Le langage Prolog

Rédaction : Jacques Tisseau, Ecole Nationale d'Ingénieurs de Brest

Révision : Fred Mesnard, université de la Réunion

– 1991/2006 –

Table des matières

Quel système Prolog utiliser ?

ISO-Prolog : norme définie en 1995

SWI-Prolog :

- `http://www.swi-prolog.org/`
- sur-ensemble ISO, LGPL, multi-plateforme
- librairies, environnement de développement
- communauté d'utilisateurs, mise à jour très fréquente

CIAO-Prolog :

- `http://www.clip.dia.fi.upm.es/Software/Ciao/`

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 apprentissage (ILP), Prolog et contraintes (CP)
- Constraint Handling Rules, Mercury

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 ..... *l'alphabet du langage*

inconnues	$x,\ y,\ z$
constantes fonctionnelles	$f,\ g,\ h$
constantes prédicatives	$P,\ Q,\ R$
connecteurs	$\neg,\ \wedge,\ \vee,\ \Rightarrow,\ \Leftrightarrow$
quantificateurs	$\forall,\ \exists$
ponctuation	$(\ )\ ,$

La logique des prédicats

Syntaxe ..... *les mots et les phrases du langage*

forme prédicative	$P(t_1, \ldots, t_n)\ n \geq 0$
terme (t_i)	inconnue ou forme fonctionnelle
forme fonctionnelle	$f(t_1, \ldots, t_m)\ m \geq 0$
formule $(A, B, \ldots)$	forme prédicative $(A), \neg A, A \wedge B, A \vee B,$ $A \Rightarrow B, A \Leftrightarrow B$ $\forall x\ A, \exists x\ A$

La logique des prédicats

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

$\mathcal{D}$ domaine de définition des termes
A formule logique

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

Formes clausales

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

1. élimination des $\Rightarrow$ et $\Leftrightarrow$
2. déplacement des $\neg$ vers les feuilles
3. déplacement des $\forall$ et $\exists$ vers la racine (“forme prénexe”)
4. élimination des $\exists$ (“skolémisation”)
5. distribution de $\wedge$ sur $\vee$
6. mise sous forme clausale

Formes clausales

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

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. $\forall x \exists y([p(y) \wedge \neg r(y,x)] \vee q(x))$
4. $\forall x([p(f(x)) \wedge \neg r(f(x),x)] \vee q(x))$
5. $(p(f(x)) \vee q(x)) \wedge (\neg r(f(x),x) \vee q(x))$
6. d'où les 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 ≡ **Pro**grammation en **log**ique

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) :- r(Z,X), v(Z).`

Un programme Prolog est un ensemble de clauses.

Mon premier interpréteur Prolog

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

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

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

demontrer(toujours_vrai).
```

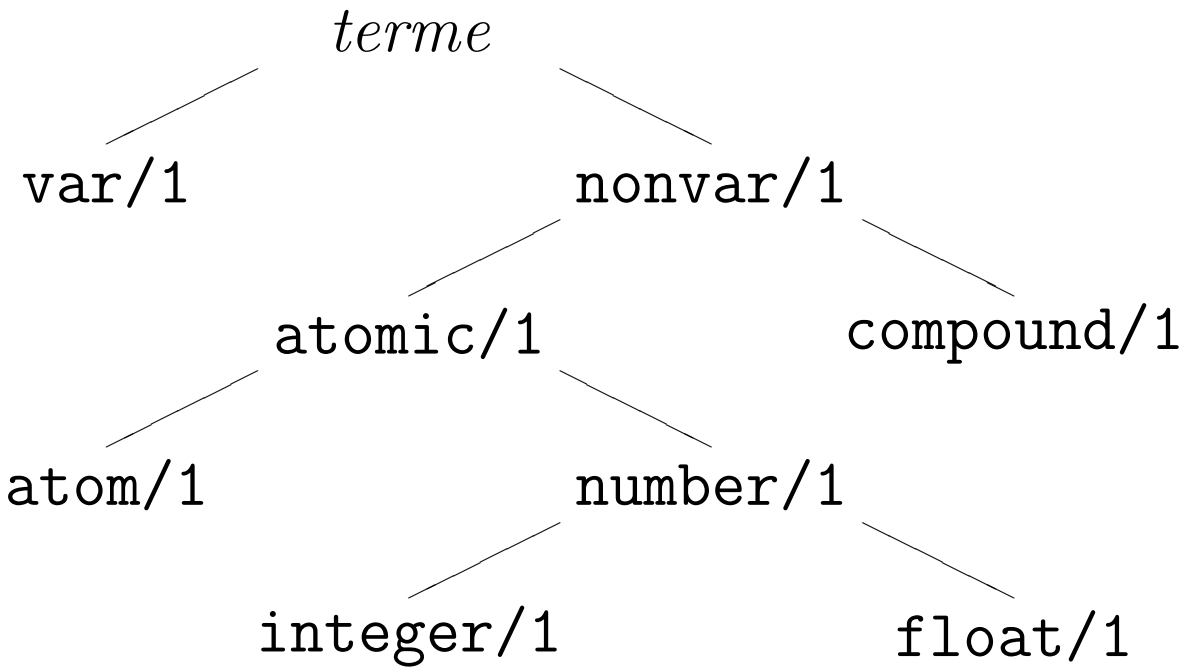
Mon premier programme

```
% Un exemple de règles généalogiques

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

% Les parents de mes ancètres sont mes ancètres
((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 ancètres de Fabien ?
?- demontrer(Qui ancetre_de fabien).
Qui = jean;
Qui = emile;
no more solution

% De qui Emile est-il l'ancètre ?
?- 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

variable
`[_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 `[a-z][a-zA-Z_0-9]*`	**opérateur** `[+-*/^<>=~:.?#&@]+`	**atome 'quoté'** `'(([^'])|('')*'`
`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.

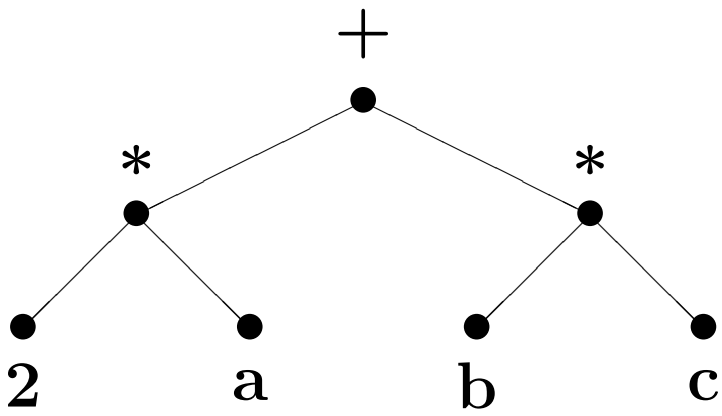
Exemples de termes composés

Foncteur	Terme composé
`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

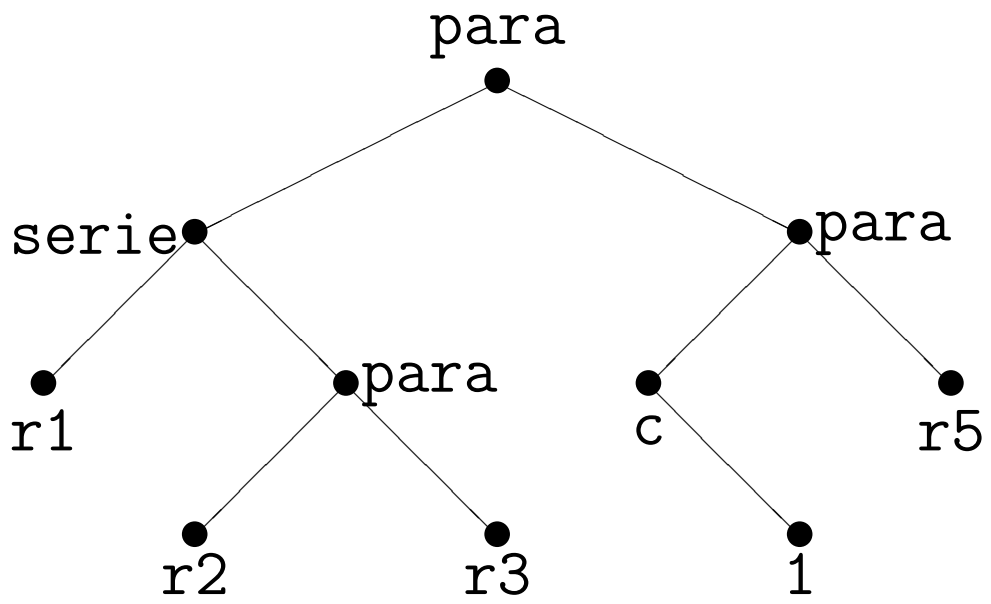
Termes ≡ arbres

```
2 * a + b * c
```



Termes ≡ arbres

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



Une fiche d'état civil

individu

état–civil

Nom	:	Martin
Prénom	:	Richard
Nationalité	:	française
Sexe	:	masculin

Date de naissance :

Jour	:	15
Mois	:	Février
Année	:	1960

adresse

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

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('Martin','Richard',francaise,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 :

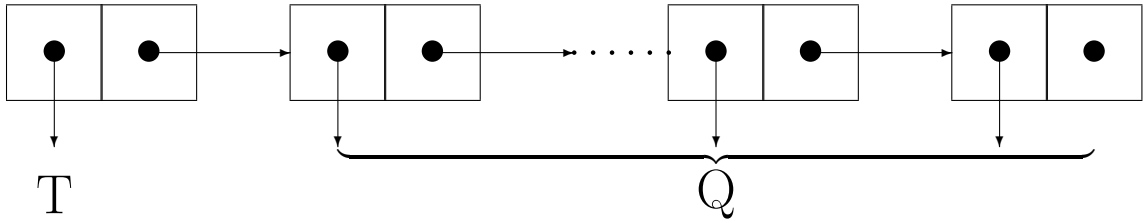
$$\forall x \in \mathcal{N}, x + z = x$$

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

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

Ma première liste

Une liste est une suite ordonnée de termes.



Nous verrons que Prolog dispose d'une syntaxe particulière pour les listes. En attendant :

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

Appartenance à une liste

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

```
?- element(X,l(a,l(b,l(c,vide)))).
X = a ;
X = b ;
X = c ;
No
```

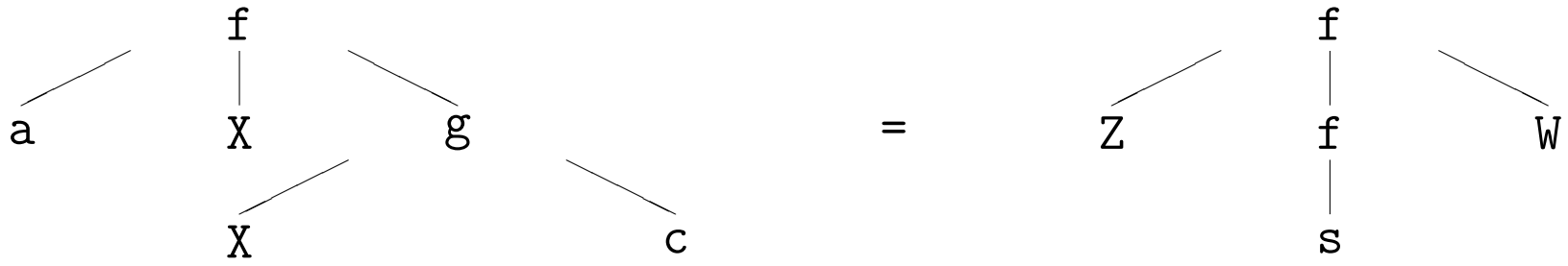
Classification des termes

`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 sur les termes

```
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 $\texttt{N}^{ième}$ argument du terme `T`
`functor(T, A, N)`	`T` est un terme de foncteur `A/N`

Termes de base

Un terme est dit de base si

1. c'est un terme atomique ou bien
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(0,T).
base(N,T) :- N > 0, arg(N,T,Arg), base(Arg),
             N1 is N-1, base(N1,T).
```

Les opérateurs en Prolog

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

en notation préfixée `~ X` $\equiv$ `~(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`	`\+ \+ x`
	non	`fx`	`- 4`
postfixée	oui	`yf`	
	non	`xf`	

Quelques opérateurs prédéfinis

priorité	opérateurs	associativité
1200	`:- -->`	`xfx`
1200	`:- ?-`	`fx`
1100	`';'`	`xfy`
1050	`->`	`xfy`
1000	`','`	`xfy`
900	`\+`	`fy`
700	`= \= =..  == \== is =:= =\=`	`xfx`
700	`< =< >= > @< @=< @>= @>`	`xfx`
500	`+ -`	`yfx`
400	`* /`	`yfx`

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
```

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`

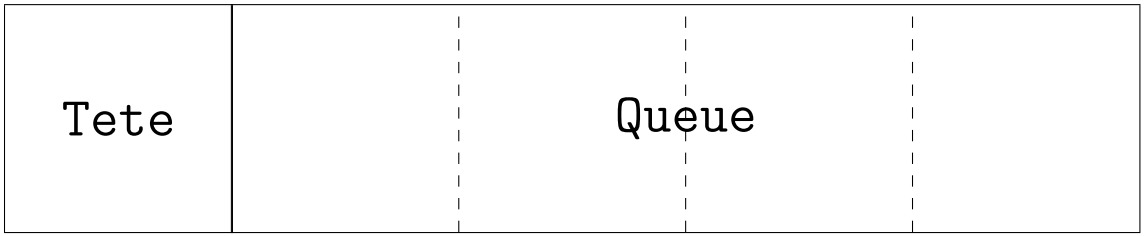
Exemples d'évaluations arithmétiques

Fonction d'Ackermann

$$
\begin{aligned}
f(0,n) &= n+1 \\
f(m,0) &= f(m-1,1) \text{ si } m>0 \\
f(m,n) &= f(m-1,f(m,n-1)) \text{ si } m>0, n>0
\end{aligned}
$$

```
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).
```

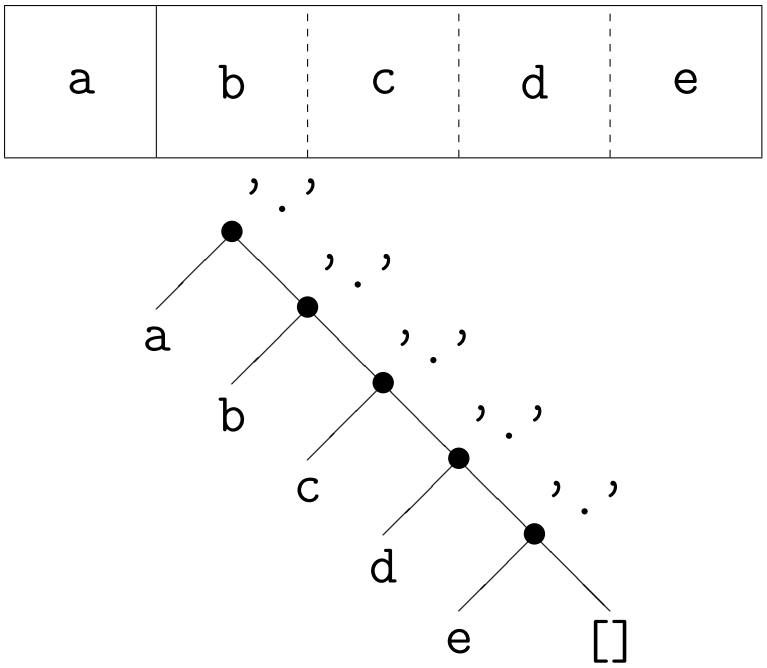
Les listes Prolog



liste vide : `[]`

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

Un exemple de liste en notation standard




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

Une autre syntaxe pour les listes 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]
```

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
```

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
```

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
```

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
?- conc([a,b],L,[a,b,c,d,e]).
L = [c,d,e] ;
No
```

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
```

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
?- inverser(L,[c,b,a]).
L = [a,b,c] ;
ERROR: Out of local stack
```

Une autre inversion de liste

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

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

```
?- inverser([a,b,c],L).
L = [c,b,a] ;
No
?- inverser(L,[a,b,c]).
ERROR: Out of local stack
```

Librairies

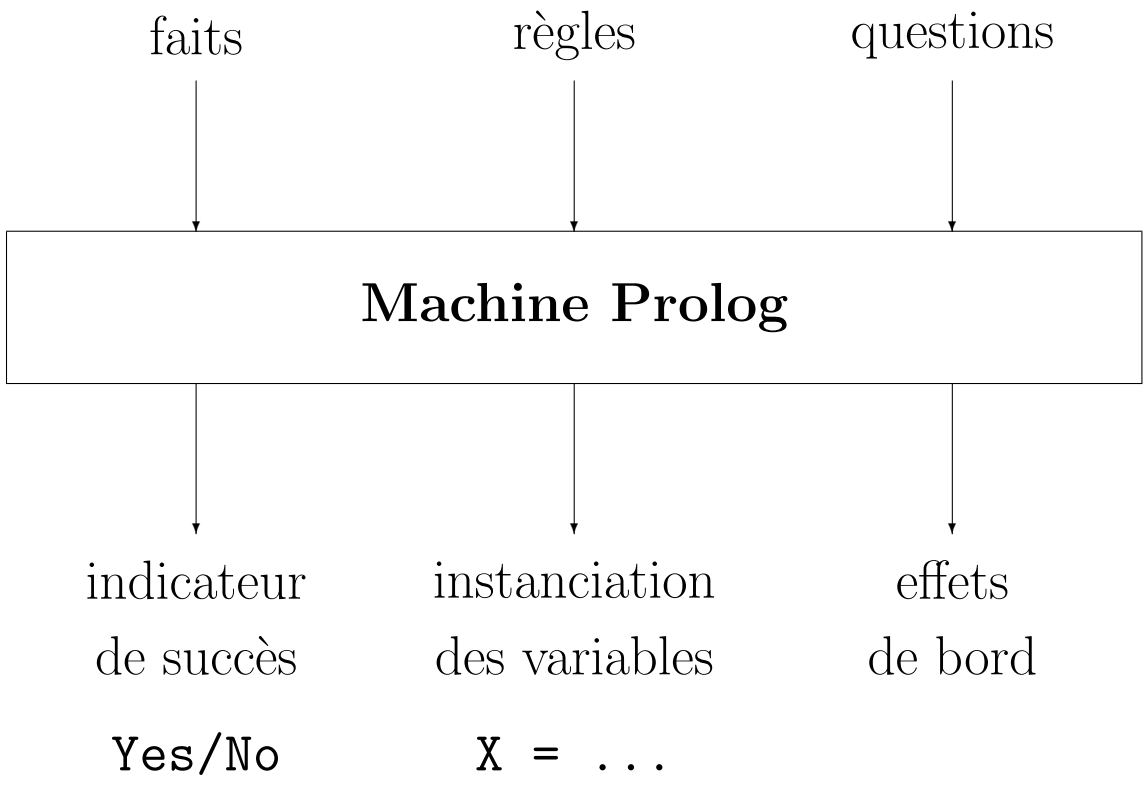
SWI-Prolog dispose de nombreuses librairies :

`http://gollem.science.uva.nl/SWI-Prolog/Manual/libpl.html`

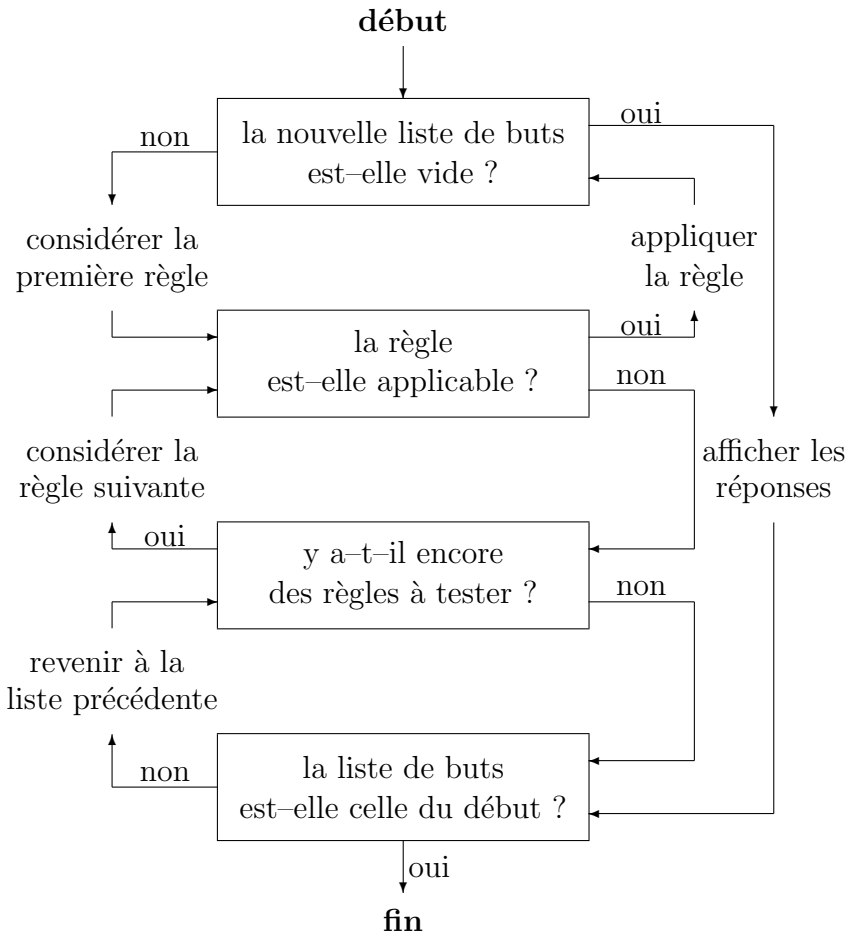
L'une d'elles est dédiée aux relations manipulant des listes :

`http://gollem.science.uva.nl/SWI-Prolog/Manual/lists.html`

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

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

$\Downarrow$

règle : t :- $q_1, q_2, \ldots, q_m$

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

$\Downarrow$

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

$\Downarrow$

$\vdots$

$\Downarrow$

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

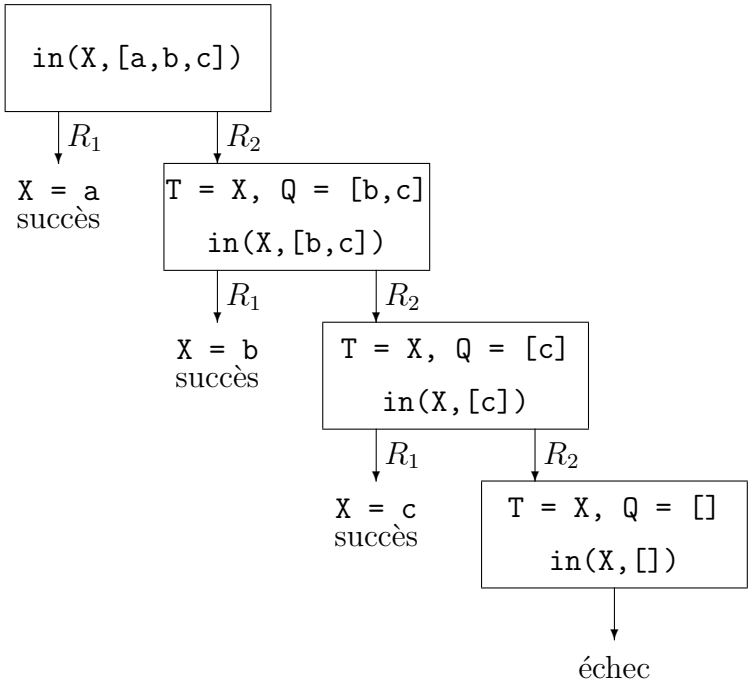
Unification Prolog

```
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(0,X,Y).
uArgs(N,X,Y) :- N > 0, uArg(N,X,Y), N1 is N-1, uArgs(N1,X,Y).

uArg(N,X,Y) :- arg(N,X,ArgX), arg(N,Y,ArgY), u(ArgX,ArgY).
```

Retour arrière

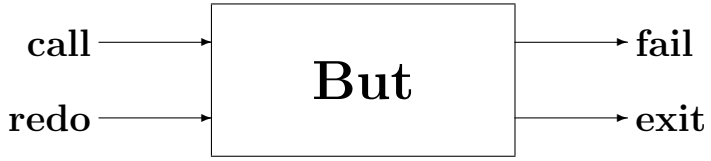
Arbre de résolution

```
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.



call	*appel initial du but*
exit	*sortie avec succès du but*
redo	*retour arrière sur un but*
fail	*échec du but initial*

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–buts 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

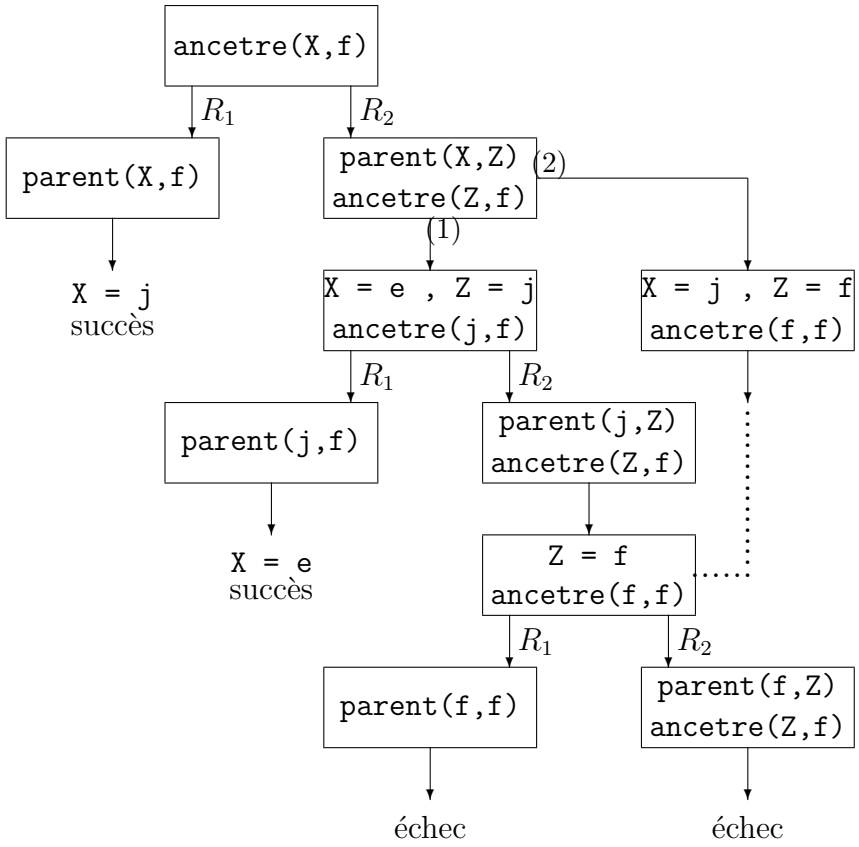
```
?- in(X,[a,b,c]).
1 call:  in(_192,[a,b,c]) >
1 exit:  in(a,[a,b,c])                          X = a;
1 redo:  in(_192,[a,b,c]) >
2 call:  in(_192,[b,c]) >
2 exit:  in(b,[b,c])
1 exit:  in(b,[a,b,c])                          X = b;
2 redo:  in(_192,[b,c]) >
3 call:  in(_192,[c]) >
3 exit:  in(c,[c])
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

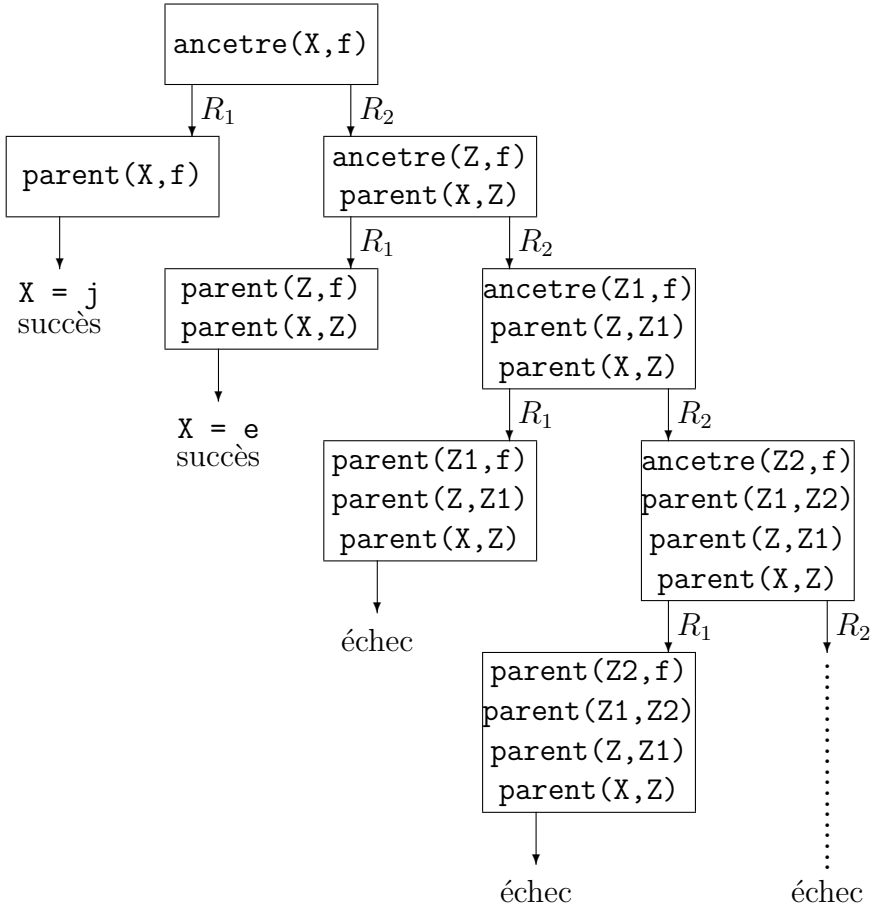


Ordonnancement des buts

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 ⇒

Récursivité terminale ou non terminale

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

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

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

Récursivité terminale ou non terminale

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

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

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

`!`	Réussit toujours mais annule tous les points de choix crées depuis l'appel du but parent.	*coupe–choix*
`call(But)`	Evalue **But**.	*interpréteur*
`fail`	Echoue toujours.	*échec*
`\+ But`	**But** n'est pas démontrable.	*négation par l'échec*
`repeat`	Réussit toujours même en cas de retour arrière.	*boucle infinie*
`true`	Réussit toujours.	*succès*

Le coupe–choix

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

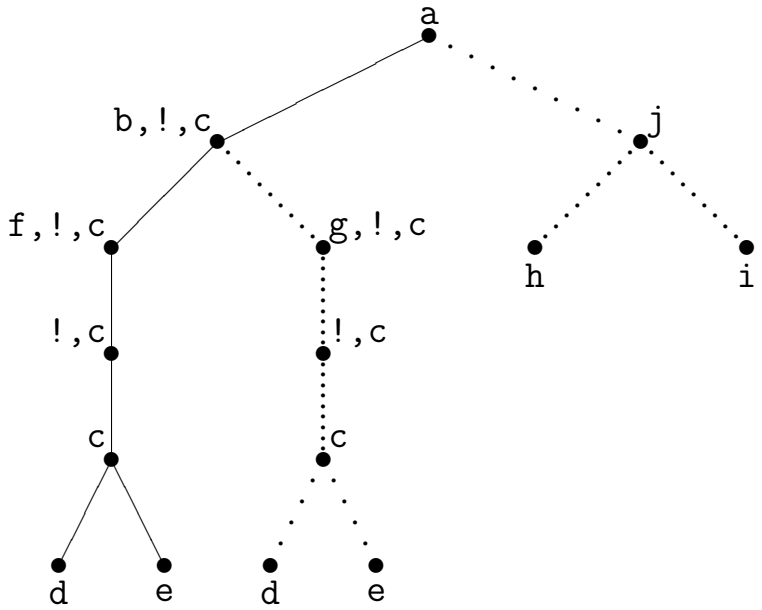
- une plus grande rapidité d'exécution,
- moins de place mémoire utilisée.

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
?- max(2,3,X).
X = 3 ;
No
?- max(3,2,2).
No
```

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

Alternative

```
max(X,Y,Z) :-
        ( X > Y
        ->    Z = X
        ;     Z = Y
        ).
```

Coupe–choix “rouge”

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

```
?- max(3,2,X).
X = 3 ;
No
?- max(2,3,X).
X = 3 ;
No
?- 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
?- ds(X).
X = 2 ;
No
```

La négation par l'échec

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

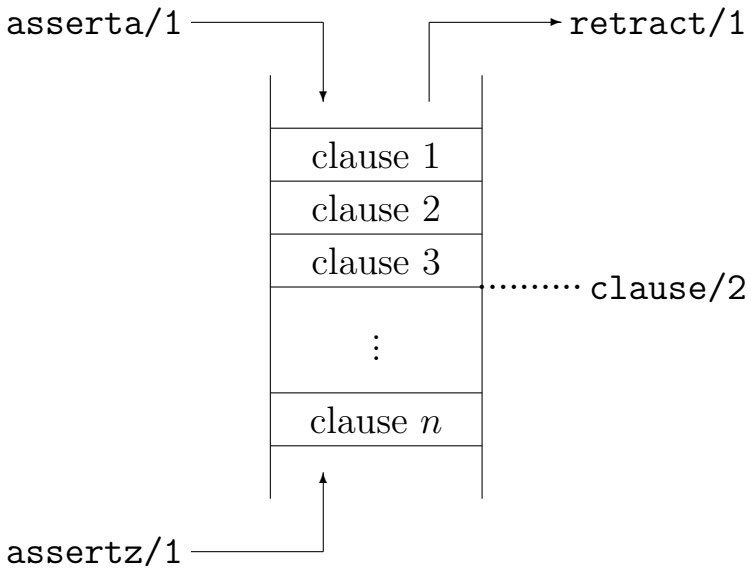
```
?- s(X).
X = 1 ; X = 2 ;
No
?- non(non(s(X))).
X = _192 ;
No
```

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



asserta (Clause) *empiler une clause*

assertz (Clause) *enfiler une clause*

clause (Tete,Corps) *inspecter une clause*

retract (Clause) *dépiler (défiler) une clause*

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

Prédicats prédéfinis ISO

Arithmétique

`Result is Expression`	*arithmetic evaluation*
`Expression1 =:= Expression2`	*arithmetic equal*
`Expression1 =\= Expression2`	*arithmetic not equal*
`Expression1 < Expression2`	*arithmetic less than*
`Expression1 =< Expression2`	*less than or equal*
`Expression1 > Expression2`	*arithmetic greater than*
`Expression1 >= Expression2`	*greater than or equal*

Type des termes

`atom(Term)`	*atom ?*
`atomic(Term)`	*atomic term ?*
`compound(Term)`	*compound term ?*
`float(Term)`	*float ?*
`integer(Term)`	*integer ?*
`number(Term)`	*number ?*
`nonvar(Term)`	*bounded variable ?*
`var(Term)`	*free variable ?*

Comparaison de termes

`Term1 == Term2`	*term identical*
`Term1 \== Term2`	*term not identical*
`Term1 @< Term2`	*term less than*
`Term1 @=< Term2`	*term less than or equal*
`Term1 @> Term2`	*term greater than*
`Term1 @>= Term2`	*term greater than or equal*

Manipulation de termes

`Term1 = Term2`	*unification*
`dif(Term1,Term2)`	*disunification* non-ISO
`arg(Integer,CompoundTerm,Term)`	*term inspection*
`functor(Term,Name,Arity)`	*term inspection*
`Term =.. List`	*term inspection*
`copy_term(Term1,Term2)`	*term duplication*

Contrôle de la résolution

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

Exceptions

`catch(Goal,Catcher,RecoverGoal)` *error catching*
`throw(Term)` *error throwing*

Ensembles de solutions

`bagof(Template,Goal,List)`	*bags of all solutions*
`findall(Template,Goal,List)`	*find all solutions*
`setof(Template,Goal,List)`	*sorted sets of all solutions*

Manipulation de clauses

`abolish(Functor)`	*clause destruction*
`asserta(Clause)`	*clause creation*
`assertz(Clause)`	*clause creation*
`clause(Head,Body)`	*clause information*
`current_predicate(Term)`	*clause retrieval*
`retract(Clause)`	*clause destruction*

Entrées/sorties de termes

`current_op(Precedence,Type,Name)`	*operator retrieval*
`op(Precedence,Type,Name)`	*operator definition*
`read(Term)`	*read a term*
`read(Stream,Term)`	*read a term from a stream*
`write(Term)`	*write a term*
`write(Stream,Term)`	*write a term to a stream*
`nl`	*output the new-line character*
`nl(Stream)`	*output the nlc to a stream*

Gestion des flots d'entrées/sorties

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

Bibliographie

Blackburn P., Bos J., Striegnitz K. *Learn Prolog Now!*, College Publications, 2006.

Bratko I. *Prolog Programming for Artificial Intelligence*, 3rd edition, Addison Wesley, 2000.

Clocksin F.W., Mellish C.S. *Programming in Prolog: Using the ISO Standard*, 5th edition, Springer, 2003.

Deransart P., Ed-Dbali A., Cervoni L. *Prolog: The Standard*, Springer, 1996.

O'Keefe R.A. *The Craft of Prolog*, MIT Press, 1990.

Sterling L., Shapiro E. *L'Art de Prolog*, Masson, 1997.