

Logique

Prolog

Thomas Pietrzak
Licence Informatique

Prolog

Langage de programmation

Programmation logique

Base de connaissance : fichier

Questions : interpréteur

Exemple

miam.pl

```
bon(chocolat).  
mange(tom,chocolat).  
mange(tom,chouxbruxelles).
```

```
$swipl
```

```
...
```

```
?- [miam].
```

```
true.
```

on charge le fichier miam.pl

le fichier a été chargé

```
?- mange(tom,chocolat).
```

```
true.
```

est-ce que je mange du chocolat ?

je mange bien du chocolat

```
?- mange(tom,soupe).
```

```
false.
```

est-ce que je mange de la soupe ?

je ne mange pas de soupe

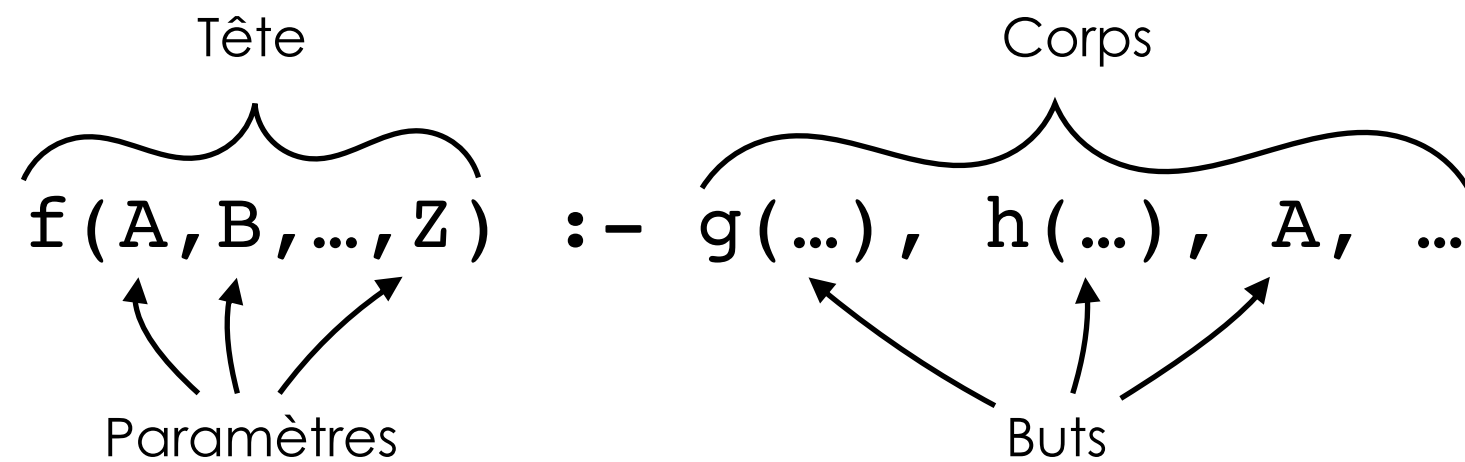
```
?- mange(tom,X), bon(X).
```

```
X = chocolat .
```

est-ce que je mange quelque chose de bon ?

oui : chocolat

Clauses



Clauses de Horn

Conjonction

$$F \text{ :- } F_1, F_2, \dots, F_n.$$

Pour démontrer F il faut montrer F_1 puis F_2 , ..., puis F_n .

Disjonction

$$F \text{ :- } F_1.$$

$$F \text{ :- } F_2.$$

...

$$F \text{ :- } F_n.$$

Pour démontrer F il suffit de montrer F_1 OU F_2 , ..., OU F_n .

Faits et Règles

Faits

```
bon(chocolat).
```

Règles

```
bonchasseur(X) :- chasseur(X), chasse(X,sanssonchien).
```

```
mauvaischasseur(X) :- chasseur(X), voit(X,Y), tire(X,Y).
```

Données

Constantes

Atomes : chaînes de caractères minuscule

patate

Nombres : entiers ou flottants

666

Variables

chaînes de caractères commençant par une majuscule

Qui

Spécial : _

_

Prédicats

nom + arité

joue(tom,baseball)

Exercice

Quand Neo et Trinity sont au restaurant, ils choisissent leur menu avec Prolog.

Trinity est végétarienne

Les nems et le canard laqué contiennent de la viande

Neo n'aime pas le caramel

Il y a du caramel dans l'île flottante et la crème brûlée

Ils choisissent une formule « entrée, plat, dessert »

Ils veulent commander des plats différents

Entrées

Salade
Nems

Plats principaux

Canard laqué
Welsch

Desserts

Île flottante
Crème brûlée
Mousse au chocolat

Corrigé

```
entree(salade).
entree(nems).
plat(canard).
plat(welsch).
dessert(ileflottante).
dessert(cremebrulee).
dessert(mousse).

aimepas(trinity,viande).
aimepas(neo,caramel).

contient(nems,viande).
contient(canard,viande).
contient(ileflottante,caramel).
contient(cremebrulee,caramel).

mangepas(X,Y) :-
    aimepas(X,I),
    contient(Y,I).

formule(X,E,P,D) :-
    entree(E),
    plat(E),
    dessert(E),
    not(mangepas(X,E)),
    not(mangepas(X,P)),
    not(mangepas(X,D)).
```

?-

Corrigé

```
entree(salade).
entree(nems).
plat(canard).
plat(welsch).
dessert(ileflottante).
dessert(cremebrulee).
dessert(mousse).

aimepas(trinity,viande).
aimepas(neo,caramel).

contient(nems,viande).
contient(canard,viande).
contient(ileflottante,caramel).
contient(cremebrulee,caramel).

mangepas(X,Y) :-
    aimepas(X,I),
    contient(Y,I).

formule(X,E,P,D) :-
    entree(E),
    plat(E),
    dessert(E),
    not(mangepas(X,E)),
    not(mangepas(X,P)),
    not(mangepas(X,D)).
```

```
?- formule(neo,E1,P1,D1),
   formule(trinity,E2,P2,D2),
   E1 \= E2,
   P1 \= P2,
   D1 \= D2.
```

Corrigé

```
entree(salade).
entree(nems).
plat(canard).
plat(welsch).
dessert(ileflottante).
dessert(cremebrulee).
dessert(mousse).

aimepas(trinity,viande).
aimepas(neo,caramel).

contient(nems,viande).
contient(canard,viande).
contient(ileflottante,caramel).
contient(cremebrulee,caramel).

mangepas(X,Y) :-
    aimepas(X,I),
    contient(Y,I).

formule(X,E,P,D) :-
    entree(E),
    plat(E),
    dessert(E),
    not(mangepas(X,E)),
    not(mangepas(X,P)),
    not(mangepas(X,D)).
```

```
?- formule(neo,E1,P1,D1),
    formule(trinity,E2,P2,D2),
    E1 \= E2,
    P1 \= P2,
    D1 \= D2.

E1 = nems,
P1 = canard,
D1 = mousse,
E2 = salade,
P2 = welsch,
D2 = ileflottante
```

Corrigé

```
entree(salade).
entree(nems).
plat(canard).
plat(welsch).
dessert(ileflottante).
dessert(cremebrulee).
dessert(mousse).

aimepas(trinity,viande).
aimepas(neo,caramel).

contient(nems,viande).
contient(canard,viande).
contient(ileflottante,caramel).
contient(cremebrulee,caramel).

mangepas(X,Y) :-
    aimepas(X,I),
    contient(Y,I).

formule(X,E,P,D) :-
    entree(E),
    plat(E),
    dessert(E),
    not(mangepas(X,E)),
    not(mangepas(X,P)),
    not(mangepas(X,D)).
```

```
?- formule(neo,E1,P1,D1),
    formule(trinity,E2,P2,D2),
    E1 \= E2,
    P1 \= P2,
    D1 \= D2.

E1 = nems,
P1 = canard,
D1 = mousse,
E2 = salade,
P2 = welsch,
D2 = ileflottante

E1 = nems,
P1 = canard,
D1 = mousse,
E2 = salade,
P2 = welsch,
D2 = cremebrulee
```

Arbre décisionnel

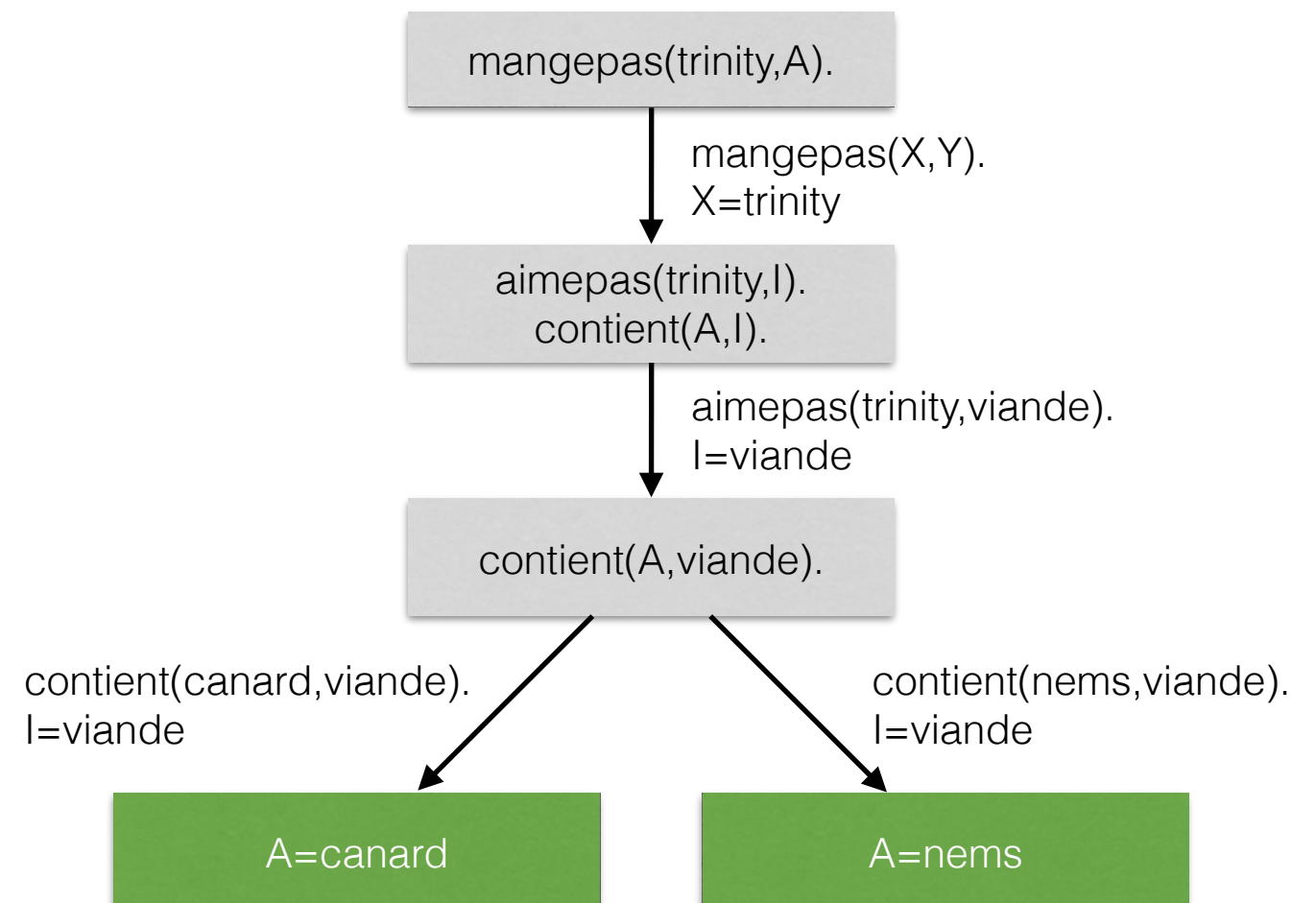
Racine : tête

Nœuds : points de choix

Feuilles : réponse ou échec

Parcours en profondeur

Backtracking



Unification

Étape cruciale du calcul

Instancie les variables

Prédicat d'unification =

Unificateur : fonction de substitution

Exemples

$f(A, B, C) = f(1, 2, 3).$

$A = 1,$

$B = 2,$

$C = 3.$

$f(X, X) = f(1, 2).$

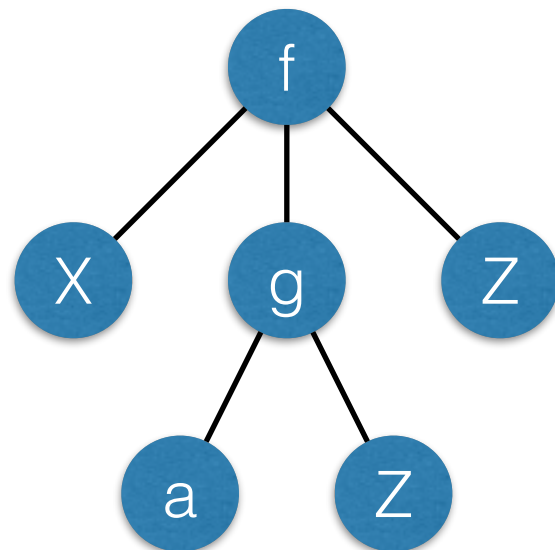
false.

Algorithme d'Unification

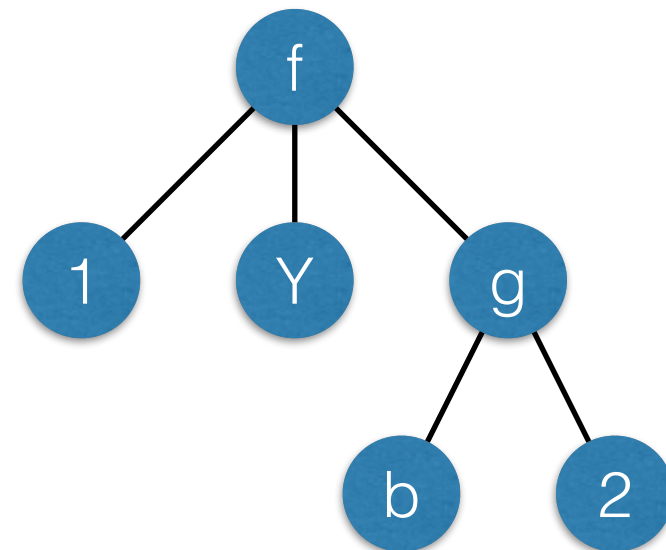
```
unifier( $T_1$ ,  $T_2$ )  
  si  $T_1$  et  $T_2$  sont des constantes identiques  $\Rightarrow$  true.  
  si  $T_1$  et  $T_2$  sont des constantes différentes  $\Rightarrow$  false.  
  si  $T_1$  est une variable  $\Rightarrow$  ajouter  $T_1 \leftarrow T_2$  à l'unificateur.  
  sinon si  $T_2$  est une variable  $\Rightarrow$  ajouter  $T_2 \leftarrow T_1$  à l'unificateur.  
  sinon  $T_1 = f(x_1, \dots, x_n)$  et  $T_2 = g(y_1, \dots, y_m)$   
    si  $n \neq m \Rightarrow$  false.  
    si  $f \neq g \Rightarrow$  false.  
    sinon pour  $i$  de 1 à  $n$   
      unifier( $x_i, y_i$ )
```

Exemple

$$t_1 = f(X, g(a, Z), Z)$$

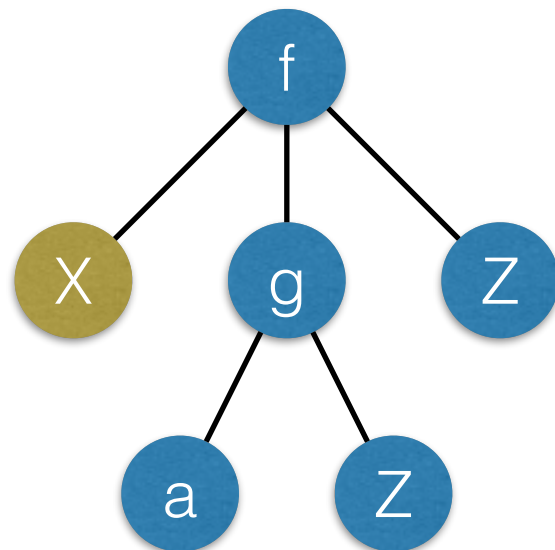


$$t_2 = f(1, Y, g(b, 2))$$

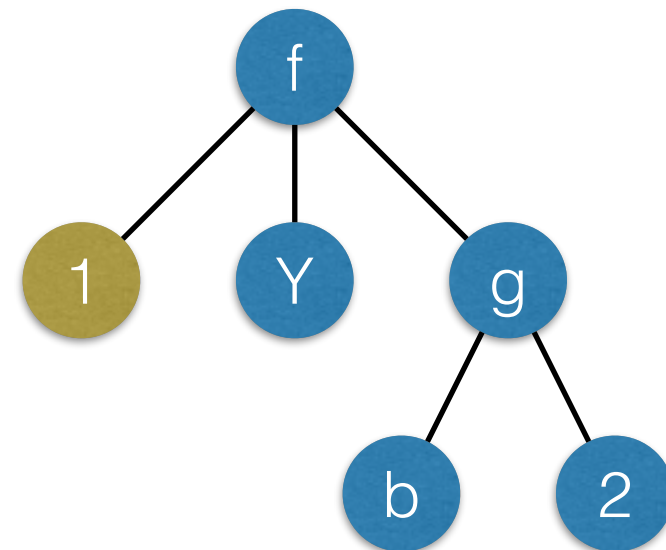


Exemple

$$t_1 = f(X, g(a, Z), Z)$$



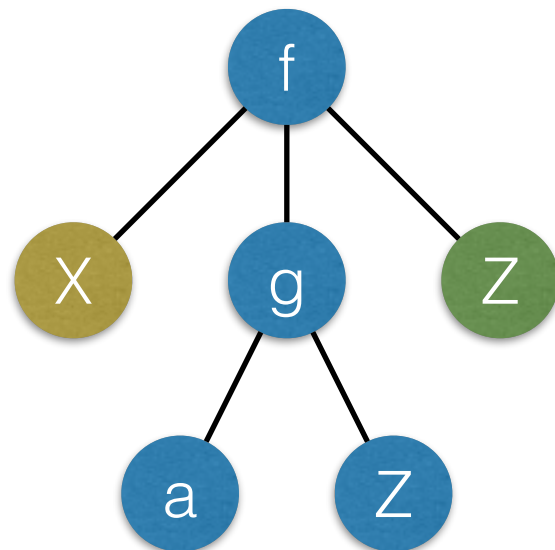
$$t_2 = f(1, Y, g(b, 2))$$



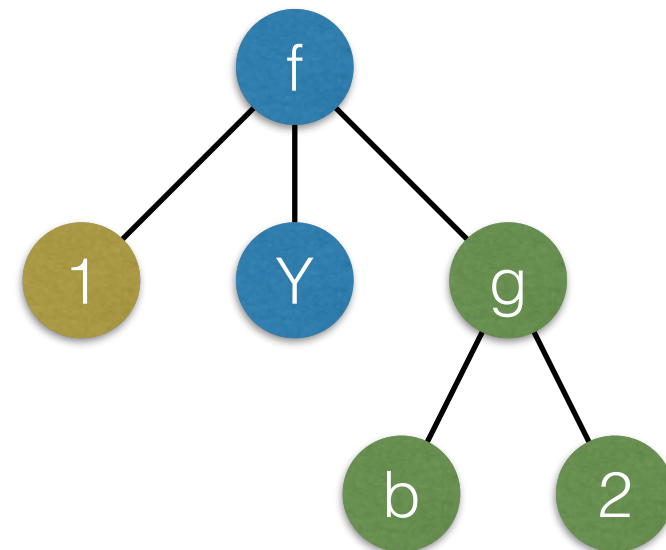
$$X = 1$$

Exemple

$$t_1 = f(X, g(a, Z), Z)$$



$$t_2 = f(1, Y, g(b, 2))$$

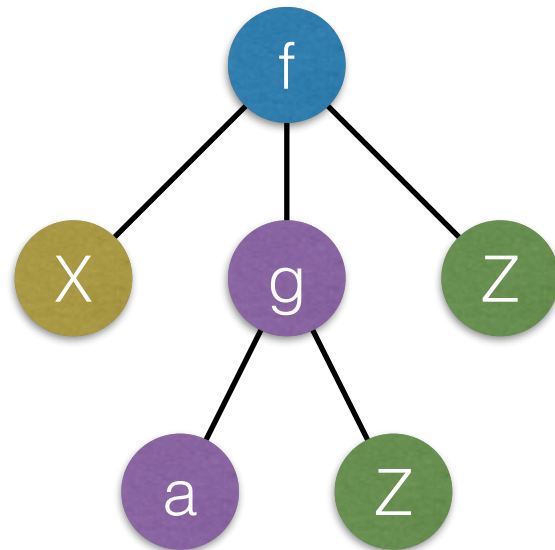


$$X = 1$$

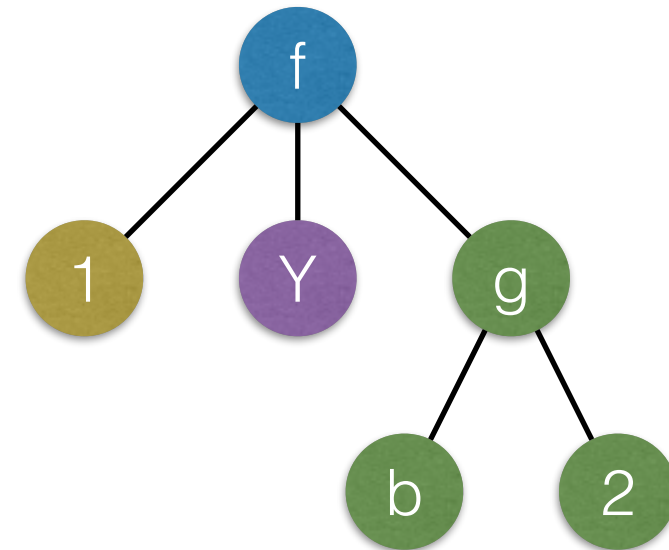
$$Z = g(b, 2)$$

Exemple

$$t_1 = f(X, g(a, Z), Z)$$



$$t_2 = f(1, Y, g(b, 2))$$



$$\begin{aligned} X &= 1 \\ Z &= g(b, 2) \\ Y &= g(a, g(b, 2)) \end{aligned}$$

Unificateur de t_1 et t_2 :

$$\begin{aligned} \sigma(X) &= 1 \\ \sigma(Z) &= g(b, 2) \\ \sigma(Y) &= g(a, g(b, 2)) \end{aligned}$$

Exercices

?- X = 42.

?- x = 42.

?- 42 = 6 * 7.

?- f(X) = X.

?- f(A,g(B,C)) = f(X,Y).

?- f(A,g(B,A)) = f(X,Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

?- 42 = 6 * 7.

?- f(X) = X.

?- f(A,g(B,C)) = f(X,Y).

?- f(A,g(B,A)) = f(X,Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

false.

?- 42 = 6 * 7.

?- f(X) = X.

?- f(A, g(B, C)) = f(X, Y).

?- f(A, g(B, A)) = f(X, Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

false.

?- 42 = 6 * 7.

false.

?- f(X) = X.

?- f(A,g(B,C)) = f(X,Y).

?- f(A,g(B,A)) = f(X,Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

false.

?- 42 = 6 * 7.

false.

?- f(X) = X.

X = f(X).

?- f(A,g(B,C)) = f(X,Y).

?- f(A,g(B,A)) = f(X,Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

false.

?- 42 = 6 * 7.

false.

?- f(X) = X.

X = f(X).

?- f(A, g(B, C)) = f(X, Y).

A = X. Y = g(B, C).

?- f(A, g(B, A)) = f(X, Y).

Exercices

?- X = 42.

X = 42.

?- x = 42.

false.

?- 42 = 6 * 7.

false.

?- f(X) = X.

X = f(X).

?- f(A, g(B, C)) = f(X, Y).

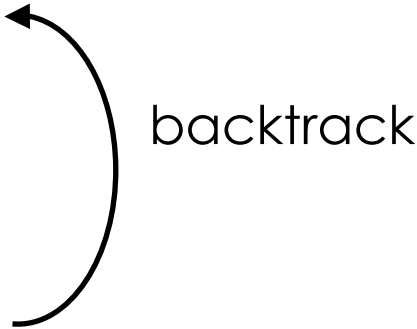
A = X. Y = g(B, C).

?- f(A, g(B, A)) = f(X, Y).

A = X. Y = g(B, X).

Résolution SLD

```
resoudre(buts, substitutions)
  si buts est vide alors
    retourner substitutions des buts initiaux
  sinon
    but, autresbuts  $\leftarrow$  buts
    listeclasses  $\leftarrow$  clauses dont la tête s'unifie avec but
    si listeclasses est vide alors echec fsi
    sinon pour chaque clause de listeclasses faire
      _,corps  $\leftarrow$  clause
       $\sigma \leftarrow$  unificateur(clause, but)
      resoudre( $\sigma$ (corps)  $\cup$   $\sigma$ (autresbuts),  $\sigma \cup$  substitutions)
    fpour
  fsi
```

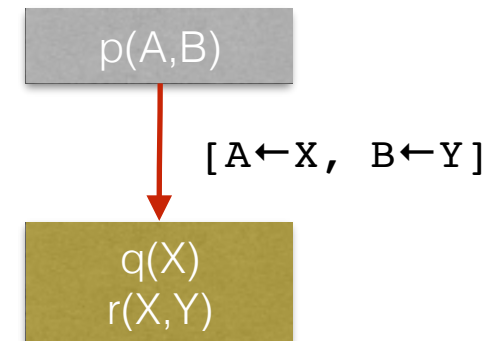


Exemple

```
but = [p(A,B)]  
substitutions = []  
  
but = p(A,B)  
autresbut = []  
listeclasses = [p(X,Y) :- q(X), r(X,Y).]  
  
clause = p(X,Y) :- q(X), r(X,Y).  
corps = [q(X), r(X,Y)]  
 $\sigma$  = [A $\leftarrow$ X, B $\leftarrow$ Y]  
resoudre([q(X), r(X,Y)], [A $\leftarrow$ X, B $\leftarrow$ Y])
```

miam.pl

```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
           r(X,Y).  
q(X) :- t(X).  
q(X) :- s(X).
```



Exemple

miam.pl

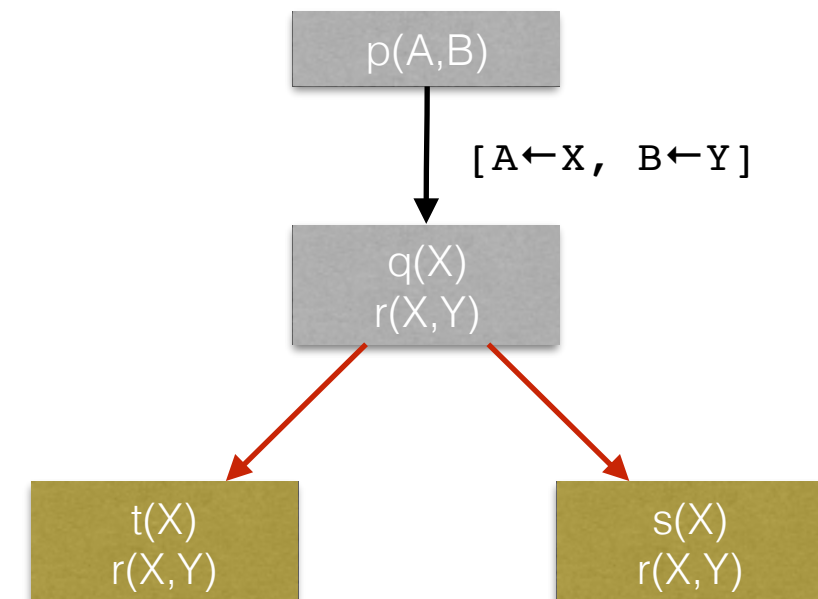
```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
           r(X,Y).  
q(X) :- t(X).  
q(X) :- s(X).
```

```
buts = [q(X),r(X,Y)]  
substitutions = [A←X, B←Y]
```

```
but = q(X)  
autresbuts = [r(X,Y)]  
listeclasses = [q(X):-t(X), q(X):-s(X)]
```

```
clause = q(X) :- t(X)  
corps = [t(X)]  
 $\sigma$  = [X←X]  
resoudre([t(X), r(X,Y)], [A←X, B←Y, X←X])
```

```
clause = q(X) :- s(X)  
corps = [s(X)]  
 $\sigma$  = [X←X]  
resoudre([s(X), r(X,Y)], [A←X, B←Y, X←X])
```



Exemple

miam.pl

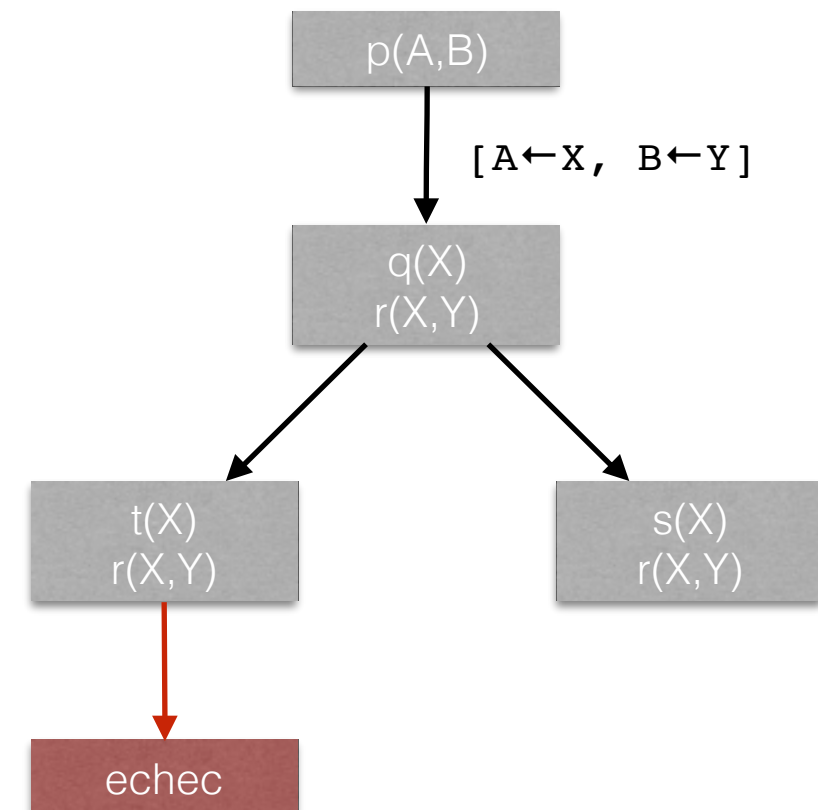
```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
           r(x,y).  
q(X) :- t(X).  
q(X) :- s(X).
```

```
but = [t(X),r(X,Y)]  
substitutions = [A←X, B←Y, X←X]
```

```
but = t(X)  
autresbut = [r(X,Y)]  
listeclasses = []
```

echec

⇒ **Backtrack**



Exemple

miam.pl

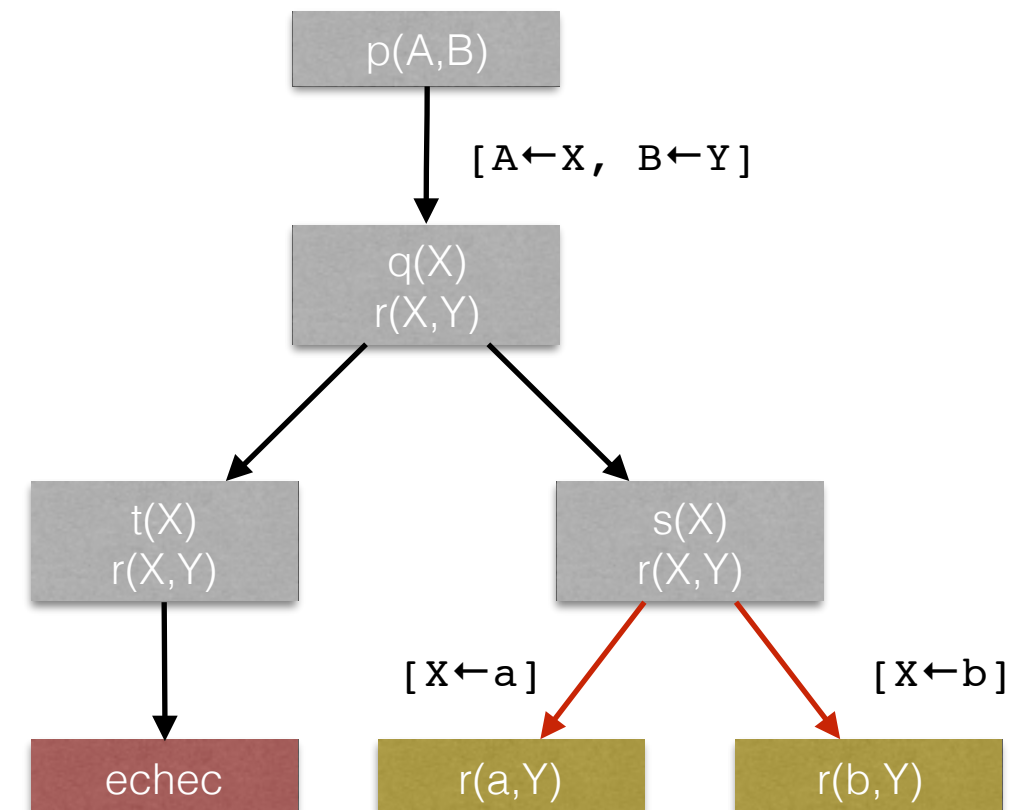
```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
          r(x,y).  
q(X) :- t(X).  
q(X) :- s(X).
```

```
buts = [s(X),r(X,Y)]  
substitutions = [A←X, B←Y, X←X]
```

```
but = s(X)  
autresbuts = [r(X,Y)]  
listeclasses = [s(a), s(b)]
```

```
clause = s(a).  
corps = []  
 $\sigma$  = [X←a]  
resoudre([r(a,Y)], [A←X, B←Y, X←X, X←a])
```

```
clause = s(b).  
corps = []  
 $\sigma$  = [X←b]  
resoudre([r(b,Y)], [A←X, B←Y, X←X, X←b])
```



Exemple

miam.pl

```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
           r(X,Y).  
q(X) :- t(X).  
q(X) :- s(X).
```

```
buts = [r(a,Y)]  
substitutions = [A←X, B←Y, X←X, X←a]
```

```
but = r(a,Y)  
autresbuts = []  
listeclasses = [r(a,b)]
```

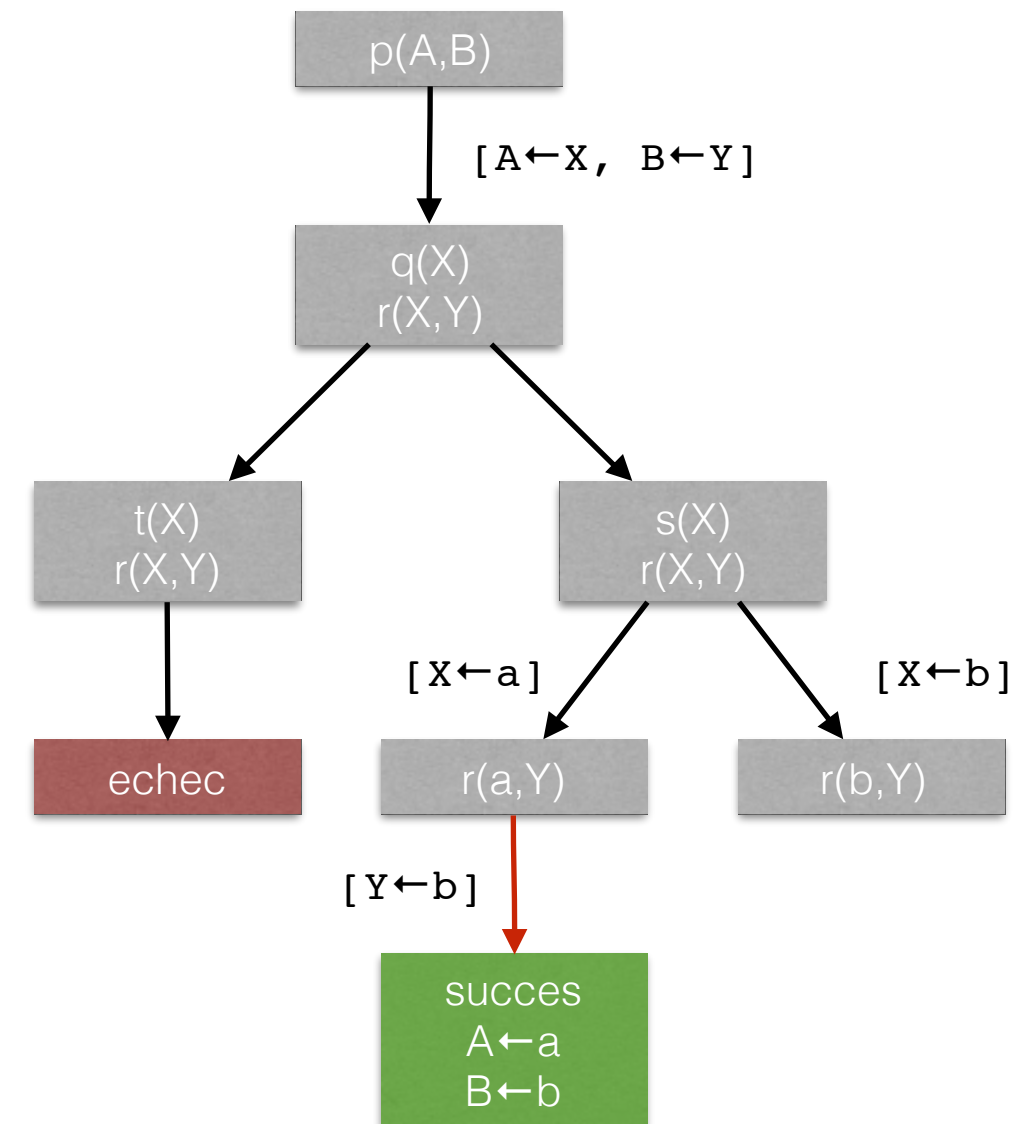
```
clause = r(a,b).  
corps = []  
 $\sigma$  = [Y←b]
```

```
resoudre([], [A←X, B←Y, X←X, X←a, Y←b])
```

succes

⇒ retourner A←a, B←b

⇒ backtrack



Exemple

miam.pl

```
s(a).  
s(b).  
r(a,b).  
r(b,c).  
p(X,Y) :- q(X),  
           r(X,Y).  
q(X) :- t(X).  
q(X) :- s(X).
```

```
buts = [r(b,Y)]  
substitutions = [A←X, B←Y, X←X, X←b]
```

```
but = r(b,Y)  
autresbuts = []  
listeclasses = [r(b,c)]
```

```
clause = r(b,c).  
corps = []  
 $\sigma$  = [Y←c]
```

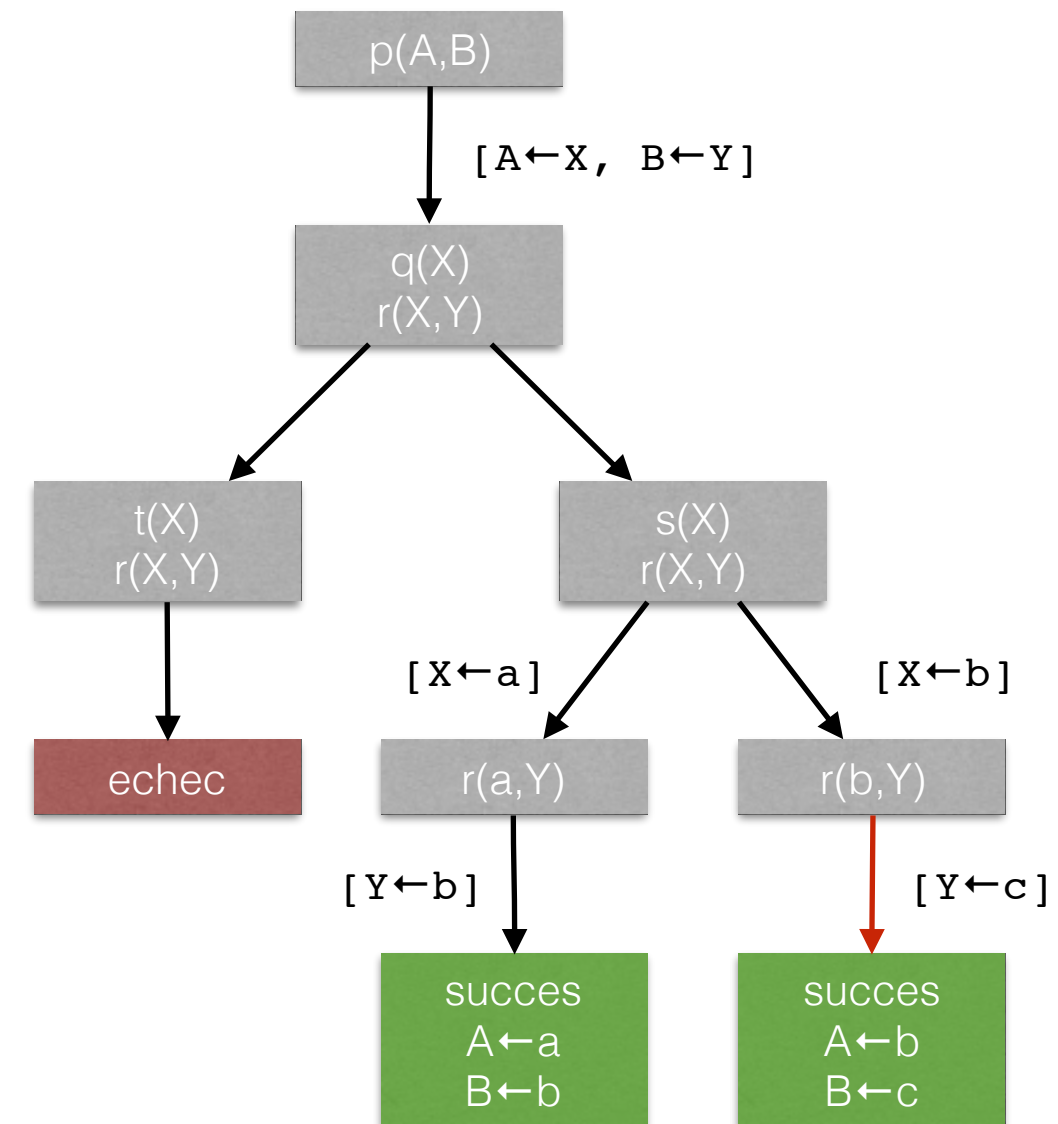
```
resoudre([], [A←X, B←Y, X←X, X←b, Y←c])
```

succes

⇒ retourner A←b, B←c

⇒ backtrack

⇒ fin



Cut

Éviter des backtracks

Green cuts : ne changent pas le résultat du programme

Red cuts : changent les résultats

Exemple

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

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

Solution qui fonctionne

MAIS si ça backtrack on risque de tester une règle inutilement.

Exemple

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

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

Le cut évite le backtrack.

Si la condition `X =< Y` est remplie, la 2^e règle ne sera jamais appelée.

Exemple

Plus besoin de la condition de la 2e règle ?

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

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

Que donne « `?- max(2,3,2).` » ?

Exemple

Plus besoin de la condition de la 2e règle ?

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

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

Que donne « ?- max(2,3,2). » ?

true. ... alors que ça devrait être false.

Exemple

Plus besoin de la condition de la 2e règle ?

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

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

Que donne « `?- max(2,3,2).` » ?

`true.` ... alors que ça devrait être `false.`

Problème : l'unification sur la 1^e règle.

Exemple

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

`max(X, Y, X).`

On enlève l'unification à gauche.

On la remet **après** le cut.

Tous les appels passent par la 1^e règle.

On n'unifie que si les conditions sont OK.

Structure avec cut

```
si cond1 alors
    action1
sinon si cond2 alors
    action2
sinon
    action3
fsi
```

```
f(X,Y,...) :-
    cond1, !,
    action1.
```

```
f(X,Y,...) :-
    cond2, !,
    action2.
```

```
f(X,Y,...) :-
    !,
    action3.
```

Exemple : factorielle

```
fact(0,1) :- !.
```

```
fact(N,_) :-  
    N < 0, !,  
    fail.
```

```
fact(N,X) :-  
    !,  
    M is N - 1,  
    fact(M, Y),  
    X is N * Y.
```

Case de base : $\text{fact}(0) = 1$

Cas où $N < 0$: on provoque le fail

Cas général

Le cut sert à ne pas mémoriser le contexte des backtracks qu'on ne fera pas.

Comparaisons, Unifications

=
\=

Unification
Inverse

is

Évaluation à droite puis unification

=:=
=\=

Évaluation des deux côtés puis unification
Inverse

==
\==

Équivalence des termes
Inverse

=@=
\=@=

Équivalence structurelle par rapport aux variables
Inverse

< >
=< >=

Plus petit que, plus grand que
Pareil, avec =

Examples

?- X = X.
true.

?- X = Y.
X = Y.

?- f(X) = f(Y).
X = Y.

?- f(X,X) = f(X,Y).
X = Y.

?- X == X.
true.

?- X == Y.
false.

?- f(X) == f(Y).
false.

?- f(X,X) == f(X,Y).
false.

?- X =@= X.
true.

?- X =@= Y.
true.

?- f(X) =@= f(Y).
true.

?- f(X,X) =@= f(X,Y).
false.

Arithmétique

`+` `-` `*`

Addition, soustraction, multiplication

`/`

Division

`//`

Division entière

`mod`

Modulo

`rem`

Reste de la division entière

`min,` `max`

Minimum / Maximum

`abs`

Valeur absolue

Listes

[]

Liste vide

[Tête|Queue]

Cons

$[E_1 | [E_2 | [E_3 | []]]] = [E_1, E_2, E_3]$

Raccourci de notation

member/2

Test si élément appartient à une liste

reverse/2

Renverse une liste

append/3

Concatène deux listes

merge/3

Fusionne deux listes triées

length/2

Longueur

sort/2

Tri

msort/2

Tri fusion

Vérification de types

<code>integer/1</code>	Entier
<code>float/1</code>	Flottant
<code>number/1</code>	Nombre (entier ou flottant)
<code>atom/1</code>	Identifieur
<code>atomic/1</code>	Identifieur ou nombre
<code>is_list/1</code>	Liste
<code>var/1</code>	Variable
<code>nonvar/1</code>	Terme instancié
<code>ground/1</code>	Terme sans variable libre
<code>compound/1</code>	Terme composé

Ordre des opérations

```
?- bon1(X).
```

```
X = a
```

```
false.
```

```
?- bon2(X).
```

```
false.
```

```
possible(a).  
possible(b).
```

```
vrai(a).  
vrai(b).
```

```
faux(b).
```

```
bon1(X) :- vrai(X), not(faux(X)).  
bon2(X) :- not(faux(X)), vrai(X).
```


Ordre des opérations

```
trace(possible).
trace(vrai).
trace(faux).
trace(bon1).
trace(bon2).
trace(not).

?- bon1(X).
  T Call: (6) bon1(_G1974)
  T Call: (7) vrai(_G1974)
  T Exit: (7) vrai(a)
  T Call: (7) not(faux(a))
  T Call: (8) faux(a)
  T Fail: (8) faux(a)
  T Exit: (7) not(user:faux(a))
  T Exit: (6) bon1(a)
X = a ;
  T Redo: (7) vrai(_G1974)
  T Exit: (7) vrai(b)
  T Call: (7) not(faux(b))
  T Call: (8) faux(b)
  T Exit: (8) faux(b)
  T Fail: (7) not(user:faux(b))
  T Fail: (6) bon1(_G1974)
false.
```

```
possible(a).
possible(b).

vrai(a).
vrai(b).

faux(b).

bon1(X) :- vrai(X), not(faux(X)).
bon2(X) :- not(faux(X)), vrai(X).
```

Ordre des opérations

```
trace(possible).  
trace(vrai).  
trace(faux).  
trace(bon1).  
trace(bon2).  
trace(not).
```

```
?- bon2(X).  
T Call: (6) bon2(_G1974)  
T Call: (7) not(faux(_G1974))  
T Call: (8) faux(_G1974)  
T Exit: (8) faux(b)  
T Fail: (7) not(user:faux(_G1974))  
T Fail: (6) bon2(_G1974)  
false.
```

```
possible(a).  
possible(b).
```

```
vrai(a).  
vrai(b).
```

```
faux(b).
```

```
bon1(X) :- vrai(X), not(faux(X)).  
bon2(X) :- not(faux(X)), vrai(X).
```

Ordre des opérations

```
trace(possible).
trace(vrai).
trace(faux).
trace(bon1).
trace(bon2).
trace(not).
```

```
?- bon2(X).
T Call: (6) bon2(_G1974)
T Call: (7) possible(_G1974)
T Exit: (7) possible(a)
T Call: (7) not(faux(a))
T Call: (8) faux(a)
T Fail: (8) faux(a)
T Exit: (7) not(user:faux(a))
T Call: (7) vrai(a)
T Exit: (7) vrai(a)
T Exit: (6) bon2(a)
```

```
X = a ;
T Redo: (7) possible(_G1974)
T Exit: (7) possible(b)
T Call: (7) not(faux(b))
T Call: (8) faux(b)
T Exit: (8) faux(b)
T Fail: (7) not(user:faux(b))
T Redo: (7) possible(_G1974)
T Exit: (7) possible(c)
T Call: (7) not(faux(c))
T Call: (8) faux(c)
T Exit: (8) faux(c)
T Fail: (7) not(user:faux(c))
T Fail: (6) bon2(_G1974)
```

ò

```
possible(a).
possible(b).
```

```
vrai(a).
vrai(b).
```

```
faux(b).
```

```
bon1(X) :- vrai(X), not(faux(X)).
bon2(X) :- possible(X),
           not(faux(X)), vrai(X).
```