La Programmation Logique et le langage Prolog

Notes de cours 2012
Etienne Kouokam

Révision : Décembre 2012

1. Introduction

Le Prolog est un langage répandu essentiellement dans le domaine universitaire et le monde de la recherche. Il est lié de manière non exhaustive :

- à la logique formelle et à de nouvelles formes de programmation;
- à une modélisation du raisonnement;
- au traitement linguistique, à l'écriture de grammaires, d'analyseurs;
- d'un point de vue plus applicatif, aux bases de données : il est facile de transcrire du SQL en Prolog et d'interfacer une base de données avec un module logique écrit en Prolog.

2. Les éléments fondamentaux du *Prolog*

2.1 Les faits

Les faits sont des affirmations qui décrivent des relations ou des propriétés, par exemple :

```
élève(jean, 1975, info, 2).
élève(catherine, 1974, info, 2).
élève(luc, 1976, _, 1).

masculin(jean).
masculin(luc).

féminin(catherine).

père(paul, jean).% paul est le père de jean

mère(isabelle, jean).
```

La forme générale d'un fait est la suivante : `prédicat(argument1, argument2, ...)` . Un prédicat est un symbole qui traduit une relation. L'arité est le nombre de ses arguments. On identifie un prédicat par son nom et son arité : `prédicat/arité`, par exemple `mère/2`, `élève/4`.

On charge ou on substitue le fichier (mise à jour) dans l'interprète Prolog avec la commande :

```
?- consult(nom_de_fichier).
```

ou bien avec le raccourci :

```
?- [nom_de_fichier].
```

2.2 Les questions ou les requêtes

Une fois le programme chargé, on peut poser des questions sur les faits :

```
?- masculin(jean).
Yes

?- masculin(françois).
No. (Pas dans la base de données)
```

On peut aussi utiliser des variables. Elles peuvent s'identifier à toutes les autres valeurs : aux constantes, aux termes composés, aux variables elles-mêmes. Les constantes commencent par une minuscule, les variables commencent par une majuscule.

```
?- masculin(X).

X = jean ;

X = luc ;

No
```

Le caractère « ; » permet de demander la solution suivante. Le caractère *Retour* arrête la recherche des solutions.

```
?- élève(X, Y, Z, 2)

X = jean, Y = 1975, Z = info ;

X = catherine, Y = 1974, Z = info.

?-
```

2.3 Les types du Prolog

En *Prolog*, tout est un terme :

- **Les constantes** ou les termes atomiques :
  1. ***Les atomes*** sont des chaînes alphanumériques qui commencent par une minuscule : `jean`, `paul`, `tOtO1`. On peut transformer une chaîne contenant des caractères spéciaux (point, espaces, etc.) dans un atome en l'entourant de caractères « ' ». Ainsi `'Mal Juin', 'Bd Pasteur'` sont des atomes.
  2. ***Les nombres*** : 19, -25, -3.14, 23E-5
- **Les variables** commencent par une majuscule ou le signe _. _ tout seul est une variable anonyme : `X XYZ Xyz _x _3 _`. Le système renomme en interne ses variables et utilise la convention `_nombre`, comme `_127` ou `_G127`.

Les structures ou **termes composés** se composent d'un foncteur avec une suite d'arguments. Les arguments peuvent être des atomes, des nombres, des variables, ou bien des structures comme par exemple `élève(robert, 1975, info, 2, adresse(6, 'mal juin', 'Caen')).` Dans l'exemple, `élève` est le foncteur principal.

Lors d'une question, si elle réussit, les variables s'**unifient** en vis-à-vis aux autres termes. Les termes peuvent être composés comme par exemple `adresse(6, 'mal juin', 'Caen')`. Ajoutons le terme `élève(robert, 1975, info, 2, adresse(6, 'mal juin', 'Caen'))` dans le ficher et consultons-le. Posons la question :

```
?- élève(X, Y, Z, T, W).

X = robert
Y = 1975
Z = info
T = 2
W = adresse(6, 'mal juin', 'Caen')
```

Le Prolog unifie le terme de la question au terme contenu dans la base de données. Pour ceci, il réalise la **substitution** des variables `X, Y, Z, T, W` par des termes, ici des constantes et un terme composé. On note cette substitution :
`{X = robert, Y = 1975, Z = info, T = 2, W = adresse(6, 'mal juin', 'Caen')}`

On dit qu'un terme A est une **instance** de B s'il existe une substitution de A à B :

- `masculin(jean)` et `masculin(luc)` sont des instances de `masculin(X)`
- `{X = jean}` ou `{X = luc}` sont les substitutions correspondantes.

Un terme est **fondé** (*ground term*) s'il ne comporte pas de variable : `f(a, b)` est fondé, `f(a, X)` ne l'est pas; une substitution est fondée si les termes qui la composent sont fondés : `{X = a, Y = b}` est fondée, `{X = a, Y = f(b, Z)}` n'est pas fondée.

2.4 Variables partagées

On utilise une même variable pour contraindre deux arguments à avoir la même valeur. Par exemple, pour chercher un élève qui porterait le nom de sa filière (l'exemple est un peu idiot) :

```
?- élève(X, Y, X, Z).
```

Les questions peuvent être des conjonctions et on peut partager des variables entre les buts. Pour chercher tous les élèves masculins, on partage la variable `X` entre `élève` et `masculin` :

```
?- élève(X, Y, Z, T), masculin(X).
```

2.5 Les règles

Les règles permettent d'exprimer des conjonctions de buts. Leur forme générale est :

TÊTE :- C_1, C_2,... ,C_n.

La tête de la règle est vraie si chacun des éléments du corps de la règle C_1,... ,C_n est vrai. On appelle ce type de règles des clauses de Horn.

```
fils(A, B):-
    père(B, A),
    masculin(A).
fils(A, B):-
    mère(B, A),
    masculin(A).

parent(X, Y):-
    père(X, Y).
parent(X, Y):-
    mère(X, Y).

grand_parent(X, Y):-
    parent(X, Z), % On utilise une variable intermédiaire Z
    parent(Z, Y).
```

Un prédicat correspond donc à un ensemble de règles ou de faits de même nom et de même arité : les clauses du prédicat. Beaucoup de Prolog demandent que toutes les règles et tous les faits d'un même prédicat soient contigus – groupés ensembles – dans le fichier du programme. On note le prédicat par son nom et son arité, par exemple `fils/2`, `parent/2`, `grand_parent/2`. Les faits sont une forme particulière de règles qui sont toujours vraies. La notation :

```
fait.
```

est en effet équivalente à :

```
fait :- true.
```

Définissons un nouveau prédicat « `ancêtre/2` » déterminant l'ancêtre `X` de `Y` par récursivité :

1. Condition de terminaison de la récursivité si c'est un parent direct.

```
ancêtre(X, Y) :-
    parent(X, Y).
```

2. Sinon `X` est ancêtre de `Y` si et seulement si il existe `Z`, tel que `X` parent de `Z` et `Z` parent de `Y`.

```
ancêtre(X, Y) :-
    parent(X, Z),
    ancêtre(Z, Y).
```

Lors de l'exécution d'une requête, Prolog examine les règles ou les faits correspondants dans l'ordre de leur écriture dans le programme : de haut en bas. Il utilise la première règle (ou le premier fait) du prédicat pour répondre. Si elle échoue, alors il passe à la règle suivante et ainsi de suite jusqu'à épuiser toutes les règles (ou tous les faits) définies pour ce

prédicat. Lorsqu'une règle est récursive, l'interprète Prolog rappelle le prédicat du même nom en examinant les règles (ou les faits) de ce prédicat dans le même ordre.

Dans le corps d'une règle, la virgule « , » est le symbole représentant un ET logique : le conjonction de buts. Le symbole « ; » représente le OU logique, la disjonction de buts :

```
A :-
    B
    ;
    C.
```

est équivalent à

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

3. Mise en œuvre d'un programme

3.1 Le lancement de l'interprète

Il existe différentes versions de *Prolog* et on en trouve pour tout type de système (Windows, Linux, Mac OS). Celui que nous utilisons ci-dessous correspond à Swi-Prolog que vous pouvez télécharger sur http://www.swi-prolog.org/ ou http://www.swi-prolog.org/Download.html

3.2 Chargement de fichiers

Les programmes sont dans des fichiers avec le suffixe « `.pl` » ou « .pro » en général. Pour compiler et charger un programme :

```
?- consult(nom_du_fichier).
```

où `nom_du_fichier` est un atome, par exemple :

```
?- consult('file.pl').
```

ou bien le raccourci avec la commande :

```
?- [nom_du_fichier].
```

par exemple

```
?- ['file.pl'].
```

Autre raccourci avec SWI :

```
?- [file].
```

Chargement de plusieurs fichiers simultanément

```
?- ['file1.pl', 'file2.pl'].
```

Une fois que les fichiers sont chargés, on peut exécuter les commandes. Elles se terminent par un point « . » :

```
?- gnagnagna.
```

Les conjonctions de buts sont séparées par des « , » :

```
?- gna1, gna2, gna3.
```

On peut aussi inclure des directives dans un fichier par l'instruction :

```
:- clause_à_exécuter.
```

Les directives sont des clauses que l'interprète exécutera lors chargement du programme.

L'affichage du contenu du fichier chargé se fait par :

```
?- listing.
```

L'affichage d'une clause particulière, ici `père`, se fait par :

```
?- listing(père).
```

Si on modifie le programme dans le fichier, on peut mettre à jour la base de données par (Pas avec SWI où on ne peut que consulter) :

```
?- reconsult('file.pl').
```

Le raccourci de rechargement est :

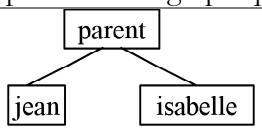
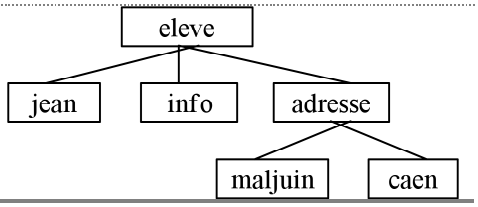
```
?- [-'file.pl'].
```

Finalement, on quitte Prolog avec

```
?- halt.
```

4. Termes et unification

Les termes peuvent se représenter par des arbres. Les nœuds sont les foncteurs.

Termes	Représentation graphique
`parent(jean, isabelle).`	
`élève(jean, info, adresse(maljuin, caen)).`	

Formellement, t_1 est une instance de t_2 s'il existe une substitution σ telle que $t_2 \sigma = t_1$.

Par exemple :

```
élève(jean, info, adresse(maljuin, caen))
```

est une instance de

```
élève(jean, X, Y)
```

avec la substitution :

σ = {(X, info), (Y, adresse(maljuin, caen)}

En effet, si on applique la substitution au dernier terme, on retrouve le premier.

On dit qu'un terme t_1 est plus général qu'un terme autre t_2 si t_2 est une instance de t_1.
On définit l'unification comme l'instance commune la plus générale de deux termes. L'opérateur Prolog de l'unification est « = ».

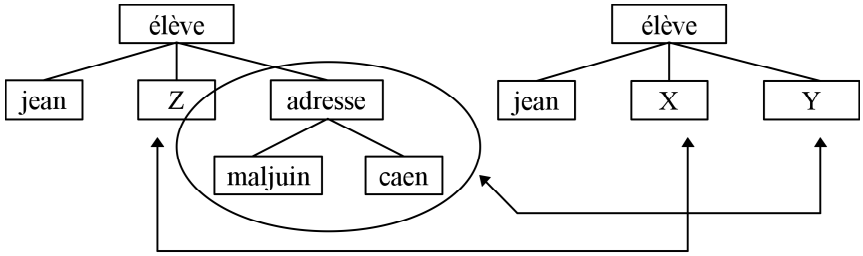
```
?- parent(X, isa) = parent(jean, Y)
X = jean, Y = isa

?- parent(jean, isa) = parent(X, X).
No

?- élève(jean, info, Z) = élève(X, info, adresse(maljuin,
caen)).
X = jean, Z = adresse(maljuin, caen))

?- élève(jean, info, adresse(Z1, Z2)) = élève(Y, info,
adresse(maljuin, caen)).
Y = jean, Z1 = maljuin, Z2 = caen

?- élève(X, Y, adresse(maljuin, caen)) = élève(T, info,
adresse(maljuin, caen)).
T = X, Y = info

?- parent(X, Y) = parent(T, isabelle), parent(X, Y),
masculin(T).
X = jean % (le père bien sûr!)
T = jean
Y = isabelle
```

Cette dernière question s'interprète de la façon suivante : tout d'abord l'unification: `X = T, Y = isabelle` puis la recherche dans la base de T tel que `parent(T, isabelle)` ET `masculin(T)` (autrement dit, `T` est le père d'isabelle). On doit avoir dans la base de données `parent(jean, isabelle)` et `masculin(jean)`.

L'unification de deux termes peut se représenter de manière graphique :

```
élève(jean, Z, adresse(maljuin, caen)) = élève(jean, X, Y).
```

Elle se traduit par une superposition des deux arbres et une identification des variables à leurs branches :



5. Les listes

Une liste est une structure de la forme :

```
[a]
[a, b]
[a, X, adresse(X, caen)]
[[a, b], [[[adresse(X, caen)]]]]
```

`[]` est la liste vide.

La notation des listes est un raccourci. Le foncteur « `.` » est le foncteur de liste : `.(a,.(b,[]))`. Il est équivalent à `[a, b]`.

La notation « | » est prédéfinie et permet d'extraire la tête et la queue. La tête est le 1er élément ; la queue est la liste restante sans le premier élément. Quelques exemples :

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

?- [a] = [X | Y].
X = a, Y = []

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

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

?- [[a, b, c], d, e] = [X | Y].
X = [a, b, c], Y = [d, e]
```

6. Quelques prédicats de manipulation de listes

Certains prédicats de manipulation de listes sont prédéfinis suivant les variantes de *Prolog* (c'est le cas à l'école).

6.1 Le prédicat `member/2`

6.1.1 Synopsis

Appartenance d'un élément à une liste :

```
?- member(a, [b, c, a]).
Yes

?- member(a, [c, d]).
No
```

6.1.2 Définition du prédicat

Programme	Commentaires
`member(X, [X | Y]).`	Le cas trivial. On peut aussi écrire `member(X, [X | _]).` Les 2 versions se valent.
`member(X, [Y | Ys]) :-` `  member(X, Ys).`	La règle récursive

6.1.3 Exemples d'utilisation

```
?- member(X, [a, b, c]).
X = a ;
X = b ;
X = c ;
No.

?- member(a, Z). % Déconseillé, car sans intérêt
Z = [a | Y] ;
Z = [Y, a | X] ;

etc.

?- not member(X, L).
```

Renvoie `yes` si `member(X, L)` vaut `no` et vice versa.

6.2 Le prédicat `append/3`

6.2.1 Synopsis

Ajout de deux listes :

```
?- append([a, b, c], [d, e, f], [a, b, c, d, e, f]).
Yes

?- append([a, b], [c, d], [e, f]).
No

?- append([a, b], [c, d], L).
L = [a, b, c, d]

?- append(L, [c, d], [a, b, c, d]).
L = [a, b]

?- append(L1, L2, [a, b, c]).
L1 = [], L2 = [a, b, c] ;
L1 = [a], L2 = [b, c] ;
```

etc., avec toutes les combinaisons

6.2.2 Définition du prédicat

Programme	Commentaires
`append([], L, L).`	La règle dans le cas trivial
`append([X | Xs], Ys, [X | Zs]) :-` `  append(Xs, Ys, Zs).`	La règle générale

6.3 Le prédicat `intersection/3`

6.3.1 Synopsis

Ce prédicat réalise l'intersection de deux listes
La syntaxe est : `intersection(LEntrée1, LEntrée2, LIntersection).`
Ici, on utilise une version approximée de ce prédicat pour simplifier le programme :

```
?- intersection([a, b, c], [d, b, e, a], L).
L = [a, b]

?- intersection([a, b, c, a], [d, b, e, a], L).
L = [a, b, a]
```

6.3.2 Définition du prédicat

Programme	Commentaires

`intersection([], _, []).`
`intersection([X | L1], L2, [X | L3]) :-`
`  member(X, L2),`
`  intersection(L1, L2, L3).`	

`intersection([_ | L1], L2, L3) :-` `  intersection(L1, L2, L3).`	si on arrive à cette règle, c'est que la tête de L_1 n'est pas dans L_2

6.3.3 Étude d'une exécution

```
?- intersection([a, b, c], [d, b, e, a], L).
```

N° appel	Clauses appelés	Variables lors de la descente			Dépilement récursif
1)	Clause 2	`X = a` `L1 = [b, c]`	`L2 = [d, b, e, a]`	`[a | L3] = L`	`L = [a, b]`
2)	Clause 2	`X' = b` `L'1 = [c]`	`L'2 = [d, b, e, a]`	`[b | L'3] = L3`	`L3 = [b]`
3)	Clause 3	`L"1 = []`	`L"2 = [d, b, e, a]`	`L"3 = L'3`	`L'3 = []`
4)	Clause 1				`L"3 = []`

6.4 Le prédicat `delete/3`

6.4.1 Synopsis

Efface un élément d'une liste.
La syntaxe est : `delete(Élément, Liste, ListeSansÉlément).`

6.4.2 Définition du prédicat

Programme	Commentaires
`delete(_,[],[]).`	Cas trivial
`delete(Élément, [Élément| Liste], ListeSansÉlément):-` `  delete(Élément, Liste, ListeSansÉlément).`	Cas où Élément est en tête de liste
`delete(Élément, [T| Liste], [T| ListeSansÉlément]):-` `  delete(Élément, Liste, ListeSansÉlément).`	Cas où Élément n'est pas en tête de liste (filtrage avec les règles précédentes.)

6.5 Le prédicat `reverse/2`

Inverse l'ordre d'une liste
1re solution coûteuse :

```
reverse([],[]).
reverse([X | XS], ZS] :-
```

```
    reverse(Xs,, Ys),
    append(Ys, [X], Zs).
```

2e solution. On passe d'une arité 2 à une arité 3 :

```
reverse(X, Y) :-
    reverse(X, [], Y).
reverse([X | Xs], Accu, Zs):-
    reverse(Xs, [X | Accu], Zs).
reverse([], Zs, Zs).
```

7. Arithmétique et opérateurs

Chaque *Prolog* dispose d'opérateurs infixés, préfixés et postfixés : + , - , * , / . L'interprète les considère comme des foncteurs et transforme les expressions en termes : `2 * 3 + 4 * 2` est un terme identique à `+(*(2, 3), *(4, 2)).`

7.1 Règles de précédence et d'associativité des opérateurs

- La précédence est la détermination récursive du foncteur principal par une priorité de 0 à 1200 (norme établie par le Prolog d'Édimbourg)
- L'associativité détermine le parenthésage de `A op B op C` :
    1. Si elle est à gauche, on a `(A op B) op C`
    2. Si elle est à droite, on a `A op (B op C)`

Un type d'associativité est associé à chaque opérateur :

	non associatif	droite	gauche
infixé	*xfx*	*xfy*	*yfx*
préfixé	*fx*	*fy*	
postfixé	*xf*		*yf*

Certains opérateurs sont prédéfinis en Prolog Standard.

Priorité	Associativité	Opérateurs
1200	xfx	(:- -->)
1200	fx	:-
1100	xfy	;
1000	xfy	,
700	xfx	= \=
700	xfx	== \==
700	xfx	=..
700	xfx	is =:= =\= < <= > >=
500	yfx	+ -
400	yfx	* /

```
200          xfy      & ^
200          fy       -
```

On peut définir de nouveaux opérateurs par :
`:- op(Précédence, Associativité, NomOpérateur).`

7.2 Opérations arithmétiques

Évaluer un terme représentant une expression arithmétique revient à appliquer les opérateurs. Ceci se fait par le prédicat prédéfini `is/2`.

```
?- X = 1 + 1 + 1.
X = 1 + 1 + 1 (ou X = +(+(1, 1), 1)).

?- X = 1 + 1 + 1, Y is X.
X = 1 + 1 + 1, Y = 3.

?- X is 1 + 1 + a.
```
erreur (`a` pas un nombre)

```
?- X is 1 + 1 + Z.
```
erreur (`Z` non instancié à un nombre)

```
?- Z = 2, X is 1 + 1 + Z.
Z = 2
X = 4
```

Si on essaie

```
?- 1 + 2 < 3 + 4.
```

Il y a évaluation des 2 termes de gauche et de droite avant la comparaison. Il importe de bien distinguer les opérateurs arithmétiques des opérateurs littéraux, ainsi que de l'unification.

	Numérique	Littérale (terme à terme)
Opérateur d'égalité	`=:=`	`==`
Opérateur d'inégalité	`=\=`	`\==`
Plus petit	`<`	`@<`
Plus petit ou égal	`=<`	`@=<`

Par exemple :

```
?- 1 + 2 =:= 2 + 1.
Yes.

?- 1 + 2 = 2 + 1.
No.
```

```
?- 1 + 2 = 1 + 2.
Yes.

?- 1 + X = 1 + 2.
X = 2.

?- 1 + 2 =:= 2 + 1.
Yes.

?- 1 + X =:= 1 + 2.
Erreur.

?- 1 + 2 == 1 + 2.
Yes.

?- 1 + 2 == 2 + 1.
No.

?- 1 + X == 1 + 2.
No.

?- 1 + a == 1 + a.
Yes.
```

7.3 Une application : le prédicat `length/2`

7.3.1 Synopsis

La longueur d'une liste :

```
?- length([a, b, c],3).
Yes

?- length([a, [a, b], c], N).
N = 3
```

7.3.2 Définition du prédicat

Programme	Commentaires
`length([],0).`	La règle dans le cas trivial de la liste vide

`length([X | Xs], N) :-`
`  length(Xs, N1),`
`  N is N1 + 1.`
	La règle générale

L'évaluation par le `is` est essentielle car *Prolog* ne comprend pas 1 + 2 comme une expression arithmétique, mais comme un terme :

```
?- L = 1 + 2
L = +(1, 2)

?- L is 1+2
L = 3
```

8. Modèle calculatoire du *Prolog*

Le calcul – l'interprétation – d'un programme Prolog repose sur l'unification et un algorithme de résolution :

- Unification :
  `pred(personne(X, marie, adresse(Nb, 'bd Mal Juin')), [T, a, b]) = pred(personne(Y, Z, adresse(1, F)), [a, G, b]).`
- Résolution :
  `T :-` B_1`,` B_2`,` B_3`, ...,` B_n. (but :- sous buts)

8.1 Unification

8.1.1 Définitions

Substitution. Remplacement dans le terme `T` des couples (variable, terme) qui forment la substitution. Par exemple, la substitution `{X = f(a, b), Y = c}` dans le terme `p(X, Y)`, produit `p(f(a, b), c)`.

Instance commune. T est une instance commune des termes t_1 et t_2 s'il existe 2 substitutions qui rendent identiques t_1 et t_2 : $t_1 \sigma_1 = t_2 \sigma_2$.

Lorsqu'on détermine une instance commune de deux termes, on ne doit pas avoir de couples X_i `=` t_i, avec X_i apparaissant dans le terme t_i, par exemple, X_i `= f(`X_i`, Y)`. En effet, il n'existe pas d'instance commune possible à ces deux termes. Cependant, pour des raisons d'efficacité, les implantations classiques de Prolog ne font pas cette vérification et la requête

```
?- X = f(X).
```

est bien exécutable et fait exploser la pile.

Un terme est *plus général* qu'un second si le second est une instance du premier : `p(X, Y)` est plus général que `p(a, Y)` ou que `p(a, Z)`. Dans ce dernier cas `Y` s'identifie à `Z` et on les appelle des variantes alphabétiques. Un terme qui comporte des variables à une infinité d'instances possibles qu'on obtient par exemple en substituant ses variables par des atomes.

Un unificateur de 2 termes est une substitution qui rend 2 termes identiques : c'est une instance commune à ces 2 termes. Comme, il existe en général une infinité de substitutions possibles, par exemple `p(a, Z)` et `p(X, Y)` ont une instance commune `p(a, b)`

qu'on obtient avec la substitution `{X = a, Z = Y = b}`, l'unification se définit par la détermination de l'instance commune la plus générale ou le plus général unificateur (PGU). Ici `{X = a, Z = Y}`.

8.1.2 Algorithme d'unification

L'unification repose sur l'algorithme de Herbrand :
Entrée : les deux termes à unifier T_1 et T_2, Sortie : la substitution σ.

Empiler $T_1 = T_2$
Tant que pile non vide
dépiler X = Y de la pile
Suivant les cas :
1) X est une variable sans occurrence dans Y (occur-check en anglais) : on substitue Y à X dans la pile et dans σ.
2) Y est une variable sans occurrence dans X : on substitue X à Y dans la pile et dans σ.
3) X et Y sont des constantes identiques ou des variables : on continue.
4) Cas général : $X = f(X_1, \ldots, X_n)$. $Y = f(Y_1, \ldots, Y_n)$. Empiler $X_i = Y_i$ pour *i* variant de 1 à n.

Sinon échec.

8.1.3 Exemple

Exécution avec un exemple : `f(g(X, h(X, b)), Z) = f(g(b, Z), Y)`
On empile les arguments en partant de la gauche du terme.
La pile après un tour est :

Terme gauche de la pile		Terme droit de la pile
`g(X, h(X, b))`	=	`g(b, Z)`
`Z`	=	`Y`

Substitution {}

On empile de nouveau les arguments et on réalise les substitutions. La pile après deux tours est :

Terme gauche de la pile		Terme droit de la pile
`X`	=	`b`
`h(X, b) ~ h(b, b)`	=	`Z`
`Z ~ h(b, b)`	=	`Y`

Substitution `{X = b, Z = h(b, b), Y = Z}`

8.2 Résolution

8.2.1 Présentation

La résolution est fondée sur le *modus ponens* :

A si B	majeure (la règle)	alors A
B	mineure (le fait)	

8.2.2 Algorithme de la résolution

L'algorithme de résolution est fondé sur les recherches de Robinson sur la démonstration automatique de théorèmes. Dans la littérature technique, on l'appelle aussi résolution SLD : stratégie linéaire avec fonction de sélection d'un sous but pour des clauses définies.

Entrée : un programme – un ensemble de règles
un but: G. On appelle le but courant – le but à un instant donné – la résolvante
Sortie : une instance de G ou échec

Tant que résolvante non vide :

1. on choisit un sous but $A \in G$,
2. on cherche une règle $A' \leftarrow B_1, \ldots B_n$. telle qu'il existe une unification entre A et A' (σ).
3. On substitue $B_1, \ldots, B_n$ dans la résolvante à A.
4. On applique σ à la résolvante et au but initial.

Sinon échec.

8.2.3 Exemple

L'interprète ne nous donne pas automatiquement une solution. Par exemple :

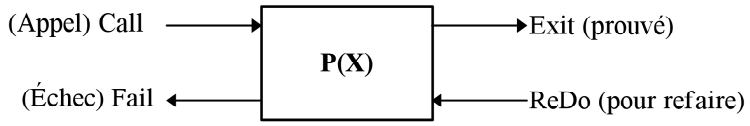
```
p(X) :-
    a(X),
    b(X).
a(a).
a(b).
b(b).
b(c).
```

Résolvante 1	Résolvante 2	Résolvante 3	Résultat
		`a(a), b(a) {X = c}`	échec
`p(X)`	`a(X) b(X)`		
		`a(b), b(b) {X = b}`	succès

Pour résolvante 2, nous avons le choix entre `a(X)` et `b(X)`. Prolog choisit le but le plus a gauche. Deux clauses s'unifient avec `a(X)`. La première mène à un échec. Pour trouver les solutions, Prolog utilise un mécanisme de retour en arrière. Lorsqu'il rencontre un échec, il revient au dernier point où un choix était possible, il annule l'unification et choisit la clause suivante possible. De cette manière, Prolog explore tous les arbres possibles jusqu'à ce qu'il trouve une solution ou bien rencontre un échec complet.

8.3 Modèle d'exécution – ou boîte d'exécution – du Prolog

L'examen des buts de la résolvante se fait de gauche à droite et les règles sont examinées de haut en bas.



Le traceur repose sur cette boîte. On l'active par :

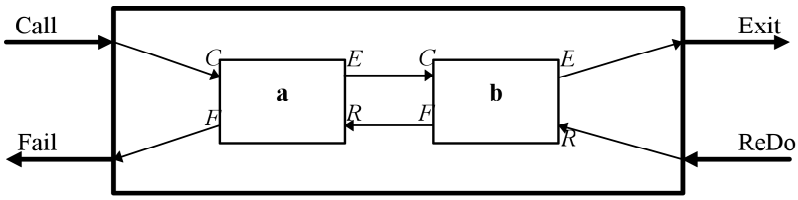
```
?- trace.
```

et on l'arrête avec

```
?- notrace.
```

Exemple :

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



Commandes du débogueur :

- `s` « skip » :sauter un prédicat
- ↲ avancer en profondeur dans le débogueur (creep)
- a quitter le débogueur

```
?- p(X).
   Call:  (  7) p(_G106) ? creep
   Call:  (  8) a(_G106) ? creep
   Exit:  (  8) a(a) ? creep
   Call:  (  8) b(a) ? creep
   Fail:  (  8) b(a) ? creep
   Redo:  (  8) a(_G106) ? creep
   Exit:  (  8) a(b) ? creep
```

```
   Call:  (  8) b(b) ? creep
   Exit:  (  8) b(b) ? creep
   Exit:  (  7) p(b) ? creep

X = b
```

9. Négation et cuts

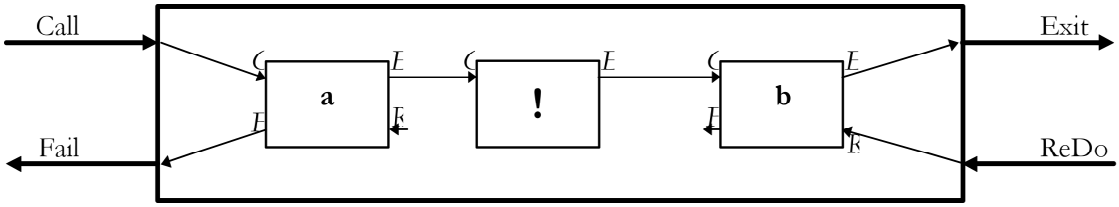
9.1 Le cut

9.1.1 Présentation

Le predicat prédéfini « ! » – le cut – permet d'empêcher le retour en arrière. La règle

```
p(X) :- a(X), !, b(X).
```

a la boîte d'exécution suivante :



Une fois un cut franchi :

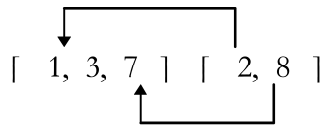
1) ! coupe toutes les clauses en dessous de lui,
2) ! coupe tous les buts à sa gauche ,
3) En revanche, possibilité de retour arrière sur les sous buts à la droite du cut.

P:- ~~$A_1, \ldots, A_i$~~, !, $A_{i+1}, \ldots, A_n$.
~~P:- $B_1, \ldots, B_m$.~~

Les conséquences du cut sont d'améliorer l'efficacité et d'exprimer le déterminisme; mais il revient à faire fonctionner le Prolog comme un langage procédural et il peux introduire des erreurs de programmation vicieuses. Il faut donc l'utiliser avec parcimonie.

9.1.2 Exemple

La fusion de 2 listes triées est déterministe :



On peut donc éliminer les possibilités de retour en arrière :

```
fusion([], L, L).
fusion(L, [], L).
```

```
fusion([X | Xs], [Y | Ys], [X | Zs]) :-
    X < Y,
    !,
    fusion(Xs, [Y | Ys], Zs).
fusion([X | Xs], [X | Ys], [X, X | Zs]) :-
    !,
    fusion(Xs, Ys, Zs).
fusion([X | Xs], [Y | Ys], [Y | Zs]) :-
    X > Y,
    !,
    fusion([X | Xs], Ys, Zs).
```

9.1.3 Règles pratiques d'utilisation

Bonne utilisation : cuts verts (ne modifie pas le sens d'un programme). Par exemple, la fusion.
Mauvaise utilisation : cuts rouges (paresseux), par exemple, le minimum

```
min(X, Y, X) :- X < Y, !.
min(X, Y, Y).
```

Ce prédicat conduit bien au calcul du minimum pour la première solution. Le cut est nécessaire pour éviter de produire une seconde solution au cours d'un retour en arrière. Si on a la règle :

```
P :- B1, …, Bi-1, min(2, 3, Z), Bi, …, Bn.
```

À la première exécution, `Z = 2`. Si on revient en arrière à cause d'un échec des buts `Bi, …, Bn` et si `min/3` ne comporte pas de cut, on obtient `Z = 3` car `min/3` aurait pu produire deux solutions. On repartirait alors avec cette valeur pour prouver buts `Bi, …, Bn`.

Autre cut rouge:

```
ifthenelse(P, Q, R):-
    P, !, Q.
ifthenelse(P, Q, R):-
    R.
```

9.2 La négation

9.1.1 Synopsis

Le symbole de la négation: « `\+` » (anciennement `not`)
Un programme logique exprime ce qui est vrai. La négation, c'est donc ce que l'on ne peut pas prouver :

- Si `G` réussi ⇒ `\+ G` échoue,
- Si `G` échoue ⇒ `\+ G` réussit.

9.2.2 Définition

Le prédicat `not` se définit par :

```
not(P) :- P, !, fail.
not(P) :- true.
```

9.2.3 Exemple et précautions

Avec un `\+` (ou bien `not`), il vaut mieux que les variables soient instanciées :

```
?- \+ member(X, [a, b]).
No.
```

Le programme identifie `X` à une des variables, réussit et le `not` le fait échouer.

De même dans une règle, il faut veiller à l'instanciation des variables. Avec le programme :

```
marié(françois).

étudiant(rené).

étudiant_célibataire(X) :-
    \+ marié(X),
    étudiant(X).
```

La requête suivante échoue :

```
?- étudiant_célibataire(X).
No.
```

car `X` s'apparie à `françois` puis le `not` fait échouer. Alors que :

```
?- étudiant_célibataire(rené).
Yes.
```

réussit car `X = rené` donc `\+ marié(rené)` est vrai.

Pour que la règle produise les résultats attendus, il faut inverser les sous buts :

```
étudiant_célibataire(X) :-
    étudiant(X)
    \+ marié(X).
```

Ainsi `X` sera instancié avant d'être soumis au `\+`.

9.3 Le prédicat `once/1`

9.3.1 Synopsis

Le prédicat `once(P)` permet de gérer le retour en arrière en évitant de revenir sur le groupe délimité par `once`. Dans la règle :

`A :- B1, B2, once((B3, …,Bi)), Bi+1, …, Bn.`

l'exécution du retour en arrière passera directement de B_{i+1} à B_2 en sautant $B_3, \ldots, B_i$.
Si `P` est une conjonction de buts, on doit la parenthéser. Sinon, écrire `once(A, B)`, par exemple, reviendrait à considérer `once` comme étant d'arité 2.

9.3.2 Définition

On définit `once(But)` de la manière suivante :

```
once(But) :-
    But, !.
```

9.3.3 Exemple

Supposons qu'on dispose d'une base de données géographique et qu'on cherche à trouver les océans communs à des pays d'Afrique et d'Amérique, la solution s'écrirait :

```
océan_commun(X) :-
    océan(X),
    once((
      frontière(X, Y),
      africain(Y),
      pays(Y))),
    %le retour arrière se fera sur océan(X).
    frontière(X, Z),
    américain(Z),
    pays(Z).
```

Si base de données comprend les océans dans l'ordre :

```
océan(indien).
océan(atlantique).
```

`once` permet d'éviter le retour en arrière sur le premier `frontière(indien, Y)` et de générer immédiatement un nouvel `océan(X)`.

10. Quelques prédicats prédéfinis

10.1 Prédicats de type

integer/1 : est-ce un entier?

```
?- integer(3).
Yes.

?- integer(X).
No.
```

number/1 : est-ce un nombre?

```
?- number(3.14).
Yes.
```

float/1 : est-ce un flottant?

atom/1 : est-ce un atome?

```
?- atom(toto).
Yes.

?- atom(3).
No
```

atomic/1 (ou constant/1 dans certains Prolog) : Est-ce un atome ou un nombre?

var/1 : est-ce une variable non instanciée?

```
?- var(X).
Yes.

?- X = f(Z), var(X).
No
```

nonvar/1. Le contraire de var/1.

```
?- nonvar(X).
No
```

compound/1. Est-ce que le terme est composé?

```
?- compound(X).
No.

?- compound(f(X, Y)).
Yes
```

ground/1. Est-ce que le terme est fondé?

```
?- ground(f(a, b)).
Yes

?- ground(f(a, Y)).
```

```
No
```

10.2 Prédicats de manipulation de termes

`functor(Terme, Foncteur, Arité)`. Mode d'utilisation (+, -, -) ou (-, +, +) seulement!

```
?- functor(père(jean, isa), F, A).
F = père, A = 2.

?- functor(T, père, 2).
T = père(X, Y).
```

`arg(N, Terme, X)`. Unifie `X` à l'argument numéro `N` de `Terme`.

```
?- arg(1, père(jean, isa), X).
X = jean.
```

Le prédicat *univ* `Terme =.. Liste`. Transforme un terme en une liste :

```
?- père(jean, isa) =.. L.
L = [père, jean, isa].

?- T =.. [a, b, c].
T = a(b, c).
```

`name/2`. Transforme un atome en une liste de ses codes ASCII.

```
?- name(toto, L).
L = [116, 111, 116, 111].

?- name(A, [116, 111, 116, 111]).
A = toto.
```

10.3 Exemples d'utilisation

10.3.1 Addition

```
plus(X, Y, Z) :-
    nonvar(X), nonvar(Y),
    Z is X + Y.
plus(X, Y, Z) :-
    nonvar(Y), nonvar(Z),
    X is Z - Y.
plus(X, Y, Z) :-
    nonvar(X), nonvar(Z),
    Y is Z - X.
```

10.3.2 Algorithme d'unification

Nous décrivons une unification simplifiée qui ne vérifie pas que la variable qu'on substitue n'apparaît pas dans le terme qu'elle remplace. En théorie, `X = f(X)`, par exemple, devrait

produire un échec. En pratique, il n'y a pas de vérification d'occurrence, si bien que cette situation fait exploser la pile.

```
%unify(T1, T2).

unify(X, Y) :-                  % cas de 2 variables
    var(X),var(Y),X = Y.
unify(X, Y) :-                  % cas variable = pas variable
    var(X),nonvar(Y),X = Y.
unify(X, Y) :-                  % cas pas variable = variable
    nonvar(X),var(Y),Y = X.
unify(X, Y) :-                  % cas atome ou nombre =
    nonvar(X), nonvar(Y),       % atome ou nombre
    atomic(X), atomic(Y),
    X = Y.
unify(X, Y) :-                  % cas composé = composé
    nonvar(X), nonvar(Y),
    compound(X), compound(Y),
    termUnify(X, Y).

termUnify(X, Y) :-              % unification terme à terme
    functor(X, F, N),           % de X et Y composés
    functor(Y, F, N),
    argUnify(N, X, Y).

argUnify(N, X, Y) :-            % unification des N
    N>0,                        % arguments de X et Y
    argUnify1(N, X, Y),
    Ns is N - 1,
    argUnify(Ns, X, Y).
argUnify(0, X, Y).

argUnify1(N, X, Y) :-           % unification de
    arg(N, X, ArgX),            % l'argument de rang N
    arg(N, Y, ArgY),
    unify(ArgX, ArgY).
```

11. Prédicats de base de données et du 2nd ordre

11.1 Prédicat de base de données

Les prédicats de bases de données permettent de manipuler les faits et les règles de la base de données.

11.1.1 Le prédicat `assert/1`

`assert(P)` permet d'ajouter P à la base de données.

État de la base à l'instant 1	Le prédicat	État de la base à l'instant 2

`personne(jean).`
`personne(isa).`
	`?- assert(personne(françois)).`	`personne(jean).`
		`personne(isa).`
		`personne(françois).`

En fait, on ne sait pas si `assert/1` ajoute au début ou à la fin des faits. Il existe deux variantes :

- On ajoute au début de la base de données avec `asserta/1`
- On ajoute en fin avec `assertz/1`

On peut aussi ajouter des règles : `assert((P :- B, C, D))`, qui modifient dynamiquement le programme. Ceci peut conduire à des comportements inattendus et il est préférable de l'éviter dans la plupart des cas.

11.1.2 Le prédicat `retract/1`

`retract(P)` permet de retirer P de la base de données.

État de la base à l'instant 2	Le prédicat	État de la base à l'instant 3

`personne(jean).`
`personne(isa).`
`personne(françois).`
	`?- retract(personne(isa)).`	`personne(jean).`
		`personne(françois).`

On peut poursuivre avec une variable :

```
?- retract(personne(X)).
X = jean ;
X = françois ;
No
```

État 4:
`rien`

11.1.3 Le prédicat `abolish/2`

`abolish(Terme, Arité)` permet de retirer tous les termes `Terme` d'arité `Arité`.

```
?- abolish(personne, 2).
```
Retire tous les prédicats « `personne` » d'arité 2.

11.1.4 Les prédicats de chargement de programmes

`consult/1` : Réalise un `assert` des faits et règles du fichier.

`reconsult/1` : Réalise un `abolish` sur les faits et règles du fichier à charger, puis un `consult`. Ce prédicat n'existe pas dans SWI.

L'interprète peut consulter un programme interactivement par une lecture du fichier « `[user]` ». On peut ainsi éviter le passage par un fichier :

```
?- [user].
père(jean, isa).% ajoute ce fait à la base.
^D

?-
```

11.1.5 Le prédicat `clause/2`

`clause(Terme, Corps)` renvoie les corps des clauses pour une tête donnée, par exemple, avec la base :

```
personne(jean).
personne(isa).

père(X, Y) :-
    fils(Y, X),
    masculin(X).

?- clause(père(X, Y),C).
C = (fils(Y, X), masculin(X)).

?- clause(personne(X),C).
X = jean, C = true;
X = isa, C = true.
```

Exemple d'application: la métaprogrammation : Prolog en Prolog.

Prouver un but revient à écrire :

```
prove(But) :- call(But).
```

Ou bien simplement:

```
prove(But) :- But.
```

On peut aussi descendre dans le code et écrire :

```
prove(true).
prove((But1, Buts2)) :-
    prove(But1),
    prove(Buts2).
prove(But) :-
    clause(But, Corps),
    prove(Corps).
```

11.2 Les prédicats du 2nd ordre

Les prédicats du second ordre donnent toutes les solutions à une question. Ce sont `findall/3`, `bagof/3`, `setof/3`.

11.2.1 Le prédicat `findall/3`

11.2.1.1 Exemple

```
élève(françois, 2, info).
élève(isa, 2, info).
élève(françois, 3, instru).

?- findall(X, élève(X, 2, info), B).
B = [françois, isa].

?- findall(X, élève(X, Y, Z), B).
B = [françois, isa, françois].
```

11.2.1.2 Définition du prédicat `findall`/3

```
findall(X, But, XListe):- Tant qu'on trouve des solutions
    But,
    assertz('$sac'(X)),
    fail.
findall(X, But, XListe):- %Lorsqu'il n'y a plus de solution
    assertz('$sac'('$fin')),
    retract('$sac'(X)),
    collecte(X, XListe),
    !.

% prédicat réunissant toutes les solutions
% dans une liste XListe
collecte(X, [X | XListe]):-
    X \== '$fin',
    retract('$sac'(Y)),
    collecte(Y, XListe),
    !.
collecte('$fin',[]).
```

11.2.2 Les prédicats `bagof/3` et `setof/3`

11.2.2.1 Exemples d'utilisation

`bagof/3` et `setof/3` sont des variantes plus élaborées de `findall/3`. Soit la base de données :

```
élève(françois, info, 2).
élève(isa, info, 2).
```

```
élève(françois, info, 3).
élève(paul, instru, 3).

masculin(françois).
masculin(paul).

féminin(isa).

?- bagof(X, élève(X, info, 2), B).
B = [françois, isa].

?- bagof(X, élève(X, info, Y), B).
B =[françois, isa], Y = 2 ;
B = [françois], Y = 3 ;
No.
```

11.2.2.2 Utilisation d'un quantificateur

On peut utiliser le quantificateur quelque soit avec le symbole ^ :

```
?- bagof(X, Y^élève(X, info, Y), B).
B = [françois, isa, françois].

?- bagof(X, élève(X, Y, Z), B).
B = [françois, isa], Y = info, Z = 2 ;
(etc...)

?- bagof(X, Z^Y^élève(X, Y, Z), B).
B = [françois, isa, françois, paul].

?- bagof(X, Y^Z^(élève(X, Y, Z), masculin(X)), B).
B = [françois, françois, paul].
```

`setof/3` est la même chose que `bagof/3`, sauf que la liste est triée et les doublons exclus :

```
?- setof(X,Y^élève(X,info,Y),B).
B = [françois, isa].
```

11.2.2.3 Définition du prédicat *quelque_soit*

Supposons qu'on cherche à prouver une propriété sur un ensemble, par exemple que les élèves d'informatique de 2e année habitent tous à Caen. Ceci peut se traduire logiquement par la démonstration de la formule :

$\forall$ *x, élève(x, info, 2)* $\Rightarrow$ *habite_caen(x).*

On peut appliquer cette formule à tout type d'ensemble *E* :

$\forall x, x \in E \Rightarrow$ *habite_caen(x).*

Il est facile d'écrire un prédicat `quelque_soit(But, Propriété)` en Prolog avec `findall` :

```
quelque_soit(But, Propriété):-
    findall(Propriété, But, Ensemble),
    vérifie(Ensemble).

vérifie([Élément | Elts]):-
    Élément,
    vérifie(Elts).
vérifie([]).
```

Avec la base de données :

```
élève(françois, info, 2).
élève(isa, info, 2).
élève(françois, info, 3).
élève(paul, instru, 3).

masculin(françois).
masculin(paul).

habite_caen(françois).
habite_caen(isa).

?- quelque_soit(élève(X, info, 2), habite_caen(X)).

yes. (ou X = _123 en fait)
```

12. Entrées / Sorties

Le Prolog standard ne connaît que l'ASCII. Les entrées/sorties sont primitives.

12.1 Lire et écrire des caractères

```
?- get0(X).
a ↲

X = 65.

?- put0(65).
a.

?- get0(X).
^D
```

```
X = -1.
```

12.2 Lire et écrire des termes

```
?- read(X).
père(françois, isa).

X = père(françois, isa).
```

`read` renvoie `end_of_file` en fin de fichier.

```
?- T = père(françois, isa), write(T).
père(françois, isa).
```

`?- nl.` insère une nouvelle ligne.

12.3 Ouvrir et fermer des fichiers

Prédicats	Commentaires
`see(fichier).`	Ouverture en lecture du fichier `fichier`. Le fichier devient le flux d'entrée courant.
`see(user).`	Le flux courant redevient l'utilisateur – le clavier.
`see(fichier).`	Rappelle `fichier` à l'endroit où il était s'il n'a pas été fermé. Et il redevient le flux courant.
`seen.`	Fermeture du fichier ouvert en lecture. L'utilisateur devient le flux courant.
`seeing(F).`	`F` s'unifie au fichier en cours.
`tell(fichier).`	Ouverture en écriture du fichier `fichier`. Le fichier devient le flux de sortie courant.
`telling(F)`	`F` s'unifie au fichier en cours.
`tell(user).`	Le flux courant de sortie redevient l'utilisateur – le clavier.
`told.`	Fermeture du fichier ouvert en écriture.

12.4 Exemples

12.4.1 Lecture d'un fichier

```
read_file(File, List) :-
    see(F),
    read_list(List),
```

```
    seen,
    !.

read_list([X | L]) :-
    get0(X),
    X =\= -1,
    !,
    read_list(L).
read_list([]).
```

12.4.2 Copie d'un fichier dans un autre (supposés correctement ouverts)

```
copie:-
    repeat,
      read(X),
      écrire(X),
      X == end_of_file,
    !.

écrire(end_of_file).
écrire(X) :-
    write(X), nl.
```

Définition du prédicat `repeat`:

```
repeat.
repeat :- repeat.
```

13. Style des programmes en Prolog

Les programmes doivent être :

- corrects,
- efficaces,
- lisibles,
- modifiables,
- robustes et
- documentés.

Ceci repose sur une bonne formulation du problème. Ensuite, on peut réaliser un développement incrémental (*stepwise* = pas à pas).
Le prototypage implique deux mouvements :

- Descendant (*penser*)
- Ascendant (*implanter*)

Les clauses doivent être courtes. Les mnémoniques doivent être clairs. La récursivité s'exprime par un ou deux cas triviaux (ou limite) et 1 ou 2 appels récursifs.

Organisation lisible (blancs, tabulations, espaces). Attention aux cut, not. Attention aux `assert` de règles. Attention au « ; » (le *ou* prolog est peu lisible)

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

est préférable à

```
A :- B ; C.
```

Bien commenter les programmes :
- foncteur nom(P_1, …, Pn).
  - type P_1: ...
  - type P_2: ...
- synopsis `(+, +, +,-, -)`, `(+, +,-, -, -)` (avec + pour entrée nécessaire et − pour sortie attendue). `?` désigne une entrée ou une sortie qui peut être ou non instanciée.
- descriptif.

Efficacité : l'ordre des buts compte dans l'exécution. Éviter

```
p1(X), p2(X).
```

avec

p_1: 1000 solutions en 1s
p_2: 1 solution et 1000 heures de calcul.

Bonne solution :

```
p2(X), p1(X).
```

14. Amélioration de programmes

14.1 Lemmes ou mémo-fonctions

Fibonacci a imaginé vers l'an 1200 des suites pour estimer une population de lapins :
- Chaque couple de lapins met au monde une autre couple de lapins tous les mois (gestation d'un mois).
- Les lapins arrivent à maturité sexuelle en un mois

Si on admet que les lapins sont immortels, on peut prédire le nombre de couples de lapins au mois *n* en fonction des couples au mois *n - 1* et *n - 2* :

lapin(n) = lapin(n - 1) + lapin(n - 2)

1re solution (coûteuse)	2e solution (optimisée)

`fib(1, 1).`
`fib(2, 1).`
`fib(M, N) :-`
`  M > 2,`
`  M1 is M - 1, fib(M1, N1),`
`  M2 is M - 2, fib(M2, N2),`
`  N is N1 + N2.`
	`fib(1, 1).`
	`fib(2, 1).`
	`fib(M, N) :-`
	`  M > 2,`
	`  M1 is M - 1, fib(M1, N1),`
	`  M2 is M - 2, fib(M2, N2),`
	`  N is N1 + N2,`
	`  asserta(fib(M, N)).`

Une solution avantageuse consiste à ajouter les faits déjà évalués dans la base de données, ce qui permet d'accélérer le programme.
La forme générique des mémo-fonctions est :

```
lemme(P):-
  P,
  asserta((P:-!)).
```

On introduit le « ! » pour empêcher le retour arrière.

14.2 Récursivité terminale

Une récursivité terminale est quand l'appel récursif apparaît au bout de la dernière règle :

```
f(X) :- X < 100.
f(X) :- B1, B2, X1 is X - 1, f(X1).
```

On peut transformer cette règle en un prédicat déterministe. En effet, `X1` est substituée à `X` dans `f(X1)` et il n'y a pas de possibilité de retour en arrière. La plupart des interprètes Prolog peuvent optimiser les règles avec ce type de récursivité. Ceci permet d'utiliser moins de place sur la pile, et ainsi empêche parfois de la faire exploser.
L'écriture précédente est donc meilleure que celle qui suit où le Prolog ne peut pas détecter l'optimisation possible :

```
f(X) :- B1, B2, X1 is X - 1, f(X1).
f(X) :- X < 100.
```

On peut transformer certains prédicats comme `length/2` en ajoutant une variable pour qu'ils présentent une récursivité terminale :

```
length(Liste, Longueur) :-
    length(Liste, 0, Longueur).

length([], N, N).
length([X | L], N1, N) :-
    N2 is N1 + 1,
    length(L, N2, N).
```

On peut aussi forcer la récursivité terminale de `f(X)` en plaçant un « ! » au bon endroit:

```
f(X) :- B1, B2, X1 is X - 1, !, f(X1).
f(X) :- X < 100.
```