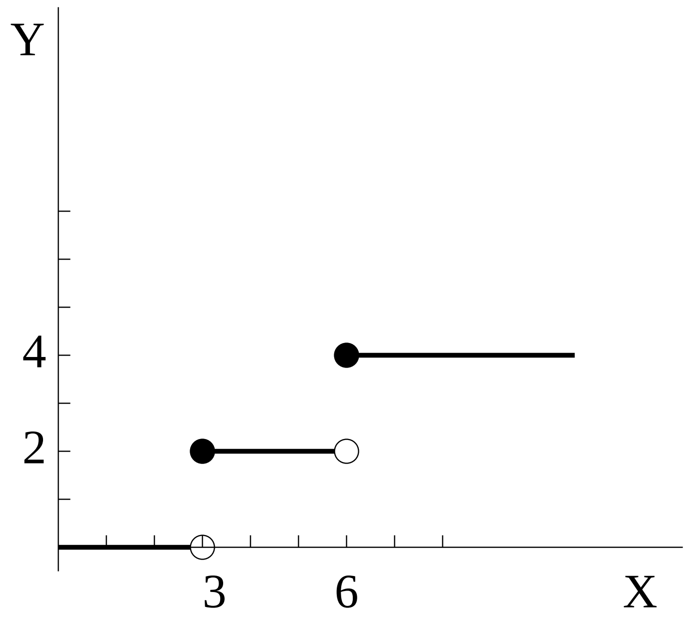
# Empêcher le retour en arrière

- Le retour en arrière est “câblé” en Prolog
- Il peut causer de l’inefficacité
- On veut le contrôler

# Exemple




La relation entre X et Y peut être définie par trois règles:

- si $X < 3$ alors $Y = 0$
- si $X \geq 3$ et $X < 6$ alors $Y = 2$
- si $X \geq 6$ alors $Y = 4$

En Prolog on définit une relation `f( X, Y)`:

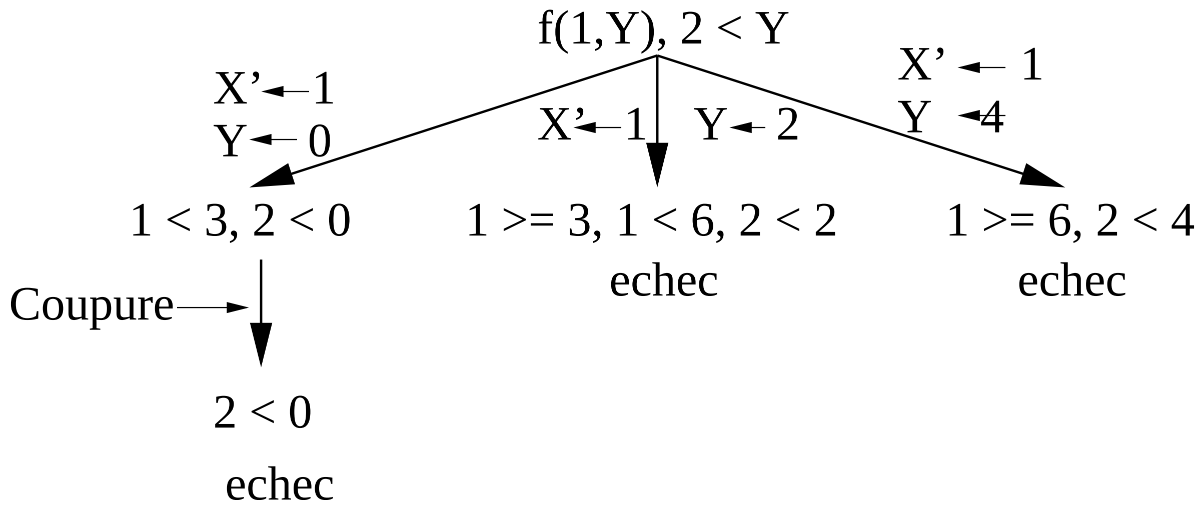
```
f( X, 0) :-  X < 3.           % règle 1
f( X, 2) :-  X >= 3, X < 6.   % règle 2
f( X, 4) :-  X >= 6.          % règle 3
```

Avant qu'on exécute `f( X, Y)`, X doit être instanciée.

Exemple :

```
?- f( 1, Y), 2 < Y.
```

- Pendant l'exécution du premier but `f( 1, Y)`, Y est instanciée par 0.
- Le deuxième but devient `2 < 0`
- C'est un échec.
- Avant de répondre `no`, Prolog fait des retours en arrière successifs.




- Les trois règles pour f sont mutuellement exclusives.

- On ajoute explicitement au programme une coupure pour ne pas faire le retour en arrière à ce point.

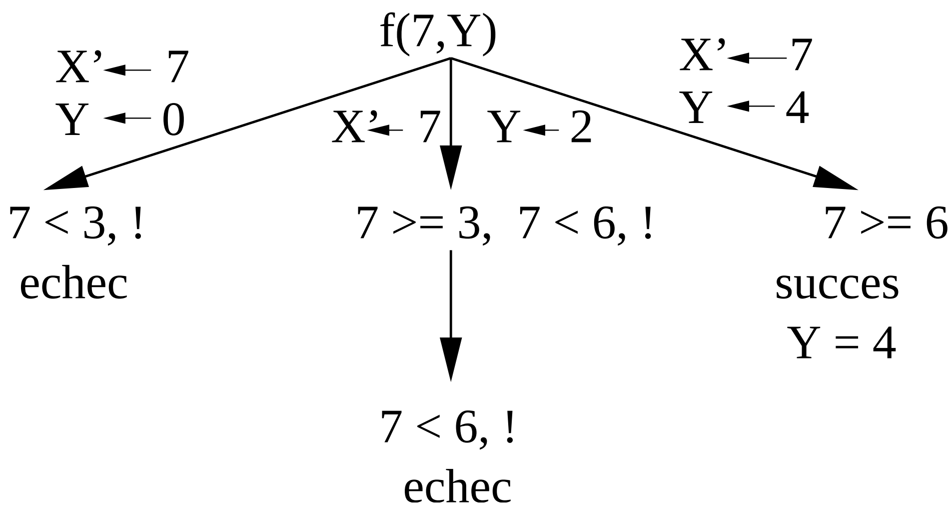
# Deuxième version du programme

```
f( X, 0) :-  X < 3, !.
f( X, 2) :-  X >= 3, X < 6, !.
f( X, 4) :-  X >= 6.
```

- Prolog répond à la question `?- f( 1, Y), 2 < Y.` plus rapidement.
- Ici, la sémantique opérationnelle du programme a changée, mais pas les résultats.
- La coupure peut aussi changer les résultats.

# Deuxième question

```
?- f( 7, Y).
Y = 4
```




- C'est inefficace.
- On teste `X < 3` et après `X >= 3`.
- Même chose pour `X < 6` et `X >= 6`.

- On peut reformuler les règles:
  - Si $X < 3$ alors $Y = 0$,
  - sinon si $X < 6$ alors $Y = 2$,
  - sinon $Y = 4$.

- En Prolog:

```
f( X, 0) :-  X < 3, !.
f( X, 2) :-  X < 6, !.
f( X, 4).
```

- Ce programme produit les mêmes résultats que l'autre mais plus efficacement.

# Exemple suite

- Si on enlève les coupures:

```
f( X, 0) :-  X < 3.
f( X, 2) :-  X < 6.
f( X, 4).
```

  on change les résultats !

- Par exemple,

```
?- f( 1, Y).
Y = 0;
Y = 2;
Y = 4;
no
```

# La coupure - Définition

- Nous appelons **but parent** le but qui s'unifie avec la tête de la clause contenant la coupure.
- Si la coupure est atteinte, elle "réussit".
- Tous les choix entre la considération du but parent et la rencontre de la coupure sont définitifs.
- Tous les autres alternatives ne seront plus considérées.

# Exemple

```
p(X) :- q(X), !, r(X).
p(X) :- u(X).
p(c).
q(X) :- s(X).
q(d).
```

```
r(a).
r(b).
r(d).
s(a).
s(b).
```

```
?- p(X).
X = a ;
No
?- r(X),!,q(Y).
X = a   Y = a;
X = a   Y = b;
X = a   Y = d;
No
```

# Exemples

```
% Maximum sans coupure
maximum( X, Y, X) :- X >= Y.
maximum( X, Y ,Y) :- X < Y.

% Maximum avec coupure
maximum2( X, Y, X) :- X >= Y, !.
maximum2( X, Y ,Y).

% Membre à une solution
membreune( X, [ X | _]) :- !.
membreune( X, [ _ | L]) :- membreune(X, L).

% Ajouter un élément sans duplication
% ajouter(+X,+L,-L1)
ajouter( X, L, L) :- membre( X, L), !.
ajouter( X, L, [X | L]).
```

# La négation

- Prolog permet la négation avec le symbole \+ (anciennement **not**).
- Exprime le fait qu'on ne peut pas prouver quelque chose.
- Si un but B réussit alors `\+ B` échoue.
- Si un but B échoue alors `\+ B` réussi.
- Ce n'est pas la négation logique.
- On peut définir `not(B)` par

```
not(B) :- B, !, fail.
not(B).
```

- Attention: Il vaut mieux que les variables soient instanciées au moment d'appeler `\+`

- Par exemple: `?- \+ member(X, [a, b]).` donne no.

# La négation (suite)

Dans une règle il faut aussi faire attention avec l'instanciation.

```
marie(francois).
etudiant(rene).

etudiant_celibataire(X) :-
        \+ marié(X),
        etudiant(X).

?- etudiant_celibataire(X).
No.

?- etudiant_celibataire(rene).
Yes.
```

```
Version correcte:

etudiant_celibataire(X) :-
    etudiant(X),
    \+ marie(X).

?- etudiant_celibataire(X).
X = rene ;
No
```

# Prédicats prédéfinis

- Prédicats simples
- Entrée/Sorties
- Prédicats de base de données
- Prédicats de deuxième ordre

# Prédicats de test

- `integer/1` teste si l'argument est un entier.

```
?- integer(3).
Yes.
?- integer(X).
No.
```

- `number/1` teste si l'argument est un nombre.
- `float/1` teste si l'argument est un flottant.
- `atomic/1` teste si l'argument est une constante.
- `atom/1` teste si l'argument est un atome.

- `var/1` teste si l'argument est une variable non instanciée.
- `nonvar/1` teste si l'argument n'est pas une variable non instanciée.
- `compound/1` teste si l'argument est un terme composé.
- `ground/1` teste si l'argument est un terme sans variables.

# Prédicats de manipulation de termes

- `functor(Terme, Foncteur, Arité).`
  Mode d'utilisation (+, -, -) ou (-, +, +) seulement !

```
?- functor(père(jean, isa), F, A).
F = père, A = 2.
?- functor(T, père, 2).
T = père(X, Y).
```

- `arg(N, Terme, X)` unifie X et argument no N de Terme.

```
?- arg(1, père(jean, isa), X).
X = jean.
```

- `Terme =.. Liste` transforme un terme en une liste :

```
?- père(jean, isa) =.. L.
L = [père, jean, isa].
?- T =.. [a, b, c].
T = a(b, c).
```

- `name/2` transforme un atome en une liste de ses codes ASCII.

```
?- name(toto, L).
L = [116, 111, 116, 111].
?- name(A, [116, 111, 116, 111]).
A = toto.
```

# Exemple d'utilisation

```
% Addition

plus(X, Y, Z) :-
    nonvar(X),
    nonvar(Y),
    Z is X + Y.

plus(X, Y, Z) :-
    nonvar(Y),
    nonvar(Z),
    X is Z - Y.

plus(X, Y, Z) :-
    nonvar(X),
    nonvar(Z),
    Y is Z - X.
```

# Exemple d'utilisation

```
% Unification

% Unifier deux Variables
unify(X, Y) :-
     var(X),var(Y),X = Y.

% Premier argument variable
unify(X, Y) :-
     var(X),nonvar(Y),X = Y.

% Deuxieme argument variable
unify(X, Y) :-
     nonvar(X),var(Y),Y = X.

% Arguments sont constantes
unify(X, Y) :-
```

```
     nonvar(X), nonvar(Y),
     atomic(X), atomic(Y),
     X = Y.

% Arguments sont composes.
unify(X, Y) :-
     nonvar(X), nonvar(Y),
     compound(X), compound(Y),
     termUnify(X, Y).


% Unifier deux termes composes
termUnify(X, Y) :-
     functor(X, F, N),
     functor(Y, F, N), % meme symb. fonctionnelle
     argUnify(N, X, Y).
```

```
% Unifier les arguments
argUnify(N, X, Y) :-
     N>0,
     argUnify1(N, X, Y),
     Ns is N - 1,
     argUnify(Ns, X, Y).

argUnify(0, X, Y).

% Unifier les arguments N
argUnify1(N, X, Y) :-
     arg(N, X, ArgX),
     arg(N, Y, ArgY),
     unify(ArgX, ArgY).
```

# Prédicat Entrées/Sorties

- Prolog communique avec des flux.
- Il y a un flux d'entrée (clavier) et de sortie standard (écran).
- Il peut être changé.

Les prédicats de base:

- `see(fichier)` change le flux d'entrée vers fichier.
- `see(user)` met le flux d'entrée standard.
- `tell(fichier)` change le flux de sortie vers fichier.
- `tell(user)` met le flux de sortie standard.

- `seen` ferme le fichier d'entrée courant.
- `told` ferme le fichier de sortie courant.
- `read(X)` lit le prochain terme (délimité par `.`) du flux d'entrée et unifie avec X.
- `write(Terme)` écrit le terme sur le flux de sortie.

# Exemple 1

```
cube :-
    write('Nouveau chiffre, s.v.p. (ou stop): '),
    read(X),
    process(X).

process(stop) :- !.

process(N) :-
    C is N * N * N,
    write('Le cube de '), write(N),
    write(' est: '), write(C), nl,
    cube.
```

# Exemple 2

```
imprimeliste([]).

imprimeliste([X | L]) :-
    write(X), nl,
    imprimeliste(L).
```

# Exemple 3

```
% etoiles( [3,4,2]) imprime sur l'écran:
% ***
% ****
% **

etoiles([]).

etoiles([N | L]) :-
   etoileslignes(N), nl, etoiles(L).

etoileslignes(N) :-
   N > 0, write(*), N1 is N - 1, etoileslignes(N1).

etoileslignes(N) :- N =< 0.
```

# Prédicats de base de données

- Un programme Prolog peut être vu comme une base de données
- Il y a des prédicats pour changer cette base pendant l'exécution du programme.
- `assert/1` et `assertz/1` ajoutent à la fin de la base de données
- `asserta/1` ajoute au début de la base de données
- `retract/1` enlève de la base de données

# Exemple 1

```
:- dynamic soleil, brouillard, pluie .
jolie :-
   soleil, \+ pluie.
bizarre :-
   soleil, pluie.
horrible :-
   pluie, brouillard.
pluie.
brouillard.

?- jolie.
No
?- horrible.
Yes
?- retract(brouillard).
Yes
```

```
?- horrible.
No
?- assert(soleil).
Yes
?- bizarre.
Yes
?- retract(pluie).
Yes
?- jolie.
Yes
```

# Exemple 2

```
:- dynamic rapide/1, lent/1.

rapide(anne).
lent(tom).
lent(marc).

?- assert((plusrapide(X,Y) :- rapide(X), lent(Y))).
X = _G456
Y = _G457
Yes
?- plusrapide( A, B).
A = anne
B = tom ;
A = anne
B = marc ;
No
```

```
?- retract( lent(X)).
X = tom ;
X = marc ;
No
?- plusrapide( anne,_).
No
```

# Exemple 3

```
maketable :-
   L = [0,1,2,3,4,5,6,7,8,9],
   member(X,L),
   member(Y,L),
   Z is X*Y,
   assert(product(X,Y,Z)),
   fail.

?- maketable.
No
3 ?- product(3,5,X).
X = 15 ;
No
```

# Prédicats de deuxième ordre

- `bagof(X, P, L)` produit la liste L de tous les objets X tel que le but P est satisfait (en tenant compte des variables dans P).
- `setof(X, P, L)` est pareil que `bagof` sauf que la liste L est ordonnée et les éléments en double sont éliminés.
- `findall(X, P, L)` comme `bagof` sans distinguer les autres variables P.

```
ex(a,1).          ex(b,1).          ex(c,1).
ex(a,2).          ex(c,2).          ex(b,2).
ex(a,3).          ex(b,3).          ex(c,3).

?- bagof(X,ex(X,A),L).

X = _G151
A = 3
```

```
L = [a, b, c] ;

X = _G151
A = 2
L = [a, c, b] ;

X = _G151
A = 1
L = [a, b, c] ;

No

?- setof(X,ex(X,A),L).

X = _G151
A = 3
L = [a, b, c] ;
```

```
X = _G151
A = 2
L = [a, b, c] ;

X = _G151
A = 1
L = [a, b, c] ;

No

?- bagof(X,A^ex(X,A),L).

X = _G151
A = _G152
L = [a, a, a, b, c, c, b, b, c] ;

No
?- setof(X,A^ex(X,A),L).
```

```
X = _G151
A = _G152
L = [a, b, c] ;

No

4 ?- setof(X/A,ex(X,A),L).
X = _G151
A = _G152
L = [a/1, a/2, a/3, b/1, b/2, b/3, c/1, c/2, c/3] ;

No
5 ?- findall(X,ex(X,A),L).
X = _G378
A = _G379
L = [a, b, c, a, c, b, a, b, c] ;
No
```

# Technique d'accumulateur

Exemple: Calculer la somme des éléments d'une liste

```
sommeliste( [] , 0).
sommeliste( [Tete | Queue], S) :-
          sommeliste( Queue, S1),
          S is Tete + S1.
```

Meilleure solution:

```
sommeliste1( Liste, Somme) :-
          sommeliste2( Liste, 0, Somme).

% le deuxième argument de sommeliste2
% contient la somme partielle

sommeliste2( [], Somme, Somme).
% Somme partielle = Somme totale

sommeliste2( [Tete | Queue], SPart, STotale) :-
           SPart2 is SPart + Tete,
           sommeliste2( Queue, SPart2, STotale).
```