

## Prolog

### Preuves automatiques et programmation logique

Système pour prouver automatiquement des conséquences logiques

- Démonstrateur de théorèmes basé sur la résolution
- Utilisation en programmation (en plus des approches procédurales et fonctionnelles)

#### Approche logique de la programmation:

calculer = prouver (chercher toutes les preuves)

## Prolog

«Programme» Prolog (= Théorie logique)

- faits
- règles
- question

La question est-elle prouvable à partir des faits et règles ?

```
humain(paul).          /* fait 1 */
animal(bonzo).         /* fait 2 */
parle(X) :- humain(X). /* règle 1 */
```

```
?- parle(bonzo).
no
?- parle(paul).
yes
?- parle(toto).
no
```

## Syntaxe

Constantes, fonctions, prédicats : `c`, `x36`, `bonne_annee`, ...

Variables: `X`, `Y`, `A`, `Arc`, `_X`, ...

Formules:

- clauses **de Horn** (un seul littéral positif)
- variables quantifiées universellement (implicitement)
- un seul littéral positif
- `:-` remplace `<=`
- `,` remplace `^`

```
parle(X, Y) :- humain(X), ami(X, Y).
```

signifie

$$\forall X \forall Y ( \text{humain}(X) \wedge \text{ami}(X, Y) \Rightarrow \text{parle}(X, Y) )$$
$$== \forall X \forall Y ( \neg \text{humain}(X) \vee \neg \text{ami}(X, Y) \vee \text{parle}(X, Y) )$$

## Prolog utilise le principe de résolution

```
1. humain(pierre)
2. humain(jean)
3. ami(médor, pierre)
4. ami(jean, pierre)
5. ami(charles, jean)
6. parle(X, Y) :- humain(X), ami(X, Y)
```

```
parle(jean, pierre) ?
7. :- parle(jean, pierre)
```

7 et 6 avec (X=jean, Y=pierre)

```
8. :- humain(jean), ami(jean, pierre)
```

8 et 1

```
9. :- ami(jean, pierre)
```

9 et 4

```
10. :-
```

## Prolog utilise le principe de résolution

```
1. humain(pierre)
2. humain(jean)
3. ami(médor, pierre)
4. ami(jean, pierre)
5. ami(charles, jean)
6. parle(X, Y) :- humain(X), ami(X, Y)
```

```
parle(médor, pierre) ?
7. :- parle(médor, pierre)
```

7 et 6 avec (X=médor, Y=pierre)

```
8. :- humain(médor), ami(médor, pierre)
```

8 et 3

```
9. :- humain(médor)
```

On ne peut plus rien résoudre  
NON

## Questions ouvertes

Si une question contient une ou plusieurs variables

Prolog cherche toutes les valeurs des variables qui rendent la question prouvable.

```
1. humain(pierre)
2. humain(jean)
3. ami(médor, pierre)
4. ami(jean, pierre)
5. ami(charles, jean)
6. parle(X, Y) :- humain(X), ami(X, Y)
7. humain(charles)
```

```
parle(X, Y) ?
```

```
X = jean, Y = pierre
X = charles, Y = jean
```

## Technique de résolution

Prolog utilise le principe de résolution en **chaînage arrière**.

Pour prouver

```
p(a)
```

chercher

soit un fait

```
p(t)
```

soit une règle

```
p(t) :- L1, L2, ..., Ln
```

tels que **a** soit unifiable avec **t**

Si c'est un fait de base, la preuve est terminée

Si c'est une règle,

il faut prouver **L1**, **L2**, ..., **Ln**, on utilise la même technique.

si la preuve ne marche pas on essaye une autre unification.

## Exemple

```
1. humain(pierre)
2. humain(jean)
4. ami(jean, pierre)
7. parle(X, Y) :- humain(X), ami(X, Y)
8. ami(X, Y) :- ami(Y, X).
```

Prouver : `parle(pierre, jean)`

avec **7.**

Prouver : `humain(pierre), ami(pierre, jean)`

Prouver : `humain(pierre)`

avec **1.**

Prouver : `ami(pierre, jean)`

avec **8.**

Prouver : `ami(jean, pierre)`

avec **4.**

## Exemple avec une négation

```
humain(paul).
humain(ernest).
humain(albert).
animal(bonzo).
bebe(ernest).
bebe(jim).

parle(X) :- humain(X), not bebe(X).

?- parle(albert).
```

Résolution avec `parle(X) :- humain(X), not bebe(X)` .

Unificateur: `X = albert`

Résolvant: `humain(albert), not bebe(albert)` .

## Négation (suite)

A prouver: `humain(albert), not bebe(albert)` .

1) `humain(albert)` est un fait --> OK

2) prouver `not bebe(albert)` .

On ne trouve aucun fait ou règle pour `bebe(albert)` .

Donc impossible de prouver `bebe(albert)`

### NEGATION PAR ECHEC

on considère que `not bebe(albert)` est prouvé

Donc `parle(albert)` est prouvé.

## Le processus de recherche en marche arrière

Problème: on a en général beaucoup de choix possibles pour appliquer la résolution.

Solution Prolog: chaînage arrière

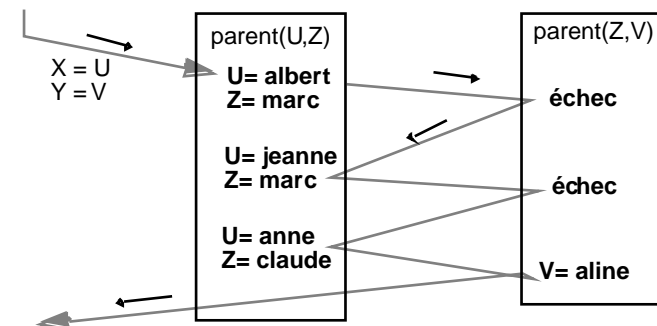
Lorsque un sous-but ne peut être satisfait,

- revient au sous-but précédent,
- essaye de le satisfaire d'une autre manière,
- repart en avant.

## Exemple.

```
parent(albert, marc). parent(jeanne, marc).
parent(anne, claudé).
parent(anne, paul). parent(claudé, aline).
grand_parent(X,Y) :- parent(X,Z), parent(Z,Y).
```

`grand_parent(U,V)`



Les instantiations de variables sont défaites lors de la marche arrière.

## Trace de la recherche

```

• 1 | 1 call grand_parent(_420,_421)
• 2 | 2 call parent(_420,_1250)
• 2 | 2 exit parent(albert,marc)
• 3 | 2 call parent(marc,_421)
• 3 | 2 fail parent(marc,_421)
• 2 | 2 redo parent(albert,marc)
• 2 | 2 exit parent(jeanne,marc)
• 4 | 2 call parent(marc,_421)
• 4 | 2 fail parent(marc,_421)
• 2 | 2 redo parent(jeanne,marc)
• 2 | 2 exit parent(anne,claudé)
• 5 | 2 call parent(claudé,_421)
• 5 | 2 exit parent(claudé,aline)
• 1 | 1 exit grand_parent(anne,aline)
• 1 | 1 redo grand_parent(anne,aline)
• 5 | 2 redo parent(claudé,aline)
• 5 | 2 fail parent(claudé,_421)

```

## Choix des sous-buts

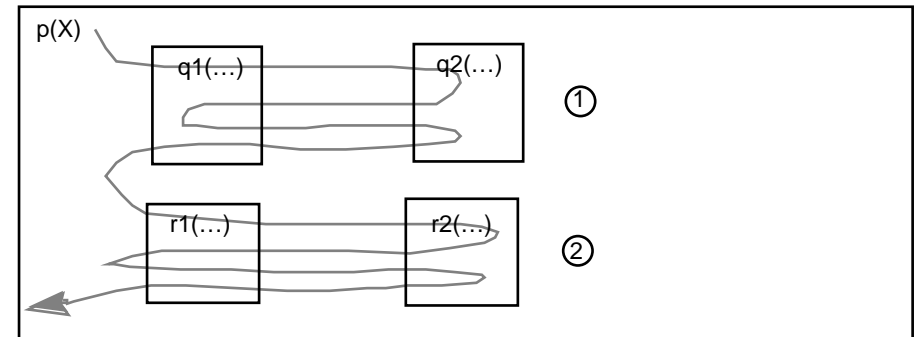
A l'intérieur d'une règle : ordre d'écriture

S'il y a plusieurs règles dont la tête est unifiable : ordre d'écriture des règles

```

p(X) :- q1(...), q2(...).      /* 1 */
p(X) :- r1(...), r2(...).    /* 2 */

```

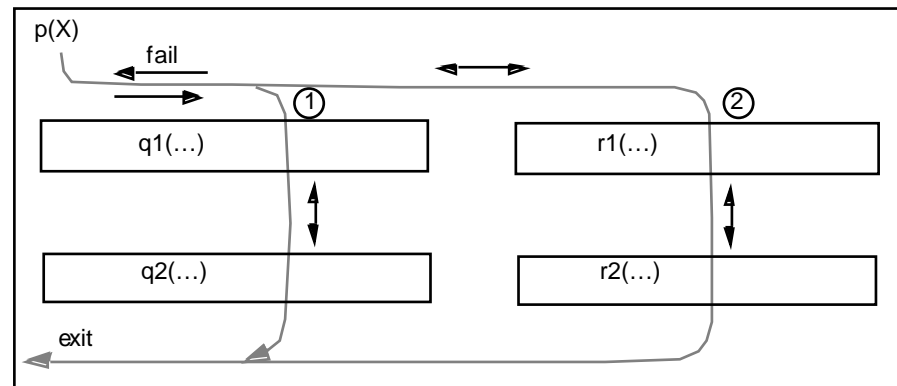


## Représentation graphique du « chemin de satisfaction »

```

p(X) :- q1(...), q2(...).
p(X) :- r1(...), r2(...).

```



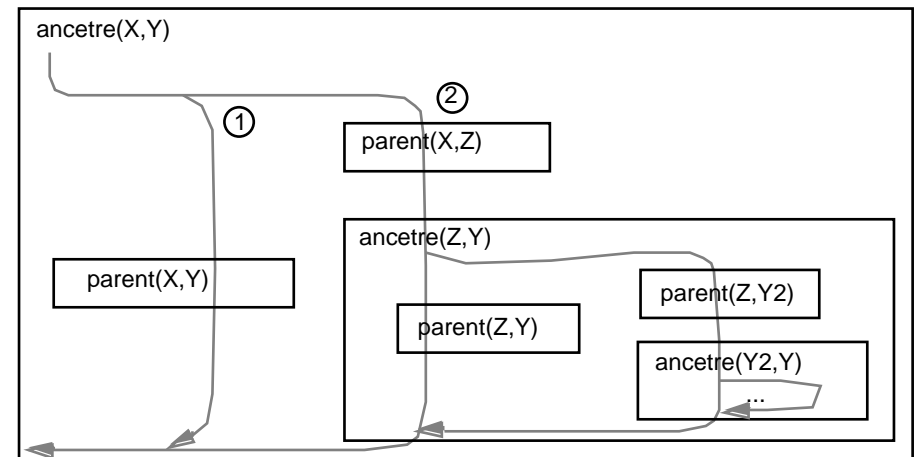
En cas d'échec on recule sur le chemin

## Règles récursives

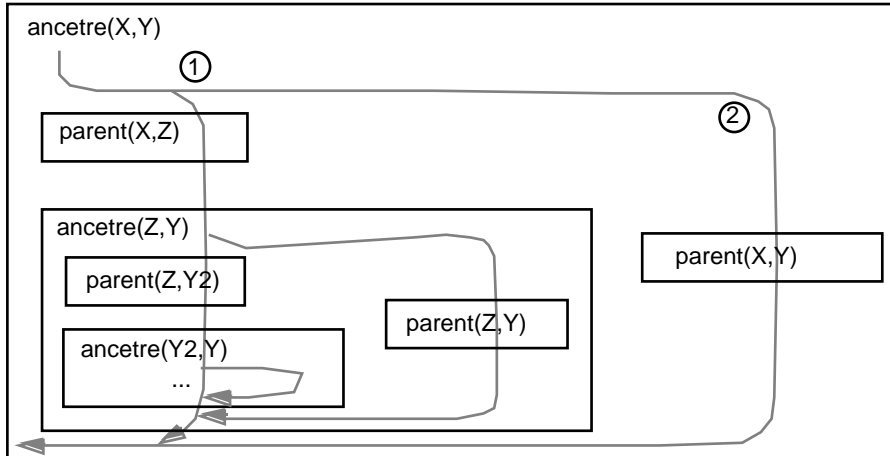
```

ancetre(X,Y) :- parent(X,Y).
ancetre(X,Y) :- parent(X,Z), ancetre(Z,Y).

```



## Importance de l'ordre de déclaration



On n'en sort jamais !

## Résolution infinie

```
1. ancetre(X, Y) :- ancetre(X, Z), parent(Z, Y).
2. ancetre(X, Y) :- parent(X, Y).
3. parent(a, b).
4. parent(b, c).
```

```
5. :- ancetre(a, c)
```

rés. 5-1

```
6. :- ancetre(a, Z), parent(Z, c).
```

rés. 6-1 (Z -> Z2 dans 6, X=a, Y=Z2)

```
7. :- ancetre(a, Z), parent(Z, Z2), parent(Z2, c)
```

rés. 7-1 (Z -> Z3 dans 7, X=a, Z3=Y)

```
8. :- ancetre(a, Z), parent(Z, Z3), parent(Z3, Z2), parent(Z2, c)
```

etc.

## Unification et fonctions

Pour satisfaire un but  $p(v_1, \dots, v_n)$

il faut trouver un fait ou une règle de la forme  $p(w_1, \dots, w_n) :- \dots$

tel que

$v_1$  unifiable avec  $w_1$ , ...,  $v_n$  unifiable avec  $w_n$

Règle d'unification

- si  $v$  est une variable et  $w$  est un objet quelconque: OK
- si  $v$  et  $w$  sont des variables : OK,  $v$  et  $w$  sont désormais «liées»
- un atome ou nombre entier ne s'unifie qu'avec lui-même
- une fonction (structure) ne s'unifie qu'avec
- une fonction de même nom
- et de même nombre de paramètres
- si les paramètres sont unifiables (récursion)

N.B. Les fonctions ne sont pas évaluées ( $\neq$  programmation impérative), sauf demande explicite.

## Exemples

Faits:  $p(35)$ ,  $p(a)$ ,  $p(f(3,b))$ ,  $p(a+3)$ ,  $p(6+7)$ .

?-  $p(35)$

yes

?-  $p(f(X,Y))$ .

X = 3

Y = b

?-  $p(c)$ .

no

?p(13).

no

?-  $p(X)$ .

X = 35;

X = a;

X = f(3,b);

X = a+3;

X = 6+7;

?p(X+Y).

X = a

Y = 3 ;

X = 6

Y = 7

## Exemple: Théorie des nombres

Représentation des nombres

- la constante 0
- la fonction s (unaire) p.ex. 5 est représenté par s(s(s(s(0))))
- la fonction + (binaire)

Prédicat: egal(binaire)

**Axiomes :**

$\forall x (x+0 = x)$

$\forall x (0+x = x)$

$\forall x \forall y \forall z (x+y = z \Rightarrow x+s(y) = s(z))$

**Prolog**

egal(X+0, X).

egal(0+X, X).

egal(X+s(Y), Z) :- egal(X, Y, Z).

## Essais

```
?- egal(s(s(0))+s(s(s(0))), s(s(s(s(0))))).
yes
?- egal(s(s(0))+s(s(0)),X).
X = s(s(s(0)))
?- egal(X+s(s(0)),s(s(s(s(0))))).
X = s(s(s(0))) ;
egal(X, s(s(s(s(0))))).
X = s(s(s(s(0))))+0 ;
X = 0+s(s(s(s(0)))) ;
X = s(s(s(0)))+s(0) ;
X = 0+s(s(s(0))) ;
X = s(s(0))+s(s(0)) ;
X = 0+s(s(s(0))) ;
X = s(0)+s(s(s(0))) ;
X = 0+s(s(s(0))) ;
X = 0+s(s(s(0))) ;
X = 0+s(s(s(0))) ;
```

## Une infinité de réponses

```
egal(X, Y).
```

```
X = _5946+0
```

```
Y = _5946 ;
```

```
X = 0+_5946
```

```
Y = _5946 ;
```

```
X = _6124+s(0)
```

```
Y = s(_6124) ;
```

```
X = 0+s(_6125)
```

```
Y = s(_6125) ;
```

```
...
```

```
X = _6124+s(s(s(s(s(s(s(s(s(0))))))))))
```

```
Y = s(s(s(s(s(s(s(s(s(_6124))))))))))
```

```
etc.
```

## Négation par échec de preuve

Le but **not P(X)** est satisfait si on n'arrive pas à prouver **P(X)**.

Particularité (non logique) : les instantiations sont défaites lors de l'échec

**Exemple**

```
cours(a1). cours(a2). cours(b1).
cours(b2). cours(c1). cours(c2).
prereq(a1,b1). prereq(a2,b1). prereq(b1,c1).
prereq(b2,c1). prereq(b3,c2).
etu(paul). etu(pierre). etu(amelie).
acquis(paul,a1). acquis(paul,a2).
acquis(pierre,a2).
acquis(amelie,b3).
```

```
ne_peut_s_inscrire(X,Y) :- prereq(Z,Y), not acquis(X,Z).
```

```
peut_s_inscrire1(X,Y) :- not ne_peut_s_inscrire(X,Y).
```

## Utilisation

```
?- peut_s_inscrire1(paul,b1).  
yes  
?- peut_s_inscrire1(paul,Y).  
no
```

Le Y qui a causé l'échec n'est pas transmis en arrière \*\*\*/

## Réparation

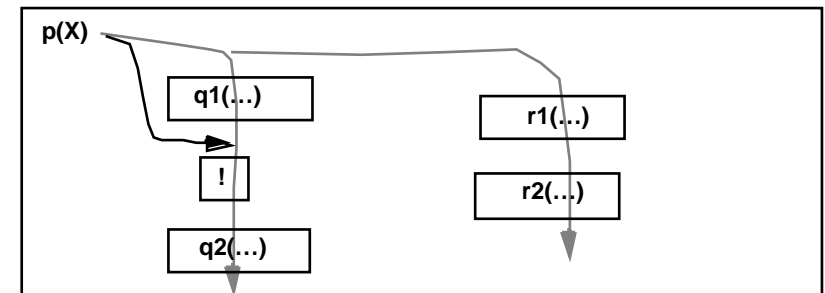
```
peut_s_inscrire2(X,Y) :- etu(X), cours(Y), not  
ne_peut_s_inscrire(X,Y).  
?- peut_s_inscrire2(paul,Y).  
Y = a1 ;  
Y = a2 ;  
Y = b1 ;  
Y = b2 ;
```

## Contrôle de l'exploration

Il est parfois utile ou nécessaire d'empêcher la marche arrière.

- n'utiliser qu'une seule règle pour résoudre un but
- détecter les situations d'échec où la recherche d'autres solutions ne conduira à rien (optimisation).
- lorsque on sait qu'il n'y a qu'une seule solution, pour éviter la recherche d'alternatives (optimisation).

### Principe



## Exemple: choix d'une seule règle

Les personnes qui ont un livre en retard n'ont accès qu'aux serves de base de la bibliothèque.

```
service(Pers, S) :- en_retard(Pers, Livre), !, serv_base(S).  
service(Pers, S) :- serv_general(S).
```

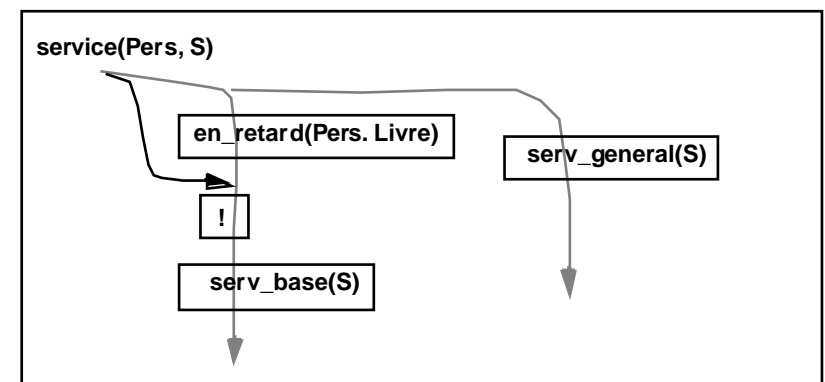
```
serv_base(reference).  
serv_base(questions).
```

```
serv_additionnel(emprunt).  
serv_additionnel(pret_inter_biblio).
```

```
serv_general(X) :- serv_base(X).  
serv_general(X) :- serv_additionnel(X).
```

```
en_retard('G. Oublié', 'Mathematical logic for CS').  
en_retard('J. Vai', 'Le grand tout').
```

## Chemins de satisfaction



## Choix d'une seule règle

```
somme(0,0).
```

```
somme(N, R) :- N1 is N - 1, /* somme des entiers de 0 à N = R */  
              somme(N1, R1),  
              R is R1 + 1
```

Que se passe-t-il dans

```
but :- p(X), somme(0, R), q(X, R).
```

lorsque q(X, R) échoue ?

Plus sûr:

```
somme(0, 0) :- !.
```

Encore plus sûr:

```
somme(N, 0) :- N < 1, !.
```

## Remplacement de ! par not

```
somme(N, 0) :- N < 1.
```

```
somme(N, R) :- N1 is N - 1,  
              not (N < 1), /* bloque la règle si N < 1 */  
              somme(N1, R1),  
              R is R1 + 1
```

Et vice-versa

```
A :- B, C.
```

```
A :- not B, D
```

Devient

```
A :- B, !, C.
```

```
A :- D
```

Plus efficace: on essaye de satisfaire B une seule fois.

## Echecs sans autres alternatives

Commencer par éliminer les cas pour lesquels

- on sait que le prédicat est faux;
- on veut que le prédicat soit faux.

```
etudiant_standard(E) :- mobile(X), !, fail.
```

```
etudiant_standard(E) :- licence(X, L), licence(X, L2), L =/ L2,  
!, fail.
```

```
etudiant_standard(E) :- .../* test compliqué */
```

## Une seule solution

Situation

- Il existe plusieurs manières de trouver la solution
- On sait qu'il n'y a qu'une solution
- 

```
habite(X, Ville) :- adresse(X, Rue, No, Ville), !.
```

```
habite(X, Ville) :- habite_chez(X, Y), habite(Y, Ville), !.
```

```
habite(X, Ville) :- sous_locataire(X, Y), habite(Y, Ville), !.
```

etc.

```
actif(Prof) :- enseigne(Prof, Matiere), !.
```

```
eligible(Prof) :- actif(Prof), engage_en(Prof, Date), Date <  
1999.
```