

SOMMAIRE

Introduction.....	5
Chapitre 1 : Prolog, le chaînage-arrière	7
Chapitre 2 : Le traitement des listes	31
Chapitre 3 : La coupure.....	53
Chapitre 4 : Problèmes avec contraintes	89
Chapitre 5 : Les expressions structurées	123
Chapitre 6 : Applications graphiques	171
Chapitre 7 : Un Prolog écrit en Lisp	179
Chapitre 8 : Applications en intelligence artificielle	195

INTRODUCTION

Prolog signifie « programmation logique », ce langage a suscité un engouement extraordinaire dans les années 1980 avant de passer quelque peu de mode. Il reste pourtant étudié dans la plupart des écoles d'ingénieurs et en master informatique. C'est en effet, un moyen de poser les problèmes qui est extrêmement puissant et, pour le programmeur classique, à la fois très déroutant et enrichissant. Et ignorer ce paradigme de programmation serait se cantonner aux premiers temps de l'informatique.

Programmer en Prolog, est en fait, très différent d'une démarche de programmation impérative classique, c'est un langage beaucoup plus évolué permettant de traduire certains des textes d'exercices présents ici, avec une grande simplicité. Il est particulièrement adapté au calcul symbolique et au raisonnement. N'étant pas typé, la liberté est très grande. C'est pourtant un langage difficile, à cause de cette liberté, mais aussi parce que poser correctement un problème est, le plus souvent, précisément la difficulté, et parfois bien plus difficile que de le résoudre (c'est-à-dire exprimer la partie algorithmique), une fois le problème formalisé. Prolog étant réservé aux problèmes de raisonnement, c'est réellement la représentation, c'est-à-dire la façon de choisir les structures de données et les relations intervenant entre elles, qui est le plus délicat.

D'autre part, il convient de bien comprendre la stratégie de fonctionnement de Prolog dans sa recherche des solutions afin de ne pas déclarer des choses, qui quoique correctes, ne sont pas interprétables par Prolog. C'est pourquoi il faut s'entraîner à schématiser l'arbre de recherche sur des exemples simples, en se mettant à la place de Prolog. Malgré ces restrictions, celui-ci est implacable, ce qui signifie qu'en cherchant bien, toutes les réponses qu'il donne aux questions de l'utilisateur s'expliquent, notamment les réponses multiples.

Dans un cours donné sur Prolog, il faut en général attendre la première utilisation interactive en travaux pratiques pour voir les étudiants reconnaître avec enthousiasme le gouffre qui le sépare des autres langages de programmation. Mais une approche de ce langage, se limitant aux principes de base est possible en une douzaine d'heures. La rentabilité pédagogique est évidente, d'autant que l'interrogation des bases de données déductives s'est largement inspirée de ce principe. Ce minimum est constitué des trois premiers chapitres, sans qu'il soit besoin, naturellement, d'en étudier tous les exercices. Les chapitres suivants essaient de montrer dans le détail, toutes les possibilités de Prolog.

Je remercie Michel Zelvelder pour sa relecture et Anne-Christelle Tauty pour ses dessins.

Chapitre 1

Prolog, le chaînage-arrière

Le Prolog est né vers 1975 à Marseille. Conçu par Colmerauer sur les idées de Herbrand (1936) et Robinson (1966) c'est un langage de représentation des connaissances, le mieux connu de la programmation déclarative, qui a influencé le domaine de l'interrogation des bases de données déductives. Le terme de programmation relationnelle serait d'ailleurs plus adapté, dans la mesure où Prolog généralise le paradigme de programmation fonctionnelle. En programmation fonctionnelle (Lisp, Caml, Haskell), comme le nom l'indique, on définit des fonctions et l'essentiel de l'utilisation d'un programme consiste à fournir des données à une fonction renvoyant le résultat. En programmation « relationnelle », l'utilisation peut consister simplement à vérifier ou non une relation entre objets, ou à donner de la même façon « fonctionnelle », l'antécédent pour obtenir l'image, mais aussi lorsque cela est possible, à donner l'image pour obtenir les antécédents, c'est-à-dire « résoudre une équation ». Un certain nombre d'exemples vont montrer qu'une telle relation peut être interrogée de toutes les manières possibles. Mais, naturellement, Prolog ne peut tout faire et ce premier chapitre montre à la fois ce que l'on peut faire et ce qui peut poser problème.

Fonctionnement d'un interpréteur prolog

Programmer en Prolog, c'est énoncer une suite de faits et de règles puis poser des questions. Si tout est fonction en Lisp, tout est relation en Prolog. Tout programme Prolog constitue un petit système-expert à savoir un ensemble de faits et de règles de déduction, qui va fonctionner en « chaînage-arrière » ou « abduction », c'est-à-dire qui va tester les hypothèses pour prouver une conclusion.

On considère une base de règles constituée d'une part de règles sans prémisses : des faits comme *frères (Caïn, Abel)* (à ceci près qu'il faudra utiliser des minuscules) ou non nécessairement clos comme par exemple *égal(X, X)*, et d'autre part, de règles sous forme de « clauses de Horn » comme :

père (X, Y) et père (Y, Z) → grand-père (X, Z).

On écrit ces règles dans le sens de la réécriture, la conclusion nommée « tête » est en premier, celle-ci devant s'effacer afin d'être remplacée par les hypothèses nommées « corps de la clause » :

On écrit donc : C si H_1 et H_2 et ... et H_n , au lieu de H_1 et H_2 et ... et $H_n \rightarrow C$.

Et avec la syntaxe de la plupart des Prolog, ce sera $C :- H_1, H_2, \dots, H_n$.

On pose un but (éventuellement structuré) Q . C'est une question que l'on pose à l'interpréteur, celui-ci va alors chercher à unifier les différentes propositions (faits) de Q avec la conclusion de chaque règle.

Pour cela, il n'y a pas de distinction dans la base de clauses entre les faits et les règles, la conclusion est toujours en tête, les prémisses suivent et si elles ne sont pas présentes, c'est que la « règle » est un fait initial, en quelque sorte un « axiome ».

Dans cette confrontation entre le but à démontrer et la conclusion d'une règle, si par un jeu de substitutions de variables, les deux expressions logiques peuvent être rendues égales, une telle « unification » est possible, alors avec ces mêmes substitutions partout dans Q ainsi que dans les hypothèses, cette conclusion est remplacée par les hypothèses qui pourraient l'entraîner.

Ces substitutions de variables par d'autres termes donnent lieu à ce qu'on appelle des « instances » provisoires de proposition logique.

C'est donc la constitution d'une « résolvante » à chaque nœud de l'arbre de recherche, dans le processus de « résolution », et « l'effacement » au cas où il n'y a plus rien à démontrer.

En Prolog une clause (Q si P_1 et P_2 et ... et P_n) s'interprète comme : pour prouver Q , il faut prouver P_1 , prouver P_2 , etc.

Le point-virgule note la disjonction « ou » $Q :- P ; R$. étant équivalent aux deux règles :

$$Q :- P.$$

$$Q :- R.$$

Prouver signifie « effacer » (unification avec un fait). Prolog ne se contente pas de fournir une telle « preuve », c'est-à-dire une instantiation *ad hoc* des variables, mais va les donner toutes à la demande de l'utilisateur, c'est en cela que l'on parle de non-déterminisme.

Si maintenant le but est structuré, par exemple, si on demande :

$$pere(X, luc), homme(X), age(X, A), A < 18.$$

Pour avoir tous les fils mineurs de Luc, alors Prolog cherchera à « effacer » chacune des propositions de ce but structuré, du premier au dernier, c'est-à-dire donner des instanciations pour X puis pour A .

Exemple

Prenons un exemple très simple où la base de règles est formée de deux faits et d'une règle :

epoux(irma, luc).
pere(luc, jean).
mere(M, E) :- pere(P, E), epoux(M, P).

En clair, on déclare que Irma a pour époux Luc, que Luc est le père de Jean, et que l'on admet la règle : « Si *P* est le père de *E* et si *M* a pour époux ce même *P*, alors *M* est la mère de *E* ». Une variable débute toujours par une majuscule, les constantes telles que *jean, luc...* par des minuscules.

On pose alors la question *mere(M, jean)*. formée par un fait dont l'effacement provoquera une sortie à l'écran de la valeur « *irma* » pour *M*.

mere(M, jean).
 ↓ Règle 3, *E ← jean*
pere(P, jean), epoux(M, P).
 ↓ Règle 2, *P ← luc*
epoux(M, luc).
 ↓ Règle 1, *M ← irma*
X = irma

Interface

Dans les différentes implémentations de Prolog, en général, le programme, c'est-à-dire l'ensemble des clauses devront être écrites dans un éditeur, puis chargées dans l'interpréteur Prolog pour être interprétées, voire compilées. Pour ce faire, il faut placer le programme situé dans *fichier*, dans Prolog en utilisant le prédicat prédéfini *consult(fichier)*. ou *reconsult(fichier)*. ou encore *[fichier].*, (une « reconsultation » pour une seconde fois). Les questions comme tous les exemples qui vont suivre sont alors posées dans Prolog qui donne la première réponse, puis les suivantes grâce au point-virgule.

Naturellement, cela diffère d'un Prolog à l'autre et le « copier-coller » est souvent plus simple. Par ailleurs certains Prolog proposent de coupler avec un autre langage, c'est ce qui est utilisé pour les applications opérationnelles de Prolog dont l'interface-utilisateur est réalisée avec un autre langage.

Syntaxe : constantes et variables

Il y a très peu de choses à savoir : essentiellement la signification des symboles constitués par la virgule, le point-virgule, etc. ; ; . :- [] | !

La première remarque est que toutes les clauses (règles ou faits) se terminent par un point.

La seconde est le signe « :- » qui est le symbole de la réécriture, il se lit « à condition que » ou bien « dès lors que » de gauche à droite, alors que de droite à gauche, il se lirait plutôt comme « implique ». Avec la syntaxe du Turbo-Prolog, on écrit *C if H₁ and H₂ and ... and H_n*, ou bien avec celle du C-Prolog, SWI-Prolog ou de Eclipse, ce sera *C :- H₁, H₂, ... , H_n*.

Voir [J.-P. Delahaye, *Cours de Prolog en Turbo-Prolog*, Eyrolles, 1987].

Le point-virgule « ; » dénote la disjonction, mais aussi, de façon interactive, lorsqu'on pose une question à l'interpréteur Prolog, frapper ce signe lui demande la solution suivante, aussi dans tous les exemples qui suivent apparaît-il entre les différentes solutions, suivies d'un « no » qui signifie qu'il n'y a pas d'autre solution.

L'essentiel sur les atomes est que les variables débutent par une majuscule ou un tiret, par exemple : *A, A1, Xa, _8, _A* ..., le symbole *_* désignant une variable anonyme.

Les constantes débutent toujours par une minuscule.

Les atomes constants sont des chaînes repérées avec des apostrophes ou « quotes » comme *'abc'* ou des minuscules *a, abc, ...* Les termes du Prolog sont donc les constantes et variables et tous les termes structurés par des « foncteurs ». Avant d'examiner ceux-ci au chapitre 5, il faut savoir que les expressions arithmétiques usuelles en sont naturellement. Ainsi les deux expressions structurées *+(3, *(4, 5))* ou *3 + 4*5* en notations préfixe ou infixes sont admissibles.

Les commentaires sont encadrés par */* ... */* ou bien précédés de *%* en Open-Prolog.

Le prédicat *trace(but)*. permet de voir tous les appels et leur résultat logique lors de la résolution d'un but.

Le prédicat *listing*. permet l'affichage des clauses correctement chargées, donc utilisables, dans l'interpréteur. Il a un intérêt au chapitre 6, lorsqu'avec les primitives *retract* et *assert*, un programme peut se modifier lui-même en cours d'exécution.

L'algorithme d'unification

Très généralement dans un système de réécriture, étant données une règle $P \rightarrow Q$ et une expression P' , on voudrait renvoyer l'expression Q' obtenue à partir de Q par substitutions de variables en cas de succès et « faux » en cas d'échec.

Par exemple si on a la règle de réécriture $(a + b)^2 \rightarrow a^2 + 2ab + b^2$; permettant de développer le carré d'une somme, mise en présence de l'expression $(x + 5y^3)^2$, on voudrait obtenir l'expression $x^2 + 2x*5y^3 + (5y^3)^2$. En ce cas où il y a eu un succès pour l'unification, la formule peut être utilisée avec un jeu de substitution de variables où a est remplacé par x et b par $5y^3$, par contre, en présence de $(2x + 3y)^5$, il n'y a pas d'unification possible car 2 est une constante, ce n'est pas 5.

Cet algorithme dû à Robinson et Pitrat se formalise ainsi :

Si P constante et $P' = P$ alors succès $Q' = Q$

Si P constante différente de P' alors échec

Si P variable X et P' constante a ,

alors $Q' = Q[X \leftarrow a]$ c'est-à-dire l'expression Q où X est remplacé par a .

Si P est la variable X et P' la variable Y

alors si Y présente dans Q , on doit renommer Y en Z différente de X et de Y et de toute autre variable figurant dans Q , puis renommage de X en Y , d'où $Q' = Q[Y \leftarrow Z] [X \leftarrow Y]$

Si P et P' sont deux termes débutant par le même symbole de fonction f avec la même arité $P = f(t_1, t_2, \dots, t_k)$ et $P' = f(t'_1, t'_2, \dots, t'_k)$

alors si unification possible pour tous les termes t_i avec une suite de substitutions s , $Q' = Q[s]$ sinon échec.

Si P et P' débutent par le même symbole de prédicat R avec même arité

$P = R(t_1, t_2, \dots, t_k)$ et $P' = R(t'_1, t'_2, \dots, t'_k)$ alors idem

Sinon échec

Ainsi par exemple, en Prolog, $pred(X, Y)$ ne peut s'unifier avec $pred(a, b, c)$ mais peut le faire avec $pred(a, b)$ grâce à la suite de substitutions $s = ((X \leftarrow a) (Y \leftarrow b))$ appelée un « unificateur ». Cette fantaisie est possible pour la raison que Prolog n'est pas typé, c'est un peu ce qui se passe dans les notations traditionnelles où « - » désigne à la fois l'opposé et la soustraction.

Par ailleurs $pred(X, Y)$ s'unifie avec $pred([], 3)$ avec $s = ((X \leftarrow []), (Y \leftarrow 3))$.

$pred(X, Y)$ s'unifie avec $pred(N + 1, [a / L])$ avec les deux substitutions $X \leftarrow N + 1$ et $Y \leftarrow$ la liste $[a / L]$ qui sont des termes structurés, mais il n'y aura pas de calcul effectué, même si N est connu.



Le principe de résolution

Pour montrer l'implication $H \wedge H_2 \wedge \dots \wedge H_n \rightarrow C$, il est équivalent de réfuter son contraire qui s'écrit $H_1 \wedge H_2 \wedge \dots \wedge H_n \wedge \neg C$. Herbrand a montré en 1930 que cela était réalisable en un nombre fini de substitutions amenant à une contradiction. Ce système est complet pour la réfutation.

Une des premières publications en 1973 fut celle de G. Battani et H. Meloui, *Interpréteur du langage de programmation Prolog*. En 1972, Colmerauer mit en œuvre le principe de résolution, constitué d'une seule règle, de Robinson (1965) :

La règle $[(P \text{ ou } R) \text{ et } (Q \text{ ou } \neg R)] \rightarrow (P \text{ ou } Q)$ est vue comme un « effacement » de R , en particulier $[(P \text{ ou } \neg R) \text{ et } R] \rightarrow P$ qui est équivalent au « modus-ponens » $[R \text{ et } (R \rightarrow P)] \rightarrow P$, preuve de P avec l'axiome R et la règle $R \rightarrow P$.

En fait, très simplement, le programme Prolog constitué de l'axiome R et de la règle $P :- R.$, mis en présence de la question P , va le prouver en effaçant R .

L'exploration de toutes les possibilités, backtrack et stratégie standard

Il y a retour en arrière (remontée dans l'arbre) chaque fois que, ou bien toutes les règles ont été examinées sans unification possible, ou bien on arrive à une feuille de l'arborescence donnant un résultat, ou encore lorsqu'une impossibilité est bien notifiée dans la base de règles pour forcer la remontée, c'est « l'impasse » (chapitre 3). Ainsi, si chaque descente dans l'arbre est associée à

une transformation du but et à une « instanciation » des variables, chaque recul correspond à l'annulation de cette transformation.

En fait le but étant examiné de gauche à droite, une seule sortie aura lieu pour l'exemple d'Irma, et cinq pour celui des animaux un peu plus loin.

La stratégie standard est l'ordre de parcours racine-gauche-droite de l'arbre de recherche (encore appelée ordre préfixe ou « recherche en profondeur d'abord » en intelligence artificielle). Cette stratégie est incomplète car en cas de branche infinie, une branche délivrant un succès risque de ne pas être atteinte, ainsi si on demande l'appartenance de a à L où a est une constante et L , l'inconnue, il existe une infinité de solutions L . Aussi dans le programme constitué des deux clauses ordonnées comme $P :- P$, puis P , la seconde clause ne sera jamais examinée puisqu'on tourne en rond sur la première.

On appelle « dénotation » d'un programme Prolog, la théorie engendrée, c'est-à-dire l'ensemble de toutes les conséquences. Cette théorie est finie ou infinie, mais elle n'est pas nécessairement celle de la logique classique ne serait-ce qu'à cause de l'exemple précédent.



Ordre des prémisses

Ce problème assez délicat à appréhender, suivant les questions qui seront posées par l'utilisateur, car pour les définitions :

$$\begin{aligned} \text{ancetre}(X, Y) &:- \text{parent}(X, Y). \\ \text{ancetre}(X, Y) &:- \text{parent}(X, Z), \text{ancetre}(Z, Y). \end{aligned}$$

on aura beaucoup de recherches inutiles lors de la question $\text{ancetre}(X, \text{max})$. L'ordre $\text{ancetre}(Z, X), \text{parent}(X, Z)$ serait plus efficace, mais par contre la situation est inversée pour la question $\text{ancetre}(\text{max}, X)$, puisque Prolog résout de gauche à droite, mais de plus, ancetre appelant ancetre , donnerait lieu à une boucle infinie.

Ce problème peut être résolu, comme on le verra dans l'exemple de la factorielle, en distinguant plusieurs clauses suivant qu'un argument est une variable ou une constante, et ceci grâce à des prédicats prédéfinis tels que $\text{var}(X)$.

Le prédicat « is »

En Prolog on dispose de quelques prédicats prédéfinis tels que : $var(X)$ et $nonvar(X)$, $number(X)$, $integer(X)$, $atom(X)$, ainsi que bien entendu les prédicats infixes $<$, $=<$, $@<$ pour les chaînes...

Il existe bien d'autres prédicats prédéfinis dans les différentes versions de Prolog. Ici, notre but n'est pas de les inventorier, mais au contraire de montrer la puissance du langage en se limitant au maximum à la syntaxe de base. De plus, comme c'est général aux autres langages, les différentes implémentations de Prolog présentent systématiquement des mots réservés légèrement différents pour désigner ces prédicats.

Mais un des principaux prédicats indispensables noté de façon infixé est le « is » (c'est le signe « = » en turbo-prolog).

L'affectation $X \text{ is } E$ s'emploie en principe avec une variable et une expression E , ainsi par exemple : $X \text{ is } 2 * exp(Y)$. L'expression E de droite est alors calculée avant d'être assignée à X . Mais aux questions $4 \text{ is } 3 + 1$. et $4 \text{ is } 3$. Prolog répond oui ou non comme si c'était le prédicat d'égalité. Cette commodité sera utilisée dans le cas où on doit vérifier une égalité sans qu'il y ait besoin d'affecter une « variable » supplémentaire.

Pendant, s'il peut servir d'égalité, ce n'est pas l'égalité symétrique, « is » réalise surtout un effet de bord : une affectation.

Par contre, le signe « = » et aussi « == » en certaines versions est l'égalité des expressions (sans évaluation) ainsi ci-dessous X et Y ne sont pas des expressions égales, pas plus que 7 et $5 + 2$ car en ce cas $7 + 2$ est pris comme le terme structuré $+(7, 2)$.

Exemples

```

7 = 5 + 2. → no
7 is 5 + 2. → yes
X = 7, Y = X + 1. → X = 7 Y = 7 + 1
% Remarquer qu'il n'y a aucun calcul effectué

X = 4, Y is X + 1. → X = 4 Y = 5
is(X, +(4, 2)). → X = 6.
```

Cette dernière écriture, étant celle du prédicat « is » en notation préfixe, ainsi que le « + ».

Ordre des clauses

Ce problème a son importance en cas de définition récursive, par exemple si on veut calculer la factorielle R de N , récursivement R est le produit de N par le résultat nommé RK de la factorielle de $K = N - 1$. Mais si la première clause est toujours examinée avant la seconde, Prolog irait chercher les factorielles de tous les entiers décroissants sans s'arrêter à 0, le programme suivant bouclerait donc indéfiniment :

$$\begin{aligned} \text{fac}(N, R) &:- K \text{ is } N - 1, \text{fac}(K, RK), R \text{ is } RK * N. \\ \text{fac}(0, 1). \end{aligned}$$

Il faut donc inverser les clauses et stopper la recherche quand la première est satisfaite (cela se fera avec une coupure, au chapitre 3), à moins de préciser une condition $0 < N$ précédant les prémisses.

Concernant la première clause (qui doit donc être placée en second), l'ordre des prémisses est impératif ; en effet, il faut d'abord calculer K afin de permettre l'évaluation de sa factorielle RK et seulement après le produit de celle-ci par N .

Vision logique et vision opérationnelle

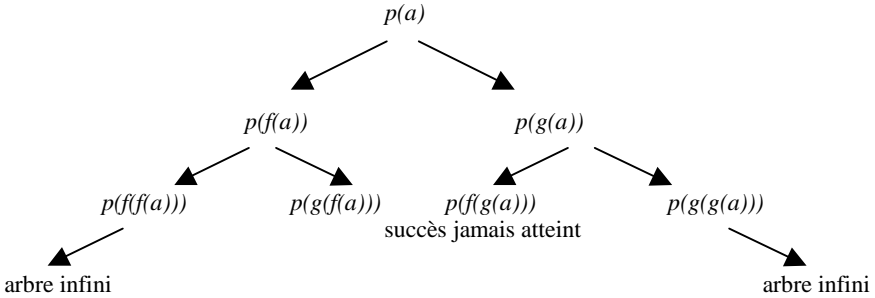
De façon générale, voir Prolog avec une vision logique sémantique plutôt que comme un démonstrateur est malheureusement insuffisant. En effet, Prolog suit une stratégie, la stratégie dite standard « en profondeur d'abord », encore appelée « racine-gauche-droite » et c'est cette vision « opérationnelle » qui compte, ce qui fait que certaines preuves évidentes ne pourront jamais être détectées par Prolog. L'interpréteur Prolog lit et tente l'unification avec la série de clauses qu'on lui a donnée, et ceci dans l'ordre où on lui a donné. C'est pourquoi, une clause récursive qui boucle sur elle-même ne doit pas être placée avant une clause « terminale », sauf à ce que les expressions d'un argument dans ces deux clauses soient incompatibles.

Ainsi par exemple, posant le but $p(a)$ avec le programme :

$$\begin{aligned} &p(f(g(a))). \\ &p(X) :- p(f(X)). \\ &p(X) :- p(g(X)). \end{aligned}$$

La sémantique prouve $p(f(g(a))) \rightarrow p(g(a)) \rightarrow p(a)$

Mais l'ordre des clauses empêchera d'y arriver, car le début de l'arbre de recherche est :



La stratégie « en largeur d'abord » explorant l'arbre étage par étage assurerait le succès, mais serait bien trop inefficace dans l'immense majorité des cas.

Problèmes liés à la transitivité et à la symétrie des relations

Si, dans un problème de graphe, on pose :

```

arc(a, b).
arc(b, c).
arc(b, d).
chemin(X, Y) :- chemin(X, Z), chemin(Z, Y).
  
```

Le programme est logiquement correct mais la question *chemin(c, X)* provoquera encore une branche infinie ; il faut donc remplacer la clause *chemin*, suivant l'exemple des ancêtres ou celui de la factorielle, par les deux clauses :

```

chemin(X, Y) :- arc(X, Y).
chemin(X, Y) :- arc(X, Z), chemin(Z, Y).
  
```

De même, donner des faits *mariés(max, eve)*. et une règle symétrique de la forme *mariés(X, Y) :- mariés(Y, X)*. est logiquement correct mais absolument pas opérationnel car la question *mariés(luc, X)* donnera lieu à une branche infinie. On peut donner alors une solution non récursive avec un second prédicat *époux* ; la même question donnera un échec si Luc ne figure pas dans la base et donnera la réponse dans le cas contraire :

```

époux(X, Y) :- mariés(X, Y).
époux(X, Y) :- mariés(Y, X).
  
```

1 L'inspecteur Maigret

L'inspecteur veut connaître les suspects qu'il doit interroger pour un certain nombre de faits : il tient un individu pour suspect dès qu'il était présent dans un lieu, un jour où un vol a été commis et s'il a pu voler la victime.

Un individu a pu voler, soit parce qu'il était sans argent, soit par jalousie. On dispose de faits sur les vols : par exemple, Marie a été volée lundi à l'hippodrome, Jean, mardi au bar, Luc, jeudi au stade.

Il sait que Max est sans argent et qu'Eve est très jalouse de Marie. Il est attesté par ailleurs que Max était au bar mercredi, Eric au bar mardi et qu'Eve était à l'hippodrome lundi (on ne prend pas en compte la présence des victimes comme possibilité qu'ils aient été aussi voleurs ce jour-là).

Ecrire le programme Prolog qui, à la question *suspect(X)*, renverra toutes les réponses possibles et représenter l'arbre de recherche de Prolog.

Le programme Prolog ci-dessous est très simple. On prend les phrases du texte dans l'ordre où elles viennent, tout en décidant de noms de prédicats les plus explicites possibles. Seule, la première phrase peut présenter une difficulté, car elle doit être décomposée dans les faits d'une présence, d'un vol commis et d'une propriété de susceptibilité qu'un individu *X* soit voleur sur la personne d'une autre *V*.

La difficulté réside aussi souvent dans le nombre, l'ordre, mais surtout la signification des paramètres d'une relation. Ainsi la présence met en jeu un individu *X* en un lieu *L* un jour *J* et un vol attesté met en jeu également trois paramètres, la victime *V*, le lieu *L* et le jour *J*.

Naturellement, il est possible de nuancer et de compliquer à loisir un tel exercice, il y a l'objet du vol, le mobile, certes, ici, très simplifié et bien d'autres choses. Mais si on suit à la lettre l'énoncé, on arrive au programme suivant :

suspect(X) :- present(X, L, J), vol(L, J, V), apuvoler(X, V).

apuvoler(X, _) :- sansargent(X).

apuvoler(X, Y) :- jaloux(X, Y).

vol(hipp, lundi, marie).

vol(bar, mardi, jean).

vol(stade, jeudi, luc).

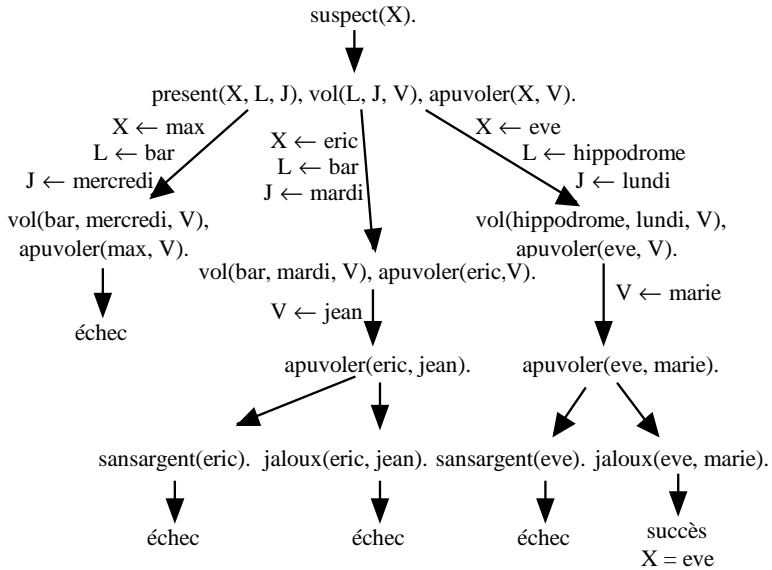
sansargent(max).

jaloux(eve, marie).

present(max, bar, mercredi).

present(eric, bar, mardi).

present(eve, hipp, lundi).



En déroulant l'arbre de recherche, on voit que la question *suspect(X)*. ne peut donner qu'une réponse :

| suspect(X). → X = eve ; no