

# Ecole Nationale d'Ingénieurs de Brest



## Le langage Prolog

MCours.com

*J. Tisseau*

– 1991/2006 –

## Table des matières

■ Programmation en logique .....	3
■ Les termes Prolog .....	15
■ La machine Prolog .....	53
■ Contrôle de la résolution .....	71
■ Prédicats prédéfinis .....	86
■ Bibliographie .....	96

## Programmation en logique

### Origines :

- Travaux des logiciens (... , *J. Herbrand, J. Robinson, ...*)
- Université de Marseille (*A. Colmerauer, Ph. Roussel, ...*)
- Université d'Edinburgh (*R. Kowalski, D. Warren, ...*)

**Idée :** Utiliser la logique comme langage de programmation

### Evolutions :

- Prolog et parallélisme
- Prolog et contraintes

## Le formalisme logique

**Aspects syntaxiques :** *ce qui peut être dit*

Vocabulaire + Syntaxe

**Aspects sémantiques :** *ce que l'on peut dire d'une expression logique*

Valeur de Vérité

Des règles de déduction décrivent les transformations que l'on peut faire subir aux expressions à partir d'expressions définies comme vraies (axiomes).

## La logique des prédicats

### Vocabulaire ..... *les mots du langage*

inconnues	$x, y, z$
constantes fonctionnelles	$f, g, h$
constantes prédictives	$P, Q, R$
connecteurs	$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
quantificateurs	$\forall, \exists$
ponctuation	$() ,$
terme ( $t_i$ )	inconnue ou forme fonctionnelle
forme fonctionnelle	$f(t_1, \dots, t_m) \ m \geq 0$

## La logique des prédicats

### Syntaxe ..... *les phrases du langage*

forme prédictive	$P(t_1, \dots, t_n) \ n \geq 0$
formule ( $A, B, \dots$ )	forme prédictive $(A), \neg A, A \wedge B, A \vee B,$ $A \Rightarrow B, A \Leftrightarrow B$ $\forall x A, \exists x A$

## La logique des prédicats

**Sémantique** ..... *la signification des phrases*

$\mathcal{D}$  domaine de définition des termes

$A$  formule logique

$$A : \mathcal{D} \longrightarrow \{\text{vrai}, \text{faux}\}$$

## Formes clauseales

Une forme clauseale est une formule logique ne contenant que des disjonctions de littéraux négatifs ou positifs.

1. élimination des  $\Rightarrow$
2. déplacement des  $\neg$  vers l'intérieur
3. élimination des  $\exists$  (“skolémisation”)
4. déplacement des  $\forall$  vers l'extérieur
5. distribution de  $\wedge$  sur  $\vee$
6. mise en formes clauseales

## Formes clauseales

$$\forall x((\forall y(p(y) \Rightarrow r(y, x)) \Rightarrow q(x))$$

1.  $\forall x(\neg(\forall y(\neg p(y) \vee r(y, x)) \vee q(x))$
2.  $\forall x((\exists y(p(y) \wedge \neg r(y, x)) \vee q(x))$
3.  $\forall x((p(f(x)) \wedge \neg r(f(x), x)) \vee q(x))$
4.  $(p(f(x)) \wedge \neg r(f(x), x)) \vee q(x)$
5.  $(p(f(x)) \vee q(x)) \wedge (\neg r(f(x), x) \vee q(x))$
6. 2 clauses :

- $p(f(x)) \vee q(x)$
- $\neg r(f(x), x) \vee q(x)$

## Clauses de Horn

Une clause de Horn est une clause possédant au plus un littéral positif.

**Faits** : pas de littéral négatif

$$p(x, y, f(x, z))$$

**Règles** : un littéral positif et au moins un littéral négatif

$$p(x, y, f(x, z)) \vee \neg q(x, y) \vee \neg r(y, z)$$

**Questions** : pas de littéral positif

$$\neg q(x, y) \vee \neg r(y, z)$$

## De la logique à Prolog

Prolog est un langage de programmation déclarative qui repose sur la logique des prédicats restreinte aux clauses de Horn.

Prolog  $\equiv$  **P**rogrammation en **l**ogique

---

**Formule logique** :  $\forall x((\exists z, r(z, x) \wedge v(z)) \Rightarrow v(x))$

**Forme clausale** :  $v(x) \vee \neg r(z, x) \vee \neg v(z)$

**Clause Prolog** :  $v(X) \text{ :- } r(Z, X), v(Z).$

---

Un programme Prolog est un ensemble de clauses.

## Mon premier interpréteur Prolog

```
:- reconsult('opérateurs.pl').
```

```
demontrer(UneConclusion et UneAutreConclusion) si
    demontrer(UneConclusion) et_si
    demontrer(UneAutreConclusion).
```

```
demontrer(UneConclusion) si
    (DesConditions => UneConclusion) et_si
    demontrer(DesConditions).
```

```
demontrer(toujours_vrai).
```

## Mon premier programme

```
% Un exemple de règles généalogiques

% Mes parents sont mes ancêtres
(X parent_de Y) => (X ancetre_de Y).

% Les parents de mes ancêtres sont mes ancêtres
((X parent_de Y) et (Y ancetre_de Z)) => (X ancetre_de Z).

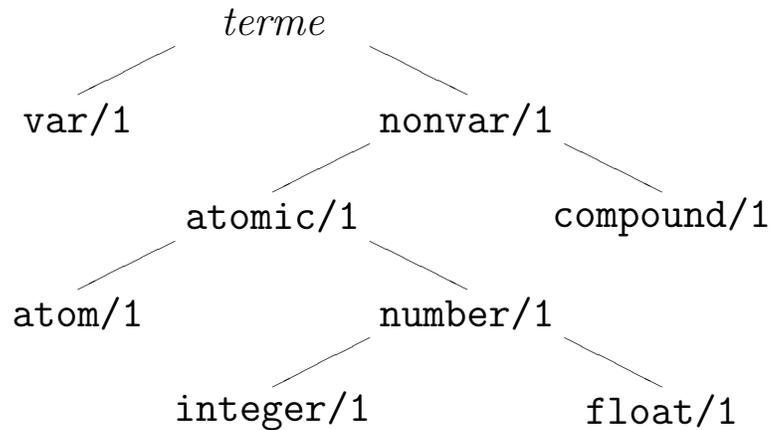
% Cas particuliers
toujours_vrai => (emile parent_de jean).
toujours_vrai => (jean parent_de fabien).
```

## Mes premières démonstrations

```
% Quels sont les ancêtres de Fabien ?
?- demontrer(Qui ancetre_de fabien).
Qui = jean;
Qui = emile;
no more solution

% De qui Emile est-il l'ancêtre ?
?- demontrer(emile ancetre_de Qui).
Qui = jean;
Qui = fabien;
no more solution
```

## Les termes Prolog



## Les variables

Une variable (ou inconnue) peut remplacer n'importe quel terme Prolog.

**variable instanciée** à un terme : la variable a pour valeur ce terme.

**variable libre** : la variable n'est instanciée à aucun terme.

**variables liées** : des variables libres peuvent être liées entre-elles : dès que l'une d'entre elles sera instanciée à un terme, les autres variables qui lui sont liées le seront aussi.

## Exemples de variables

<b>variable</b>
[_A-Z] [_a-zA-Z0-9]*
X
Nom_compose
_variable
_192
_

## Les termes atomiques

**Atomes** : un atome (constante symbolique) permet de représenter un objet quelconque par un symbole.

identificateur	opérateur	atome 'quoté'
[a-z] [_a-zA-Z0-9]*	[+*/^<>= ~: .?#&@]+	'(([^']) (''))*'
atome	==	'ATOME'
bonjour	:?:?:	'ca va ?!'
c_est_ca	--->	'c''est ca'

**Nombres** : entiers ou réels

## Les termes composés

- Un terme composé permet de décrire des données structurées.
- Un terme composé (exemple : `date(25,mai,1988)`) est constitué
  - d'un atome (`date`)
  - d'une suite de termes (arguments : `(25,mai,1988)` ); le nombre d'arguments (`3`) est appelé arité
- Le couple atome/arité (`date/3`) est appelé foncteur du terme composé correspondant.

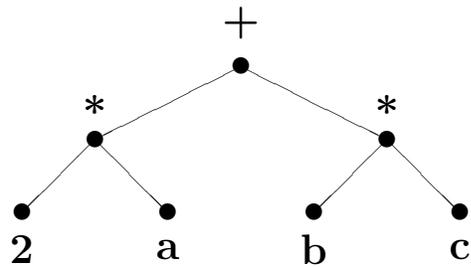
## Exemples de termes composés

Foncteur	Terme composé
<code>date/3</code>	<code>date(25,mai,1988)</code>
<code>'etat-civil'/3</code>	<code>'etat-civil'('ac''h',luc,date(1,mars,1965))</code>
<code>c/2</code>	<code>c(3.4,12.7)</code>
<code>c/4</code>	<code>c(a,B,c(1.0,3.5),5.2)</code>
<code>parallele/2</code>	<code>parallele(serie(r1,r2),parallele(r3,c1))</code>
<code>list/2</code>	<code>list(a,list(b,list(c,'empty list')))</code>

**Terme composé  $\equiv$  Structure de données**

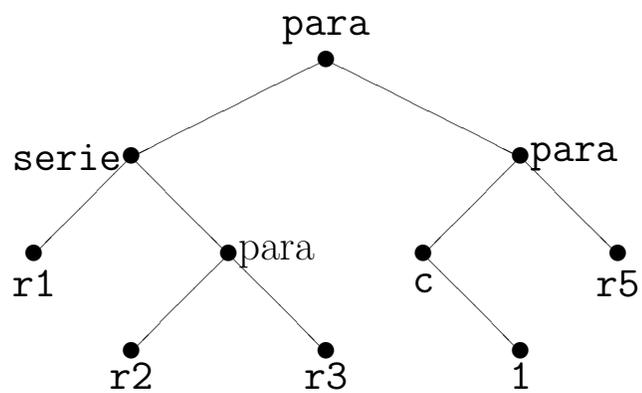
### Termes $\equiv$ arbres

$2 * a + b * c$



### Termes $\equiv$ arbres

`para(serie(r1,para(r2,r3)),para(c(1),r5))`



## Une fiche d'état civil

individu

état-civil	
Nom	: NGAOUNDERE
Prénom	: Richard
Nationalité	: camerounaise
Sexe	: masculin
Date de naissance :	
Jour	: 15
Mois	: Février
Année	: 1960

adresse	
Rue	: 4 rue Leclerc
Ville	: Brest
Code postal	: 29200

# MCours.com

## Une fiche d'état civil en Prolog

```
% Structures de données
% etat_civil(Nom,Prenom,Nationalite,Sexe,Date)
% date(Jour,Mois,Annee)
% adresse(Rue,Ville,Code_postal)

% Base de données
% individu(Etat_civil,Adresse)
individu(
    etat_civil('Ngaoundere','Richard',camerounaise,masculin,
              date(15,'Fevrier',1960)),
    adresse('4 rue Leclerc','Brest',29200)
).
```

## Les entiers naturels

Le vocabulaire initial de la théorie des entiers naturels comprend:

- une constante  $z$  qui représente l'entier 0 (zéro)
- une fonction unaire  $s(X)$  qui traduit la notion de successeur d'un entier  $X$
- un prédicat unaire  $entier(X)$  ( $X$  est un entier)

$$z \in \mathcal{N} \text{ et } \forall x \in \mathcal{N}, s(x) \in \mathcal{N}$$

## Les entiers naturels en Prolog

**Définition/Génération :**

$$z \in \mathcal{N} \text{ et } \forall x \in \mathcal{N}, s(x) \in \mathcal{N}$$

```
entier(z).
entier(s(X)) :- entier(X).
```

**Addition/Soustraction :**

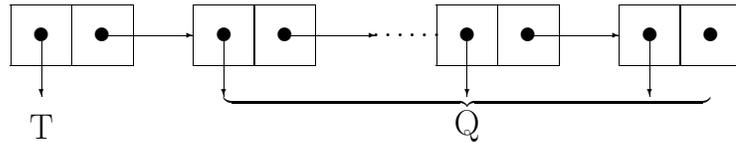
$$\forall x \in \mathcal{N}, x + z = x$$

$$\forall x, y \in \mathcal{N}, x + s(y) = s(x + y)$$

```
plus(X,z,X).
plus(X,s(Y),s(Z)) :- plus(X,Y,Z).
```

## Ma première liste

Une liste est une suite ordonnée de termes.



```
liste(0).           % 0 ≡ liste vide
liste(l(T,Q)) :- liste(Q).
```

## Appartenance à une liste

```
dans(T,l(T,_)).
dans(T,l(_,Q)) :- dans(T,Q).
```

```
?- dans(X,l(a,l(b,l(c,0)))).
X = a;
X = b;
X = c;
no more solution
```

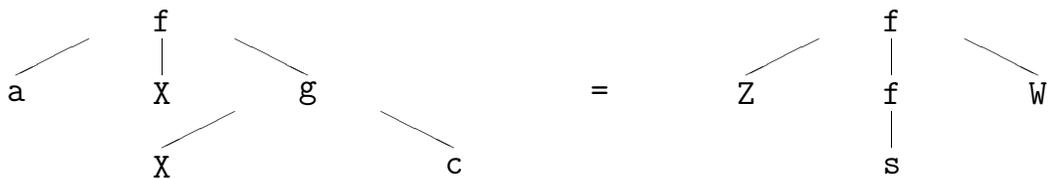
## Classification des termes

atom(T)	T est un atome
atomic(T)	T est un terme atomique
compound(T)	T est un terme composé
float(T)	T est un réel
integer(T)	T est un entier
nonvar(T)	T n'est pas une variable libre
number(T)	T est un nombre
var(T)	T est une variable libre

## Unification de termes

mise en correspondance d'arbres syntaxiques

Terme1 = Terme2



## Comparaison de termes

ordre lexicographique

Terme1 == Terme2

Terme1 \== Terme2

Terme1 @< Terme2

Terme1 @=< Terme2

Terme1 @>= Terme2

Terme1 @> Terme2

## Inspection de termes

accès à la structure interne des termes

`arg(N, T, X)` X est le N<sup>ième</sup> argument du terme T

`functor(T, A, N)` T est le terme de foncteur A/N

## Termes de base

Un terme est dit de base si

1. c'est un terme atomique
2. c'est un terme composé dont tous les arguments sont des termes de base

```
base(T) :- atomic(T).
base(T) :- compound(T), functor(T,F,N), base(N,T).

base(N,T) :- N > 0, arg(N,T,Arg), base(Arg),
             N1 is N-1, base(N1,T).
base(0,T).
```

## Les opérateurs en Prolog

Un opérateur permet une représentation syntaxique simplifiée d'un terme composé, unaire ou binaire.

**en notation préfixée**  $\sim X \equiv \sim(X)$

**en notation postfixée**  $X : \equiv :(X)$

**en notation infixée**  $X + Y \equiv +(X,Y)$

Aucune opération n'est *a priori* associée à un opérateur.

## Priorité des opérateurs

- Chaque opérateur a une priorité comprise entre 1 et 1200.
- La priorité détermine, dans une expression utilisant plusieurs opérateurs, l'ordre de construction du terme composé correspondant.

$$a + b * c \equiv a + (b * c) \equiv +(a, *(b,c))$$

- La priorité d'une expression est celle de son foncteur principal.
- Les parenthèses donnent aux expressions qu'elles englobent une priorité égale à 0.
- La priorité d'un terme atomique est de 0.

## Associativité des opérateurs

position	associativité	notation	exemple
infixée	à droite	xfy	a , b , c
	à gauche	yfx	a + b + c
	non	xfx	x = y
préfixée	oui	fy	not not x
	non	fx	- 4
postfixée	oui	yf	
	non	xf	

## Quelques opérateurs prédéfinis

priorité	opérateurs	associativité
1200	<code>:- --&gt;</code>	<code>xfx</code>
1200	<code>:- ?-</code>	<code>fx</code>
1100	<code>','</code>	<code>xfy</code>
1000	<code>','</code>	<code>xfy</code>
900	<code>not</code>	<code>fy</code>
700	<code>= =.. == \== is =:= =\= &lt; &gt;</code>	<code>xfx</code>
700	<code>&lt; =&lt; &gt;= &gt; @&lt; @=&lt; @&gt;= @&gt;</code>	<code>xfx</code>
500	<code>+ -</code>	<code>yfx</code>
400	<code>* /</code>	<code>yfx</code>

## Définir ses propres opérateurs

Adapter la syntaxe Prolog aux besoins de l'utilisateur.

- `op(P,A,Op)` définit un nouvel opérateur de nom `Op`, de priorité `P` et d'associativité `A`.
- `A` est l'un des atomes `xfx`, `xfy`, `yfx`, `fx`, `fy`, `xf`, `yf`.

```
a et b et c et d.           % erreur
:- op(200, xfy, et).
a et b et c et d.           % correct
```

## Evaluateur arithmétique

X is Expression

X est le résultat de l'évaluation de l'expression arithmétique Expression.

```
?- 8 = 5+3.           % unification
no
?- 8 is 5+3          % évaluation
yes
?- X is 5+3
X = 8;
no more solution
```

## Comparaisons arithmétiques

$e_1 < e_2$	Expr1 < Expr2
$e_1 \leq e_2$	Expr1 =< Expr2
$e_1 > e_2$	Expr1 > Expr2
$e_1 \geq e_2$	Expr1 >= Expr2
$e_1 = e_2$	Expr1 ::= Expr2
$e_1 \neq e_2$	Expr1 =\= Expr2

## Exemples d'évaluations arithmétiques

### Fonction d'Ackermann

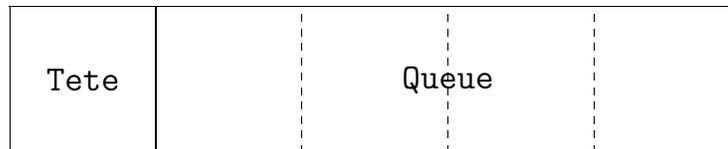
$$f(0, n) = n + 1$$

$$f(m, 0) = f(m - 1, 1) \text{ si } m > 0$$

$$f(m, n) = f(m - 1, f(m, n - 1)) \text{ si } m > 0, n > 0$$

```
f(0,N,A) :- A is N+1.  
f(M,0,A) :- M > 0, M1 is M-1, f(M1,1,A).  
f(M,N,A) :- M > 0, N > 0, M1 is M-1, N1 is N-1,  
            f(M,N1,A1), f(M1,A1,A).
```

## Les listes Prolog

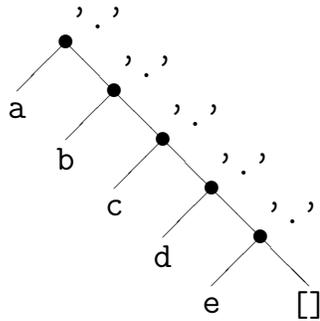


liste vide : []

liste non vide : ' .' (Tete, Queue)

## Un exemple de liste en notation standard

a	b	c	d	e
---	---	---	---	---



`'.'(a,'.(b,'.(c,'.(d,'.(e,[]))))`

## Une autre syntaxe pour les listes Prolog

$[T_1, T_2, \dots, T_n | Reste]$

- $T_1, T_2, \dots, T_n$  représente les  $n$  ( $n > 0$ ) premiers termes de la liste
- $Reste$  représente la liste des éléments restants
- on a l'équivalence

$[T_1, T_2, \dots, T_n] \equiv [T_1, T_2, \dots, T_n | []]$

## Notations équivalentes pour les listes Prolog

```
'.'(a, '.'(b, '.'(c, [])))  
≡ [a|[b|[c|[]]]]  
≡ [a|[b|[c]]]  
≡ [a|[b,c|[]]]  
≡ [a|[b,c]]  
≡ [a,b|[c|[]]]  
≡ [a,b|[c]]  
≡ [a,b,c|[]]  
≡ [a,b,c]
```

## Appartenance à une liste

```
in(T, [T|_]).  
in(T, [_|Q]) :- in(T,Q).
```

```
?- in(b, [a,b,c]).  
yes  
?- in(d, [a,b,c]).  
no  
?- in(X, [a,b,c]).  
X = a; X = b; X = c;  
no more solution
```

## Permutation d'une liste

```
permutation([], []).  
permutation(L, [T|Q]) :-  
    select(T, L, L1), permutation(L1, Q).  
  
select(T, [T|Q], Q).  
select(X, [T|Q], [T|L]) :- select(X, Q, L).
```

```
?- permutation([1,2,3], L).  
L = [1,2,3]; L = [1,3,2]; L = [2,1,3];  
L = [2,3,1]; L = [3,1,2]; L = [3,2,1];  
no more solution
```

## Tri d'une liste

```
tri(L1, L2) :- permutation(L1, L2), ordonnee(L2).  
  
ordonnee([T]).  
ordonnee([T1, T2|Q]) :- T1 < T2, ordonnee([T2|Q]).
```

```
?- tri([3,1,2], L).  
L = [1,2,3];  
no more solution  
?- tri(L, [1,2,3]).  
L = [1,2,3]; L = [1,3,2]; L = [2,1,3];  
L = [2,3,1]; L = [3,1,2]; L = [3,2,1];  
no more solution
```

## Concaténation de listes

```
conc([],L,L).  
conc([T|Q],L,[T|QL]) :- conc(Q,L,QL).
```

```
?- conc([a,b],[c,d,e],[a,b,c,d,e]).  
yes  
?- conc([a,b],[c,d,e],L).  
L = [a,b,c,d,e];  
no more solution  
?- conc([a,b],L,[a,b,c,d,e]).  
L = [c,d,e];  
no more solution
```

## Concaténation de listes

```
conc([],L,L).  
conc([T|Q],L,[T|QL]) :- conc(Q,L,QL).
```

```
?- conc(L1,L2,[a,b,c]).  
L1 = []      L2 = [a,b,c];  
L1 = [a]     L2 = [b,c];  
L1 = [a,b]   L2 = [c];  
L1 = [a,b,c] L2 = [];  
no more solution
```

## Inversion d'une liste

```
% processus récursif  
inverser([], []).  
inverser([T|Q],L) :- inverser(Q,L1), conc(L1,[T],L).
```

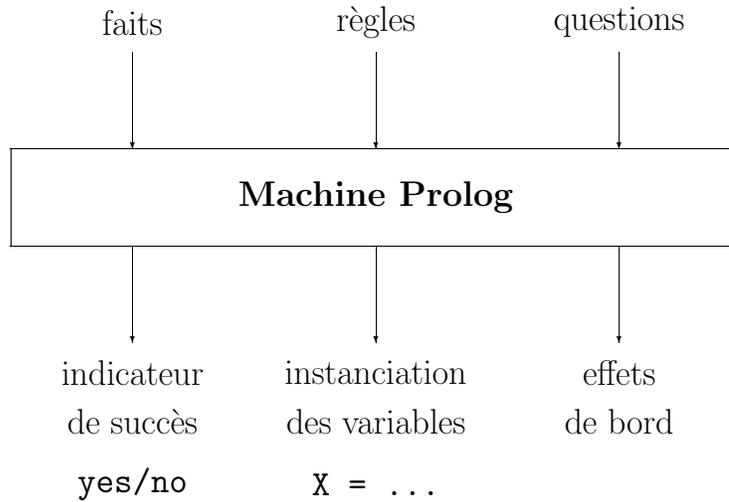
```
?- inverser([a,b,c],L).  
L = [c,b,a];  
no more solution  
?- inverser(L,[c,b,a]).  
L = [a,b,c];  
!!! Stack Overflow !!!
```

## Une autre inversion de liste

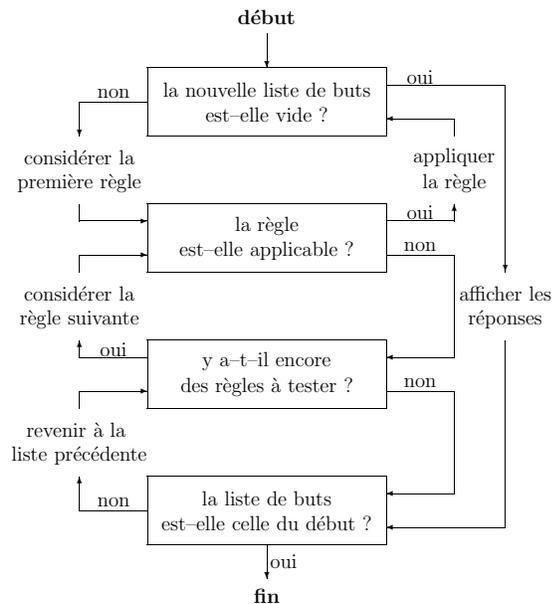
```
% processus itératif  
inverser(L1,L2) :- inverser(L1,[],L2).  
  
inverser([T|Q],Pile,L) :- inverser(Q,[T|Pile],L).  
inverser([],L,L).
```

```
?- inverser([a,b,c],L).  
L = [c,b,a];  
no more solution  
?- inverser(L,[a,b,c]).  
!!! Stack Overflow !!!
```

## La machine Prolog



## L'algorithme Prolog



## Effacer un but

1. Chercher une règle (dans l'ordre où elles apparaissent dans le programme) dont la tête s'unifie avec le but à effacer :
  - même foncteur (atome/arité)
  - arguments unifiables
2. Remplacer le but par le corps de la règle applicable en tenant compte des substitutions de variables effectuées lors de l'unification.

Si le corps de la règle est vide, le but est effacé.

## Effacer un but

$$\{x_1, \dots, x_r\} [b_1, b_2, \dots, b_n]$$

$$\Downarrow$$

règle :  $t :- q_1, q_2, \dots, q_m$

unification :  $t = b_1$ , avec substitution :  $\{x_i/t_i, x_j/t_j, \dots\}$

$$\Downarrow$$

$$\{x_1, \dots, x_i/t_i, \dots, x_r\} [q_1, q_2, \dots, q_m, b_2, \dots, b_n]$$

$$\Downarrow$$

$$\vdots$$

$$\Downarrow$$

$$\{x_1/t_1, x_2/t_2, \dots, x_r/t_r\} []$$

## Unification Prolog

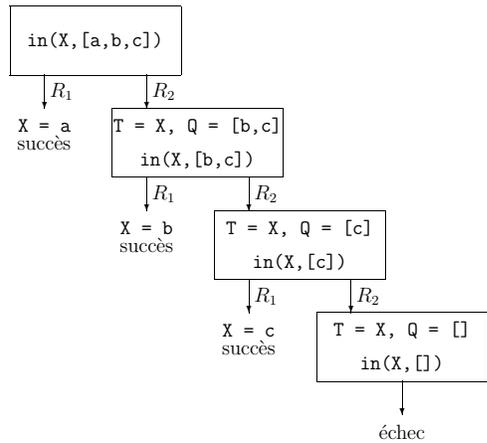
```
u(X,Y) :- var(X), var(Y), X = Y.  
u(X,Y) :- var(X), nonvar(Y), X = Y.  
u(X,Y) :- nonvar(X), var(Y), X = Y.  
u(X,Y) :- atomic(X), atomic(Y), X == Y.  
u(X,Y) :- compound(X), compound(Y), uTerme(X,Y).  
  
uTerme(X,Y) :- functor(X,F,N), functor(Y,F,N), uArgs(N,X,Y).  
  
uArgs(N,X,Y) :- N > 0, uArg(N,X,Y), N1 is N-1, uArgs(N1,X,Y).  
uArgs(0,X,Y).  
  
uArg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY), u(ArgX,ArgY).
```

## Retour arrière

## Arbre de résolution

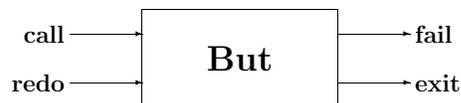
```

in(T, [T|_]).           % R1
in(T, [_|Q]) :- in(T,Q). % R2
    
```



## La trace Prolog

Le mode **trace** permet de visualiser la résolution d'un but de manière interactive.



<b>call</b>	<i>appel initial du but</i>
<b>exit</b>	<i>sortie avec succès du but</i>
<b>redo</b>	<i>retour arrière sur un but</i>
<b>fail</b>	<i>échec du but initial</i>

## Les ports de trace

**call** : L'entrée par ce port de trace s'effectue avant la première tentative d'unification du but à une tête de clause de la base de clauses.

**exit** : La sortie par ce port de trace s'effectue lorsque le but a été unifié à une tête de clause et que tous les sous-butts éventuels du corps de la clause ont pu être prouvés.

**redo** : L'entrée par ce port de trace s'effectue lors d'un retour arrière pour unifier le but à la tête de clause suivante dans la base de clauses.

**fail** : La sortie par ce port de trace s'effectue lorsque le but ne peut pas être unifié à une tête de clause de la base de clauses, ni à aucun prédicat prédéfini.

## Un exemple de trace

```
?- in(X,[a,b,c]).
1 call: in(_192,[a,b,c]) >
1 exit: in(a,[a,b,c])           X = a;
1 redo: in(_192,[a,b,c]) >
2 call: in(_192,[b,c]) >
2 exit: in(b,[b,c])
1 exit: in(b,[a,b,c])           X = b;
2 redo: in(_192,[b,c]) >
3 call: in(_192,[c]) >
3 exit: in(c,[c])
2 exit: in(c,[b,c])
1 exit: in(c,[a,b,c])           X = c;
3 redo: in(_192,[c]) >
4 call: in(_192,[]) >
4 fail: in(_192,[])           no more solution
```

## Ordonnancement des clauses

L'inversion des clauses ne modifie pas l'arbre de résolution, seul l'ordre des solutions est modifié.

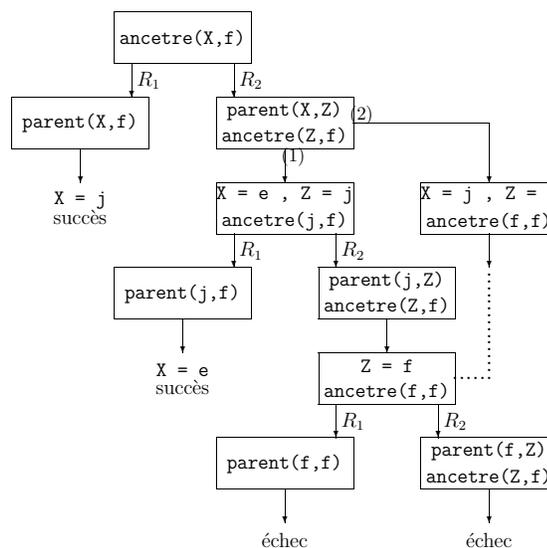
```

parent(e,j). parent(j,f).

                                % ancetre/2 : version 1
ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z), ancetre1(Z,Y).

                                % ancetre/2 : version 2
ancetre2(X,Y) :- parent(X,Z), ancetre2(Z,Y).
ancetre2(X,Y) :- parent(X,Y).
    
```

## ancetre1/2 ↔ ancetre2/2



## Ordonnancement des buts

L'inversion des buts dans une clause modifie l'arbre de résolution.

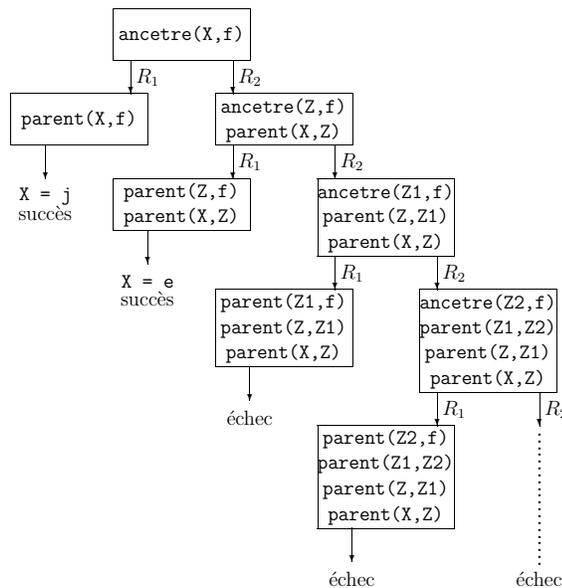
```

                                % ancetre/2 : version 1
ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z), ancetre1(Z,Y).

                                % ancetre/2 : version 3
ancetre3(X,Y) :- parent(X,Y).
ancetre3(X,Y) :- ancetre3(Z,Y), parent(X,Z).
    
```



## ancetre3/2 ↔ ancetre4/2



## Branches infinies

```

                                % ancetre/2 : version 4
ancetre4(X,Y) :- ancetre4(Z,Y), parent(X,Z).
ancetre4(X,Y) :- parent(X,Y).
```

```

?- ancetre4(X,fabien).
1 call:  ancetre4(_192,fabien)
2 call:  ancetre4(_248,fabien)
3 call:  ancetre4(_278,fabien)
4 call:  ancetre4(_308,fabien)
5 call:  ancetre4(_338,fabien)
...
EXCEPTION : stack overflow
```

## Les 4 versions d'ancetre/2

⇐ Permutations des buts ⇒

```

ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z), ancetre1(Z,Y).
?- ancetre1(X,f).
X = j; X = e;
no more solution
```

```

ancetre3(X,Y) :- parent(X,Y).
ancetre3(X,Y) :- ancetre3(Z,Y), parent(X,Z).
?- ancetre3(X,f).
X = j; X = e;
... Stack Overflow
```

↑

↑

Permutation des clauses

↓

↓

```

ancetre2(X,Y) :- parent(X,Z), ancetre2(Z,Y).
ancetre2(X,Y) :- parent(X,Y).
?- ancetre2(X,f).
X = e; X = j;
no more solution
```

```

ancetre4(X,Y) :- ancetre4(Z,Y), parent(X,Z).
ancetre4(X,Y) :- parent(X,Y).
?- ancetre4(X,f).
... Stack Overflow
```

⇐ Permutations des buts ⇒

## Récurtivité terminale ou non terminale

```
                                % lgr1(N,L)
                                % processus récurisif
lgr1(0, []).
lgr1(N, [_|Q]) :- lgr1(NQ,Q), N is NQ + 1.

                                % lgr2(N,L)
                                % processus itératif
lgr2(N,L) :- lgr(N,0,L).

lgr(N,Sum, [_|Q]) :- Sum1 is Sum + 1, lgr(N,Sum1,Q).
lgr(N,N, []).
```

## Récurtivité terminale ou non terminale

```
                                % inverser1(L1,L2)
                                % processus récurisif
inverser1([], []).
inverser1([T|Q],L) :- inverser1(Q,L1), conc(L1, [T],L).

                                % inverser2(L1,L2)
                                % processus itératif
inverser2(L1,L2) :- inverser(L1, [],L2).

inverser([T|Q],Pile,L) :- inverser(Q, [T|Pile],L).
inverser([],L,L).
```

## Contrôle de la résolution

Certains prédicats ont un comportement procédural; leurs effets ne sont pas effacés par retour arrière.

**Coupe-choix** : élagage de l'arbre de résolution

**Gestion de la mémoire** : ajout et/ou retrait de clauses à l'exécution

**Entrées/Sorties** : écriture ou lecture de termes

## Prédicats de contrôle

<b>!</b>	Réussit toujours mais annule tous les points de choix créés depuis l'appel du but parent.	<i>coupe-choix</i>
<b>call(But)</b>	Evalue <b>But</b> .	<i>interpréteur</i>
<b>fail</b>	Echoue toujours.	<i>échec</i>
<b>not But</b>	<b>But</b> n'est pas démontrable.	<i>négation par l'échec</i>
<b>repeat</b>	Réussit toujours même en cas de retour arrière.	<i>boucle infinie</i>
<b>true</b>	Réussit toujours.	<i>succès</i>

## Le coupe-choix

Le coupe-choix (*cut*) a pour but d'élaguer l'arbre de résolution. Cet élagage *peut* conduire à

- une plus grande rapidité d'exécution,
- moins de place mémoire utilisée.

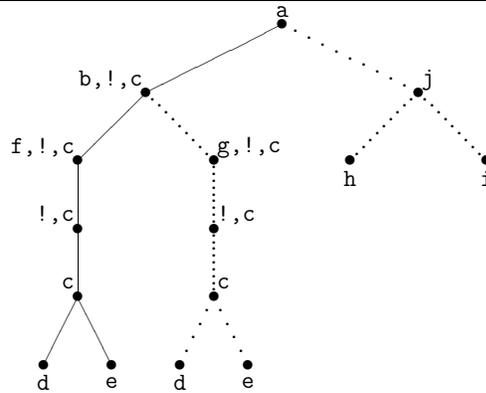
## Coupe-choix rouge ou vert ?

- Dans *tous les cas*, l'interprétation procédurale du programme est modifiée : le programme ne s'exécute pas de la même manière avec ou sans coupe-choix.
- Dans *certains cas*, la signification déclarative du programme est conservée (coupe-choix "vert") : le programme a la même interprétation logique avec ou sans coupe-choix.
- Dans *les autres cas*, la signification déclarative du programme est modifiée (coupe-choix "rouge") : le programme n'a pas la même interprétation logique avec ou sans coupe-choix.

## Elagage de l'arbre de résolution

```

a :- b, !, c.
a :- j.
b :- f. b :- g. c :- d. c :- e.
f. g. j :- h. j :- i.
    
```



## Coupe-choix "vert"

```

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

```

?- max(3,2,X).
X = 3;
no more solution
?- max(2,3,X).
X = 3;
no more solution
?- max(3,2,2).
no
    
```

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est conservée.

## Alternative

```
:- op(900,fx,si), op(850,xfx,alors), op(800,xfx,sinon).

si Cond alors True sinon False :-
    call(Cond), !, call(True).
si Cond alors True sinon False :-
    not call(Cond), !, call(False).

                                     % exemple d'utilisation
max(X,Y,Z) :- si X > Y alors Z = X sinon Z = Y.
```

## Coupe-choix “rouge”

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

```
?- max(3,2,X).
X = 3;
no more solution
?- max(2,3,X).
X = 3;
no more solution
?- max(3,2,2).
yes                                     % !!!!
```

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est modifiée.

## La première solution

```

s(1). s(2). s(3).
           % la première solution
ps(X) :- s(X), !.
           % la deuxième solution
ds(X) :- ps(Y), s(X), X \== Y, !.

?- ps(X).
X = 1;
no more solution
?- ds(X).
X = 2;
no more solution

```

## La négation par l'échec

```

           % négation par l'échec
non(But) :- call(But), !, fail.
non(But).

```

```

?- s(X).
X = 1; X = 2;
no more solution
?- non(non(s(X))).
X = _192;
no more solution

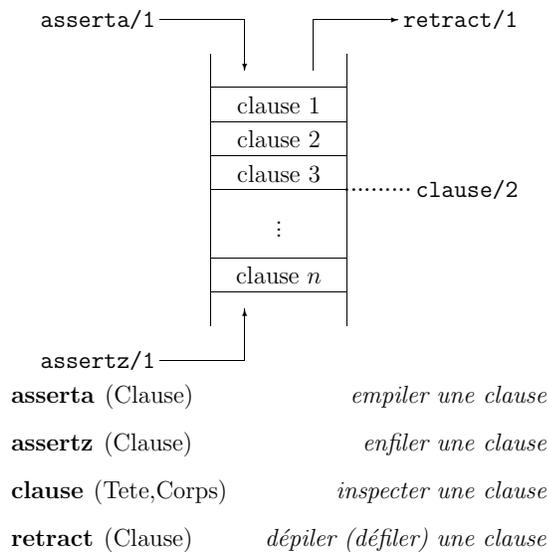
```

## Une boucle de lecture

```
lecture :-
  repeat,
    read(Terme), write('-> '), write(Terme), nl,
  Terme == fin, !.
```

```
?- lecture.
bonjour.
-> bonjour
salut.
-> salut
fin.
-> fin
yes
```

## L'espace des clauses



## Un chaînage avant

```

deduire :- regle(B,C), test(C), affirmer(B),
           !, deduire.
deduire.

test((C1,C2)) :- !, test(C1), test(C2).
test((C1;C2)) :- !, ( test(C1) ; test(C2) ).
test(C) :- regle(C,vrai) ; regle(C,affirme).

affirmer(B) :- not test(B), !,
              assert(regle(B,affirme)),
              write(B),write(' affirme'),nl.

```

## Ensemble de solutions

```

findall(T,But,L) :- call(But), assertz(sol(T)), fail.
findall(T,But,L) :- assertz(sol('fin')), fail.
findall(T,But,L) :- recup(L).

recup([T|Q]) :-
    retract(sol(T)), T \== 'fin', !, recup(Q).
recup([]).

```

## Exemples avec findall/3

```
s(b,1). s(a,1). s(c,1). s(a,1).
s(d,2).
```

```
?- findall(T,s(T,1),L).
T = _192 , L = [b,a,c,a];
no more solution
?- findall(T,s(T,X),L).
T = _192 , X = _194 , L = [b,a,c,a,d];
no more solution
?- findall(T,s(T,3),L).
T = _192 , L = [];
no more solution
```

## Prédicats prédéfinis

Arithmétique .....	87
Types des termes .....	88
Classification de termes .....	89
Manipulations de termes .....	90
Contrôle de la résolution .....	91
Ensembles de solutions .....	92
Manipulations de clauses .....	93
Entrées/Sorties de termes .....	94
Gestion des flots d'entrées/sorties .....	95

## Arithmétique

Result is Expression	<i>evaluate expression</i>
Expression1 := Expression2	<i>arithmetic equal</i>
Expression1 =\= Expression2	<i>arithmetic not equal</i>
Expression1 < Expression2	<i>arithmetic less than</i>
Expression1 =< Expression2	<i>less than or equal</i>
Expression1 > Expression2	<i>arithmetic greater than</i>
Expression1 >= Expression2	<i>greater than or equal</i>

## Classification des termes

atom(Term)	<i>atom ?</i>
atomic(Term)	<i>atomic term ?</i>
compound(Term)	<i>compound term ?</i>
float(Term)	<i>float ?</i>
integer(Term)	<i>integer ?</i>
number(Term)	<i>number ?</i>
nonvar(Term)	<i>bounded variable ?</i>
var(Term)	<i>free variable ?</i>

## Comparaisons de termes

Term1 == Term2	<i>term identical</i>
Term1 \== Term2	<i>term not identical</i>
Term1 @< Term2	<i>term less than</i>
Term1 @=< Term2	<i>term less than or equal</i>
Term1 @> Term2	<i>term greater than</i>
Term1 @>= Term2	<i>term greater than or equal</i>

## Manipulations de termes

Terme1 = Terme2	<i>term unification</i>
Terme1 \Terme2	<i>non term unification</i>
arg(Integer,CompoundTerm,Term)	<i>term inspection</i>
functor(Term,Name,Arity)	<i>term inspection</i>
Term =.. List	<i>term inspection</i>
copy_term(Term1,Term2)	<i>term duplication</i>

## Contrôle de la résolution

call(Goal)	<i>metacall</i>
Goal1 , Goal2	<i>conjunction</i>
Goal1 ; Goal2	<i>disjunction</i>
!	<i>cut</i>
fail	<i>failure</i>
halt	<i>top-level stop</i>
Condition -> Then ; Else	<i>if-then-else</i>
\+(Goal)	<i>not provable</i>
once(Goal)	<i>once only</i>
repeat	<i>infinite loop</i>
true	<i>success</i>

## Ensembles de solutions

bagof(Template,Goal,List)	<i>bags of all solutions</i>
findall(Template,Goal,List)	<i>find all solutions</i>
setof(Template,Goal,List)	<i>sorted sets of all solutions</i>

## Manipulations de clauses

abolish(Functor)	<i>clause destruction</i>
asserta(Clause)	<i>clause creation</i>
assertz(Clause)	<i>clause creation</i>
clause(Head,Body)	<i>clause information</i>
current_predicate(Term)	<i>clause retrieval</i>
retract(Clause)	<i>clause destruction</i>

## Entrées/Sorties de termes

current_op(Precedence,Type,Name)	<i>operator retrieval</i>
op(Precedence,Type,Name)	<i>operator definition</i>
read(Term)	<i>read a term</i>
read(Stream,Term)	<i>read a term from a stream</i>
write(Term)	<i>write a term</i>
write(Stream,Term)	<i>write a term to a stream</i>

## Gestion des flots d'entrées/sorties

<code>at_end_of_stream</code>	<i>end of file</i>
<code>at_end_of_stream(Stream)</code>	<i>end of file</i>
<code>close(Stream)</code>	<i>close a stream</i>
<code>current_input(Stream)</code>	<i>input stream identification</i>
<code>current_output(Stream)</code>	<i>output stream identification</i>
<code>flush_output(Stream)</code>	<i>flush buffered output stream</i>
<code>open(File,Mode,Stream,Opts)</code>	<i>open a stream</i>
<code>set_input(Stream)</code>	<i>set the input stream</i>
<code>set_output(Stream)</code>	<i>set the output stream</i>
<code>stream_position(Str,Old,New)</code>	<i>set the position in a stream</i>

## Bibliographie

- Bratko I.** *Programmation en Prolog pour l'intelligence artificielle*, InterEditions, 1988
- Clocksin F.W., Mellish C.S.** *Programmer en Prolog*, Eyrolles, 1986
- Coello H., Cotta J.C.** *Prolog by example. How to learn, teach and use it*, Springer Verlag, 1988
- Deransart P., Ed-Dbali A., Cervoni L.** *Prolog : the standard*, Springer Verlag, 1996
- O'Keefe R.A.** *The craft of Prolog*, MIT Press, 1990
- Sterling L., Shapiro E.** *L'art de Prolog*, Masson, 1990