

Introduction à Prolog

http://www.enseeiht.fr/lima/ia/Ens/2I_N7/Cours/SIA

MCours.com

Lien avec la logique formelle

Prolog met en œuvre une procédure de réfutation dite linéaire par entrées contrainte par :

- la clause centrale de départ C_0 est une clause négative de la forme clausale associée à la négation de la conjecture
- il n'y a pas vérification que $G \cup \{C_0\}$ est satisfiable
 - donc pas de garantie de succès (obtention de la clause vide)
- les clauses sont des clauses de Horn (au plus 1 littéral positif)
 - il faut savoir que certains dialectes outrepassent cette contrainte
 - donc pas de garantie de succès
- Prolog propose des prédicats prédéfinis

Lien avec la logique formelle

- certains sont des méta-prédicats de contrôle de la recherche d'une réfutation (ex: not, assert, cut, findall, freeze, etc.)
- la plupart des dialectes sortent du cadre de la logique du premier ordre (ordre supérieur à 1)
- la résolution est binaire sans factorisation
- le littéral effacé de la clause centrale courante est le 1er littéral négatif dans l'ordre d'écriture des littéraux
- il est remplacé par les littéraux non effacés d'une clause de bord (une clause d'entrée) en conservant l'ordre d'écriture :

si $\sigma(l_1) = \sigma(h_1)$ alors

$$\begin{aligned} \text{Res}(\neg l_1 \vee \neg l_2 \vee \dots \vee \neg l_n, h_1 \vee \neg h_2 \vee \dots \vee \neg h_m) \\ = \sigma(\neg h_2 \vee \dots \vee \neg h_m \vee \neg l_2 \vee \dots \vee \neg l_n) \end{aligned}$$

Lien avec la logique formelle

- les clauses d'entrées (programme Prolog) sont toujours examinées dans le même ordre (ordre d'écriture du programme)
 - la plupart des dialectes obligent les clauses à ce regrouper par nom: suites de faits (atomes) ou de règles (clauses) utilisant un même nom de prédicat
 - la recherche d'une clause vide se fait en profondeur (et non pas niveau par niveau comme en largeur)
 - pas de garantie de terminaison, donc de succès
 - Prolog recherche toutes les clauses vides (pas seulement la 1ère).
- Certaines de ces caractéristiques font de Prolog une stratégie non complète

Résolution de problèmes

La programmation logique est particulièrement adaptée à la résolution de problèmes (ex: en IA)

1. Représentation sous forme logique

Un problème est résolu par une démonstration.

Exemple:

On utilise $p(X,Y)$ pour dire X est le père de Y ,

$gp(X,Y)$ pour dire X est le grand père de Y .

On définit le grand père ainsi

règle: $\forall X,Y,Z p(X,Y) \wedge p(Y,Z) \Rightarrow gp(X,Z)$

faits: $p(m,a)$, $p(m,j)$ et $p(j,e)$

question: $gp(m,e)$? "Marc est-il le grand père d'Eric?"

Résolution de problèmes

Le problème, représenté logiquement, peut être résolu en utilisant le principe de résolution. Il exige de s'appliquer sur des formes clausales:

$c_1: gp(X,Z) \vee \neg p(X,Y) \vee \neg p(Y,Z)$ $c_2: p(m,a)$ $c_3: p(m,j)$ $c_4: p(j,e)$

On y ajoute le contraire de la conclusion $c_0: \neg gp(m,e)$

On cherche une réfutation de $\{c_0, c_1, c_2, c_3, c_4\}$ en utilisant une variante de:

Trouver $n \in \mathbb{N}$ tel que $\square \in S_n$ où

$S_0 = \{c_1, c_2, c_3, c_4\}$

$S_n = \{ \text{certains résolvants de } c \text{ et } c' / \text{ pour certains } c \in S_0 \cup \dots \cup S_{n-1} \text{ et certains } c' \in S_{n-1} \}.$

Résolution de problèmes

2. Cadre théorique

1. Les clauses de Horn

Les éléments de S_n sont tous des clauses de Horn (au plus un atome non nié).

Les faits sont des atomes de la forme (syntaxe Edimbourg):

$p(m,a)$. % les arguments peuvent être des variables ou des constantes

$p(m,j)$. % ou des expressions fonctionnelles (i.e.: des termes)

$p(j,e)$.

Les règles sont de la forme:

$gp(X,Z):- p(X,Y), p(Y,Z)$.

MCours.com

Faits et règles forment le programme (!-respecter la casse-!)

Résolution de problèmes

Le démenti (question) est de la forme (conjonction d'atomes):

?- gp(G,e),p(P,e). % quel est le grand père G et le père P d'Eric

2. La procédure de preuve de Prolog

- Prolog utilise le démenti comme clause initiale (C_0).
- Il parcourt le programme (du début à la fin) à la recherche de la première clause dont l'atome de tête (situé à gauche de :-) s'unifie (σ) avec l'atome de tête du démenti courant.
- Il lui substitue la queue de clause (située à droite de :-) en remplaçant certaines variables par les valeurs appropriées (σ).
Le résultat forme le nouveau démenti courant.

Résolution de problèmes

- Il recherche ainsi en profondeur la clause vide.
- Il les recherche toutes par retour arrière (backtrack).

3. Résolution (voir aussi [session](#))

?- gp(G,e) , p(P,e). % 1er démenti

→ gp(X,Z) :- p(X,Y) , p(Y,Z). % appel de la 1ère clause en "gp"

 % gp(G,e) et gp(X,Z) s'unifient: $G \leftarrow X$ et $Z \leftarrow e$

 % gp(G,e) est remplacé par $p(X,Y)$, $p(Y,e)$.

?- p(X,Y) , p(Y,e) , p(P,e). % 2ème démenti

→ p(m,j). % appel de la 1ère clause en "p"

 % p(X,Y) et p(m,j) s'unifient: $X \leftarrow m$ et $Y \leftarrow j$

 % p(m,j) est un fait, on ne substitue rien à p(X,Y) qui disparaît.

?- p(j,e) , p(P,e). % 3ème démenti

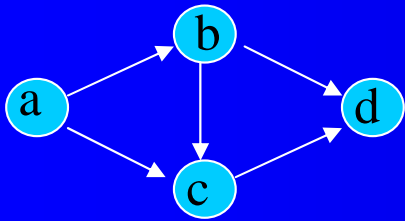
Résolution de problèmes

```
→ p(j,e).          % unification triviale: égalité et disparition de p(j,e)
?- p(P,e).         % 4ème démenti
→ p(j,e).         % P ← j
?- □                % disparition de p(j,e), rien ne reste, c-à-d false
← succès_1          % succès: succès_1: G ← X, X ← m, P ← j
                    % pas d'autre appel possible que p(j,e) pour p(P,e)
← succès_1          % pas d'autre appel possible pour p(j,e)
→ p(j,e).         % appel de p(j,e) pour p(X,Y), X ← j, Y ← e
?- p(e,e) , p(P,e). % aucun appel possible pour p(e,e)
← échec             % échec pour la seule tentative d'appel de p(j,e)
← succès_1          % pas d'autre appel possible pour p(X,Y), 1 seul succès trouvé
                    % pas d'autre appel possible pour gp(G,e): fin avec ce seul succès.
```

Programmation logique

1. La Récursivité

Exemple 1: On note $\text{arc}(X,Y)$ le fait qu'il y ait un arc orienté de X vers Y . On note $\text{chem}(X,Y)$ le fait qu'il y ait un chemin orienté entre X et Y . Le programme est (voir [session](#)):



<code>arc(a,b).</code>	<code>chem(X,Y):-</code>
<code>arc(a,c).</code>	<code>arc(X,Y).</code>
<code>arc(b,c).</code>	<code>chem(X,Y):-</code>
<code>arc(b,d).</code>	<code>arc(X,I),</code>
<code>arc(c,d).</code>	<code>chem(I,Y).</code>

Il y a un chemin entre X et Y si il y a un arc entre ces nœuds ou bien s'il y a un arc entre X et un nœud intermédiaire I et un chemin entre I et Y .

Programmation logique

Exemple 2:

La définition mathématique récursive de la fonction factorielle est:

factorielle(0)=1.

pour tout entier n supérieur à 0,

factorielle(n)=n*factorielle(n-1).

La fonction factorielle est une relation binaire notée fact(X,Y). Le programme est:

```
fact(0,1).
```

```
fact(X,Y) :- X > 0, R is X-1, fact(R,S), Y is X*S.
```

(voir [session](#))

Programmation logique

Les listes en SWI-Prolog sont:

- la liste vide [].
- la liste énumérative [a, 0, f(_), X] des éléments a, 0, f(_), X.
- la liste générique [X|Y] d'élément de tête X et de queue de liste Y.
- la liste mixte [a, 0, f(_), X|Y] commençant par les éléments a, 0, f(_), X et de queue de liste Y.

Exemple 3:

L'appartenance à une liste peut être définie langagièremment:

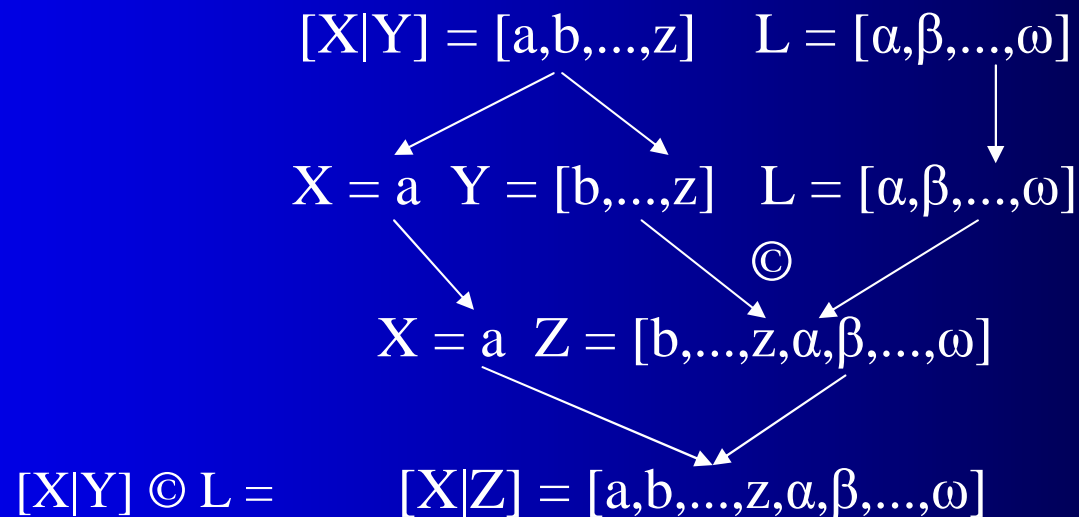
- C'est un fait qu'un élément appartient à une liste qui commence par cet élément.
- Si un élément appartient à une liste alors il appartient à cette liste précédée de n'importe quel élément.

Programmation logique

Le programme est (voir [session](#)):

```
element(X,[X|_]).  
element(X,[_|L]):- element(X,L).
```

Exemple 4: La concaténation \odot de 2 listes peut se définir en supposant le problème résolu dans un cas plus simple. Résolution graphique:



Programmation logique

L'argument est similaire à celui de l'exemple 3. Si l'on sait concaténer Y à L, ce qui donne Z, alors on sait concaténer à L la liste Y précédée de n'importe quel élément X. Cela donne [X|Z].

Le programme est:

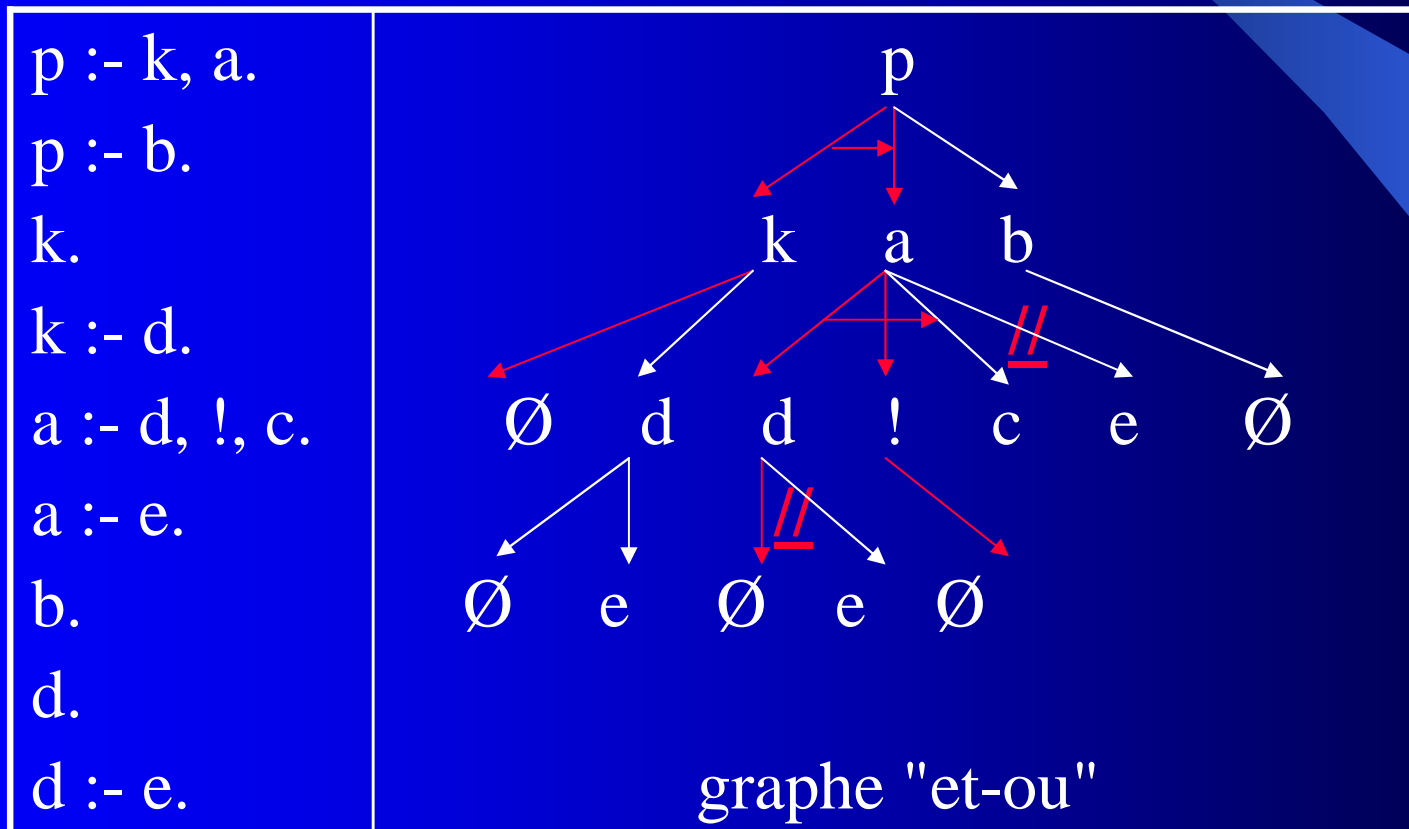
```
concat([], L, L).  
concat([X|Y], L, [X|Z]):-  
    concat(Y,L,Z).
```

N.B.: A force d'appels récursifs (sur le 1er argument de concat) de [X|Y] à Y, il arrivera que ce 1er argument soit vide. Il faut donc traiter ce cas. (voir [session](#))

Programmation logique

2. Le contrôle

On limite la recherche des solutions à la première d'entre elles grâce au méta prédicat cut (!).



Programmation logique

A la question "est-ce que p?" Prolog se comporte ainsi

```
→ p                % 1er cas de p défini par: p:-k,a
  → k              % 1er sous but k de p, 1er cas de k défini par: k.
  ← k succès      % k est un fait avéré, k disparaît
1  → a            % 2ème sous but a de p, 1er cas de a défini par: a:-d,!,c
    → d          % 1er sous but d de a, 1er cas de d défini par: d.
    ← d succès   % d est un fait avéré, d disparaît
    → !          % 2ème sous but ! de a (appel n'apparaissant pas en SWI-Prolog)
    ← ! succès   % ! est toujours avéré, il supprime le backtrack sur d (cut 1)
                    % et sur a (cut 2)
    → c          % 3ème sous but c de a, c non défini
    ← c échec    % c ne peut être résolu, en principe backtrack sur les frères
                    % précédents de c et sur le père de c: pas de backtrack sur !
    → d          % pas d'appel du 2ème cas de d défini par d:-e
    ....         % à cause du cut (cut 1), disparition de toute la partie
    ← ....       % correspondante
← a échec        % a ne peut être résolu (or en principe backtrack sur a)
```

Programmation logique

```
→ a          % pas d'appel du 2ème cas de a défini par: a:-e
...          % à cause du cut (cut 2) et disparition
2 ← échec    % de toute la partie correspondante
→ k          % backtrack sur k, 2ème cas de k défini par: k:-d.
  → d        % 1er sous but d de k
  ← d succès % 1er cas, d est un fait avéré
← k succès   % donc k est résolu
1' → a.      % k disparaît, appel du 2ème sous but a de p
...         % et reproduction à l'identique (entre 1' et 2') de toute
2' ← a échec % la partie concernant l'appel à "a" (de 1 à 2)
  → d        % backtrack sur d du 2ème cas de k défini par: k:-d.
    → e      % 2ème cas de d défini par: d:-e
    ← e échec % e ne peut être résolu, backtrack sur d puis sur k puis sur p
  ← d échec  % d ne peut être résolu, backtrack sur k puis sur p
← k échec    % k ne peut être résolu, backtrack sur p
→ b          % 2ème cas de p défini par: p:-b
← b succès   % b est un fait avéré, b disparaît
← p succès   % p est résolu
```

Programmation logique

En résumé:

Le cut empêche tout backtrack sur ses frères qui le précèdent dans la clause où il apparaît ainsi que sur son père (pas de backtrack sur "d" ni sur "a", voir // du graphe et-ou précédent). Le backtrack reste actif pour tout le reste. (voir [session](#))

MCours.com

3. L'unification

Quelques opérateurs en rapport avec l'égalité:

$=$, $\backslash=$, $==$, $\backslash==$, $=@=$, $\backslash=@=$

Utiliser `help(opérateur)` sous Prolog pour les opérateurs numériques:

is , $==:$, $=\backslash=$

Programmation logique

L'opérateur = ressemble à une affectation

?- $X = a$.

$X = a$

Yes

?- $X = a, Y = [b], Z = [X|Y]$.

$X = a$

$Y = [b]$

$Z = [a, b]$

Yes

Mais c'est plus que cela

?- $p(a, f(X), g(Z)) = p(Y, f(Y), W)$.

$X = a$

$Z = _G159$

$Y = a$

$W = g(_G159)$

Yes

Programmation logique

?- $p(X,Z,g(f(Y))) = p(Y,f(Y),W)$.

$X = _G157$

$Z = f(_G157)$

$Y = _G157$

$W = g(f(_G157))$

Yes

?- $p(a,f(X),g(Z)) = p(X,Z,g(f(Y))), p(X,Z,g(f(Y))) = p(Y,f(Y),W)$.

$X = a$

$Z = f(a)$

$Y = a$

$W = g(f(a))$

Yes

L'opérateur = est donc l'unification et \= l'impossibilité à toute unification

Programmation logique

?- X \= a.

No % c'est un échec car il n'est pas impossible d'unifier X et a, il suffirait que X capture a

?- f(X) \= a. % c'est un succès car il est impossible que f(X) s'unifie à a, et ce pour n'importe
% quelle valeur de X

X = _G157

Yes

L'opérateur == est le partage d'une valeur "syntaxiquement" identique

?- [a] == [a]. % même valeur de part et d'autre

Yes

?- a == b. % pas même valeur de part et d'autre

No

?- X == a. % X n'a pas de valeur, aucune valeur commune n'est partagée

No

Programmation logique

?- X == Y.

% X et Y n'ont pas de valeur

No

?- X == X.

% quelle que soit la valeur de X, il la partage avec X

X = _G157

Yes

?- X = a, Y = a, X == Y.

% X et Y ont une valeur et c'est la même

X = a

Y = a

Yes

L'opérateur \== est sa négation

?- X \== a.

X = _G157

Yes

Programmation logique

?- X = a, X \== b.

X = a

Yes

L'opérateur =@= est l'identité de structure ou de signature (de nom pour les ctes)

?- a =@= a. % même signature (= nom)

Yes

?- a =@= b. % pas même signature (= nom)

No

?- X =@= a. % une variable et une constante n'ont pas même signature

No

?- X =@= Y. % 2 variables ont même signature (si non liées)

X = _G157 % attention: ?- X=a,X=@=Y. retourne un échec

Y = _G158

Yes

Programmation logique

```
?- [A,[B,C]] =@= [D,[E,F]].    % mêmes structures
```

```
A = _G157
```

```
B = _G160
```

```
C = _G163
```

```
D = _G169
```

```
E = _G172
```

```
F = _G175
```

```
Yes
```

```
?- [a,[b,c]] =@= [d,[e,f]].    % les listes ont mêmes structures mais pas les éléments constitutifs
```

```
No
```

L'opérateur $\backslash=@=$ est sa négation

```
?- a \=@= b.
```

```
Yes
```

Programmation logique

4. La négation

Il faut d'abord comprendre les échecs à la résolution d'un but en Prolog.

Certains dialectes de Prolog échouent sur des buts dont les prédicats ne sont pas définis (cas des versions antérieures de SWI-Prolog).

Il y a alors échec quand il ne trouve aucun moyen de résoudre le but.

La négation en Prolog fonctionne de cette façon. Elle est définie par l'échec.

Not est un méta-prédicat puisque son argument n'est pas un terme mais une expression prédicative (Not est un prédicat d'ordre 2).

Programmation logique

```
1: not(B) :-  
    B,           % si B réussit  
    !,          % on ne procèdera à aucun backtrack: ni sur B ni sur not (de la clause n°2)  
    fail.       % on retourne un échec (nom de prédicat réservé pour l'échec forcé)  
2: not(_).      % si B échoue, not(B) de la clause n°1 échoue, il y a backtrack (car le cut  
                % de la clause n°1 n'est pas atteint), la clause n°2 est appelée et elle réussit
```

Exemple:

```
?- not(member(a,[b,c,d])).
```

Yes

```
?- not(member(a,[b,a,c])).
```

No

```
?- not(member(X,[b,c,d])).           % B = member(X,[b,c,d]) qui réussit (car  $\exists X \in [b,c,d]$ )
```

No

Comparaison avec d'autres langages

1. Langages procéduraux

Une clause de la forme $A :- B_1, \dots, B_n$. peut s'assimiler à une procédure

```
procedure A( $\vec{x}$ )  
begin  
    call B1( $\vec{x}$ ),  
    ...  
    call Bn( $\vec{x}$ ),  
end
```

à la différence que les variables de \vec{x} ne peuvent pas être réaffectées: une variable pointe vers un individu et non pas vers un emplacement mémoire.

Comparaison avec d'autres langages

2. Langages applicatifs

Comme Lisp:

D'un point de vue conceptuel à " $y = f(x)$ " correspond " $f(x,y)$ "

```
(de fact(x)
(cond((zerop x)1)
      (t (times(x,fact(sub1 x))))))
```

```
? (setq x 3)(setq y (fact x))
```

6

```
fact(X,1):- zerop(X),!.
fact(X,Y):- sub1(X,X1),
             fact(X1,X2),
             times(X,X2,Y).
```

```
?- X is 3, fact(X,Y).
```

X=3

Y=6

Programmation logique en SWI - Prolog

3. Références bibliographiques

Pour des détails sur SWI-Prolog: <http://www.swiprolog.org/>

Voir aussi le support de cours de 2ème année Informatique de l'ENSEEIHHT intitulé "Programmation logique"

http://www.enseeiht.fr/lima/ia/Ens/2I_N7/Programmation_Logique/

MCours.com