

Cours 6: Coupure et Négation

- Objectifs:
 - Expliquer comment **contrôler** les retours arrière de Prolog (**backtracking**) avec le prédicat **cut**
 - Introduire la **négation**
 - Expliquer comment cut peut être inclus dans une forme plus structurée, la **négation par l'échec**
- Correspond au **chapitre 10 de LPN**

La coupure (cut)

- Le **retour arrière** automatique est l'un des caractéristiques de la recherche de solution de Prolog
- Potentiel d'inefficacité:
 - Prolog peut **perdre du temps à explorer** des possibilités qui ne mènent nulle part
- Le prédicat de **coupure !/0** offre un contrôle direct

Exemple de coupure

- La coupure est **atome spécial** de Prolog, qu'on peut utiliser en écrivant des clauses.
 - Exemple:
p(X):- b(X), c(X), !, d(X), e(X).

Exemple de coupure

- La coupure est **atome spécial** de Prolog, qu'on peut utiliser en écrivant des clauses.
 - Exemple:
 $p(X) \text{:- } b(X), c(X), !, d(X), e(X).$
- La coupure est un but qui **réussit toujours**

Exemple de coupure

- La coupure est **atome spécial** de Prolog, qu'on peut utiliser en écrivant des clauses.
 - Exemple:
 $p(X):- b(X), c(X), !, d(X), e(X).$
- La coupure est un but qui **réussit toujours**
 - Oblige Prolog à **conserver tous choix faits depuis l'appel au but parent**, ce dernier inclus (ici: $p(X)$).

Explication de la coupure

- Pour expliquer la coupure nous allons
 1. Inspecter un fragment de code *sans* coupure, en particulier son retour arrière

Explication de la coupure

- Pour expliquer la coupure nous allons
 1. Inspecter un fragment de code *sans* coupure, en particulier son retour arrière
 2. **Insertion** d'une coupure

Explication de la coupure

- Pour expliquer la coupure nous allons
 1. Inspecter un fragment de code *sans* coupure, en particulier son retour arrière
 2. **Insertion** d'une coupure
 3. **Re-inspecter** le fragment de code modifié et les **effets** de la coupure **sur le retour en arrière** (backtracking)

1: fragment sans coupure

```
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).
```

1: fragment sans coupure

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).

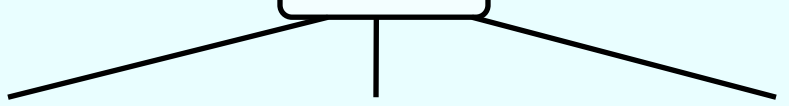
?- p(X).

1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

$?- p(X).$

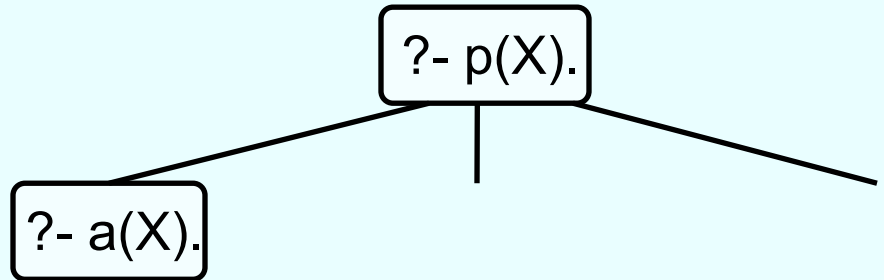
$?- p(X).$



1: fragment sans coupure

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).

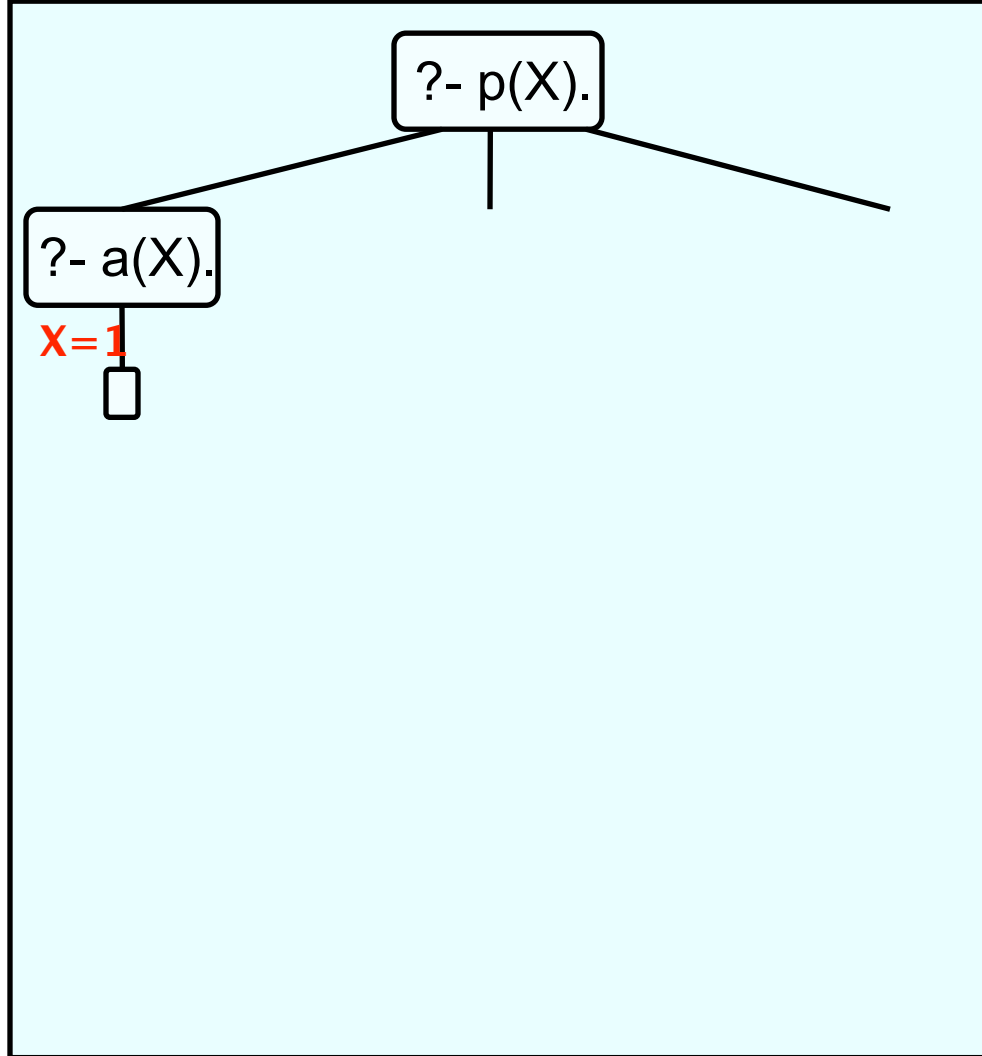
?- p(X).



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

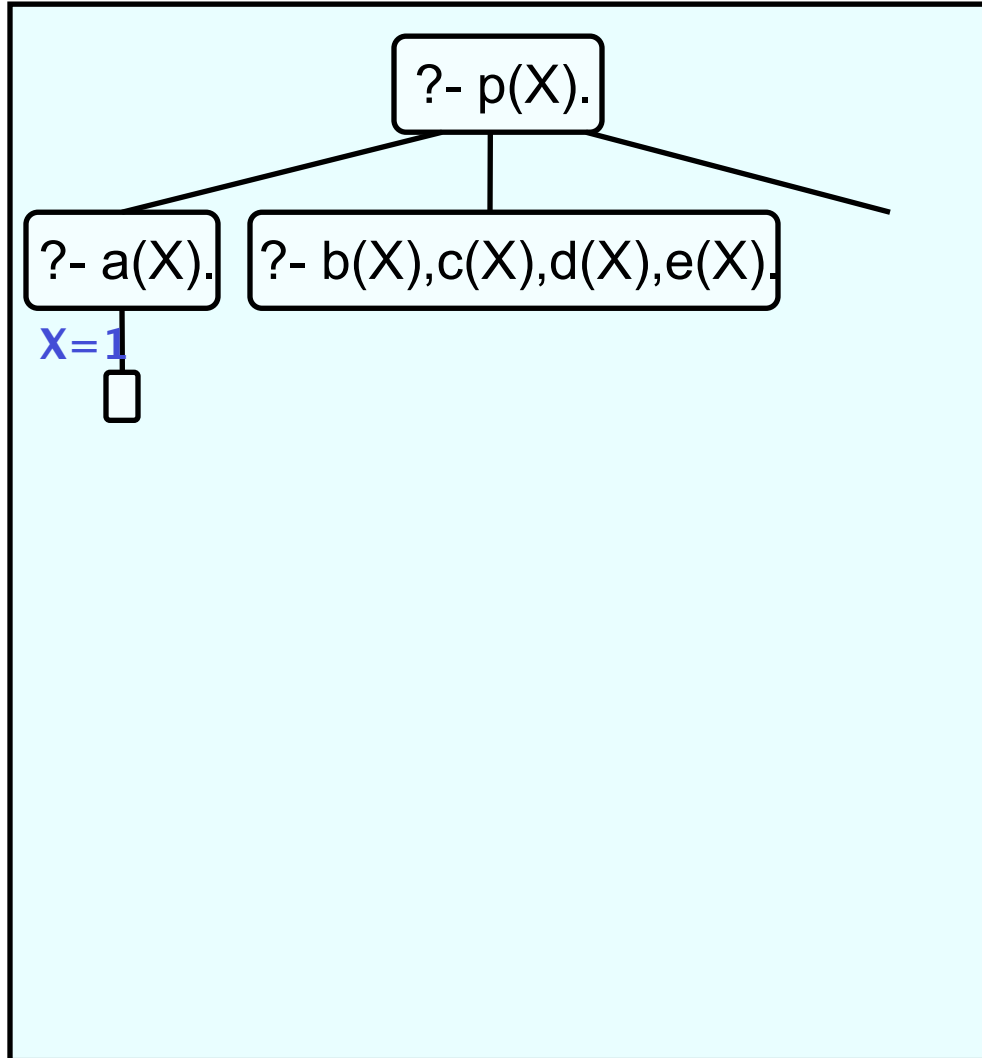
$?- p(X).$
 $X=1$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

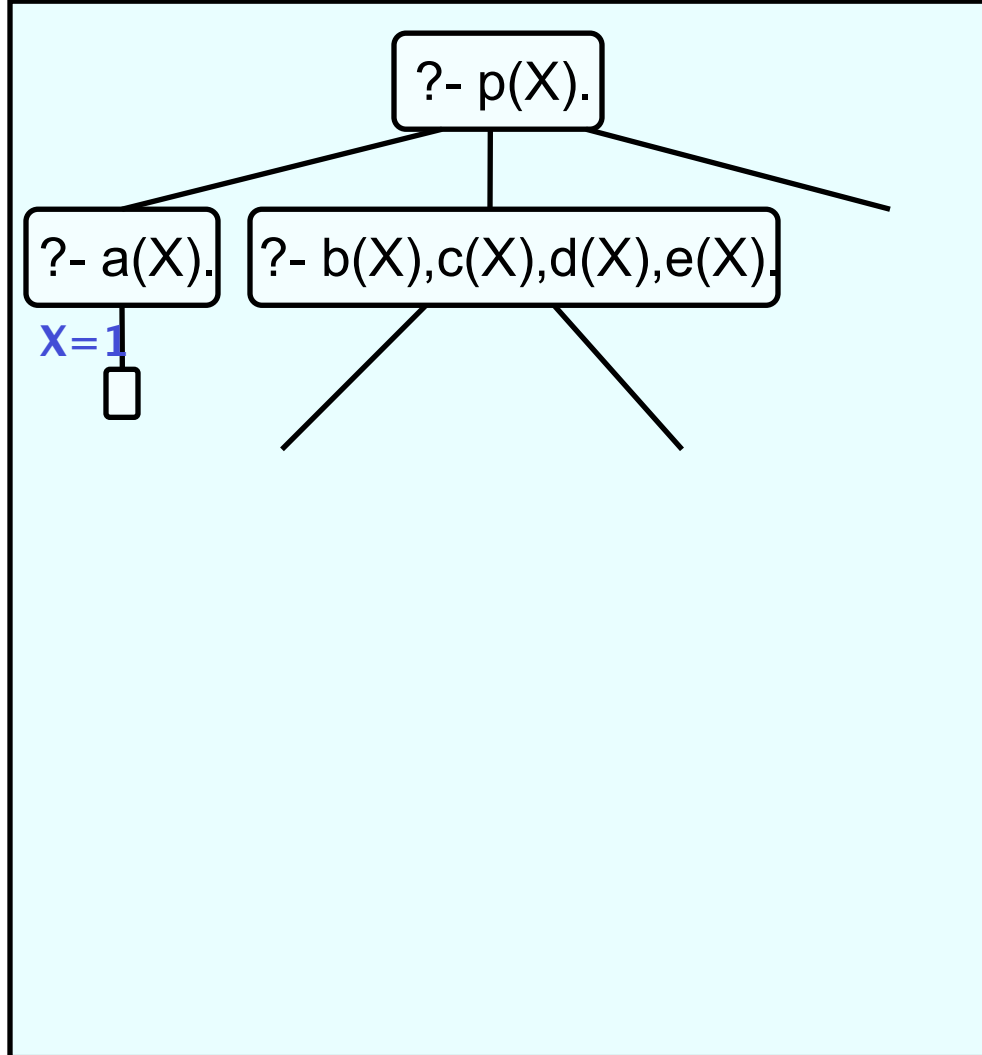
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

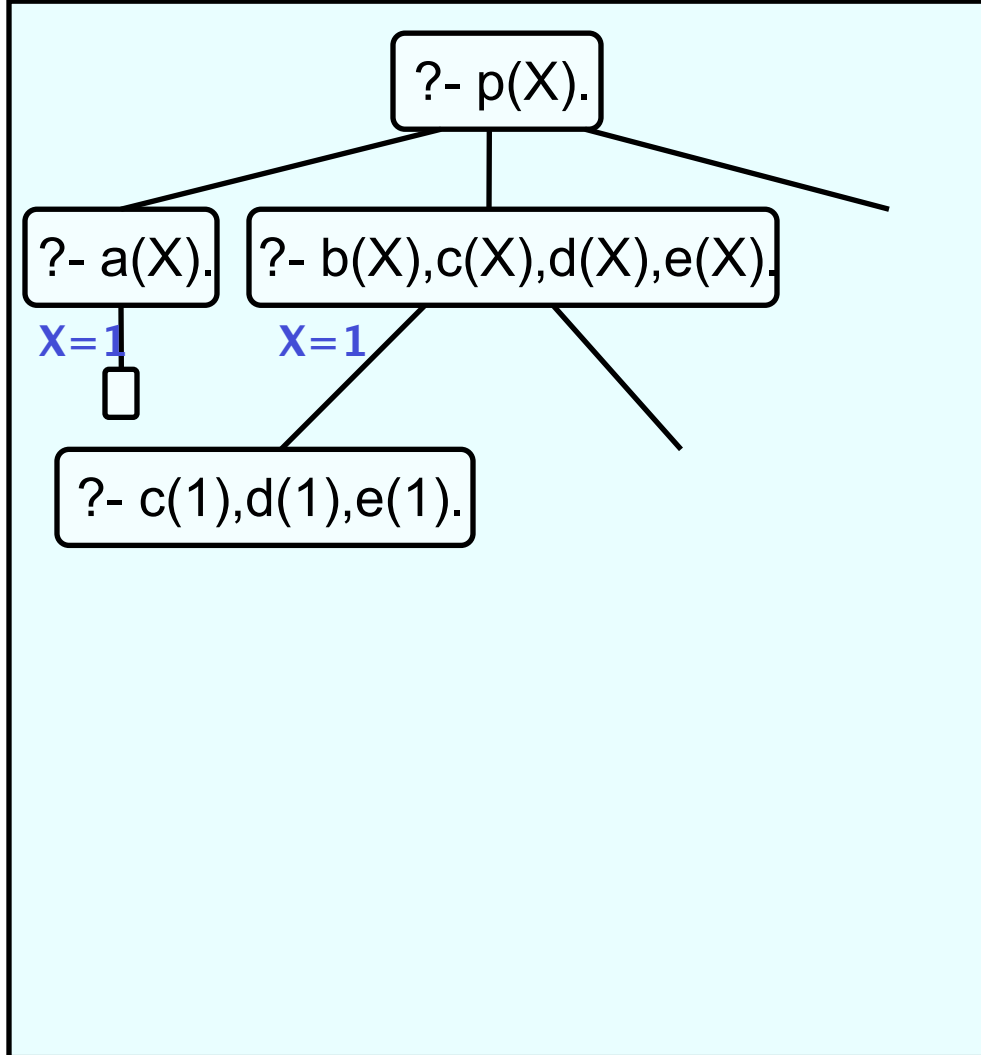
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

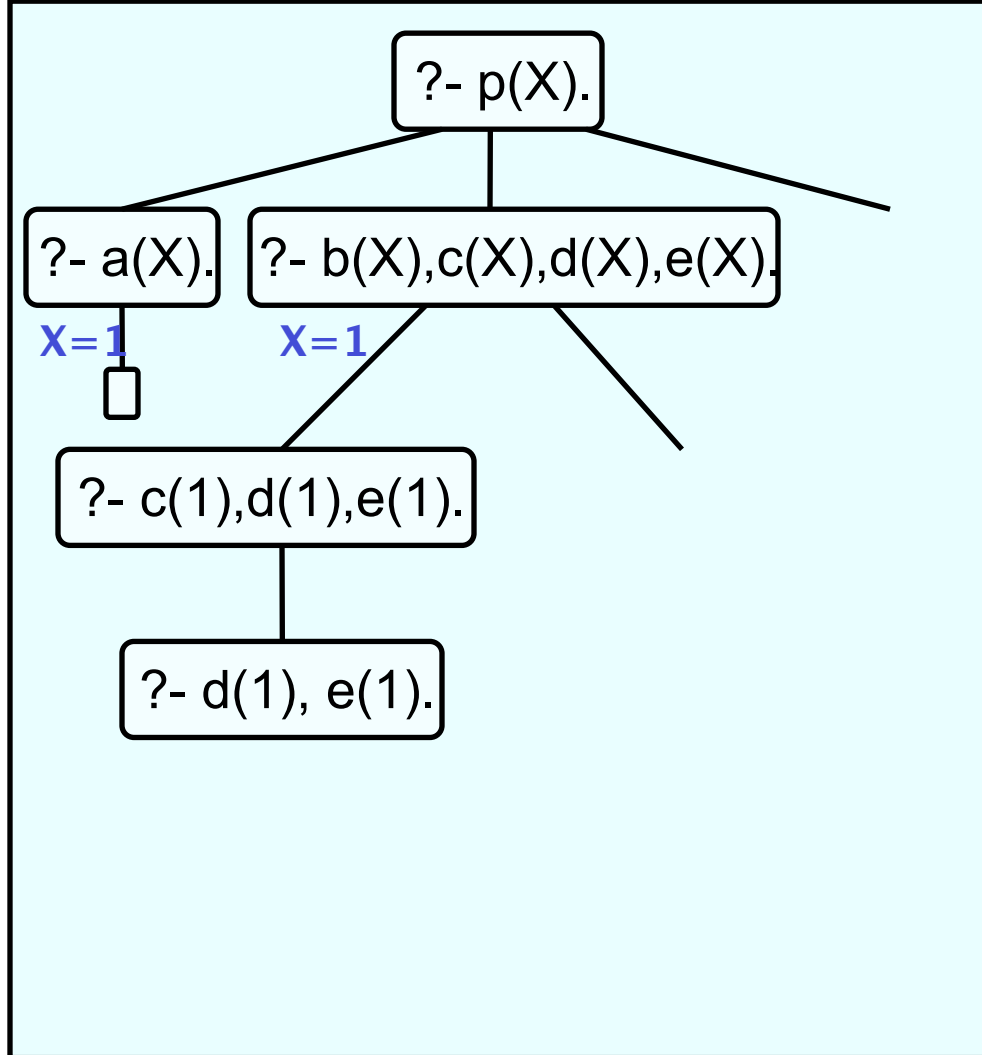
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

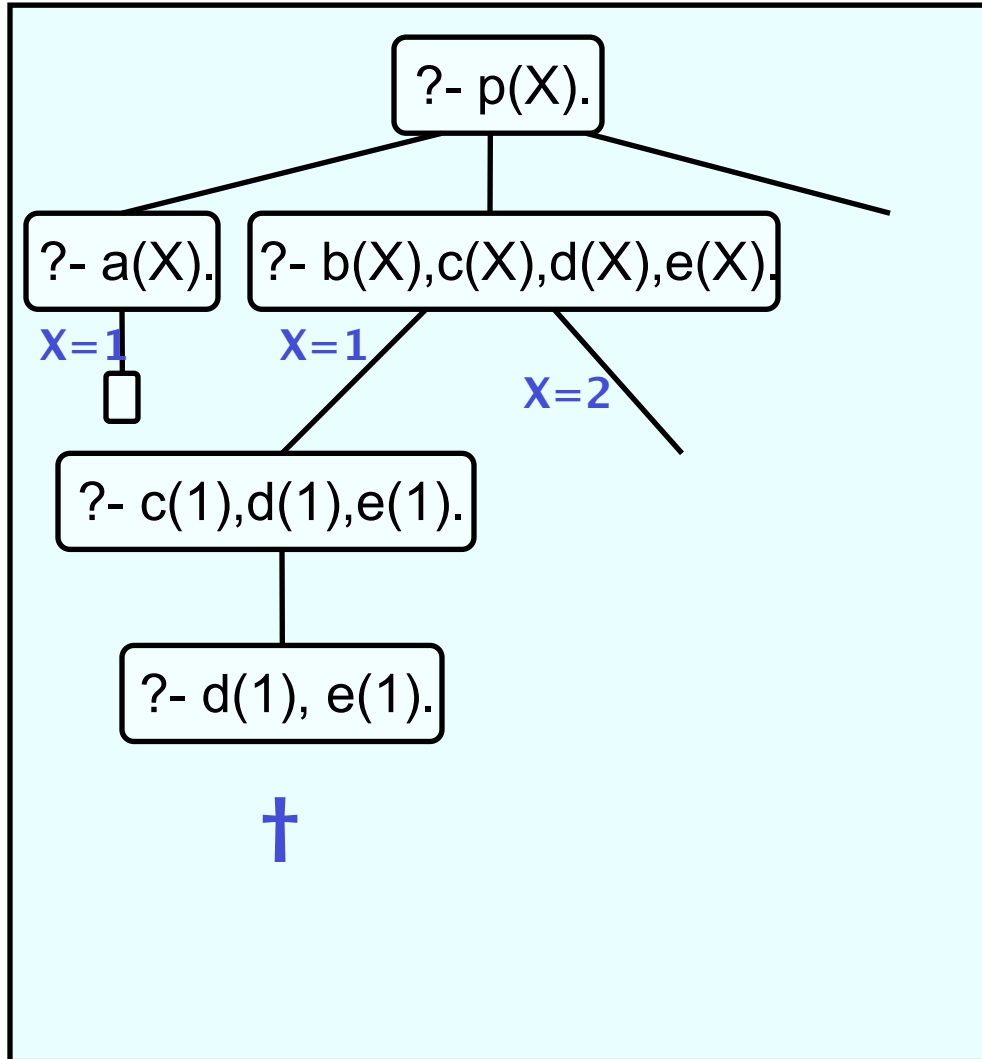
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

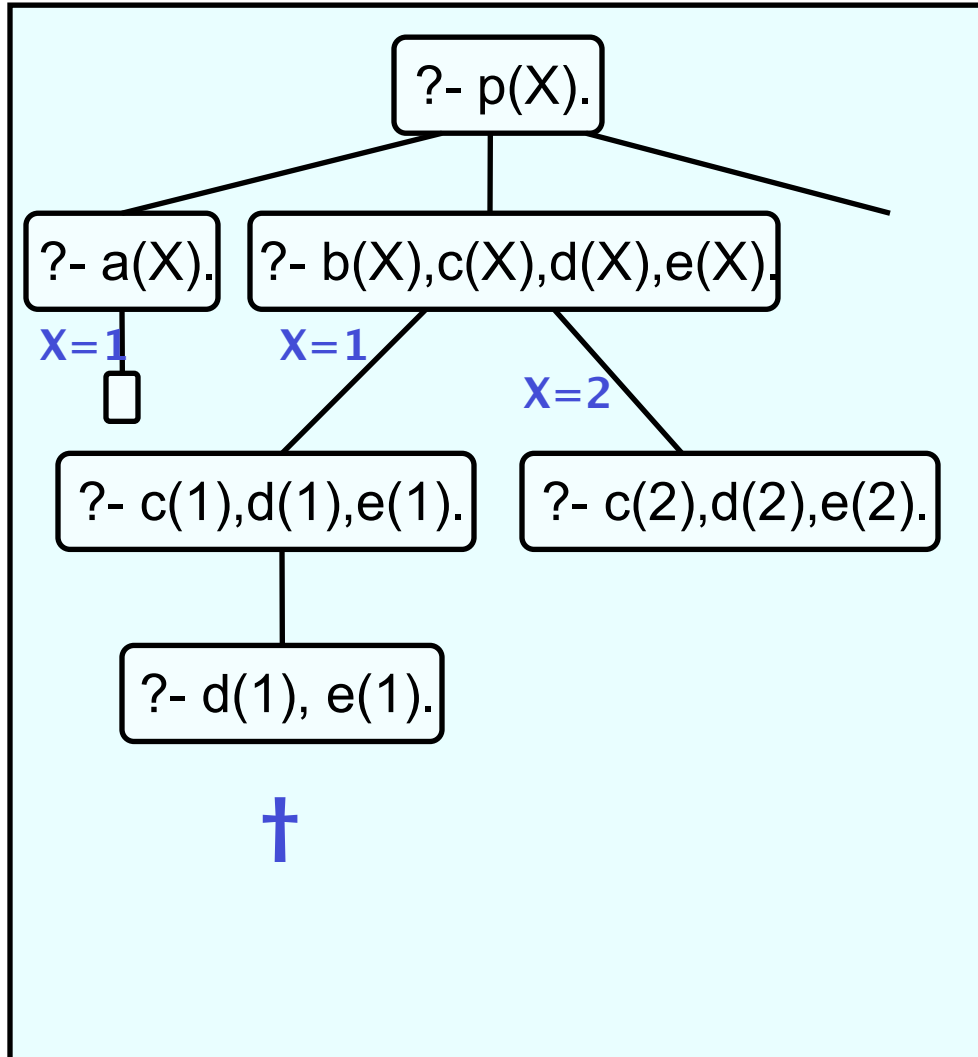
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

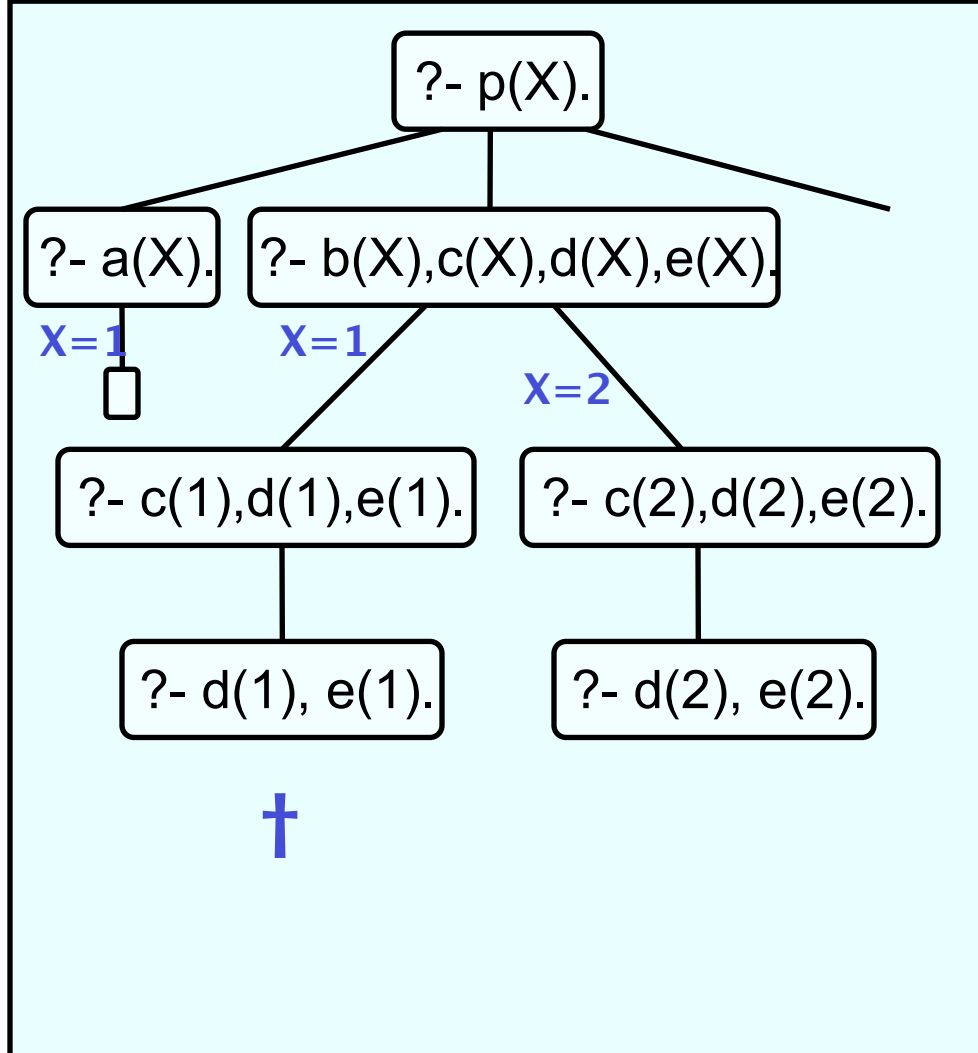
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

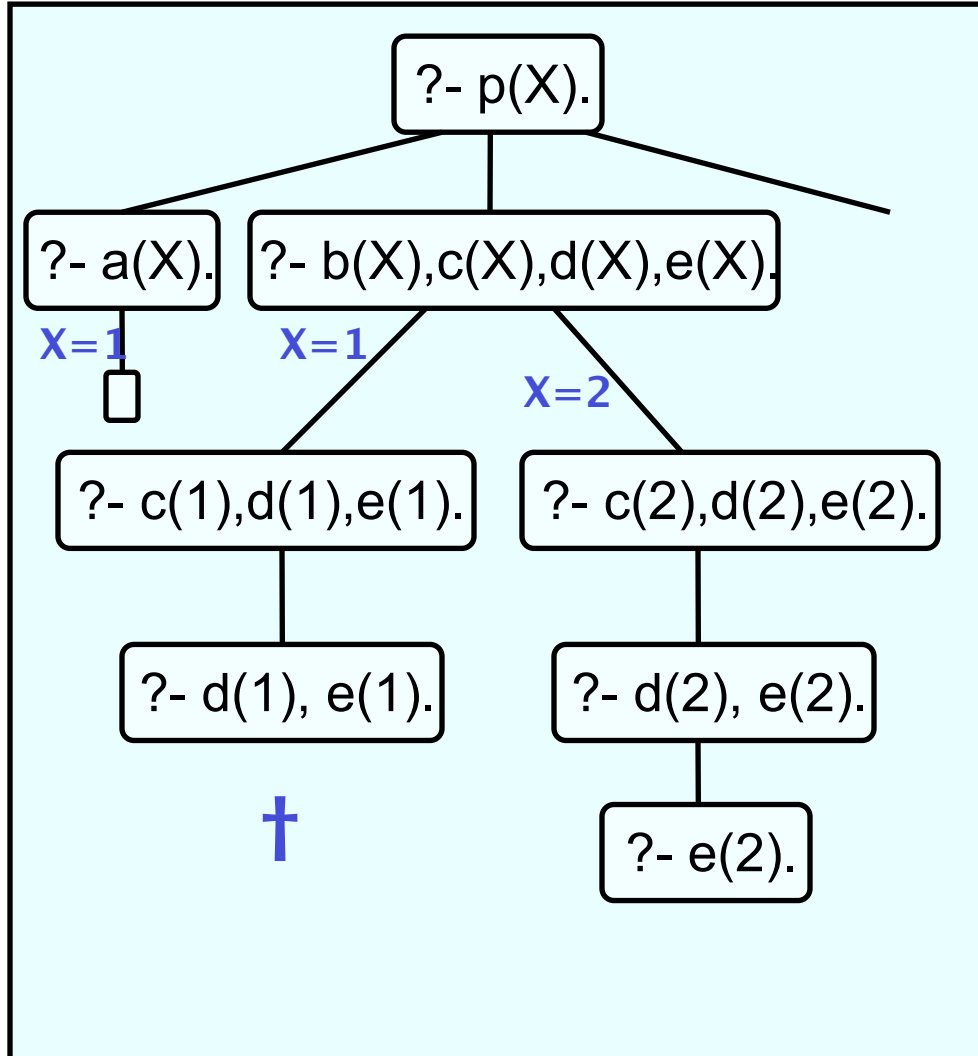
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

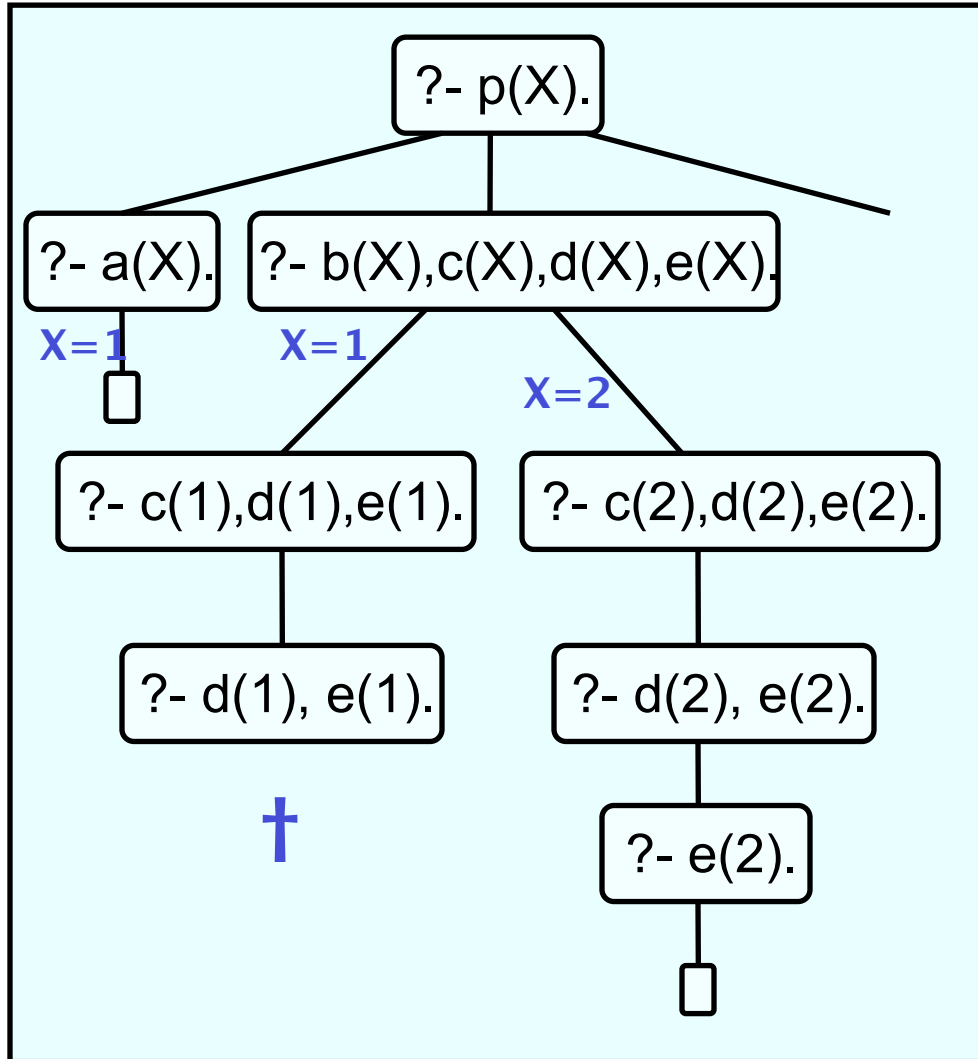
$?- p(X).$
 $X=1;$



1: fragment sans coupure

$p(X) \text{:- } a(X).$
 $p(X) \text{:- } b(X), c(X), d(X), e(X).$
 $p(X) \text{:- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

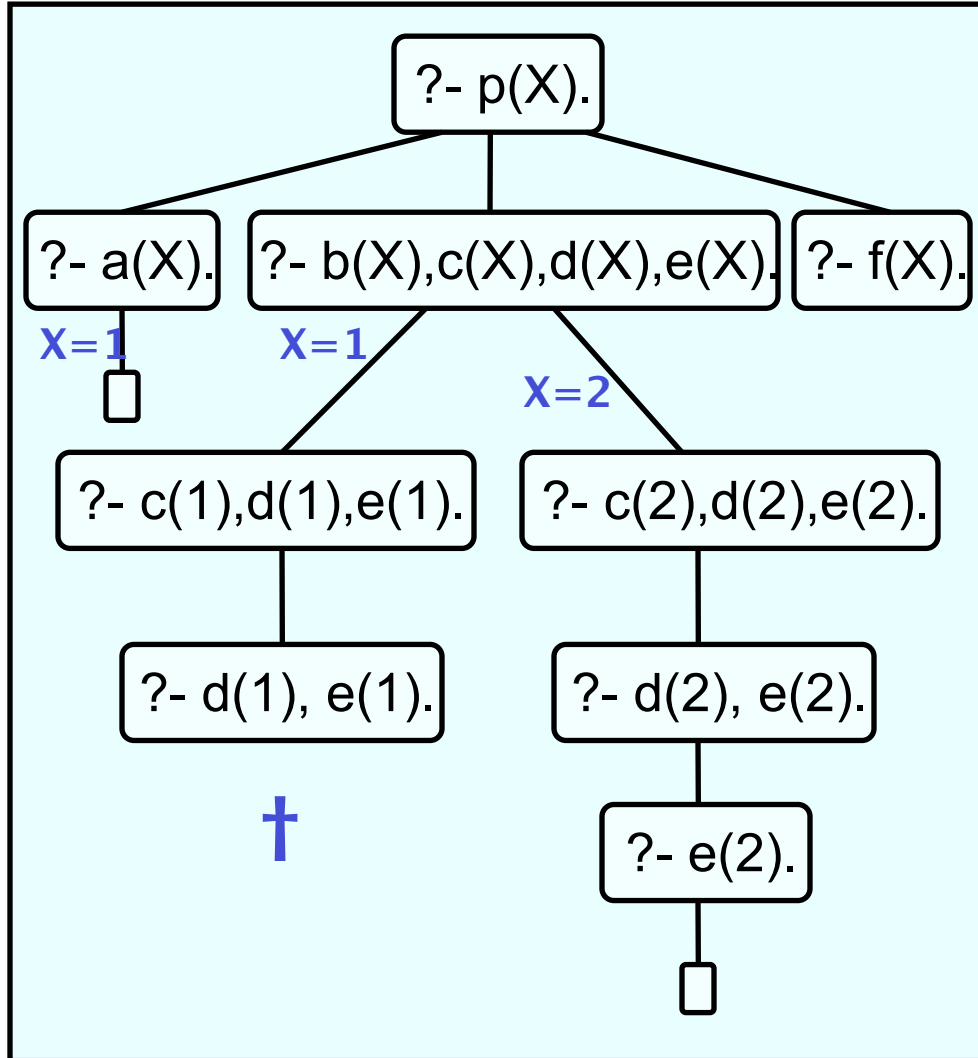
$?- p(X).$
 $X=1;$
 $X=2$



1: fragment sans coupure

$p(X) \text{ :- } a(X).$
 $p(X) \text{ :- } b(X), c(X), d(X), e(X).$
 $p(X) \text{ :- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

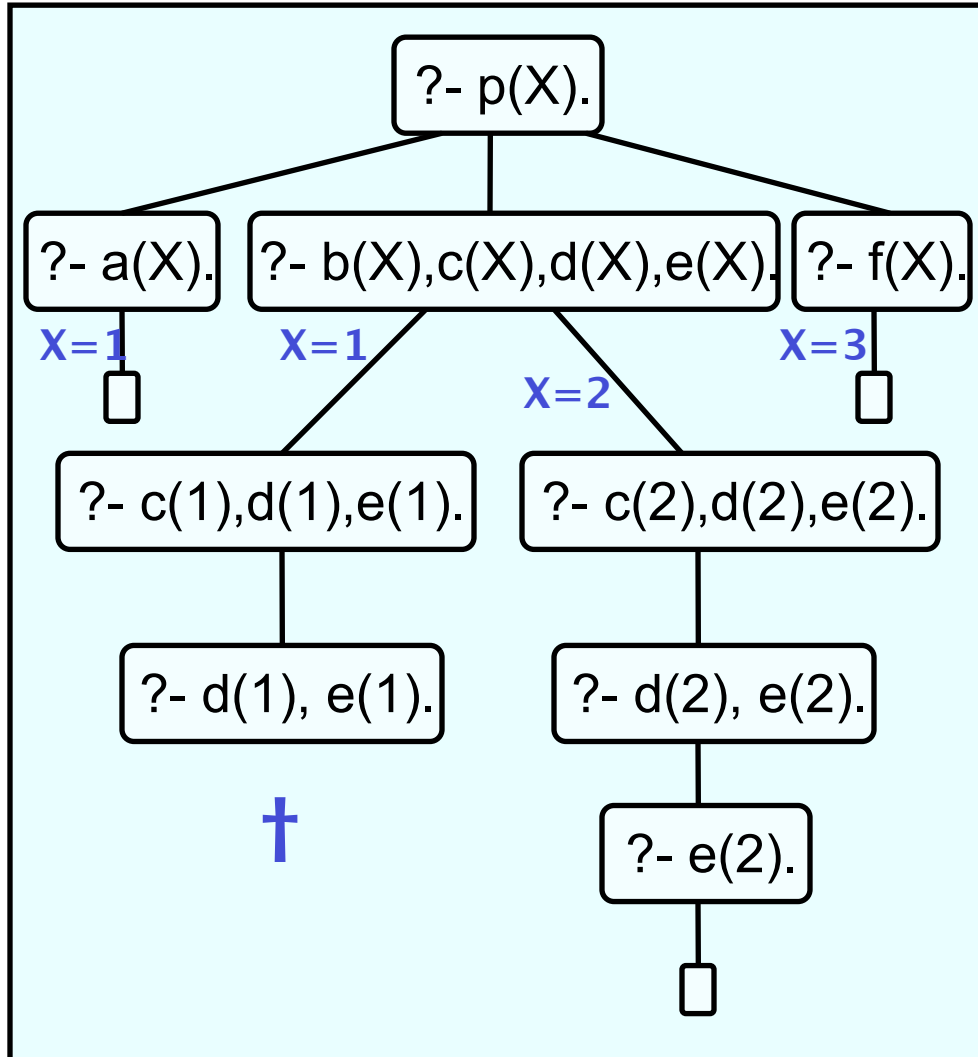
$?- p(X).$
 $X=1;$
 $X=2;$



1: fragment sans coupure

$p(X) \text{ :- } a(X).$
 $p(X) \text{ :- } b(X), c(X), d(X), e(X).$
 $p(X) \text{ :- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

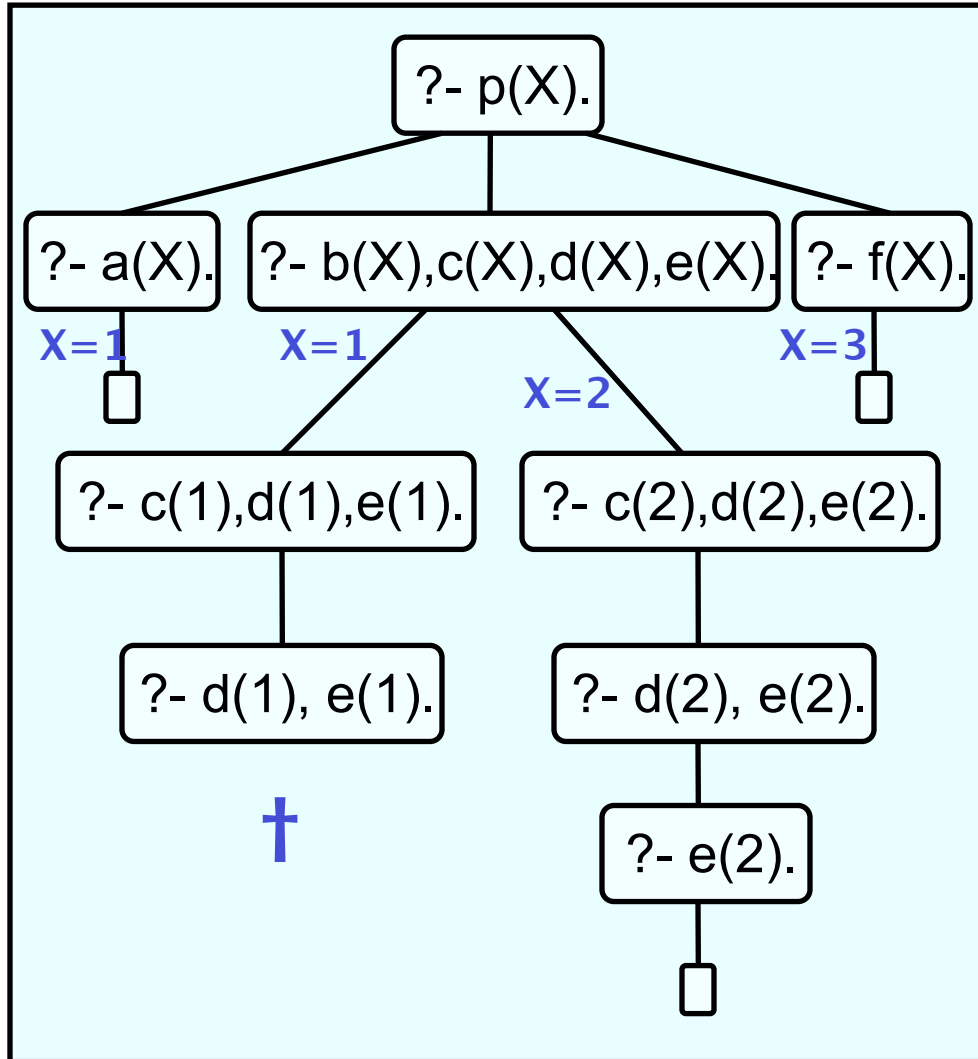
$?- p(X).$
 $X=1;$
 $X=2;$
 $X=3$



1: fragment sans coupure

$p(X) \text{ :- } a(X).$
 $p(X) \text{ :- } b(X), c(X), d(X), e(X).$
 $p(X) \text{ :- } f(X).$
 $a(1).$
 $b(1). \quad b(2).$
 $c(1). \quad c(2).$
 $d(2). \quad e(2).$
 $f(3).$

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



2: insertion d'une coupure

- On insère un cut dans la deuxième clause:

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

- La même requête donne la réponse suivante:

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

3: effet de la coupure

```
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).
```

3: effet de la coupure

```
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).
```

3: effet de la coupure

```
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).
```

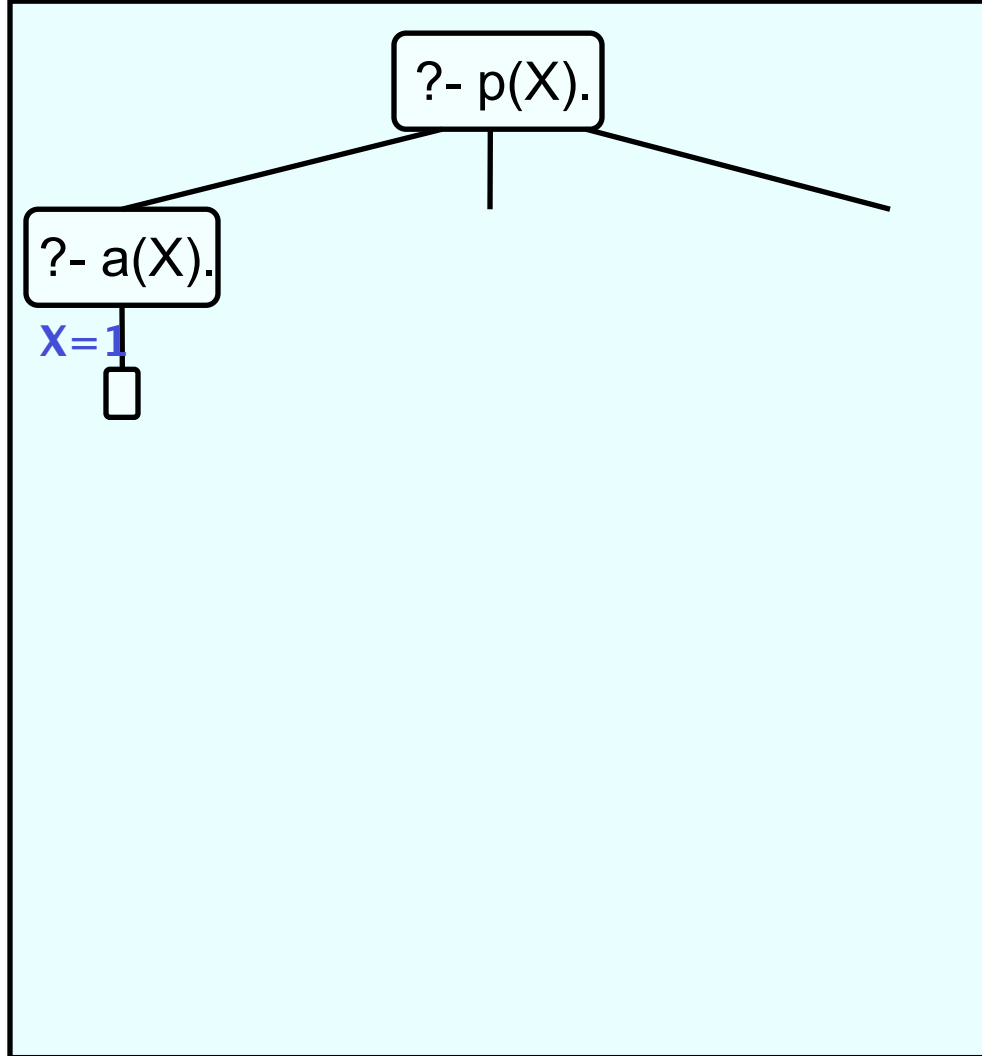
?- p(X).



3: effet de la coupure

```
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).
```

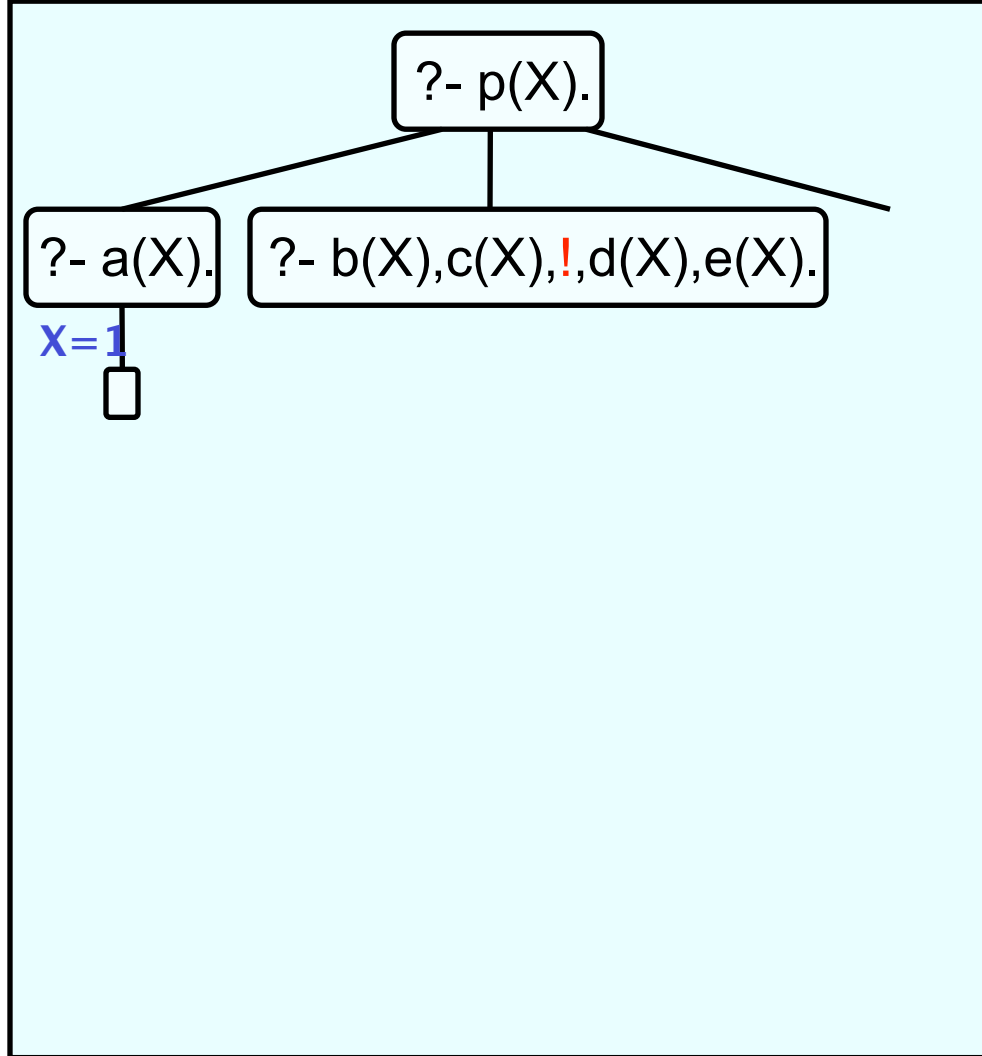
```
?- p(X).  
X=1
```



3: effet de la coupure

```
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).
```

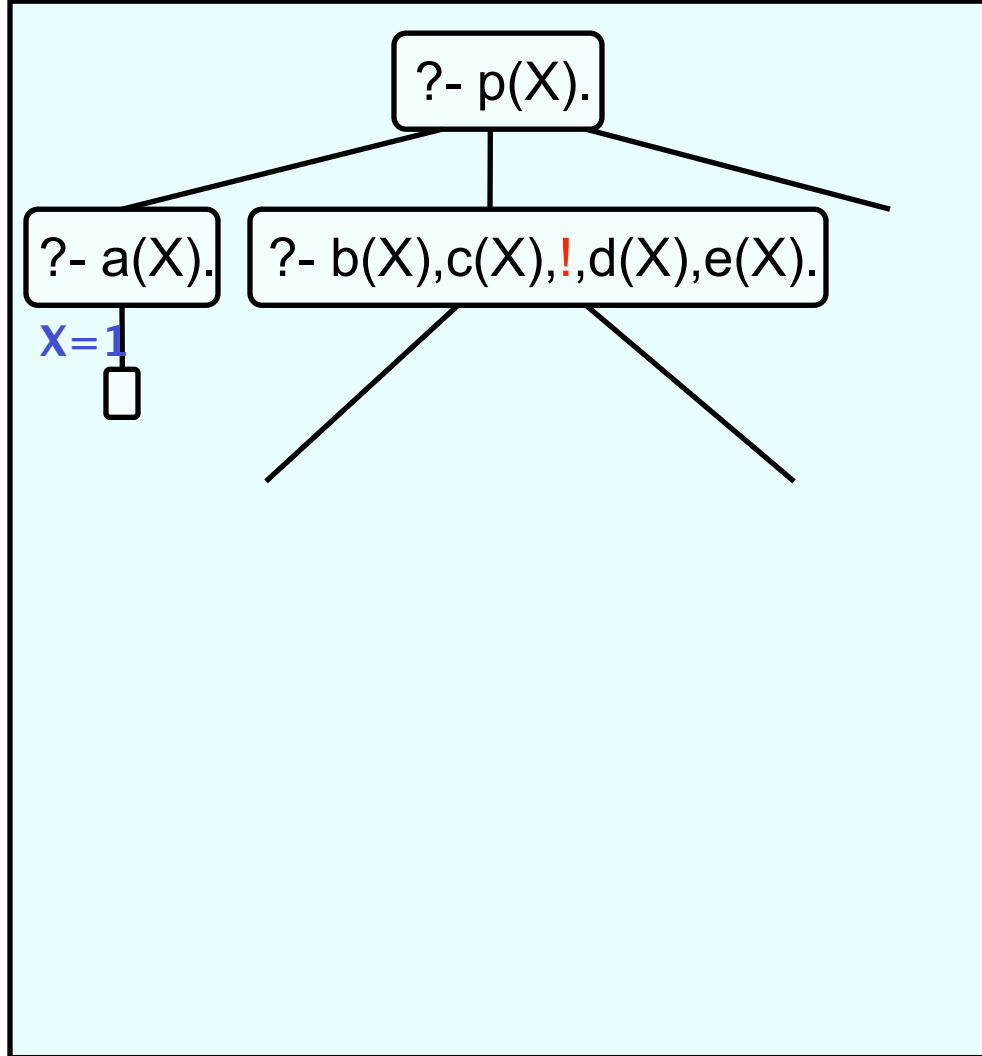
```
?- p(X).  
X=1;
```



3: effet de la coupure

```
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).
```

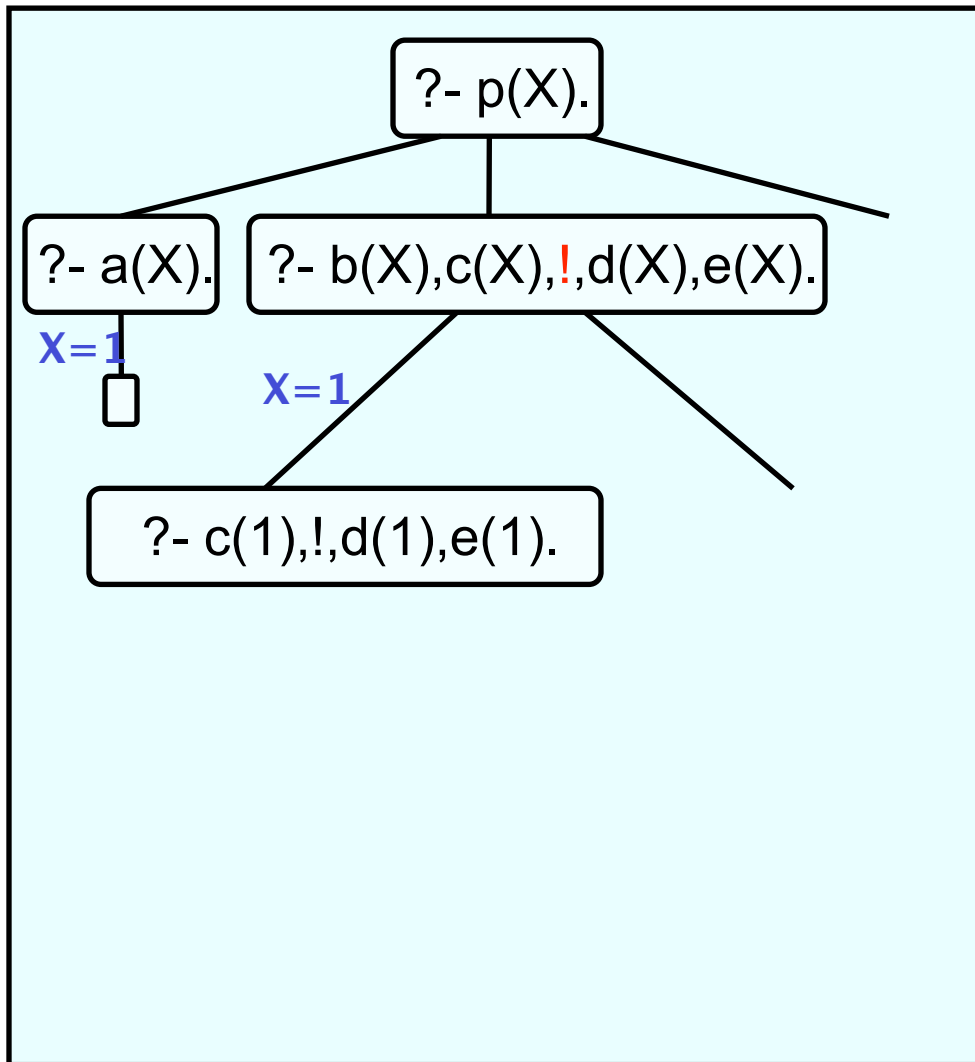
```
?- p(X).  
X=1;
```



3: effet de la coupure

```
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).
```

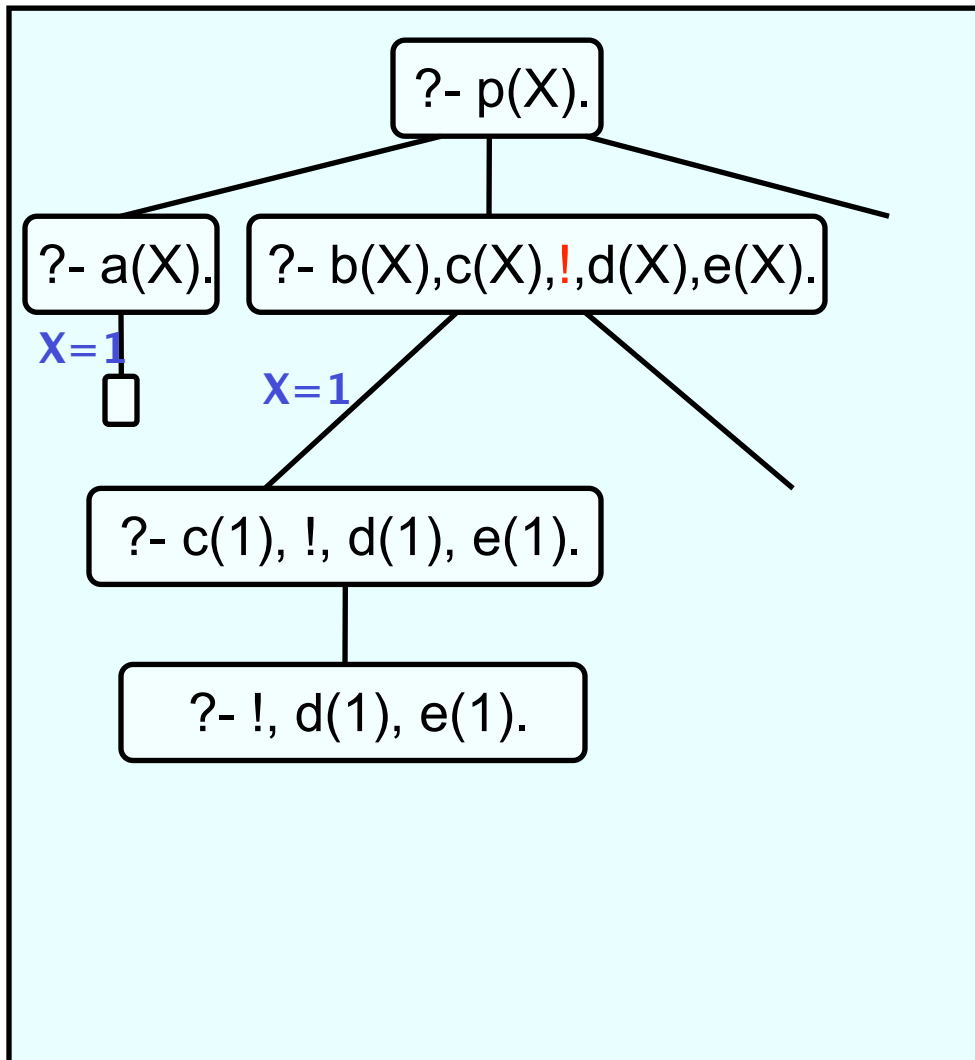
```
?- p(X).  
X=1;
```



3: effet de la coupure

```
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).
```

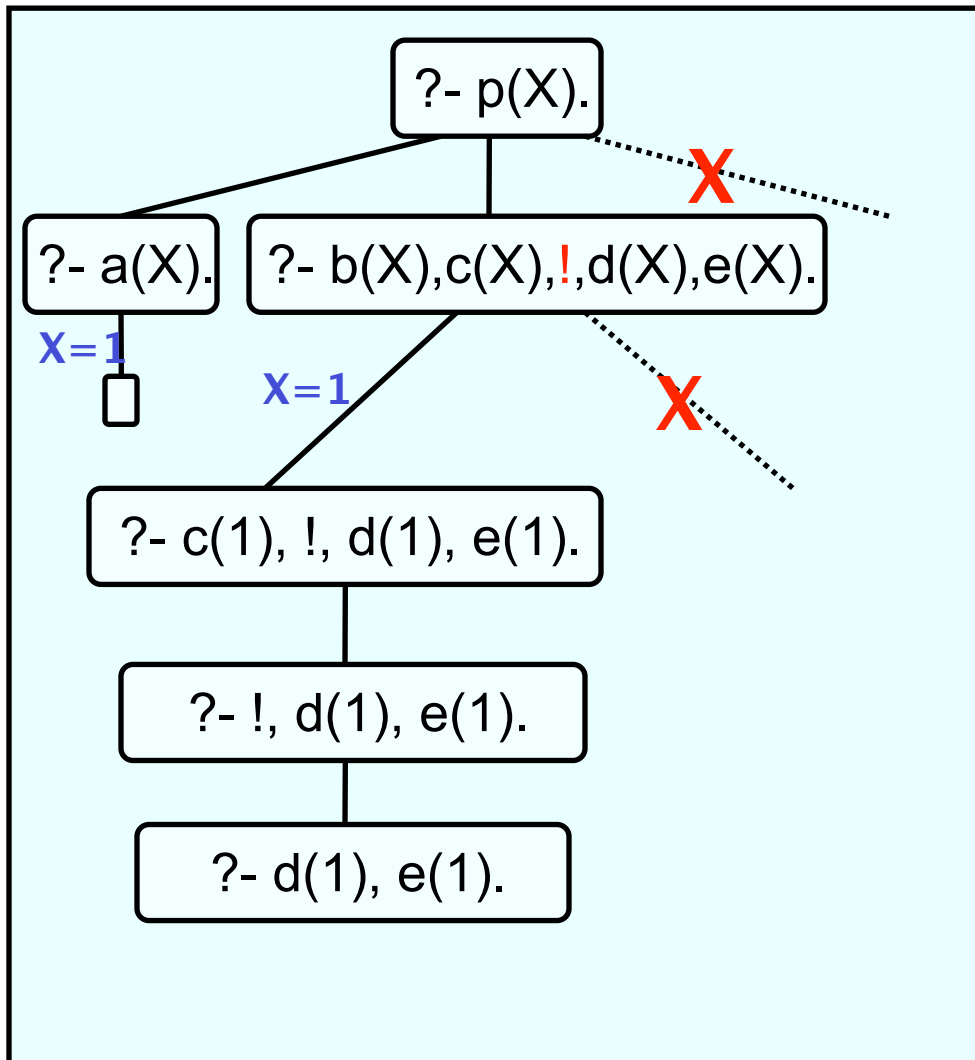
```
?- p(X).  
X=1;
```



3: effet de la coupure

```
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).
```

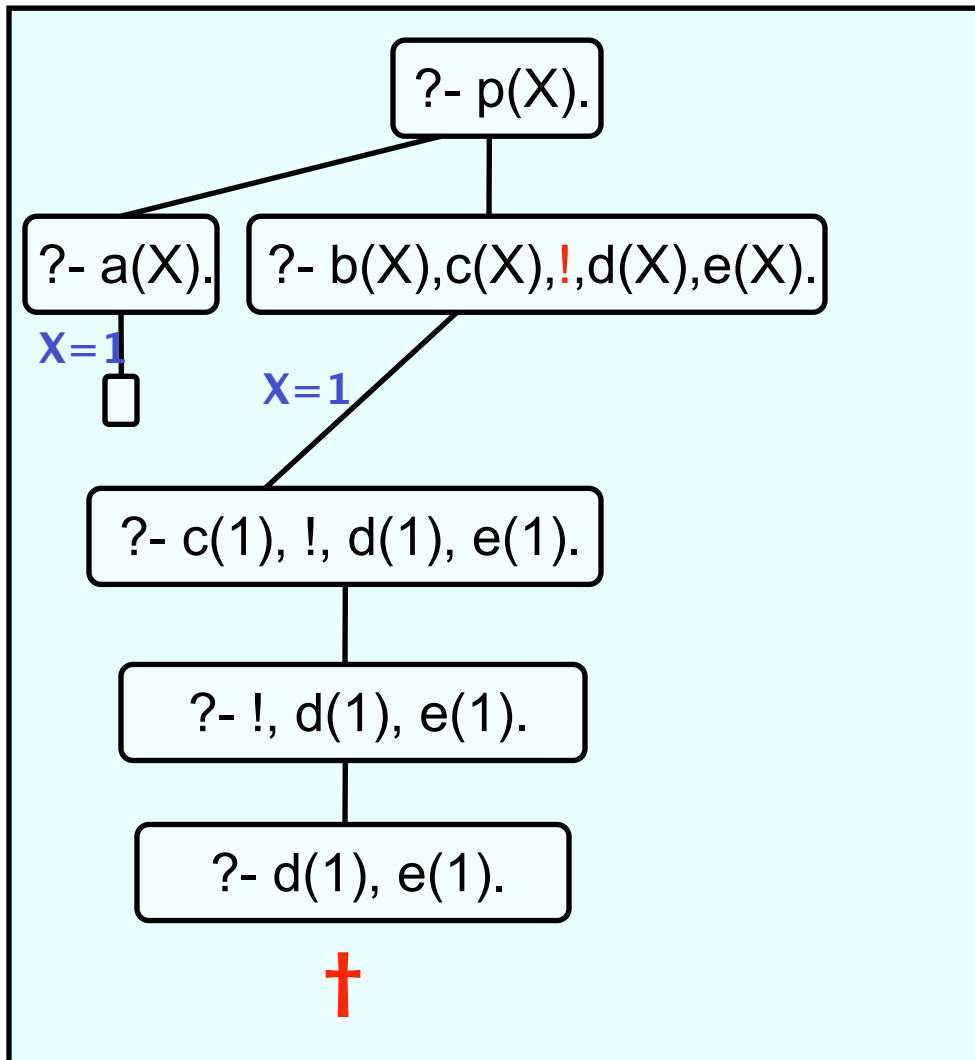
```
?- p(X).  
X=1;
```



3: effet de la coupure

```
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;  
fail
```



Ce que fait la coupure

- La coupure n'oblige que de conserver les choix faits depuis l'unification du but parent avec la tête de la règle contenant la coupure
- Exemple

$q:- p_1, \dots, p_n, !, r_1, \dots, r_n.$

en arrivant à la coupure on doit conserver:

1. le choix de **cette clause pour le prédicat q**
2. les choix faits par **p_1, \dots, p_n**
3. MAIS: les choix pour r_1, \dots, r_n restent ouverts

Exo 1

- Supposez la base de connaissances suivante:

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

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

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

```
?- p(X), ! , p(Y).
```

- Quelles sont les réponses?

Exo 1

- Réponses Q1:

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

- prochaine question:
?- p(X), p(Y).

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

Exo 1

- Réponses Q2:

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

- prochaine question:
?- p(X), !, p(Y).

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

```
X = 1
```

```
Y = 1;
```

```
X = 1
```

```
Y = 2;
```

```
X = 2
```

```
Y = 1;
```

```
X = 2
```

```
Y = 2;
```

```
fail
```


Exo 1

- Réponses Q3:

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

```
?- p(X), ! , p(Y).
```

```
X = 1
```

```
Y = 1;
```

```
X = 1
```

```
Y = 2;
```

```
fail
```

Utiliser la coupure

- Prédicat max/3 réussit si le 3^{ème} argument est le maximum des deux autres

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

Utiliser la coupure

- Prédicat max/3 réussit si le 3^{ème} argument est le maximum des deux autres

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

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

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

Utiliser la coupure

- Prédicat max/3 réussit si le 3^{ème} argument est le maximum des deux autres

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

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

```
?- max(2,3,5).  
fail
```

Le prédicat max/3

- Où est le problème?
- Potentiellement inefficace
 - Un appel a `max(3,4,Y)` à l'intérieur d'un programme.
 - Unification correcte de `Y` avec `4`.
 - Dans `p`, on force un retour en arrière. Donc on essaie de satisfaire la deuxième clause de `max/3`.
 - Quel gaspillage!

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

max/3 avec coupure

- On répare par une coupure.

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

- Comment ça fonctionne:
 - Si $X \leq Y$ réussit, la coupure fixe les substitutions des variables. On ne touche pas à la deuxième clause de max/3.
 - Si $X \leq Y$ échoue, Prolog entre dans la deuxième clause.

Coupures vertes

- Les **coupures vertes** ne changent pas le sens d'un prédicat
- max/3 en est un exemple:
 - le nouveau code donne les mêmes réponses que l'ancienne version,
 - mais est **plus efficace**

Autre max/3 avec coupure

- Pourquoi pas éliminer le corps de la deuxième clause? Après tout il est **répétitif**.

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

% X > Y effacé

- Ça marche...?

Autre max/3 avec coupure

- Pourquoi pas éliminer le corps de la deuxième clause? Après tout il est répétitif.

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

- Ça marche...?
 - ça marche!

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

Autre max/3 avec coupure

- Pourquoi pas éliminer le corps de la deuxième clause? Après tout il est répétitif.

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

- Ça marche...?
 - ça marche!

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

Autre max/3 avec coupure

- Pourquoi pas éliminer le corps de la deuxième clause? Après tout il est répétitif.

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

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

- Ça marche...?
— oops....

Version finale avec coupure

- Unification *après* passage de la coupure

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

- Ca tient la route!

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

Coupures rouges

- Nous appelons rouges les coupures qui changent le sens d'un prédicat.
- La version finale de max/3 est un exemple de **coupure rouge**:
 - Nous l'éliminons et le programme n'est plus équivalent
- Les programmes avec coupures rouges
 - Sont **moins déclaratifs**
 - Peuvent être **moins lisibles**
 - Sont susceptibles à **fautes de programmations** subtiles

Exo 2: coupures vertes

- D'abord, expliquez ce programme:

```
classe(Nombre,positif) :- Nombre > 0.
```

```
classe(0,zero).
```

```
classe(Nombre,negatif) :- Nombre < 0.
```

- Maintenant, améliorez-le en ajoutant des *cuts* verts

Exo 2: coupures vertes

```
classe(Nombre,positif) :- Nombre > 0 , ! .
```

```
classe(0,zero) :- ! .
```

```
classe(Nombre,negatif) :- Nombre < 0 , ! .
```

Exo 3: split/3 avec et sans !

- Sans utiliser de coupure, écrivez le prédicat `split/3` qui découpe une liste d'entiers en deux listes: la première doit contenir les nombres positifs (ou nuls), l'autre les négatifs.

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

`P = [3,4,0,4]`

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

Exo 3: split/3 sans cut

```
split([ ],[ ],[ ]).
spit([Num | L ], [Num|Pos],Neg):-
    Num >= 0,
    split(L,Pos,Neg).
spit([Num | L ], Pos, [Num|Neg]):-
    Num < 0,
    split(L,Pos,Neg).
```

Exo 3: split/3 avec coupure verte

```
split([ ],[ ],[ ]) :- ! .  
split([Num | L ], [Num|Pos],Neg):-  
    Num >= 0, ! ,  
    split(L,Pos,Neg).  
split([Num | L ], Pos, [Num|Neg]):-  
    Num < 0, ! ,  
    split(L,Pos,Neg).
```

Autre prédicat pré-défini: fail/0

- Comme indique son nom, le but fail échoue dès que Prolog le rencontre dans sa recherche de preuve.
- Ceci pourrait sembler peu utile
- Rappelez-vous: quand Prolog échoue, il essaie de revenir en arrière

Vincent et le fast food

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

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

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

- Combinaison **cut-fail** pour **exceptions** à règles

Vincent et burgers

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

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

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

- Combinaison **cut-fail** pour **exceptions** à règles

```
?- adore(vincent,a).  
true
```

Vincent et burgers

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

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

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

- Combinaison **cut-fail** pour **exceptions** à règles

```
?- adore(vincent,b).  
fail
```

Vincent et burgers

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

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

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

- Combinaison **cut-fail** pour **exceptions** à règles

```
?- adore(vincent,c).  
true
```

Vincent et burgers

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

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

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

- Combinaison **cut-fail** pour **exceptions** à règles

```
?- adore(vincent,d).  
true
```


Négation par l'échec

- La combinaison coupure-fail semble offrir une sorte de négation
- Qu'on appelle **négation par l'échec**, définie comme:

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

Vincent et hamburgers ré-écrit

```
adore(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 et burgers ré-écrit

```
adore(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).
```

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

Autre prédicat pré-défini: \+

- Comme la **négation par l'échec** sert souvent, elle est pré-définie
- Par l'opérateur préfix **\+**
- Ré-écrivons les préférences alimentaires de Vincent:

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

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

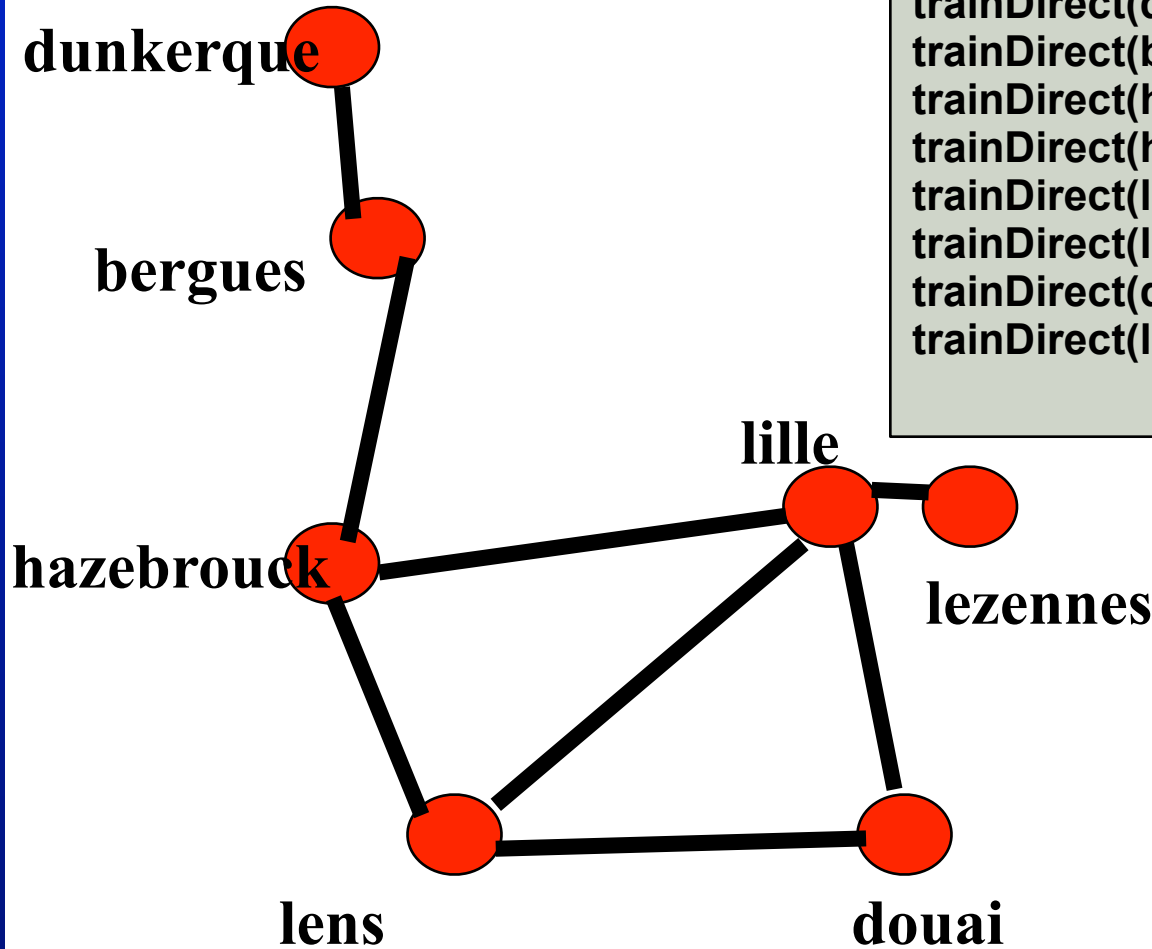
La négation par l'échec et la logique

- La négation par l'échec n'est **pas la négation logique**
- Si nous invertissons **l'ordre** des buts du programme, nous observons un autre comportement:

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

```
?- adore(vincent,X).  
fail
```

Parcours de graphes



```
trainDirect(dunkerque,bergues).  
trainDirect(bergues,hazebrouck).  
trainDirect(hazebrouck,lens).  
trainDirect(hazebrouck,lille).  
trainDirect(lens,lille).  
trainDirect(lens,douai).  
trainDirect(douai,lille).  
trainDirect(lille,lezennes).
```

Parcours de graphe

- Supposez que s'il est possible de prendre un train direct de A à B, il est aussi possible de prendre un train de B à A. Ajoutez cette information à la base de connaissances (sous forme d'un prédicat `liaisonTrain/2`). Ensuite, écrivez le prédicat `route/3` qui nous donne la liste des villes qu'on visite en prenant le train d'une ville à l'autre:

```
?- route(bergues,lezennes,X).  
X = [bergues, hazebrouck, lille, lezennes] ;  
X = [bergues, hazebrouck, lens, lille, lezennes] ;  
X = [bergues, hazebrouck, lens, douai, lille, lezennes] ;  
fail.
```

Prochain sujet

- Manipulation de bases de données et collections de solutions...
 - mais non: fini la programmation logique,
 - nous passerons à la programmation fonctionnelle.