Introduction à Prolog

http://www.depinfo.enseeiht.fr/N7/IA/IMAdpt/2IN/ProgLog

Lien avec la démonstration automatique

La programmation logique est particulièrement adaptée à la résolution de (toutes sortes de) problèmes.

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(=m) est-il le grand père d'Eric(=e)?"

Lien avec la démonstration automatique

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

$c1: gp(X,Z) \vee \neg p(X,Y) \vee \neg p(Y,Z)$ $c2: p(m,j)$ $c3: p(j,e)$

On y ajoute le contraire de la conclusion $c0: \neg gp(m,e)$

On cherche une réfutation de $\{c0,c1,c2,c3,c4\}$ en utilisant une variante de:

Trouver $n \in N$ tel que $\square \in S_n$ où

$S_0 = \{c0,c1,c2,c3\}$

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

Lien avec la démonstration automatique

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,j).      % les arguments peuvent être des variables ou des constantes
p(j,e).      % ou des expressions fonctionnelles (i.e.: des termes)
```

Les règles sont de la forme:

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

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

Lien avec la démonstration automatique

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

Lien avec la démonstration automatique

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 ← X et Z ← 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 ← m et Y ← 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

Lien avec la démonstration automatique

→ p(j,e). % unification triviale: égalité et disparition de p(j,e) (prog)

?- 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)

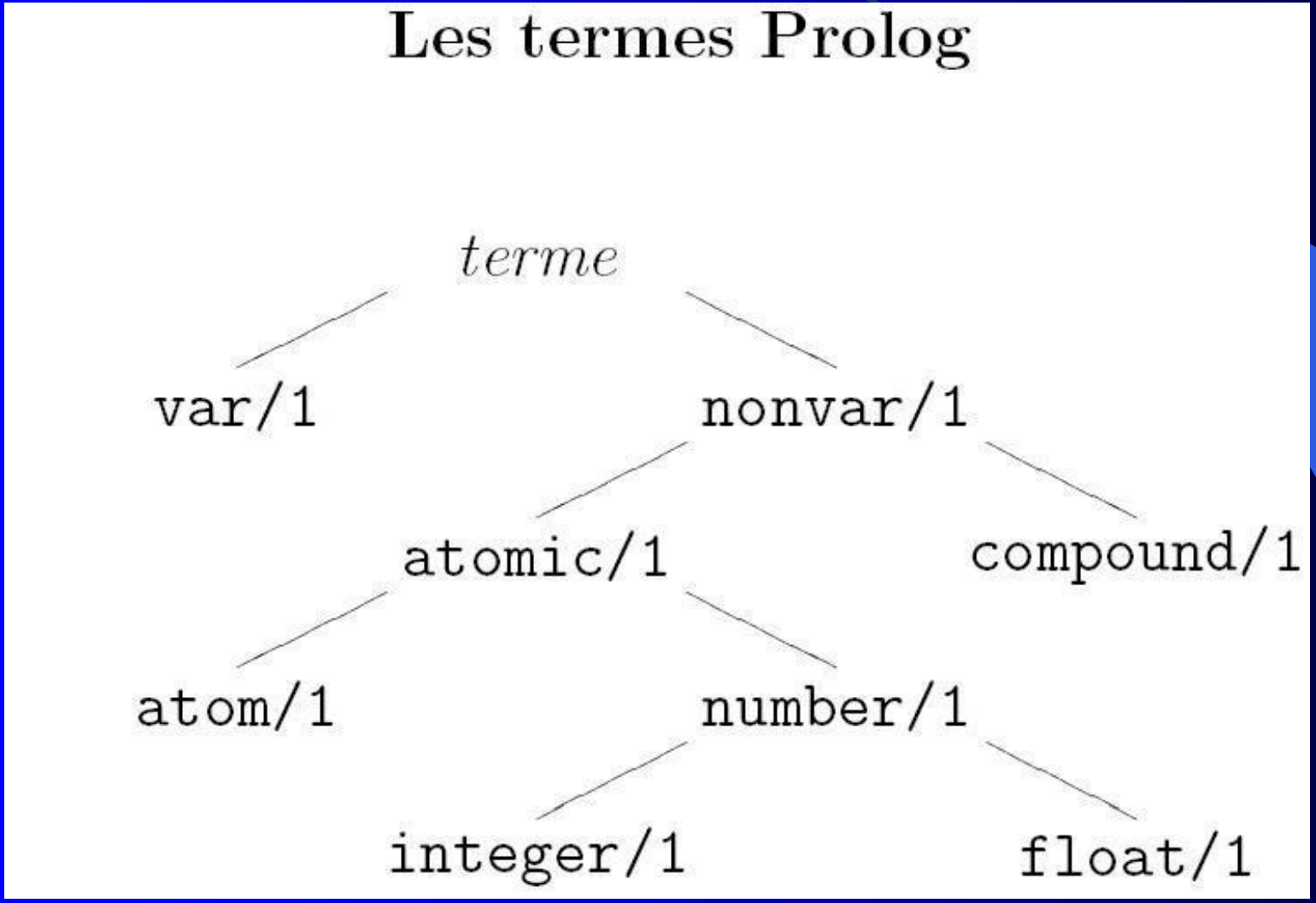
← échec % échec pour la seule tentative d'appel de p(j,e)

← 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.

Le langage Prolog

1. Les éléments principaux du langage Prolog: les termes



Le langage Prolog

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

atome

identificateur	atome 'quoté'
[a-z][a-zA-Z_0-9]*	'([^]|(''))*'
atome	'ATOME'
bonjour	'ca va ?!'
c_est_ca	'c''est ca'

* signifie un nombre quelconque de fois, ^ signifie n'importe quel caractère sauf: '

nombre

entier	réel
[0-9]+	([0-9]+.[0-9]+)
0 012 12034	0.0 01.10 12.34

Le langage Prolog

termes composés

- Un terme composé `date(25,mai,1988)` est constitué
  - d'un atome (`date`)
  - d'une suite de termes (`(25,mai,1988)`)

    le nombre d'arguments (3) est appelé arité
- Le couple atome/arité (`date/3`) est appelé foncteur

  ou opérateur du terme composé correspondant.

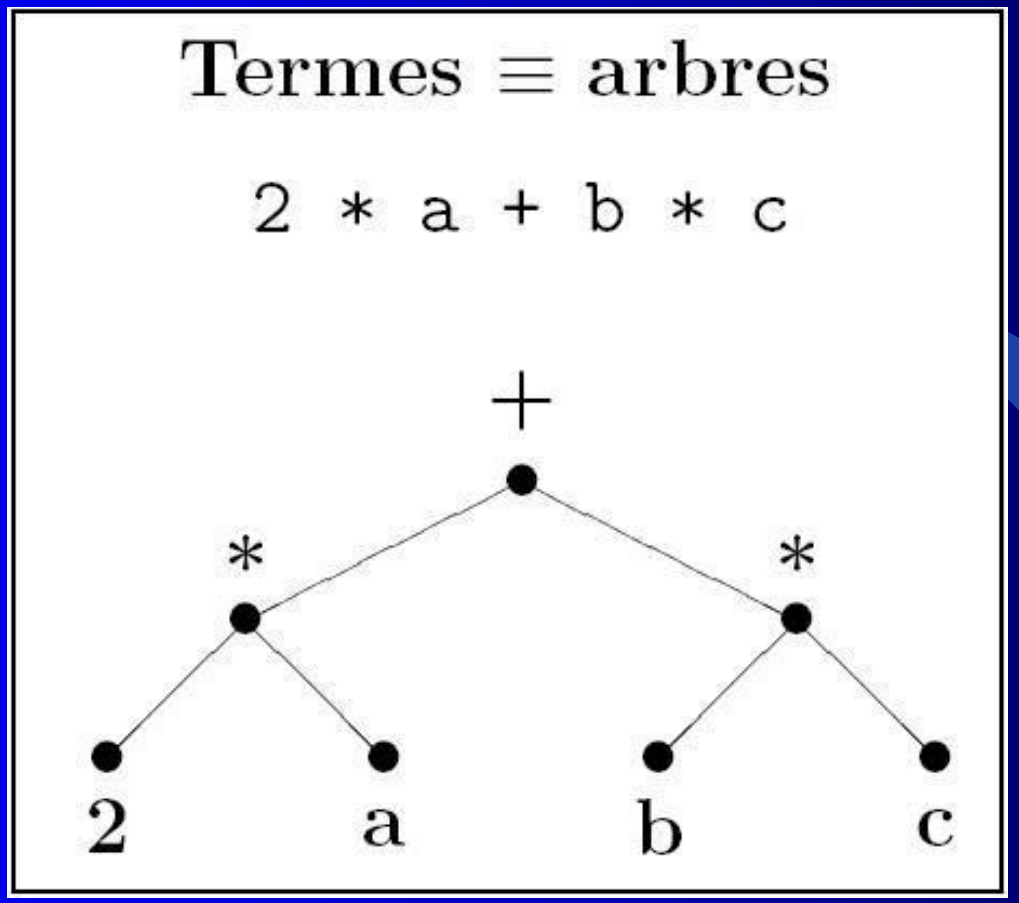
Le langage Prolog

Exemples de termes composés

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

Terme composé ≡ Structure de données

Le langage Prolog

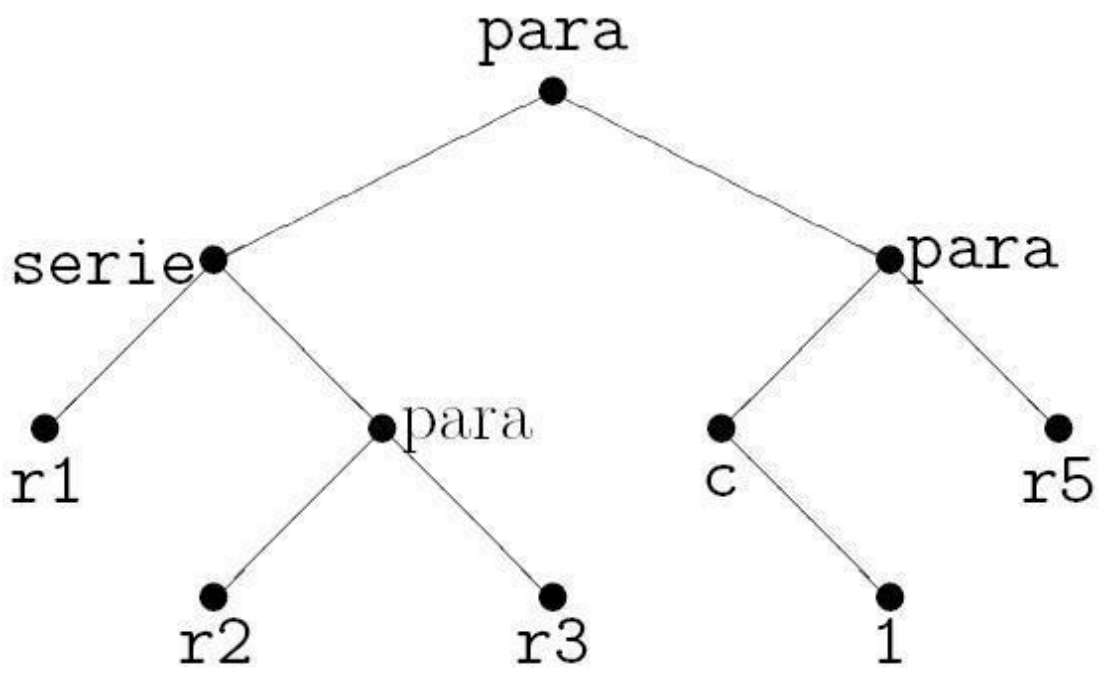


Le tableau précédent donne l'opérateur principal + (de priorité 500), l'opérateur suivant est * (de priorité 400), etc.

Le langage Prolog

Terme ≡ arbre

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



Le langage Prolog

2. Prédicats de test du type d'un terme

Il existe des prédicats qui permettent de connaître le type d'un terme.

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

```
1 ?- X=X, var(X).
X = _G263
Yes
2 ?- X=3, var(X).
No
```

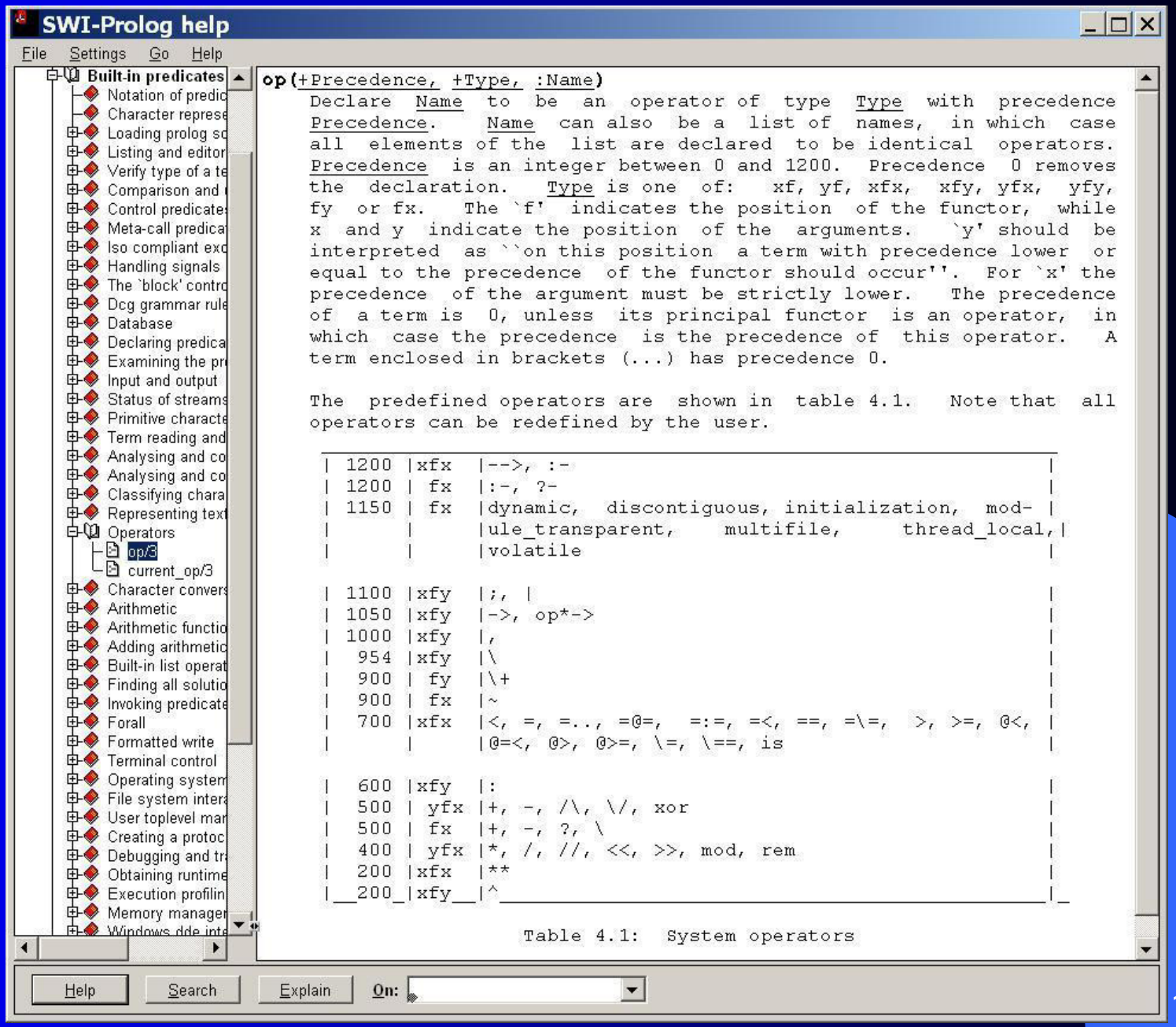
C'est la valeur de T qui est testée. Prolog renomme les variables.

Le langage Prolog

3. Les opérateurs

Il existe des opérateurs prédéfinis en Prolog.

Ces opérateurs peuvent être préfixés, postfixés ou infixés par rapport à leurs arguments.



Le langage Prolog

Un opérateur en Prolog permet de connaître l'opérateur principal d'un terme ainsi que ses arguments, c'est: =..

```
1 ?- X = 2*a+b*c**3, X =.. L.

X = 2*a+b*c**3
L = [+, 2*a, b*c**3]

Yes
2 ?- Y = b*c**3, Y =.. L.

Y = b*c**3
L = [*, b, c**3]

Yes
```

Inspection de termes

accès à la structure interne des termes

`arg(N, T, X)`	`X` est le $N^{ième}$ argument du terme `T`
`functor(T, A, N)`	`T` est le terme de foncteur `A/N`

Le langage Prolog

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`.

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

L'expression :- op(P,A,Op) est une déclaration qu'un nouvel opérateur Op sera utilisé dans un programme Prolog. Il est ainsi défini syntaxiquement. Il n'a aucune sémantique, ce n'est pas comme + (par exemple) qui dans certaines expressions réalise une addition.

Le langage Prolog

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` ≡ `a + (b * c)` ≡ `+(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.

Le langage Prolog

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	

Un opérateur binaire d'associative gauche (comme +) signifie que s'il apparaît plusieurs fois dans une expression, ses arguments sont emboîtés sur la gauche.
Exemple:
a + b + c + d + e = (((a + b) + c) + d) + e

1 ?- X = a+b+c, X =.. L.

X = a+b+c
L = [+, a+b, c]

Yes

Le langage Prolog

4. 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
yes
```

Le prédicat d'affectation d'une valeur à celle d'une expression arithmétique n'est pas = (qui réalise une unification) mais is.

Le langage Prolog

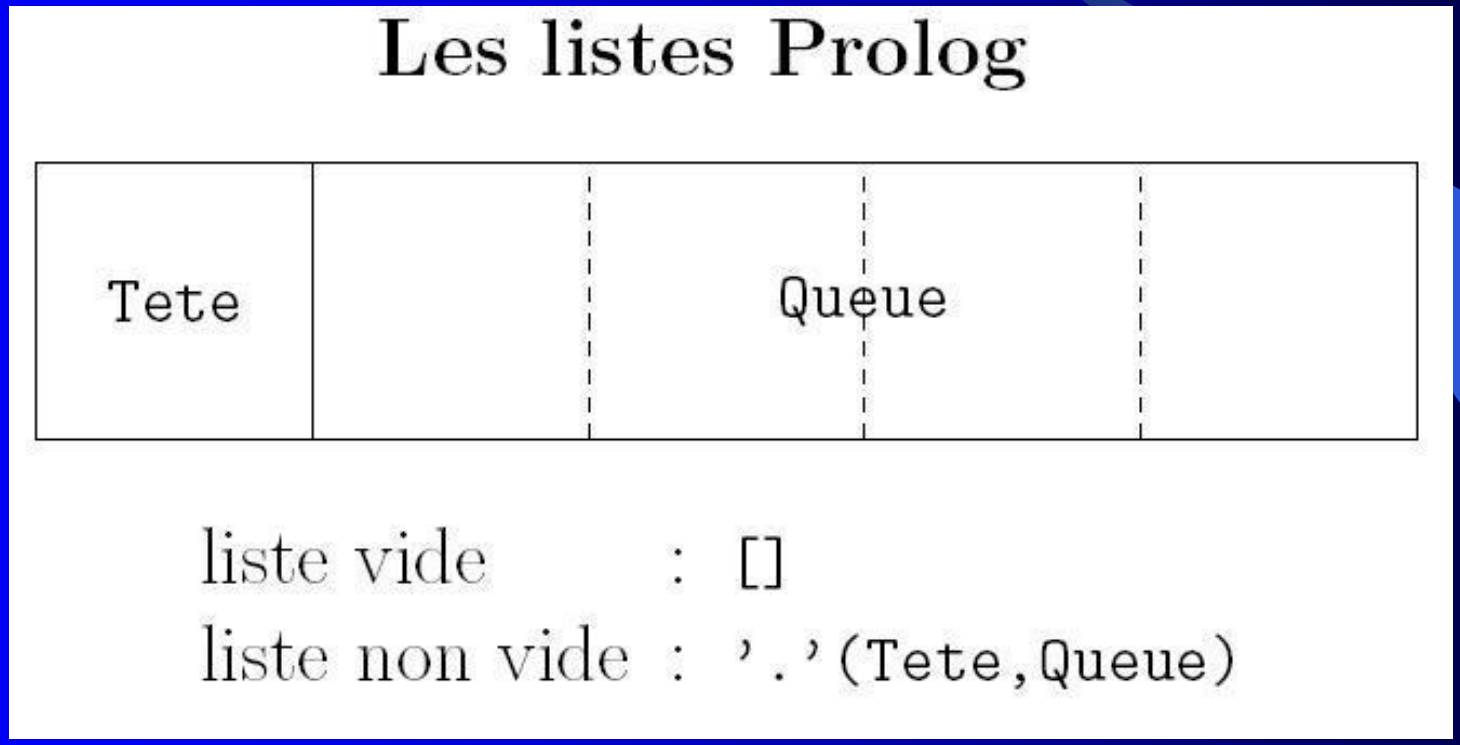
Comparaisons arithmétiques

$e_1 < e_2$	`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`

Le langage Prolog

5. Les listes en Prolog

Les listes sont des termes construits à partir de l'opérateur de construction '.'.



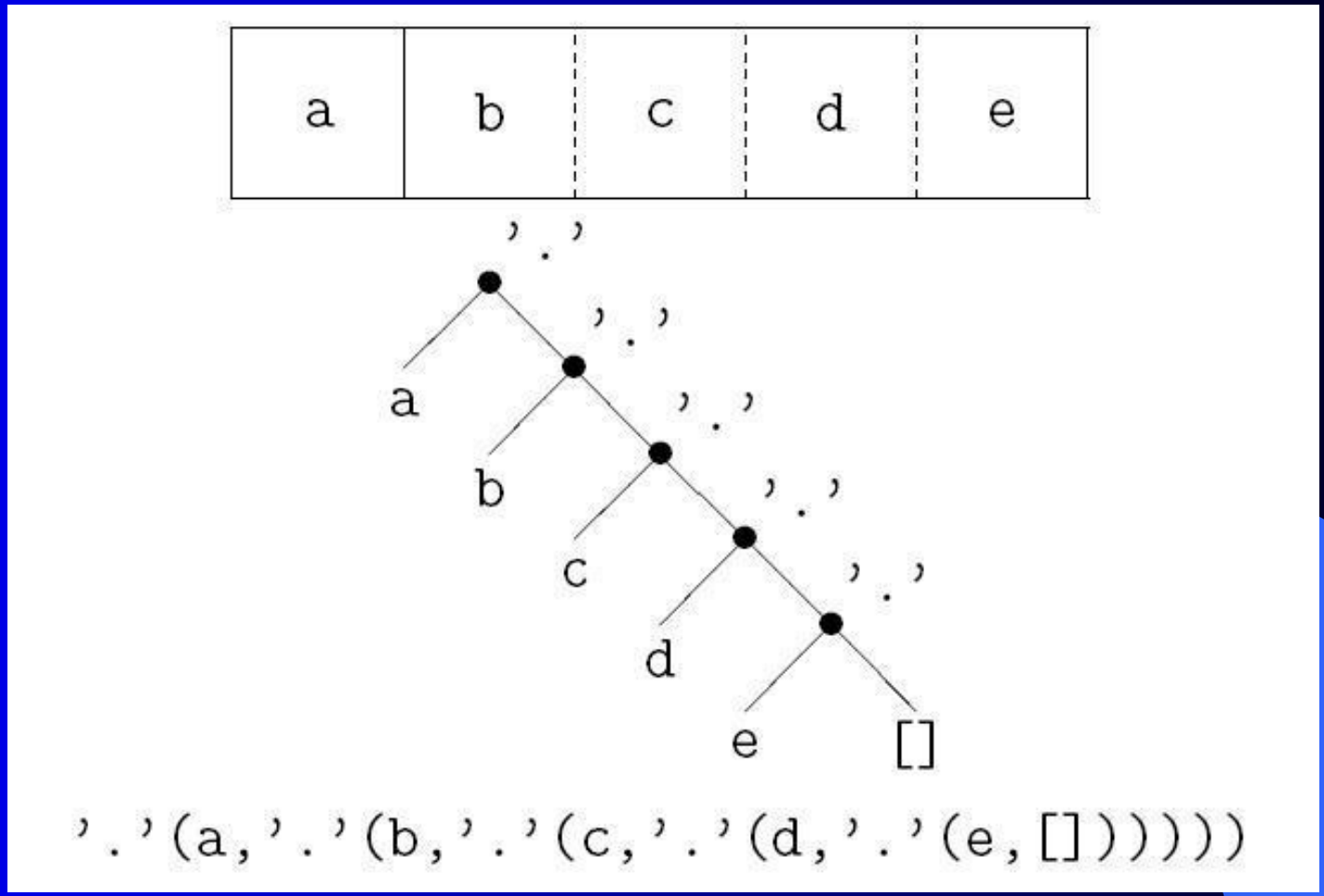
Le langage Prolog

Liste qu'en Prolog on écrit [a,b,c,d,e].

1 ?- X = [a,b,c,d], X =.. L.

X = [a, b, c, d]
L = ['.', a, [b, c, d]]

Yes



Le langage Prolog

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

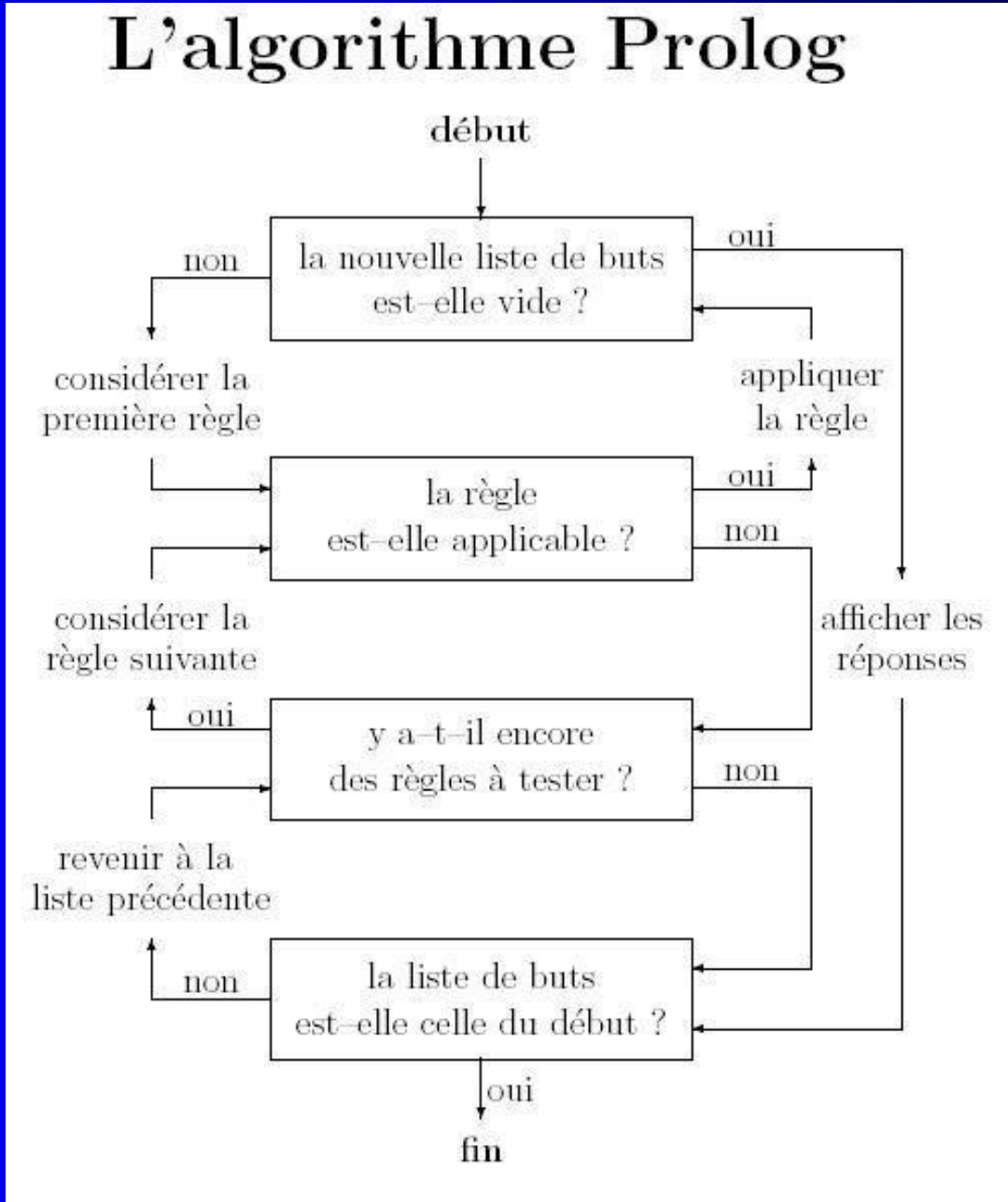
- $T_1, T_2, \ldots\ , T_n$ représente les n $(n > 0)$ premiers termes de la liste
- *Reste* représente la liste des éléments restants
- on a l'équivalence $[T_1, T_2, \ldots\ , T_n] \equiv [T_1, T_2, \ldots\ , T_n | []]$

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]
```

Le langage Prolog

6. Algorithme



Le langage 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.

Le langage Prolog

Effacer un but

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

$\Downarrow$

règle : t :- $q_1, q_2, \ldots, q_m$

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

$\Downarrow$

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

$\Downarrow$

$\vdots$

$\Downarrow$

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

Tout repose donc sur l'unification du littéral de tête t d'une règle et du 1er but courant b_1 ainsi que de leurs arguments.

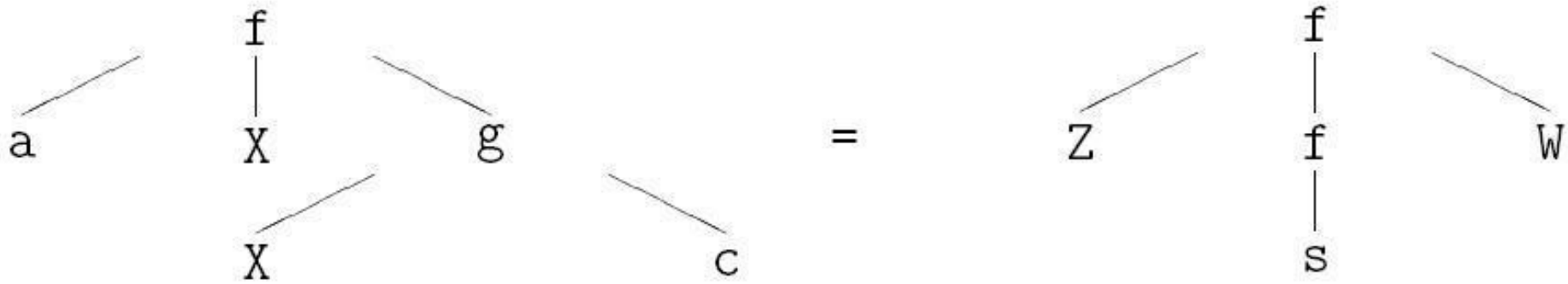
Le langage Prolog

7. Unification

Unification de termes

mise en correspondance d'arbres syntaxiques

```
Terme1 = Terme2
```



Le langage Prolog

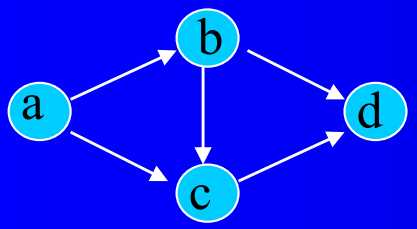
Unification Prolog

```
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).
```

Programmation logique

1. La Récursivité

Exemple 1: On note arc(X,Y) le fait qu'il y ait un arc orienté de X vers Y. On note chem(X,Y) le fait qu'il y ait un chemin orienté entre X etY. Le programme est (voir session):




```
arc(a,b).
arc(a,c).
arc(b,c).
arc(b,d).
arc(c,d).
```

```
chem(X,Y):-
        arc(X,Y).
chem(X,Y):-
        arc(X,I),
        chem(I,Y).
```

Il y a un chemin entre X et Y si il y a un arc entre ces noeuds ou bien s'il y a un arc entre X et un noeud 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

Exemple 3:

L'appartenance à une liste peut être définie langagièrement:

- C'est un fait qu'un élément appartient à une liste qui commence par cet élément.

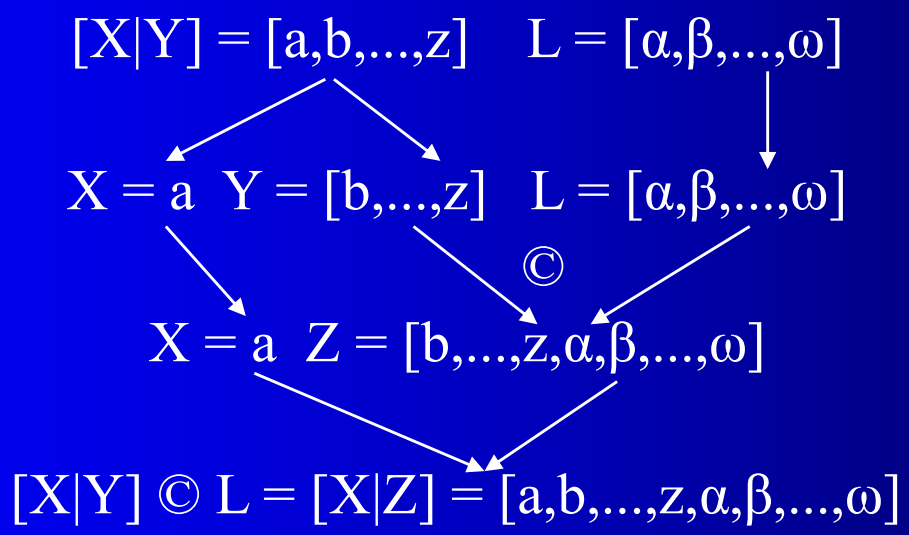
-Si un élément appartient à une liste alors il appartient à cette liste précédée de n'importe quel élément.

Le programme est (voir session):

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

Programmation logique

Exemple 4: La concaténation © 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

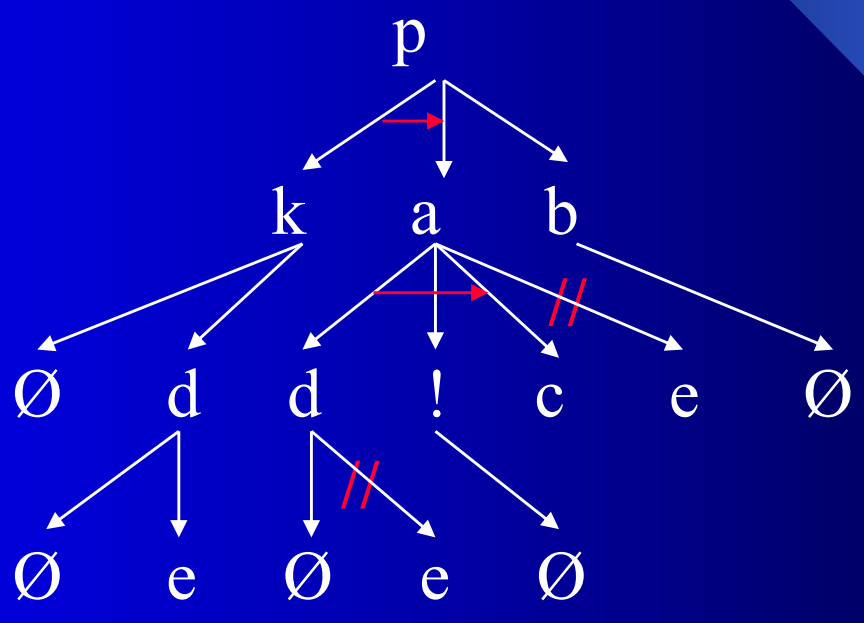
Récursivité terminale ou non terminale

```
                                                    % lgr1(N,L)
                                                % processus récursif
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,[]).
```

Programmation logique

2. Le contrôle

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

p :- k, a. p :- b. k. k :- d. a :- d, !, c. a :- e. b. d. d :- e.	 hypergraphe "et-ou"

Programmation logique

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

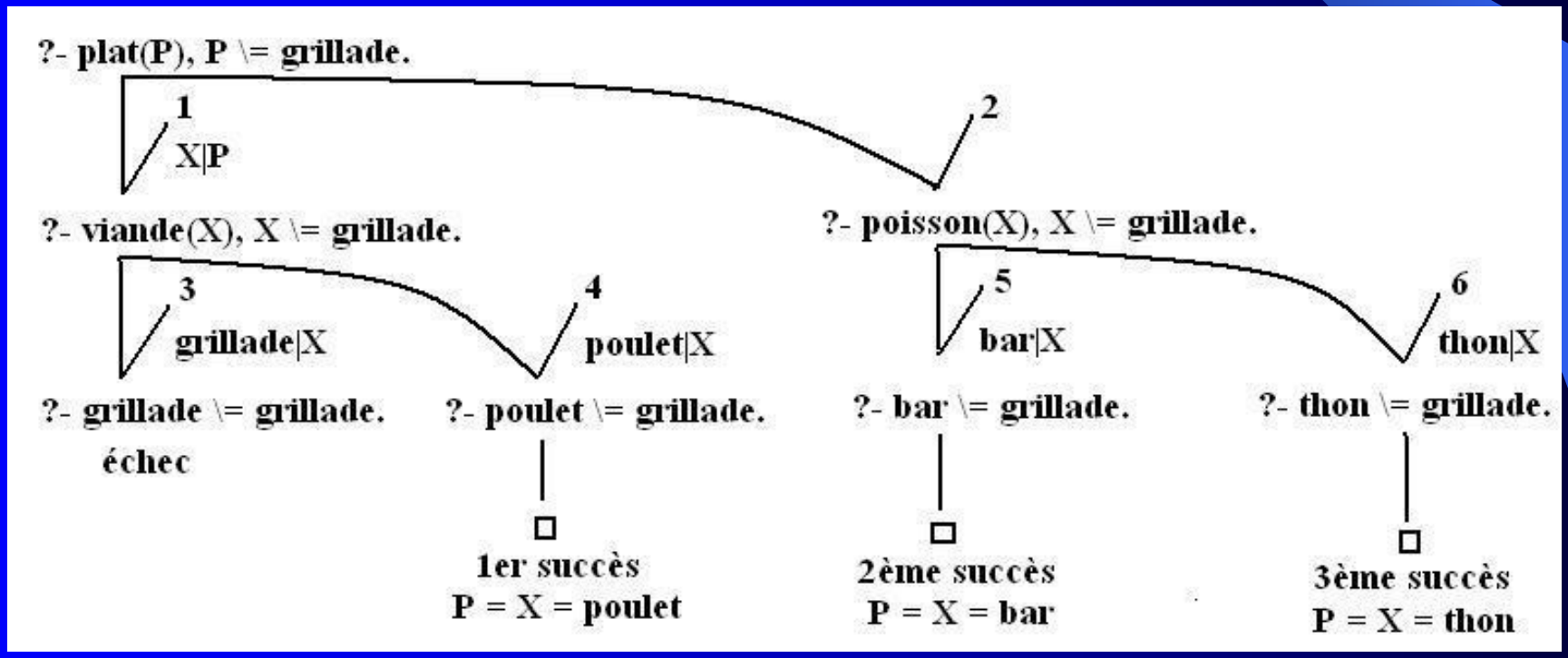
```
→ p                 % 1er cas de p défini par: p:-k,a
    → k             % 1er sous but k de p, 1er cas de k défini par: k.
    ← k succès      % k est un fait avéré, k disparaît
1   → a             % 2ème sous but a de p, 1er cas de a défini par: a:-d,!,c
        → d         % 1er sous but d de a, 1er cas de d défini par: d.
        ← d succès  % d est un fait avéré, d disparaît
        → !         % 2ème sous but ! de a (appel n'apparaîssant 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

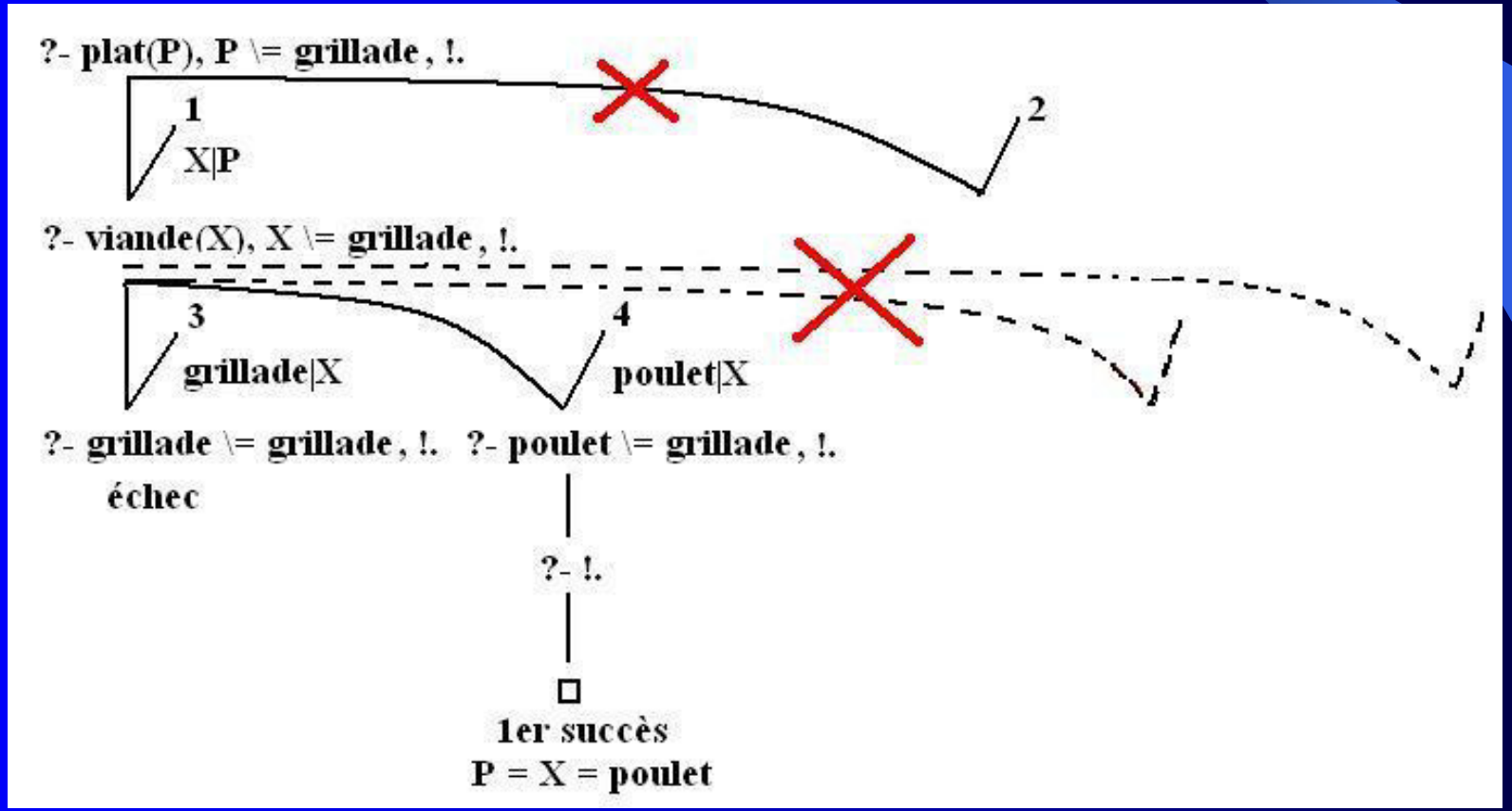
Programmation logique

Soit le programme:	Sous Prolog on pose le but:
1: plat(X) :- viande(X). 2: plat(X) :- poisson(X). 3: viande(grillade). 4: viande(poulet). 5: poisson(bar). 6: poisson(thon).	?- plat(P), P \= grillade.



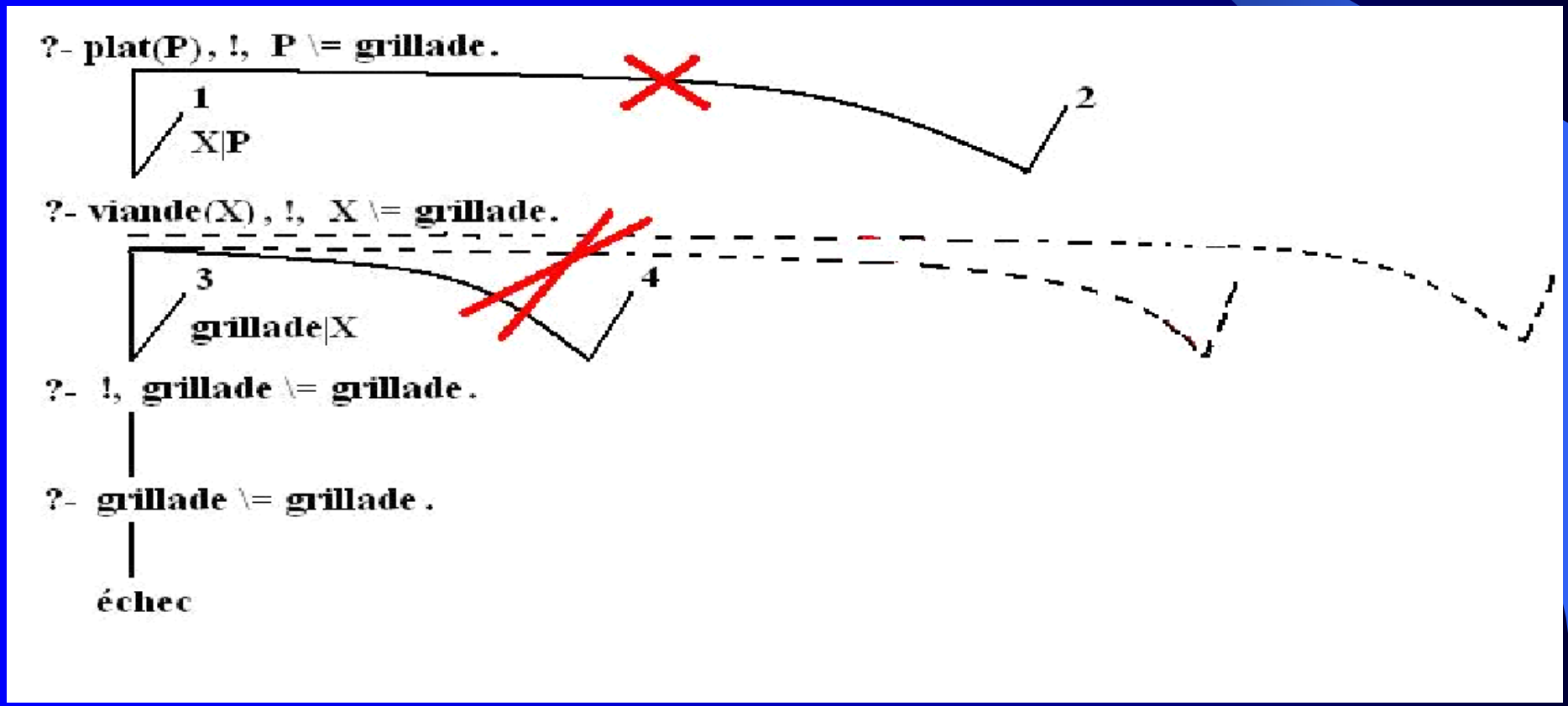
Programmation logique

Soit le programme:	Sous Prolog on pose le but:
1: plat(X) :- viande(X). 2: plat(X) :- poisson(X). 3: viande(grillade). 4: viande(poulet). 5: poisson(bar). 6: poisson(thon).	?- plat(P), P \= grillade, !.



Programmation logique

Soit le programme:	Sous Prolog on pose le but:
1: plat(X) :- viande(X). 2: plat(X) :- poisson(X). 3: viande(grillade). 4: viande(poulet). 5: poisson(bar). 6: poisson(thon).	?- plat(P), !, P \= grillade.



Programmation logique

Interprétation:

?- plat(P), P \= grillade.

on cherche tous les plats P différents d'une grillade
on trouve poulet, bar, thon

?- plat(P), P \= grillade, !.

on cherche le premier plat différents d'une grillade
on trouve poulet

?- plat(P), !, P \= grillade.

on cherche si le premier plat est différent d'une grillade
on ne trouve pas car c'est une grillade

Programmation logique

Effacer un but

$\{x_1, \ldots, x_r\}\ [b_1, b_2, \ldots, b_n]\ \ (a_1,\ldots,a_p)$

$\Downarrow$

règle : t :- $q_1, q_2, \ldots, q_i, !, \ldots, q_m$

unification : $t = b_1$, avec substitution : $\{x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_p}\}$

$\Downarrow$

$\{x_1, \ldots, x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_p}, \ldots, x_r, \ldots\}[q_1, q_2, \ldots, q_i, !, \ldots, q_m, b_2, \ldots, b_n]\ (a_1,\ldots,a_p, b_1)$

$\Downarrow$

$\vdots$

$\{x_1, \ldots, x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_p}, \ldots, x_r, \ldots\}[!, \ldots, q_m, b_2, \ldots, b_n]\ (a_1,\ldots,a_p, b_1, q_1, q_2, \ldots, q_i)$

$\Downarrow$

$\{x_1, \ldots, x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_p}, \ldots, x_r, \ldots\}[q_{i+1}, \ldots, q_m, b_2, \ldots, b_n]\ (a_1,\ldots,a_p)$

$\Downarrow$

$\vdots$

$\{x_1, \ldots, x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_p}, \ldots, x_r, \ldots, \ldots\}[b_2, \ldots, b_n]\ (a_1,\ldots,a_p, q_{i+1}, \ldots, q_m)$

$\Downarrow$

$\vdots$

$\{x_1/t_1, x_2/t_2, \ldots, x_r/t_r\}\ [\,]\ (a_1,\ldots,a_p, q_{i+1}, \ldots, q_m, b_2, \ldots, b_n)$

Programmation logique

Les x_i (i=1,r) sont les variables, apparaissant dans la conjonction de buts $b_1,...,b_n$, dont on cherche les valeurs.

En résolvant b_1 grâce à t_1:-$q_1,...,q_i,!,...,q_m$ le nombre de variable augmente de celles propres aux q_j, tandis que certaines autres (les x_{jk}) sont instanciées suite à l'unification entre b_1 et t_1.

Les a_i (i=1,p) sont les buts sur lesquels on procèdera à un retour arrière (backtrack). Cette liste s'augmente progressivement de tous les buts de la conjonction de buts et de leurs sous-buts.

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. Dans l'exemple de résolution propositionnelle (page 39), il n'y a pas de backtrack sur "d" ni sur "a", voir // du graphe et-ou précédent). Le backtrack reste actif pour tout le reste. (voir session)

3. Différents opérateurs d'unification

Quelques opérateurs en rapport avec l'égalité:

= , \= , == , \== , =@= , \=@=

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

is , =:= , =\=

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 c[tes])

```
?- 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 \=@= 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(But) réussit si But échoue, Not(But) échoue sinon.

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 (pas appel à 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])).          % But = member(X,[b,c,d]) qui réussit (car ∃X ∈ [b,c,d])
No
```

Comparaison avec d'autres langages

1. Langages procéduraux

Une clause de la forme A :- B1,…,Bn. peut s'assimiler à une procédure

```
procedure   A( x⃗ )
begin
        call   B1( x⃗ ),
        …
        call   Bn( x⃗ ),
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

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

Et bien sûr SWI–Prolog: http://www.swiprolog.org/