Programmation Logique

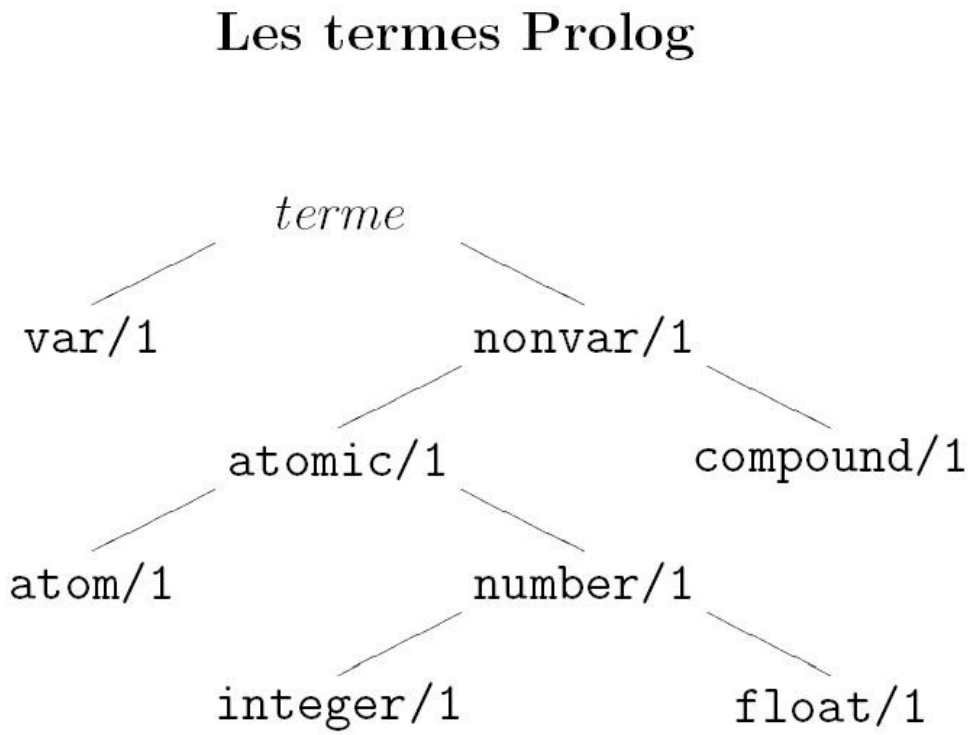
RAYAR Frédéric
frederic.rayar@univ-tours.fr

Année 2011-2012

Sommaire

Sommaire

Termes


Les termes Prolog
terme
var/1
nonvar/1
atomic/1
compound/1
atom/1
number/1
integer/1
float/1

Atomes

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: '

Nombres

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

Termes composés

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.

Termes composés

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

Variables

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

Variables anonymes

Variables anonymes :

Elles se comportent comme des variables ordinaires dans la recherche des correspondances, mais elles ne prennent jamais de valeurs et n'apparaissent pas dans les réponses.
Notation : _ (*underscore*)

Exemples : *?- livre('Hugo', _, _).*
yes

La base de fait contient effectivement au moins un livre écrit par Hugo.

Variables anonymes

Variables anonymes (suite et fin) :

Alors que : *?- livre('Hugo', X, Y).*

X = 'Hermani', Y = edition(gallimard, 1975) ;

X = 'Les misérables', Y = edition(poche, 1984) ;

no

?- livre(A, _, _).

A = 'Hugo' ;

A = 'Giannesini' ;

no

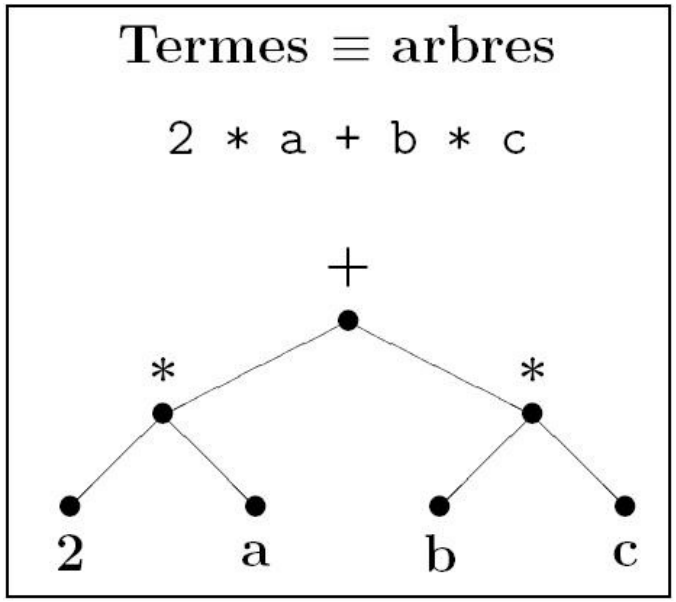
Remarque : différentes occurrences du trait de soulignement _ désignent des variables anonymes différentes.

Types 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

Arbres



Arbres : A vous !

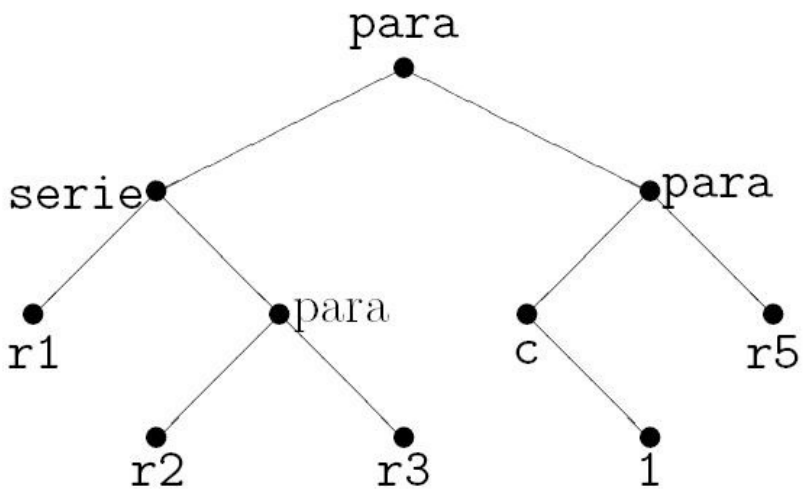
$$\textbf{Terme} \equiv \textbf{arbre}$$

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

Arbres : A vous !

Terme $\equiv$ arbre

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



Sommaire

Unification

`X is Expression`

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

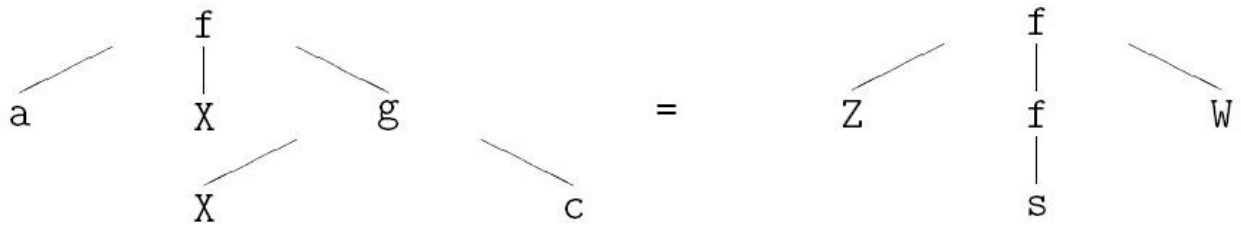
```
?- 8 = 5+3.                    % unification
no
?- 8 is 5+3                    % évaluation
yes
?- X is 5+3
X = 8
yes
```

Unification

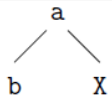
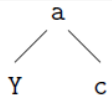
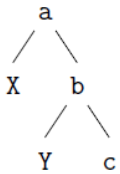
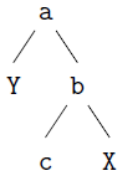
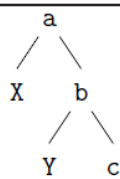
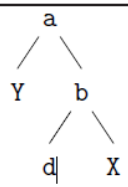
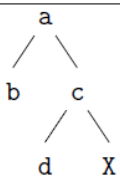
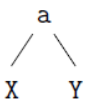
Unification de termes

mise en correspondance d'arbres syntaxiques

Terme1 = Terme2



Unification

Terme 1	Terme 2	?ok	substitution	commentaire
aa	aa	oui	{ }	deux termes atomiques sont unifiables s'ils sont identiques
aa	bb	non		
A	bb	oui	{A=bb}	si un des deux termes est une variable libre, l'unification est toujours possible
		oui	{X=c,Y=b}	
		oui	{X=c,Y=c}	les variables X et Y sont liées
		non	{X=Y, X=c, Y=d}	l'unification de X avec c nécessiterait l'unification des deux termes atomiques c et d.
		oui	{X=b, Y=c(d,b)}	on donne la valeur de Y en remplaçant X par sa valeur

Arbre de recherche

Arbre de recherche

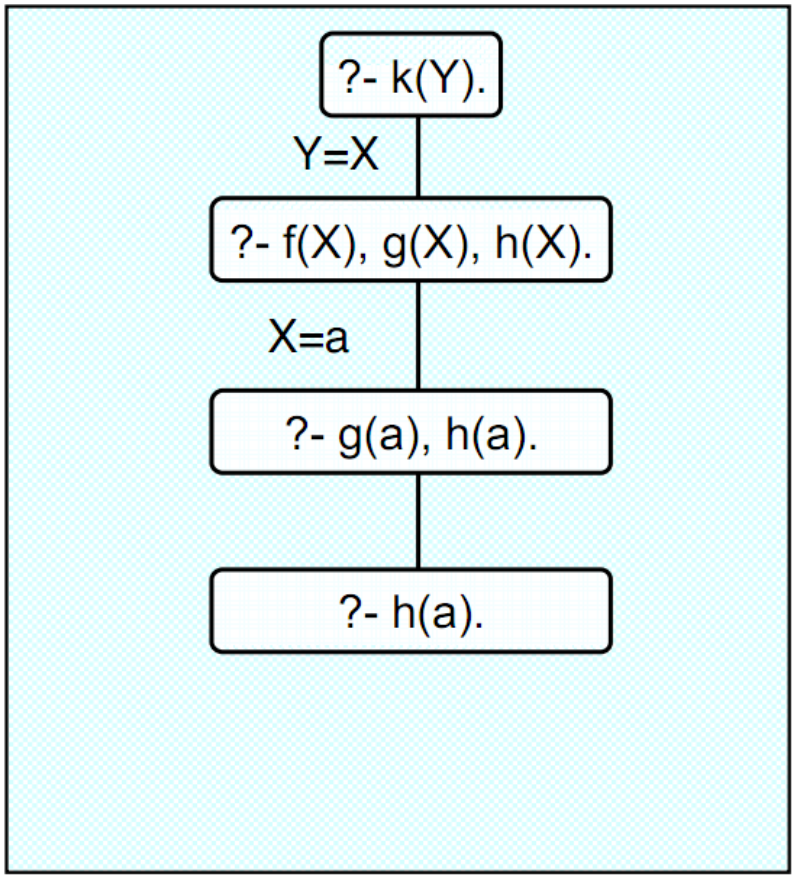
Pour résoudre une question, Prolog construit l'arbre de recherche de la question

- Racine de l'arbre : question
- Noeud : formules à démontrer
- Branche : règle et unification à considérer
- **Noeuds d'échec** : aucune règle ne permet de démontrer la première formule du noeud
- **Noeud de succès** : ne contient plus aucune formule, tout a été démontré et les éléments de solution sont trouves en remontant vers la racine de l'arbre

Arbre de recherche : Exemple

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

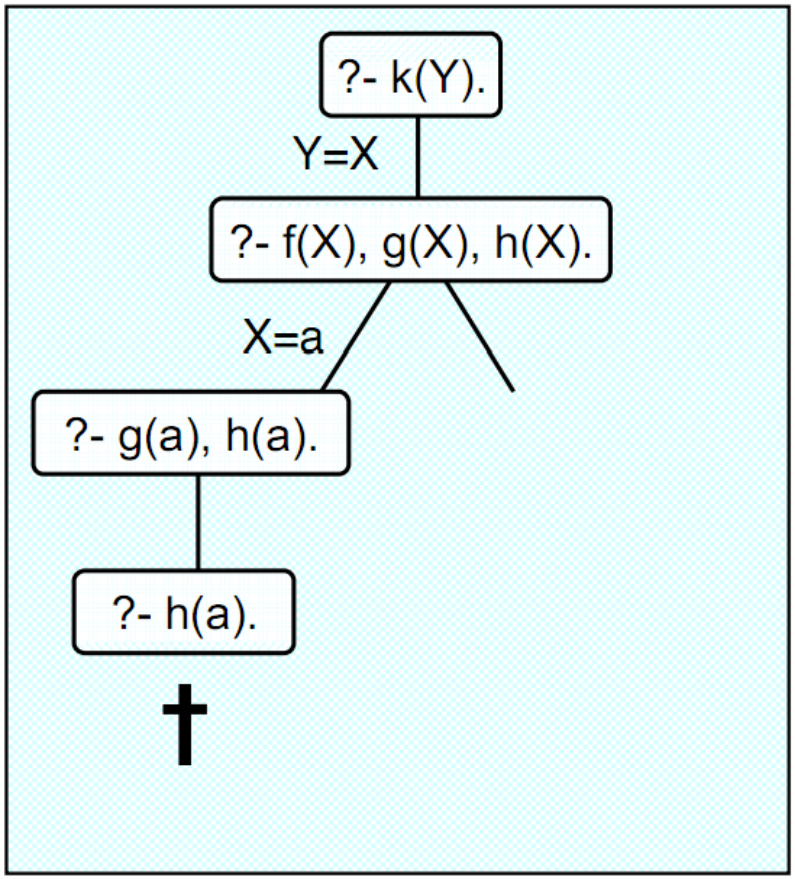
```
?- k(Y).
```



Arbre de recherche : Exemple

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

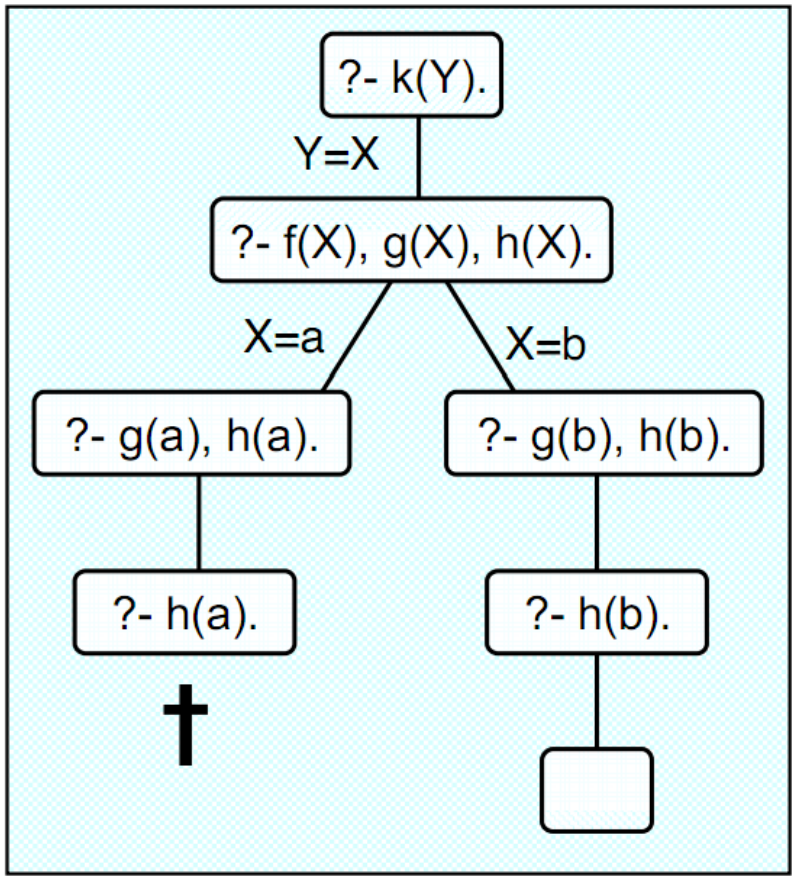
```
?- k(Y).
```



Arbre de recherche : Exemple

```
f(a).
f(b).
g(a).
g(b).
h(b).
k(X):- f(X), g(X), h(X).
```

```
?- k(Y).
Y=b;
no
?-
```



Backtracking

Backtracking

- Les faits et règles sont examinés dans leur ordre d'apparition dans le programme (stratégie linéaire)
- Si une sous-preuve échoue, la recherche revient sur ses pas (**retour-arrière, backtracking**) pour tenter la prochaine alternative de preuve
- Possibilité de forcer le backtrack en cas de réussite pour trouver une autre solution (**;** dans l'interpréteur)
- **Parcours en profondeur** de l'arbre de recherche

Non-déterminisme

Non-déterminisme

- Le retour-arrière illustre le non-déterminisme de Prolog
- Le **non-déterminisme** est la propriété qui permet à deux exécutions/évaluations successives de fournir des résultats différents
- Le non-déterminisme se situe au niveau du choix de la tête de clause (pour les faits/règles ayant plusieurs clauses) et des liaisons de variables correspondantes.
- Chaque emplacement où il y a non-déterminisme est considéré comme un **point de choix** vers lequel on reviendra en cas d'échec d'une sous-preuve

Attention

Attention

- **Stratégie linéraire + Non-déterminisme**
  $\Rightarrow$ Attention à l'ordre des clauses du programme.
  $\Rightarrow$ Attention à l'ordre des littéraux dans la queue de clause.
  $\Rightarrow$ Attention aux clauses croisés.

Exemple

```
pere(X,Y) :- parent(X,Y), male(X).
parent(X,Y) :- pere(X,Y).
```

Stratégie de Prolog

Pour répondre à une question, PROLOG tente d'**effacer** les buts un à un, dans l'ordre de leur apparition dans le programme.

L'effacement est réalisé ainsi, en fonction du type d'énoncé :

- une question comme $Q_1(...)$, $Q_2(...)$, ..., $Q_n(...)$, s'interprête par :

effacer Q_1(...), effacer Q_2(...), ..., effacer Qn(...).

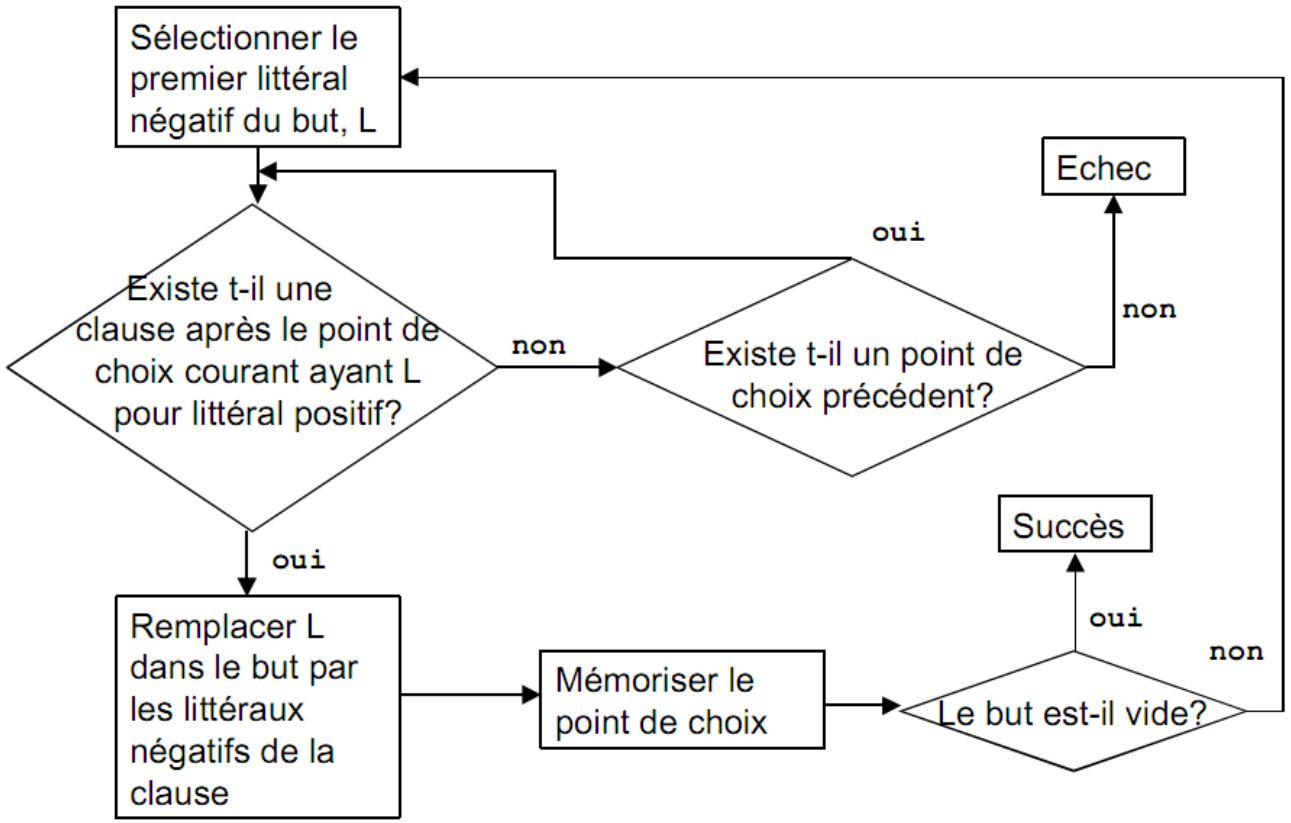
- une règle comme $P(...) \rightarrow Q(...),R(...),S(...)$, s'interprête par :

Pour effacer P(...), il faut effacer Q(...), puis R(...) et enfin R(...).

- un fait comme $P(...) \rightarrow .$ s'interprête par :

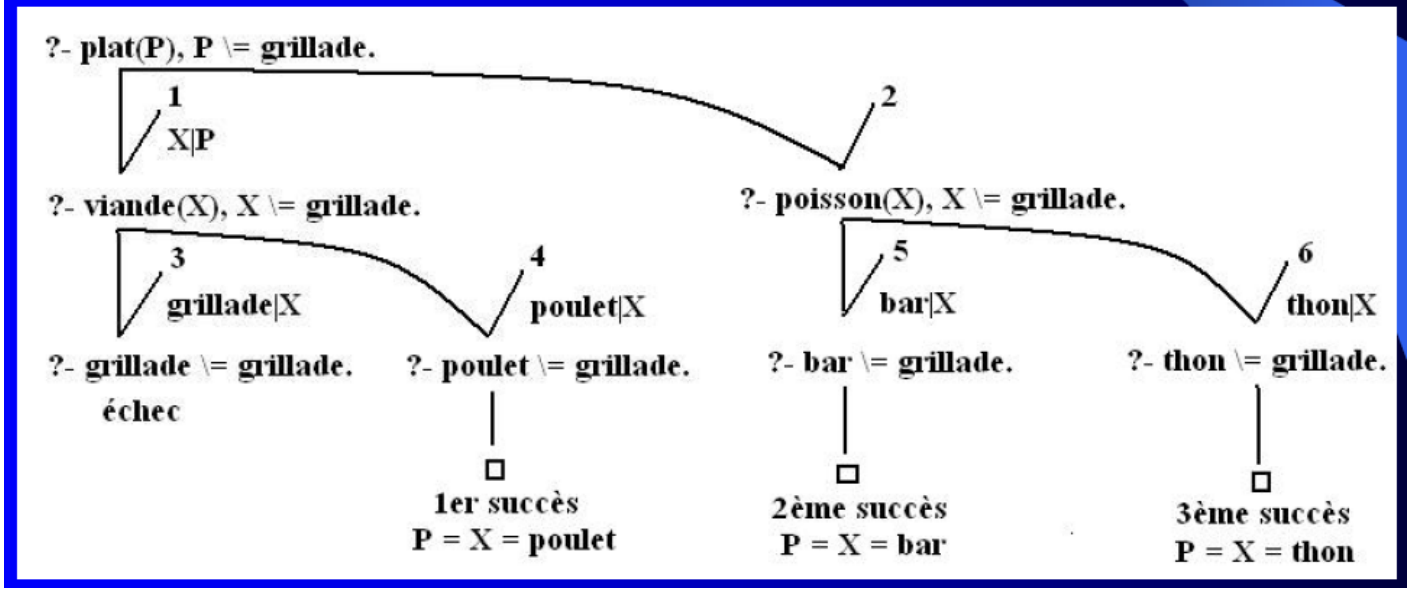
P s'efface.

Stratégie de Prolog



Autre exemple

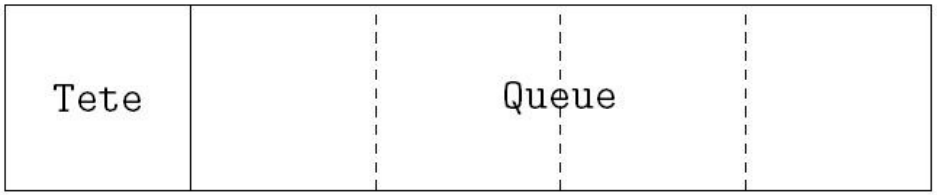
Soit le programme:	Sous Prolog on pose le but:
1: plat(X) :- viande(X). 2: plat(X) :- poisson(X). 3: viande(grillade). 4: viande(poulet). 5: poisson(bar). 6: poisson(thon).	?- plat(P), P \= grillade.



Sommaire

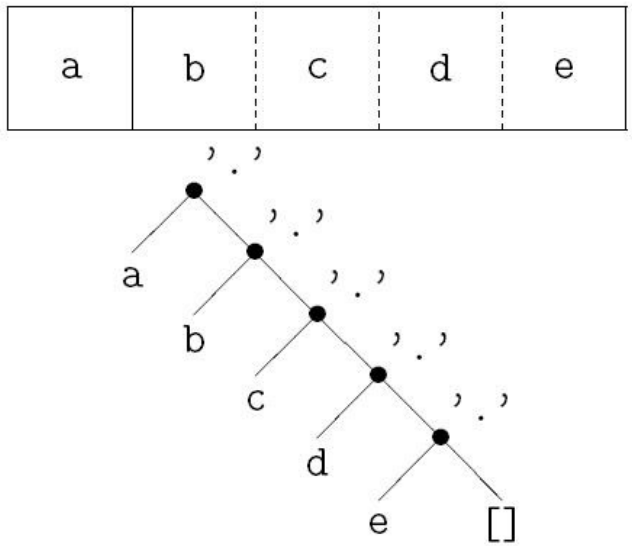
Listes

Les listes Prolog



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

Listes




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

Listes

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]
```

Recursivité

Exemple : fonction d'appartenance d'un élément à une liste

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

Tout prédicat défini par une règle récursive doit évidemment posséder au moins un cas trivial (appel terminal). Dans le cas contraire, le programme bouclerait.
Tout cas trivial d'une règle récursive doit apparaître avant le(s) appel(s) récursif(s).

Recursivité

Fibonacci

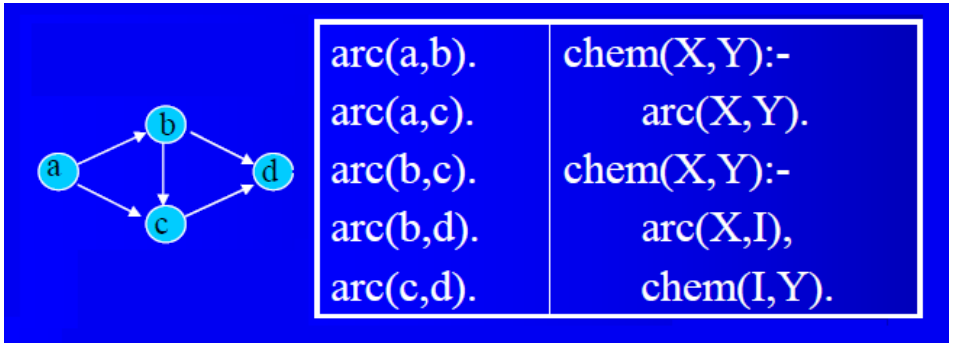
```
/* N et R entiers >=0 fibo(+N, ?R)
R est le Ni‘eme nombre de la suite de Fibonacci */

(r1) fibo(0,1).

(r2) fibo(1,1).

(r3) fibo(N,R) :- N>=2, N1 is N-1, N2 is N-2,
 fibo(N1,R1),fibo(N2,R2),R is R1+R2.
```

Recursivité



Sommaire

Coupure

Exemple

Soit une fonction f dont une définition Prolog est :
f(X, 0) :- $X < 3$.
f(X, 2) :- $3 \leq X$, $X < 6$.
f(X, 4) :- $6 \leq X$.
Que se passe-t-il si on pose la question :
?- f(1,Y), $2 < Y$.

Réponse

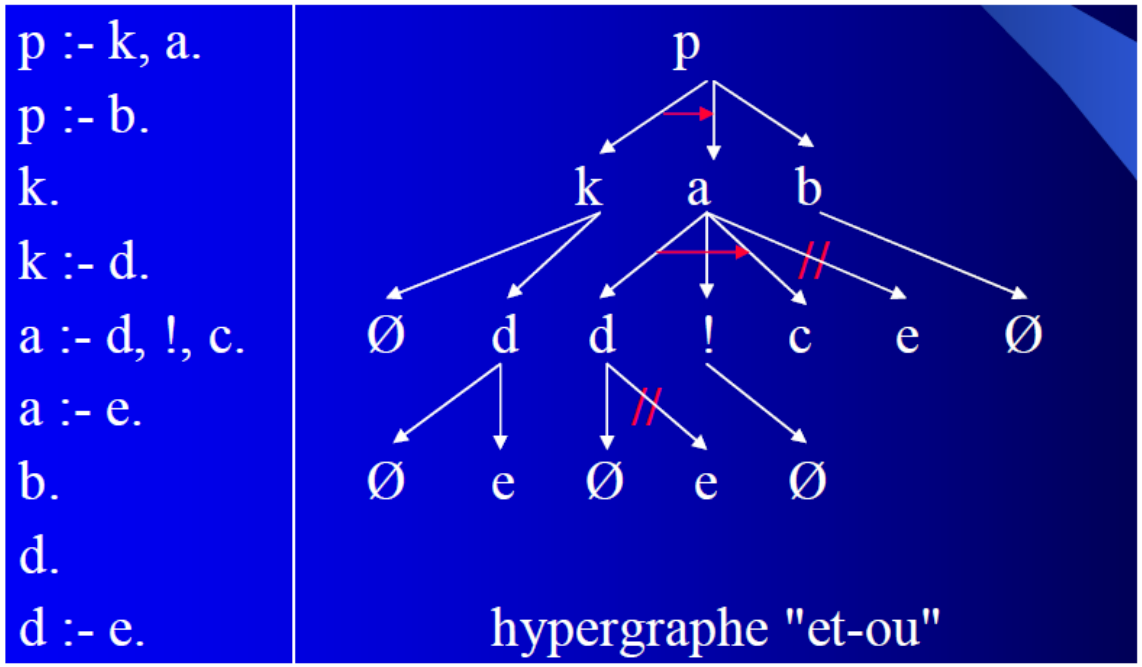
Première règle : $\mathbf{Y = 0}$
Démonstration de $2 < Y$:- **échec**
Deuxième règle : **échec**
Troisième règle : **échec**

Coupure

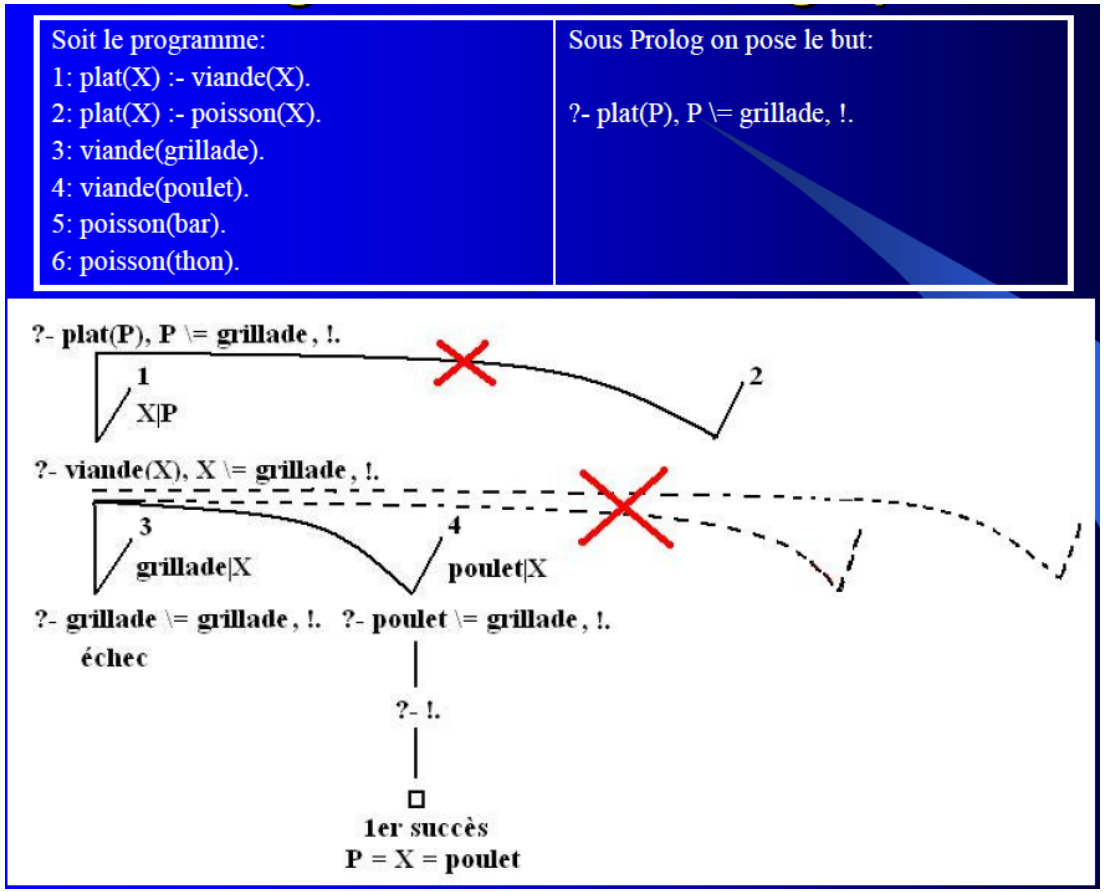
Coupure

- Nous avons vu que l'effacement d'un terme se fait de façon non deterministe. On garde en réserve les divers points de choix pour y revenir ultérieurement lors de la remontrée dans l'arbre.
- Il est possible de supprimer certains de ces points de choix de choix, donc d'empêcher certains retour-arrière de Prolog au moyen de la **coupure** ou **cut**, en utilisant le prédicat noté **!**
- Si on a la clause suivante : **p :-** n_1**, , ...,** n_k**, !, ... ,** n_p
  Alors tous les points de choix mémorisés depuis l'appel de la tête de clause jusqu'à l'éxécution du cut sont supprimés.

Coupure

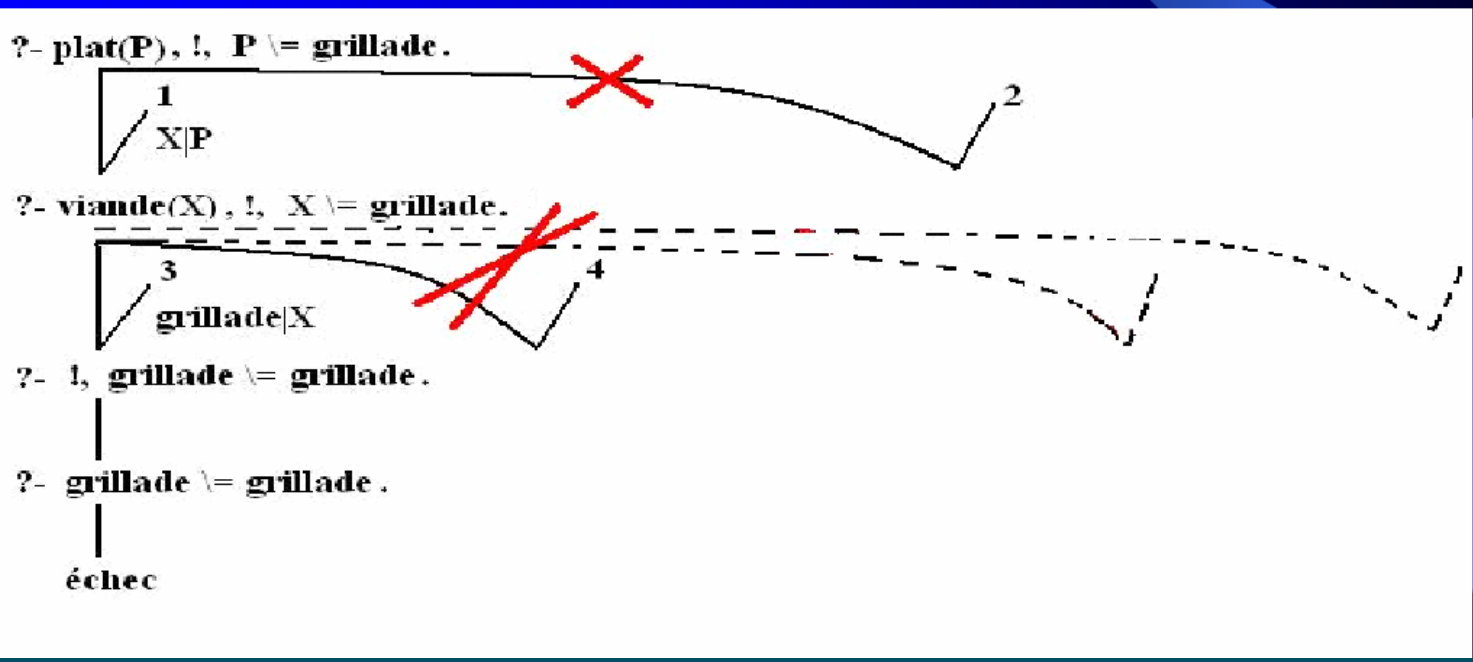


Coupure


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

Coupure

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.



Coupure

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.

Négation par l'échec

En PROLOG, la négation s'écrit comme un prédicat à un argument dont le nom est *not* et l'argument le prédicat à nier.
Par exemple, si f est une formule, sa négation est notée *not(f).*

Remarque :

PROLOG pratique la **négation par l'échec** : pour démontrer *not(f)*, PROLOG tente de démontrer f. Si la démonstration de f échoue, PROLOG considère que *not(f)* est démontrée.

Autrement dit, *not(f)* réussit lorsque :

- f est faux, ou
- f n'est pas déductible, c'est-à-dire **inexistant** dans la base.

Hypothèse du monde clos : tout ce qui n'est pas vrai est faux.

Négation par l'échec

Des conclusions douteuses peuvent alors parfois être émises...

Exemple #1 : Soit la base de connaissance suivante :

```
homme( 'Pierre' ).
femme( 'Eve' ).
femme( X ) :- not( homme( X ) ).
```

A la question :

```
?- femme( adam ).
```

l'interpréteur PROLOG retournera :

yes !?!

Négation par l'échec

Un prédicat nié est considéré comme VRAI s'il n'y a pas moyen de démontrer le prédicat débarassé de la négation.
Ce point de vue pratique n'est pas celui de la logique classique.

En PROLOG, *not(f)* signifie donc que la formule f n'est pas démontrable.

Sommaire

Opérateurs arithmétiques

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`

Opérateurs arithmétiques

Exemples de comparaisons arithmétiques :

?- 1 =:= 2 .	**no**
?- 1 =:= 1 .	**yes**
?- 1 =:= 1.0.	**yes**
?- 3*2.0 =:= 12.0 / 2 .	**yes**
?- 5.0 =< 24 .	**yes**

Opérateurs arithmétiques

'//'/2
mod/2

abs/1
sqrt/1
sign/1

floor/1
truncate/1
round/1
ceiling/1

log/1
exp/1

sin/1
cos/1

*'**'/2*

Opérateurs personalisés

- PROLOG dispose des opérateurs arithmétiques courants, tels que +, -, * et /, en notation infixée : 1+2, 8*(5-3)...

- PROLOG permet aussi à l'utilisateur de définir de nouveaux opérateurs, et de modifier ainsi sa syntaxe.

Un opérateur est alors défini par :

- sa priorité (entre 0 et 1200)
- son type (infixé | préfixé | postfixé) et son caractère associatif (à gauche | à droite | non associatif)
- son nom

Opérateurs personalisés

Déclaration des opérateurs :

En enregistrant une nouvelle règle sans tête, comme :

:- op(Priorite, Specification, Nom).

où :

- Priorité : comprise entre 0 (plus forte priorité) et 1200
- Specification : type et caractère associatif de l'opérateur
- Nom : nom de l'opérateur

Opérateurs personalisés

Trois types d'opérateurs :

- Opérateur **binaire infixé** : il figure entre ses deux arguments et désigne un arbre binaire.

  Exemple : X **+** Y

- Opérateur **unaire préfixé** : il figure avant son argument et désigne un arbre unaire

  Exemple : **-**X

- Opérateur **unaire postfixé** : il figure après son argument et désigne un arbre unaire

  Exemple : X**^2**

Opérateurs personalisés

Tableau des spécificateurs (types et associativité) :

Spécificateur	Type	Associativité
fx	préfixe	non associatif
fy	préfixe	associatif à droite
xfx	infixe	non associatif
xfy	infixe	associatif à droite
yfx	infixe	associatif à gauche
xf	postfixe	non associatif
yf	postfixe	associatif à gauche

Opérateurs personalisés

Exemple de déclaration d'un opérateur :

:- op(1000, xfx, aime) : définit un opérateur infixé non associatif *aime.*

Dans ce cas, Prolog traduira une expression du type *X aime Y* en le terme *aime(X, Y)*, et si Prolog doit afficher le terme *aime(X, Y)*, il affichera *X aime Y*.

Sommaire

Prédicats ensemblistes

```
pere(tarzan, fils).
pere(tarzan, fille).

age(fils, 5).
age(fille, 2).
age(tarzan, 25).
```

- *On veut la liste de tous les enfants de* ***tarzan*** et leur âge.

Prédicats ensemblistes

pere(**tarzan**, **fils**).
pere(**tarzan**, **fille**).

age(**fils**, 5).
age(**fille**, 2).
age(**tarzan**, 25).

- *On veut la liste de tous les enfants de **tarzan*** et leur âge.
- **pere(tarzan, X), age(X,A)** ne donne les solutions qu'une par une
- Comment les récupérer toutes dans une liste ?

Prédicats ensemblistes

```
?- setof(X/A, (pere(tarzan, X), age(X,A)), L).
   L = [fils/5, fille/2]
```

- ***setof**/3 résultat trié (**fail** si pas de solution)*
- ***bagof**/3 résultat tel que le backtrack (**fail** si pas de solution)*
- ***findall**/3 résultat tel que le backtrack (**[]** si pas de solution)*

```
findall(X, member(X/Y, [b/1, a/2]), L)      --> [b,a]
setof(X, member(X/Y, [a/1, b/2]), L         --> L=[b]  Y=1 ;
                                                L=[a]  Y=2
setof(X, Y^member(X/Y, [b/1, a/2]), L)      --> [a,b]
```

Prédicats de manipulation de clauses

`assert(C)`
ajoute la clause C. Sa position dans la liste des clauses dépend de l'implémentation.

`asserta(C)`
ajoute C *par le haut*, c'est-à-dire avant la première clause du paquet correspondant.

`assertz(C)`
ajoute C *par le bas*, c'est-à-dire après la dernière clause du paquet correspondant.

`clause(P,Q)`
cherche une clause de tête P et de corps Q. (Q = true pour une clause positive).

`retract(C)`
supprime la première clause qui s'unifie avec C.

`abolish(N, A)`
supprime toutes les clauses de nom N et d'arité A.

Remarques syntaxiques:
- on n'écrit pas le . de fin de clause, soit
  `assert(p(a));`
- en raison des priorités des opérateurs, on devra écrire
  `assert((p:-q,r))` et non `assert(p:-q,r)` .

Prédicats entrées/sorties

`see(Fic)`	:	lecture sur fichier `Fic ;`
`seeing(Fic)`	:	donne l'entrée courante dans `Fic;`
`seen`	:	ferme l'entrée courante;
`tell(Fic)`	:	sortie sur fichier `Fic ;`
`telling(Fic)`	:	donne la sortie courante dans `Fic;`
`told`	:	ferme la sortie courante;

Lectures - Ecritures

`read(X)`	:	lecture dans l'entrée courante d'un terme (qui doit être suivi d'un point, d'un blanc ou d'un [Entrée]) qui est unifié avec `X`
`write(X)`	:	écriture de `X` sur la sortie courante
`nl`	:	passage à la ligne
`tab(N)`	:	saute `N` espaces
`get0(N)`	:	renvoie dans `N` le code ASCII du caractère lu si on rencontre la fin de fichier, renvoie [CTRL] Z et le fichier est fermé
`get(N)`	:	même chose mais pour le premier caractère non blanc
`put(N)`	:	écrit le caractère de code N

Trace

Visiter l'arbres de dérivation: le prédicat `trace`.

On peut visualiser les traces des calculs en utilisant le prédicat prédéfini `trace/0` (`notrace/0` pour quitter le traceur).
Tracer un but donne lieu à une séquence de messages, qui nous informent sur l'arbre de dérivation que l'interpreteur PROLOG est en train de construir (utile pour debug). Ces messages sont de 4 types:

- `call p(t1,...,tn)` : le but `p(t1,...,tn)` est en tête de la pile des buts. La recherche d'une clause dont la tête s'unifie avec `p(t1,...,tn)` commence.
- `exit p(t1,...,tn)`: le but `p(t1,...,tn)` a été prouvé.
- `redo p(t1,...,tn)`: on revient au but `p(t1,...,tn)` par *backtracking*.
- `fail p(t1,...,tn)`: il n'existe plus aucune possibilité d'unifier le but `p(t1,...,tn)` avec la tête d'une clause.

Trace

Considérons par exemple le programme `ex.pl`:

```
q(a).
q(b).
r(b).
r(c).
p(X):-q(X),r(X).
```

et le but:

```
?- P(X).
```

Trace

```
 ?- consult(exemple).
yes
   ?- trace.
[ Trace mode on. ]
yes
[trace]
   ?- p(X).
   (1) call:p(_172) ?
   (2) call:q(_172) ?
   (2) exit:q(a) ?
   (3) call:r(a) ?
   (3) fail:r(a) ?
   (2) redo:q(a) ?
   (2) redo:q(_172) ?
   (2) exit:q(b) ?
   (4) call:r(b) ?
   (4) exit:r(b) ?
   (1) exit:p(b) ?
```