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**

MCours.com

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 :- $H_1, H_2, \ldots, H_n$.
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 instanciation *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, $E \leftarrow jean$

pere(P, jean), epoux(M, P).

↓ Règle 2, $P \leftarrow luc$

epoux(M, luc).

↓ Règle 1, $M \leftarrow irma$

$X = irma$

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* H_1 *and* H_2 *and ... and* H_n. ou bien avec celle du C-Prolog, SWI-Prolog ou de Eclipse, ce sera C :- H_1, H_2, ... , 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, A*1*, 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 infixe 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 $P = f(t_1, t_2, \ldots t_k)$ et $P' = f(t'_1, t'_2, \ldots t'_k)$
    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
    $P = R(t_1, t_2, \ldots t_k)$ et $P' = R(t'_1, t'_2, \ldots t'_k)$ 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 $H \wedge H_2 \wedge ... \wedge H_n \rightarrow C$, il est équivalent de réfuter son contraire qui s'écrit $H_1 \wedge H_2 \wedge ... \wedge H_n \wedge \neg C$. 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 $[(P$ ou $R)$ et $(Q$ ou $\neg R)] \rightarrow (P$ ou $Q)$ est vue comme un « effacement » de *R*, en particulier $[(P$ ou $\neg R)$ et $R] \rightarrow P$ qui est équivalent au « modus-ponens » $[R$ 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 :

ancetre(X, Y) :- *parent(X, Y).*
ancetre(X, Y) :- *parent(X, Z), ancetre(Z, Y).*

on aura beaucoup de recherches inutiles lors de la question *ancetre(X, max).* L'ordre *ancetre(Z, X), parent(X, Z).* serait plus efficace, mais par contre la situation est inversée pour la question *ancetre(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 *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 infixe est le « is » (c'est le signe « = » en turbo-prolog).

L'affectation *X is E* s'emploie en principe avec une variable et une expression *E*, ainsi par exemple : *X is* 2**exp(Y)*. L'expression *E* de droite est alors calculée avant d'être assignée à *X*. Mais aux questions 4 *is* 3 + 1. et 4 *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.
Cependant, 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 :

```
fac(N, R) :- K is N – 1, fac(K, RK), R is RK*N.
fac(0, 1).
```

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*.

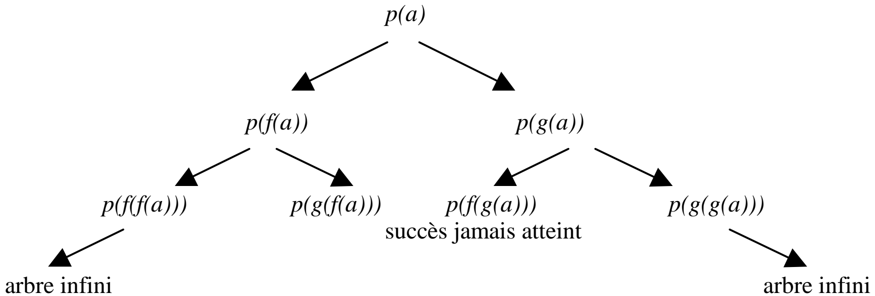
Vision logique et vision opérationnelle

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 :

```
p(f(g(a))).
p(X) :- p(f(X)).
p(X) :- p(g(X)).
```

La sémantique prouve *p(f(g(a)))* → *p(g(a))* → *p(a)*
Mais l'ordre des clauses empêchera d'y arriver, car le début de l'arbre de recherche est :

MCours.com



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).*

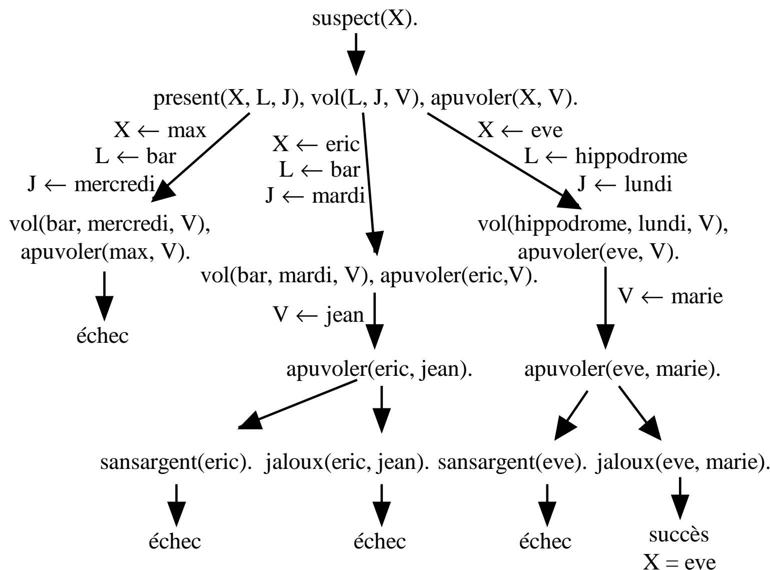
De même, donner des faits *mariés(max, eve).* et une règle symétrique de la forme *mariés(X, Y)* :- *mariés(Y, X).* est logiquement correct mais absolument pas opérationnel car la question *mariés(luc, X)* donnera lieu à une branche infinie. On peut donner alors une solution non récursive avec un second 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

2 Jeu des pièces

Un jeu consiste à mettre trois pièces du même côté en en retournant simultanément deux, et ceci trois fois exactement. On devra demander par exemple *jeu(pile, face, pile)* et les trois modifications devront être affichées. Voir [F. Giannesini, H. Kanoui, R. Pasero, N. Van Caneghem, *Prolog,* Interéditions, 1985]

```
opp(pile, face).
opp(face, pile).

modif(X, Y1, Z1, X, Y2, Z2) :- opp(Y1, Y2), opp(Z1, Z2).
modif(X1, Y, Z1, X2, Y, Z2) :- opp(X1, X2), opp(Z1, Z2).
modif(X1, Y1, Z, X2, Y2, Z) :- opp(Y1, Y2), opp(X1, X2).
```

jeu(X1, Y1, Z1) :- modif(X1, Y1, Z1, X2, Y2, Z2), modif(X2, Y2, Z2, X3, Y3, Z3),
modif(X3, Y3, Z3, R, R, R), aff(X1, Y1, Z1), aff(X2, Y2, Z2),
aff(X3, Y3, Z3), aff(R, R, R).

aff(X, Y, Z) :- write(X), write(' '), write(Y), write(' '), write(Z), nl.

On demandera par exemple :

```
jeu(pile, face, pile). →
pile face pile
pile pile face
pile face pile
face face face
```

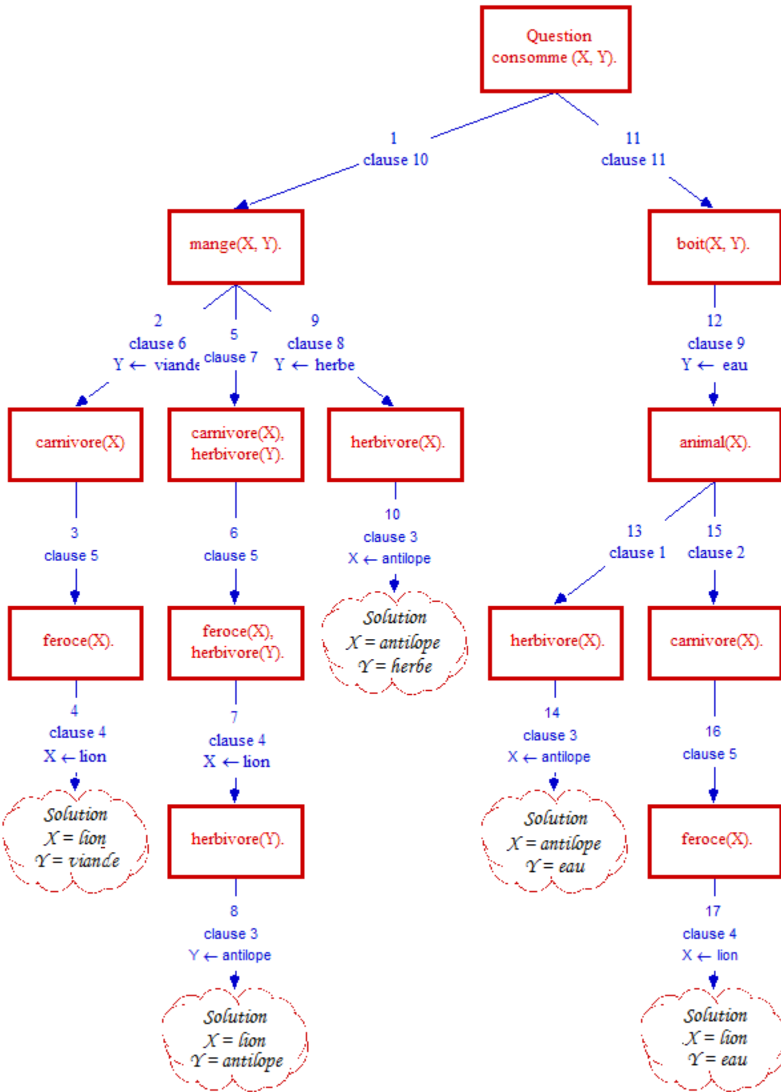
3 Exemple des carnivores
Ecrire les clauses Prolog correspondant au fait que les animaux sont herbivores ou carnivores, l'antilope est un herbivore, le lion est féroce et d'ailleurs tous les animaux féroces sont des carnivores. Les carnivores mangent de la viande et des herbivores, lesquels mangent de l'herbe. Tous boivent de l'eau.
Qui consomme quoi ? Développer l'arbre de recherche.
[J.L. Laurière *Intelligence artificielle, Tomes I et II*, Eyrolles 1986, 1987].

En suivant exactement l'ordre de l'énoncé, nous avons :

1 *animal(X) :- herbivore(X).*
2 *animal(X) :- carnivore(X).*
3 *herbivore(antilope).*
4 *feroce(lion).*
5 *carnivore(X) :- feroce(X).*
6 *mange(X, viande) :- carnivore(X).*
7 *mange(X, Y) :- carnivore(X) , herbivore(Y).*
8 *mange(X, herbe) :- herbivore(X).*
9 *boit(X, eau) :- animal(X).*
10 *consomme(X, Y) :- mange(X, Y).*
11 *consomme(X, Y) :- boit(X, Y).*

L'arbre de recherche de Prolog se parcourt dans le sens « racine-gauche-droite » en suivant les numéros des clauses qui sont marquées ici, exceptionnellement, de 1 à 11, le cheminement se faisant de 1 à 17 sur le schéma. On posera le but suivant qui donne 5 solutions que sont les feuilles de l'arbre ci-après.

consomme(X, Y). → X = lion Y = viande ; X = lion Y = antilope ; X = antilope Y = herbe ; X = antilope Y = eau ; X = lion Y = eau ; no



Remarque, la clause *animal(X) :- herbivore(X) ; carnivore(X).* résume deux règles grâce au point-virgule, et on pourrait faire de même pour *consomme*.

4 Opérations sur une base de données
Etant donnée une relation notée *rel* à 3 arguments, définir la relation *pr* « projection » de *rel* sur les 2 premiers arguments, puis la « sélection » *sr* des objets dont les deux premiers arguments vérifient une propriété *prop*. Si *r* et *s* sont deux relations à deux arguments, définir leur « jointure » *jrs* comme l'ensemble des triplets *(x, y, z)* vérifiant *r(x, y)* et *s(x, z),* enfin leur union comme l'union des couples de *r* et de *s*.

La projection d'une relation à trois arguments sur les deux premiers est l'ensemble des couples *(X, Y)* tels qu'un triplet au moins vérifie la relation *rel.* C'est donc la définition mathématique aussi bien que celle utilisée en base de données.

```
pr(X, Y) :- rel(X, Y, _).
sr(X, Y, Z) :- rel(X, Y, Z), prop(X, Y).
jrs(X, Y, Z) :- r(X, Y), s(X, Z).
urs(X, Y) :- r(X, Y).
urs(X, Y) :- s(X, Y).
```

5 Mythologie germanique
Au commencement était Ginungap ou le chaos, au nord était l'amas de glaces Niffelheim, et au sud le Muspelheim embrasé. De ces deux contraires naquirent Ymer ancêtre de tous les géants et la vache Audumbla qui engendra Bure père de Bôr. Ce dernier épousa la géante Bestla qui le rendit père d'Odin (Wotan), de Vil et de Vé. Odin tua Ymer, dont le sang provoqua le fameux déluge, et grâce à Frigga, engendra Thor (la guerre), Balder (la lumière), Braga (la sagesse), Heimdal (sentinelle). Thor eut deux fils Mod (le courage) et Magni (la force).
Compléter éventuellement en introduisant les Valkyries, Hilda, Mista, Rota, des Elfes, Trolls et autres Nornes, puis faire l'arbre généalogique, et définir des relations diverses telles que oncle, grand-oncle, cousin, etc.

```
parent(P, E) :- pere(P, E); mere(P, E).
homme(H) :- pere(H, _).

pere(niffelham, ymer).
mere(muspelheim, ymer).
mere(muspelheim, audumbla).
pere(ymer, G) :- geant(G).
homme(ymer).
pere(niffelham, audumbla).
mere(audumbla, bure).
```

pere(bure, bor).
mere(bor, vil).
homme(vil).
mere(bor, ve).
mere(bor, odin).
pere(odin, bilder).
pere(odin, thor).
pere(thor, mod).
pere(thor, magni).

oncle(O, N) :- *parent(P, O), homme(O), parent(P, E), parent(E, N).*

> oncle(*O*, bure). % recherches inutiles pour la même raison que les ancêtres
>
> → *O* = ymer ; *O* = ymer ; no
> oncle(vil, *X*). → *X* = bilder ; *X* = thor ; no

Mais attention, poursuivre ce genre de parentés ne peut se faire qu'avec le prédicat « différent », sinon, le père serait considéré comme oncle !
C'est le cas pour :
gdoncle(O, N) :- *parent(P, O), homme(O), parent(P, E), parent(E, PE), parent(PE, N).*

> gdoncle(*O*, mod). → *O* = vil ; *O* = odin
> Alors que :
> gdoncle(ymer, *N*). → *N* = bor

ancetre(P, E) :- *parent(P, E).*
ancetre(A, D) :- *parent(A, E), ancetre(E, D).*

On pourrait utiliser *var(A)* ou *nonvar(A)* pour découper la seconde en deux clauses afin de limiter les recherches inutiles, mais le maniement de ces prédicats est délicat. Voir plus loin l'exemple de la factorielle comme relation pouvant s'interroger dans les deux sens.

> ancetre(muspelheim, *D*).
> → *D* = ymer ; *D* = audumbla ; *D* = bure ; *D* = bor ; *D* = vil ; *D* = ve ; *D* = odin ; *D* = bilder ; *D* = thor ; *D* = mod ; *D* = magni ; no
>
> ancetre(*A*, mod).
> → *A* = thor ; *A* = niffelham ; *A* = bure ; *A* = odin ; *A* = muspelheim ; *A* = audumbla ; *A* = bor ; no

6 Rencontres

Eve est une petite femme blonde qui désire rencontrer un homme, Irma est une brune mesurant 1m55 favorable à tout homme qui veut bien d'elle. Julie la rousse mesure 1m65 et cherche un homme plus grand qu'elle. Carmela est une blonde de 1m59 qui ne sait pas ce qu'elle veut.
Luc fait 1m70, est très attiré par une rousse, mais ne sait plus son prénom. Max adore les petites femmes brunes. Marc mesure 1m90 et aimerait aussi rencontrer une brune, Hector cherche une petite blonde. En admettant que *petit* signifie moins de 1m60, peut-on les aider ?

On prend les phrases, dans l'ordre où elles sont données, en prenant garde que *taille* est un prédicat à deux arguments et que, pour simplifier, le désir de chacun soit exprimé par une liste de conditions évaluables, par exemple pour les tailles.
A la fin, le prédicat *possible* est construit comme traduisant deux désirs compatibles, mais il faut éviter que la recherche de Prolog tourne en rond, simplement en précisant le sexe des arguments.

```
femme(eve).
petit(eve).
cheveux(eve, blond).
voudrait(eve, H) :- homme(H).
femme(irma).
cheveux(irma, brun).
taille(irma, 155).
voudrait(irma, H) :- homme(H), voudrait(H, irma).
femme(julie).
taille(julie, 165).
cheveux(julie, roux).
voudrait(julie, H) :- taille(julie, TJ), homme(H), taille(H, TH), TJ < TH.
femme(carmela).
cheveux(carmela, blond).
taille(carmela, 159).
homme(luc).
taille(luc, 170).
voudrait(luc, F) :- femme(F), cheveux(F, roux).
homme(max).
voudrait(max, F) :- femme(F), petit(F), cheveux(F, brun).
homme(marc).
taille(marc, 190).
voudrait(marc, F) :- femme(F), cheveux(F, brun).
homme(hector).
```

voudrait(hector, F) :- femme(F), petit(F), cheveux(F, blond).
petit(X) :- taille(X, T), T < 160.
possible(H, F) :- homme(H), voudrait(H, F), voudrait(F, H).

Utilisation

> possible(H, F).
> → H = luc F = julie ; H = max F = irma ; H = marc F = irma ;
> H = hector F = eve ; no
> Comme on le voit, il y a juste un petit conflit, qui peut facilement se régler, si on arrive à décider Carmela.

7 Division entière
On cherche le quotient entier et le reste de la division de A par B.

divise(A, B, 0*, A) :- A* < *B.*
divise(A, A, 1*,* 0*).*
divise(A, B, Q, R) :- B < *A, AS is A − B, divise(AS, B, QS, R), Q is QS* + 1.

> divise(23, 7, Q, R). → $Q = 3$ $R = 2$
> divise(58, 3, Q, R). → $Q = 19$ $R = 1$

8 PGCD de deux entiers
L’algorithme d’Euclide peut se faire par divisions successives ou, ce qui revient au même, par soustractions successives entre le plus grand et le plus petit, jusqu’à obtenir deux entiers identiques.

Le PGCD de deux nombres identiques est lui-même, sinon, dans l’hypothèse où $X < Y$, c’est le même que celui du plus petit X avec la différence des deux, la troisième clause renvoyant à ce cas. Ainsi, par soustractions successives, on peut montrer que cet algorithme aboutit obligatoirement au résultat.

pgcd(X, X, X).
pgcd(X, Y, Z) :- X < *Y, YS is Y – X, pgcd(X, YS, Z).*
pgcd(X, Y, Z) :- Y < *X, pgcd(Y, X, Z).*

> pgcd(12, 16, X). → $X = 4$
> pgcd(12, 20, X). → $X = 4$
> pgcd(27, 16, X). → $X = 1$

9 Combinaisons
Faire une programmation de complexité linéaire pour le coefficient C_n^p. Dessiner l'arbre de résolution de *comb(*5, 3 *X).* par la méthode naïve puis par celle-ci en comptant le nombre de nœuds de chacune.

La relation de Pascal (hors les bords du triangle de Pascal) $C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$ donnerait lieu à un arbre de recherche exponentiel où la plupart des calculs sont refaits plusieurs fois, c'est pourquoi, pour donner un arbre de recherche réduit à une branche, il vaut mieux utiliser la relation de récurrence $C_n^p = (n.C_{n-1}^p) / p$.
On a donc dans l'ordre, pour ce calcul, à poser $M = N - 1$ et $Q = P - 1$, appeler le calcul de C_M^Q puis le multiplier par N et le diviser par P.

```
comb(_, 0, 1).
comb(N, P, R) :- 0 < N, M is N – 1, Q is P – 1, comb(M, Q, S), R is N*S / P.
```

comb(6, 3, *R*). → *R* = 20
comb(7, 4, *R*). → *R* = 35
comb(8, 4, *R*). → *R* = 56

10 Suite de Fibonacci
Programmer de manière astucieuse pour ne pas refaire les mêmes calculs la suite $u_0 = u_1 = 1$, $u_{n+2} = u_{n+1} + u_n$. L'arbre de recherche doit être réduit à une branche de $N + 1$ nœuds pour l'appel de *fib(N, R).*
Par exemple *fib(*5, *X).* → $X = 8$ avec 7 nœuds.

De la même façon que dans l'exemple précédent, la relation de récurrence donnée donne lieu à une complexité exponentielle des calculs. Aussi, faut-il prendre en compte deux paramètres U et V représentant deux termes consécutifs de la suite et, si le but n'est pas atteint, renvoyer le problème aux deux termes consécutifs suivants qui sont V et $W = U + V$.
De cette manière, *fibo*, s'il ne termine pas, provoque une réécriture en *fibo*, laquelle entraîne une autre réécriture de *fibo* et ainsi de suite, le nombre « d'appels » étant de l'ordre du rang N demandé, on a donc une complexité linéaire.

Ce qui est possible ici, car Prolog n'est pas typé, est de prendre le même nom de prédicat *fibo* pour le prédicat principal appelé avec *N*, devant fournir le résultat *R* et pour le prédicat auxiliaire qui lui, ayant quatre arguments, ne peut être unifiable au premier aucune confusion ne peut donc se produire.

```
fibo(N, R) :- fibo(R, 0, 1, N).

fibo(R, _, R, 0).
fibo(R, U, V, N):- 0 < N, M is N – 1, W is U + V, fibo(R, V, W, M).
```

fib(5, X). → X = 8
fib(6, X). → X = 13
fib(7, X). → X = 21
fibo(20, X). → X = 10946
fibo(40, X). → X = 165580141

Cette suite bien connue peut rendre compte, par exemple, sous réserve d'une grande régularité, du nombre de couples de lapins obtenus, de mois en mois, à partir d'un seul couple de lapins.

11 Suite de Fibonacci (suite)
Soit la suite fibonaccienne d'ordre 3 définie par $u_0 = 2$, $u_1 = –3$, $u_2 = –3$ et la relation de récurrence $u_{n+3} = 4u_{n+2} – u_{n+1} – 6u_n$.
On ne demande pas de traduire ces quatre clauses telles quelles, mais un programme de 5 clauses dont la première *u(N, R) :- aux(R, 2, –3, –3, N).* sera vraie dès lors que *R* est le terme de la suite *u* de rang donné *N*. L'exécution du programme par le but *u(N, R).* où *N* est donné doit toujours donner lieu à une branche, donc de complexité linéaire en *N*.
Remarque, à titre de vérification, les formules de Binet donnent :
$u_n = 2(–1)^n – 3*2^n – 3^{n+1}$, l'équation caractéristique ayant trois racines simples réelles.

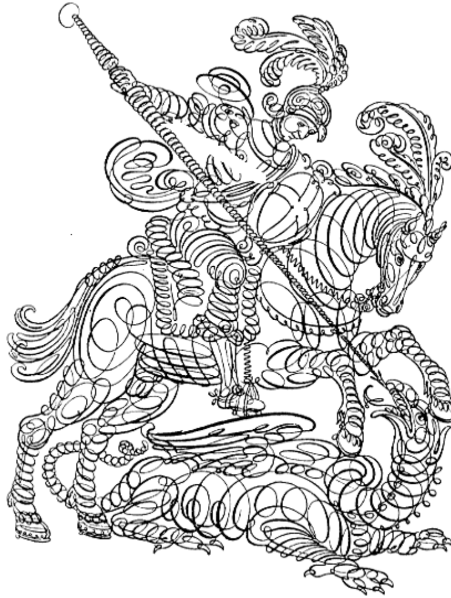
Là encore, lorsqu'on demande la valeur *R* de la suite pour un rang donné *N*, le jeu des réécritures appellera *aux* un nombre de fois de l'ordre de *N*, soit une complexité linéaire.

```
u(N, R) :- aux(R, 2, –3, –3, N).

aux(R, R, _, _, 0).
aux(R, _, R, _, 1).
aux(R, _, _, R, 2).
aux(R, U, V, W, N) :- 2 < N, M is N – 1, X is 4*W – V – 6*U,
          aux(R, V, W, X, M).
```

Exemples

u(0, X). → $X = 2$
u(1, X). → $X = -3$
u(2, X). → $X = -3$
u(3, X). → $X = -21$
u(4, X). → $X = -63$
u(5, X). → $X = -213$

Ci-dessous, Leonardo Fibonacci de Pise terrassant l’explosion combinatoire grâce à des paramètres supplémentaires.



12 Tours de Hanoï

Prolog résout magistralement les problèmes pouvant se résoudre par un algorithme décrit récursivement. Il s'agit, dans ce fameux problème, de déplacer une tour formée de *N* disques de diamètres échelonnés (le plus petit en haut), d'un emplacement dit *gauche* à un autre dit *droite*, en se servant d'un troisième (*milieu*). La règle est de ne jamais placer un disque sur un autre dont le diamètre serait plus petit.

Le problème va se programmer par un seul prédicat qui réalise les déplacements et un autre qui se contente de l'appeler au départ.
Le prédicat *hanoi* associé au nombre *N* de disques permet de démarrer.
Le prédicat *mouv* traduit récursivement ce qu'il faut faire comme mouvements.

L'explication de *mouv* est transparente ; pour déplacer les *N* disques de *A* sur *C*, il faut déplacer les $N - 1$ premiers sur *B*, puis le dernier (qui est le plus grand) sur *C*, et enfin les $N - 1$ qui ont été placés sur *B*, les transporter sur *C*. (2^{N-1} transports en tout). Pour bien comprendre, il faut faire à la main la suite des réécritures pour $N = 2$ par exemple.
Le prédicat prédéfini *write* n'accepte qu'un seul argument en C-prolog, il faut donc en écrire plusieurs et se servir du retour à la ligne *nl*.
Le programme s'écrit alors :

```
mouv(0, _, _ , _).

mouv(N, A, B, C) :-   K is N – 1, mouv(K, A, C, B),
                      write('transport de '), write(A), write(' sur '), write(C), nl,
                      mouv(K, B, A, C).

hanoi(N) :- mouv(N, gauche, milieu, droite).
```

Exemple

```
hanoi(3).
→
transport de gauche sur droite
transport de gauche sur milieu
transport de droite sur milieu
transport de gauche sur droite
transport de milieu sur gauche
transport de milieu sur droite
transport de gauche sur droite
yes
```

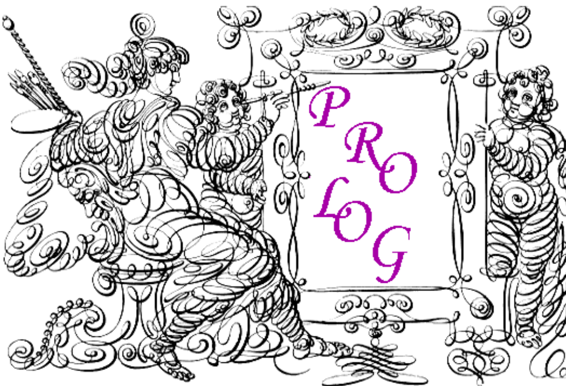
13 Factorielle en tant que relation

On souhaite reprendre la fonction factorielle de façon à pouvoir l'interroger dans les deux sens, trouver par exemple l'antécédent de 24 qui est 4 mais ne pas en trouver pour 25. Il existe un prédicat prédéfini *integer(X)* disant si *X* est un entier, de même qu'un prédicat *var(X)* indiquant qu'il s'agit d'une variable.

On reprend la relation de base liant 0 et 1, puis la définition récursive construisant *R* à partir de *N*, enfin, dans le cas où au contraire *R* est donné, on se sert d'un prédicat auxiliaire *aux* qui va effectuer une suite de quotients entiers. Cette suite s'achève par un succès consistant à trouver l'entier 1 ou bien n'est pas résolue.

```
fac(0, 1).
fac(N, R) :- integer(N), 0 < N, K is N – 1, fac(K, F), R is N*F.
fac(N, R) :- integer(R), aux(1, R, N).
aux(N, 1, N).
aux(K, R, N) :- 1 < R, NK is K + 1, NR is R//NK, R is NK*NR, aux(NK, NR, N).
```

> fac(6, *R*). → R = 720 ? ; no
> fac(*N*, 6). → *N* = 3 ? ; no
> fac(*N*, 7). → no
> fac(N, 1). → *N* = 0 ? ; *N* = 1 ? ; no
> fac(4, *R*). → *R* = 24 ? ; no
> fac(*N*, 25). → no
> fac(*N*, 5040). → *N* = 7 ? ; no
> fac(*N*, 5041). → no



Chapitre 2

Le traitement des listes

La liste est une structure de données très utilisée dans un langage admettant la récursivité. Ce n'est pas du tout un tableau à accès direct mais un enchaînement de doublets d'adresses formées par le premier élément et la suite de la liste, deux termes auxquels on accède très rapidement. Et cette structure permet donc de traduire facilement beaucoup de traitements définis récursivement. Il s'agit en fait d'un cas particulier de terme structuré (voir plus loin les arbres).

Le traitement des listes en Prolog

Avec la symbolique commune à la plupart des versions de Prolog, on écrit [ ] pour la liste vide, [*a, b, c, d*] pour une liste définie par énumération de tous ses objets, et [*a* | *X*] si *a* en est le premier élément et *X* la liste de ce qui suit, ou encore par exemple [*a, b, c* | *X*] si l'on distingue les trois premiers éléments.
La barre verticale est l'opérateur « cons » de construction de listes, il n'a rien à voir avec la concaténation ou enchaînement des listes, appelée plus loin *conc* ou *concat*.
A la gauche de *cons* se trouve un nombre fini d'éléments quelconques qui seront placés au début du résultat, et à sa droite se trouve nécessairement une liste, celle devant être complétée.

Ainsi [*a* / [*b, c, d*]] = [*a, b* / [*c, d*]] = [*a, b, c* / [*d*]] = [*a, b, c, d* / []] = [*a, b, c, d*].

Ce qui surprend les habitués d'un langage de programmation classique est qu'en Prolog, grâce au travail d'unification, il est possible d'écrire des expressions structurées. Ainsi pour les listes, le fait d'écrire une liste [*X* / *Q*] suffit pour que dans toute la clause où cette expression apparaît, *X* désigne la tête et *Q* la queue de cette liste qui n'a pas obligatoirement besoin d'être nommée.
Dans les premiers Prolog, l'opérateur de construction de liste était le point, la liste [*a, b, c*] était donc notée *a.b.c.nil* ou encore *a.L* si *L* est la liste [*b, c*] définie antérieurement sous le nom de *L* (*nil* est le nom traditionnel de la liste vide). Le prédicat prédéfini : *name(X, L)* qui relie un atome *X* à la liste *L* de ses codes Ascii sera utilisé dans quelques exercices.

1 Appartenance
Le prédicat indiquant si oui ou non un objet appartient à une liste (au premier niveau simplement) se définit par induction sur la construction de la liste.

En rappelant que le symbole _ signifie un objet quelconque, il va s'unifier avec le premier élément du second argument. La première clause est un axiome disant que *X* appartient à toute liste qui débute par *X*. La seconde clause indique que *X* appartient à une liste si elle appartient à la queue de cette liste (condition suffisante). Il est inutile de rajouter une clause indiquant que *X* quelconque ne peut appartenir à l'ensemble vide étant donné qu'une requête du type *app(X,* []*).* ne peut s'unifier avec la tête d'aucune des deux clauses ; le résultat sera *no*.

app(X, [*X* / _]*).*
app(X, [_ / *L*]*) :- app(X, L).*

> Exemples :
> app(a, [c, b, a]). → yes
> app(d, [c, b, a]). → no
> app(*X*, [c, b, a]). → *X* = c ; *X* = b ; *X* = a ; no

2 Ecrire une liste
Affichages successifs des éléments d'une liste.

On se sert ici d'un prédicat prédéfini *write(...)* qui ne possède qu'un seul argument, qui est toujours évalué à vrai dans le travail de Prolog, mais qui a juste l'utilité de provoquer un effet de bord lors de son évaluation, à savoir un affichage.

ecrire([]).
ecrire([X | L]) :- *write(X), write(' '), ecrire(L).*

> Exemple
> ecrire([a, b, c, d]). → a b c d yes

ecrire([]).
ecrire([X | L]) :- *write(X), write(" "), ecrire(L).*

> ecrire([a, b, c, d]). → a[32]b[32]c[32]d[32] yes
> En effet "*A*" désigne la liste des codes Ascii soit [65], alors que '*A*' sera la lettre A, 32 étant le code Ascii de l'espace.

3 Etre une liste
Le prédicat « être une liste » doit se définir avec deux clauses.

Il s'agit de deux axiomes qui seront vérifiés par filtrage avec toute expression entre crochets, vide ou non, donc avec des listes et seulement avec des listes.

liste([]).
liste([_ / _]).

4 Préfixes d'une liste
Définir les clauses nécessaires à *prefix(P, L).* où *P* est une sous-liste commençante de *L*, par exemple que *prefix([a, b, c], [a, b, c, d, e]).* est vrai.

La seconde clause, examinée par Prolog, consistera à avancer simultanément d'un cran dans les deux arguments tant que c'est possible, pour constater qu'ils débutent par les mêmes objets.

prefix([], _).
prefix([X / L], [X / M] :- *prefix(L, M).*

Exemples
prefix([a, d], [a, b, c, d]). → no
prefix([a, b], [a, b, c, d]). → yes
prefix(*P*, [a, b, c, d]). → *P* = [] ; *P* = [a] ; *P* = [a, b] ; *P* = [a, b, c] ;
P = [a, b, c, d] ; no

5 Suffixes d'une liste
Définir les clauses nécessaires à *suffixe(S, L).* où S est une sous-liste finissante de *L*, par exemple que *suffixe([d, e], [a, b, c, d, e]).* est vrai.

suffixe(L, L).
suffixe(S, [_ / L]) :- *suffixe(S, L).*

suffixe([c, d], [a, b, c, d]). → yes
suffixe([a, d], [a, b, c, d]). → no
suffixe(*S*, [a, b, c, d]).
→ *S* = [a, b, c, d] ; *S* = [b, c, d] ; *S* = [c, d] ; *S* = [d] ; *S* = [] ; no

6 Dernier élément d'une liste
Construire les prédicats *dernier(X, L).* où *X* est le dernier élément de *L*, et *zajouter(X, L, R).* où *R* est la liste *L* augmentée de *X* en dernière position.

```
last(X, [X]).
last(X, [_ / Y]) :- last(X, Y).

zajouter(X, [], [X]).
zajouter(X, [Y / Z], [Y / T]) :- zajouter(X, Z, T).
```

Ou bien en utilisant la concaténation :

```
last(X, L) :- conc(_, [X], L).
zajouter(X, L, R) :- conc(L, [X], R).
```

Exemples

> last(*X*, [a, b, c, d]). → *X* = d
> zajouter(z, [a, b, c, d], *R*). → *R* = [a, b, c, d, z]
> zajouter(*X*, [a, b, c], [a, b, c, d]). → *X* = d
> zajouter(*X*, L, [a, b, c, d]). → *X* = d *L* = [a, b, c]
> zajouter(d, [a, b], [a, b, c, d]). → no

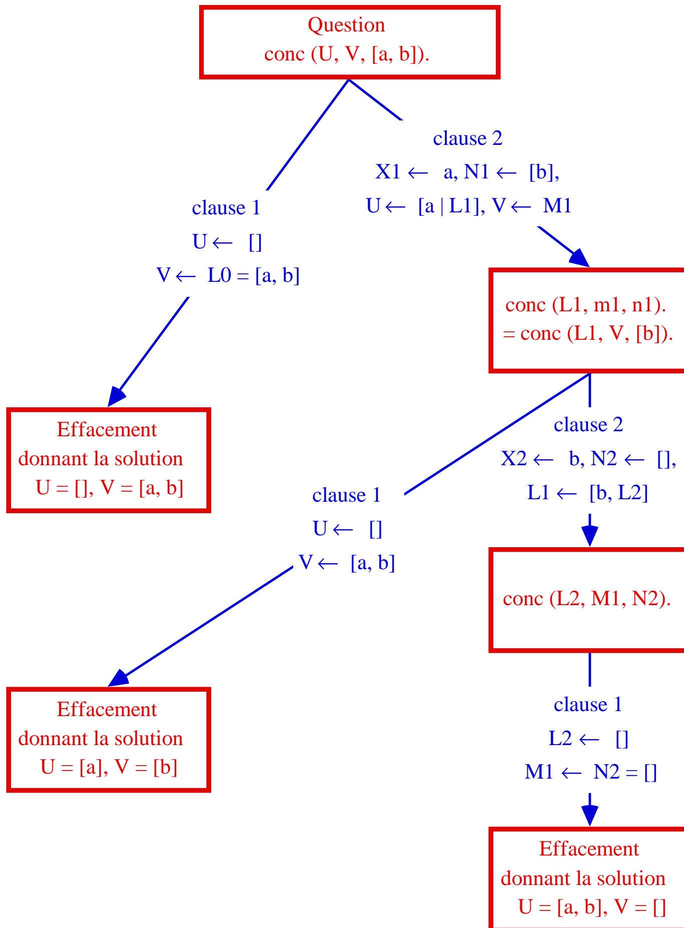
7 Concaténation
Définir les clauses nécessaires à la concaténation des listes, par exemple il faut que la relation *conc([a], [b, c], [a, b, c]).* soit vraie et que *conc(X, [a], [b]).* soit impossible.
Dessiner l'arbre de résolution de *conc(U, V, [a, b]).* en marquant sur chaque arc les instanciations des variables.

Deux clauses suffisent, l'une constituant l'axiome terminal et l'autre une règle récursive.
Remarquons ici qu'elles sont intervertibles car les unifications du premier argument sont incompatibles, soit cet argument est vide, soit c'est une liste contenant au moins un *X*.

```
conc([], L, L).
conc([X / L], M, [X / N]) :- conc(L, M, N).
```

MCours.com

Comme au chapitre précédent, il est extrêmement instructif de suivre à la trace le travail de remplacement des buts par Prolog en parcourant l'arbre en profondeur d'abord, ce qui permet de voir que les solutions sont données dans l'ordre [], [*a, b*] puis [*a*], [*b*] et enfin [*a*, *b*], [].



8 Retrait
Définir le prédicat *ret(X, L, R).* où *R* est la liste *L* dans laquelle la première occurrence de *X* est effacée.

Pour le retrait de la première occurrence d'un élément dans une liste, il est possible de prévoir que le retrait d'un absent échoue (première version) ou bien ne change rien, cas de la seconde version.

ret(X, [X | L], L).
ret(X, [Y | L], [Y | M]) :- *ret(X, L, M).*

> ret(a, [f, a, d, a], *R*). → *R* = [f, d, a] ; R = [f, a, d] ; no
> ret(u, [a, b, c], *R*). → no
> Il n'est pas possible de retirer u d'une liste qui ne la contient pas en ce cas.

Seconde version :

ret(_, [], []).
ret(X, [X | L], L).
ret(X, [Y | L], [Y | M]) :- *ret(X, L, M).*

> ret(a, [f, a, d, a], *R*). → *R* = [f, d, a] ; *R* = [f, a, d] ; *R* = [f, a, d, a] ; no
> ret(u, [a, b, c], *R*). → *R* = [a, b, c]
> Ici, on a choisi de laisser invariante une liste à qui on retire un élément ne lui appartenant pas.

Maintenant pour retirer toutes les occurrences d'un élément d'une liste.

ret(_, [], []).
ret(X, [X | L], R) :- *ret(X, L, R).*
ret(X, [Y | L], [Y | M]) :- *ret(X, L, M).*

> ret(a, [f, a, d, a], *R*).
> → *R* = [f, d] ; *R* = [f, d, a] ; *R* = [f, a, d] ; *R* = [f, a, d, a] ; no
> Ce qui est parfaitement logique, car rien n'indique dans la troisième clause que *X* soit différent de *Y*, aussi, en forçant la poursuite de l'exploration, toutes les solutions possibles pour *R* sont calculées. Pour y remédier, il faut utiliser une coupure (chapitre suivant).
> ret(u, [a, b, c], *R*). → *R* = [a, b, c]

9 Sous-liste consécutive
C'est la relation qui peut exister entre un « morceau » ou « segment » d'une liste, pris d'un seul tenant à l'intérieur de la liste.

Le fait pour *SL* d'être une sous-liste consécutive de *GL* se traduit par l'existence de listes *A* et *B* telles que *GL* soit la concaténation de *A*, *SL*, et *B*, alors *A* et *B* sont notées avec le symbole « _ » qui signifie ici « une liste quelconque » et *D* représente la concaténation de *A* avec *SL*. Ici, l'ordre des prémisses est impératif afin de ne pas tomber dans une recherche infinie.

souliste(SL, GL) :- conc(D, _, GL) , conc(_, SL, D).

> souliste([c, d], [a, b, c, d, e]). → yes
> souliste([c, e], [a, b, c, d, e]). → no
>
> On demandera toutes les sous-listes de [a, b, c] par souliste(*X*, [a, b, c]).
>
> souliste(*X*, [a, b, c]).
> → *X* = [] ; *X* = [a] ; *X* = [] ; *X* = [a, b] ; *X* = [b] ; *X* = [] ; *X* = [a, b, c] ; *X* = [b, c] ; *X* = [c] ; *X* = [] ; no

10 Sous-liste pure
Le premier argument doit être une suite extraite du second, c'est-à-dire que l'ordre est conservé, par exemple *sliste([a, c, d, e], [a, b, c, d, e, f]).* doit être vrai.

En parcourant la « sur-liste » constituée par le second argument, arrivé à *X*, soit cet *X* fait partie de la sous-liste (seconde clause), soit il n'en fait pas partie (troisième clause). Il y a là, avec ces deux dernières clauses, l'expression d'un choix, c'est-à-dire que Prolog essaie la première puis la suivante.

Le « backtracking », ou essais avec retour en arrière, s'exprime tout simplement par ce choix.

sliste([], _).
sliste([X / Q], [X / L]) :- sliste(Q, L).
sliste(S, [_ / L]) :- sliste(S, L).

Exemples

> sliste([a, c, d], [a, b, c, d, e, f]). → yes
>
> sliste(X, [a, b, c]).
> → *X* = [] ; *X* = [a] ; *X* = [a,b] ; *X* = [a, b, c] ; *X* = [a, b] ; *X* = [a] ; *X* = [a, c] ; *X* = [a] ; *X* = [] ; *X* = [b] ; *X* = [b, c] ; *X* = [b] ; *X* = [] ; *X* = [c] ; *X* = [] ; no

Ou bien, utilisant la concaténation, deux clauses suffisent alors :

sliste([X | L], GL) :- *conc(D, B, GL), conc(_, [X], D), sliste(L, B).*
sliste([], _).

11 Eléments consécutifs d'une liste
consec(X, Y, L). doit être vrai dès lors que *X* et *Y* sont consécutifs dans cet ordre au sein d'une liste *L*.

consec(X, Y, L) :- *souliste([X, Y], L).*

Ou bien :
consecutifs(X, Y, [X, Y | _]).
consecutifs(X, Y, [_ | L]) :- *consecutifs(X, Y, L).*

Ou encore une seule clause signifiant que *L* est la concaténation de trois listes, un préfixe non nommé, la liste *[X, Y]* et un suffixe anonyme lui aussi.

consec(X, Y, L) :- *conc(_, F, L), conc([X, Y], _, F).*

> consecutifs(*X*, *Y*, [a, b, c, d, e]).
> → *X* = a *Y* = b ; *X* = b *Y* = c ; *X* = c *Y* = d ; *X* = d *Y* = e ; no
>
> mais aussi :
>
> consecutifs(e, *Y*, [a, b, c, a, a, d, a, e]). → no
> consecutifs(a, *Y*, [a, b, c, a, a, d, a, e]). → *Y* = b ; *Y* = a ; *Y* = d ; *Y* = e ; no
> consecutifs(*X*, a, [a, b, c, a, a, d, a, e]). → *X* = c ; *X* = a ; *X* = d ; no

12 Rang d'un élément dans une liste
On convient que le premier élément d'une liste a pour rang zéro.

rg(X, 0*, [X / _]).*
rg(X, M, [_ / L]) :- *rg(X, K, L), M is K* + 1.

Fonctionnera, également dans le sens de la demande du *N*-ième élément d'une liste, ainsi par exemple *rg(X,* 2*, [a, b, c]).* répondra *X* = *c*.

Erreurs classiques *rg(X, N* + 1*, [_ / L])* :- *rg(X, N, L).* répond 0 + 1 + 1 pour le rang de *c*, et :
rg(X, N, [_ / L]) :- *rg(X, N* – 1*, L).* répond *no*.

Quant à l'inversion *M is K* + 1*, rg(X, NK, L),* elle provoque une erreur.

Exemples

rg(a, *R*, [a, b, a, c, a, d]). → *R* = 0 ; *R* = 2 ; *R* = 4 ; no
rg(*X*, 3, [a, b, c, d]). → *X* = d ; no
rg(e, *R*, [a, b, c, d]). → no
rg(*X*, *R*, [a, b, c]). → *X* = a R = 0 ; *X* = b *R* = 1 ; *X* = c *R* = 2 ; no
rg(a, *X*, [f, a, d, a]). → *X* = 1 ; *X* = 3 ; no % deux solutions

13 Insertion
Insérer *X* à la *N*-ième place dans une liste *L*, ici la convention est que le premier élément d'une liste a le rang 1.

ins(X, 1*, L, [X / L]).*
ins(X, N, [Y / L], [Y / M]) :- *ins(X, K, L, M), N is K* + 1.

Exemples

ins(x, 3, [a, b, c, d, e, f, g, h], R). → *R* = [a, b, x, c, d, e, f, g, h] ; no
ins(*X*, 3, [a, b, c, d, e, f, g, h], [a, b, x, c, d, e, f, g, h]). → *X* = x ; no
ins(x, *N*, [a, b, c, d, e, f, g, h], [a, b, x, c, d, e, f, g, h]). → *N* = 3 ; no
ins(*X*, *N*, [a, b, c, d, e, f, g, h], [a, b, x, c, d, e, f, g, h]). → *X* = x *N* = 3 ; no
ins(c, 3, [a, b, c, d, e, f, g, h], [a, b, x, c, d, e, f, g, h]). → no

14 Extraction d'un élément *X* de rang *N* d'une liste *L*.

*extraire(*0*, [X | _], X).*
extraire(N, [_ | L], X) :- *P is N* – 1*, extraire(P, L, X).*
% différence avec rang (exercice 12)

> extraire(0, [a, b], *X*). → *X* = a ; no
> extraire(3, [a, b, c, d], *X*). → *X* = d ; no
> extraire(3, [a, b], *X*). → no
> % car aucune clause ne permet d'extraire de la liste vide
> extraire(1, [a, b], b). → yes
> extraire(*N*, [a, b, c], *X*). → *N* = 0 *X* = a ; puis erreur
> extraire(*N*, [a, b], b). → erreur
> % mêmes remarques que pour le rang

15 Le compte est bon
Le problème consiste à décomposer un entier *S* donné en choisissant des nombres dans une liste *L* donnée.

Il serait maladroit d'utiliser le prédicat *sliste* précédent (exercice 10) pour vérifier qu'une sous-liste d'une liste de nombres donnés admet *S* pour somme ! C'est un des exemples les plus simples de « backtracking » où, précisément il n'y a rien à programmer, puisque Prolog fait le travail de recherche de toutes les solutions ; ce travail de recherche avec essai de *X* qui se déduit de *S*, et retour en arrière, se trouve dans le retrait *ret* de la troisième clause de *compte*.

Ajoutons que le « backtracking » est ce qui se fait de plus difficile en algorithmique et est assez laborieux dans les langages moins évolués.
Le troisième argument est la liste des nombres choisis.
Noter que « compte » est défini avec trois arguments et passe la main à un autre « compte » qui a quatre arguments. Cela n'est bien sûr pas recommandé, mais amusant dans un langage non typé !

ret(X, [X | L], L).
ret(X, [Y | L], [Y | M]) :- *ret(X, L, M).*

compte(S, L, R) :- *compte(S, L, [ ], R).*
*compte(*0*, _, R, R).*
compte(S, L, LC, R) :- *ret(X, L, LS), X =< S, T is S – X, compte(T, LS, [X | LC], R).*

compte(7, [1, 2, 3, 4, 5, 6], *R*).
→ *R* = [4, 2, 1] ; *R* = [2, 4, 1] ; *R* = [6, 1] ; *R* = [4, 1, 2] ; *R* = [1, 4, 2] ; *R* = [5, 2] ; *R* = [4, 3] ; *R* = [2, 1, 4] ; *R* = [1, 2, 4] ; *R* = [3, 4] ; *R* = [2, 5] ; *R* = [1, 6] ; no

compte(7, [2, 3, 5, 2, 1], *X*).
→ *X* = [2, 3, 2] ; *X* = [5, 2] ; *X* = [3, 2, 2] ; *X* = [2, 2, 3] ; *X* = [2, 2, 3] ; *X* = [2, 5] ; *X* = [2, 5] ; *X* = [3, 2, 2] ; *X* = [2, 3, 2] ; *X* = [5, 2] ; no

compte(4, [4, 1, 2, 3], *X*).
→ *X* = [4] ; *X* = [3, 1] ; *X* = [1, 3] ; no

compte(512, [33, 49, 72, 84, 58, 93, 68, 25, 16, 5, 77, 34], *R*).
→ *R* = [34, 5, 16, 68, 93, 58, 84, 72, 49, 33] avec autres permutations

16 Filtre suivant une étoile
Il s’agit d’établir que deux listes sont identiques aux étoiles près, qu’elles soient égales après retrait de toutes les étoiles. Mais naturellement, ce serait maladroit de retirer les étoiles dans chacune pour ensuite comparer les résultats, il est bien plus simple de les parcourir simultanément.
On veut par exemple que *equ([a, *, *, b, *, c], [*, a, b, *, c, *]).* soit vrai.

Les clauses correspondent à toutes les situations qui peuvent arriver en confrontant deux listes où l’on avance simultanément (cas de la seconde clause) ou séparément. Etant devant deux symboles différents et différents de l’étoile, aucune clause ne dit ce qu’il faut faire, il y a donc échec.

equ([], []).
equ([X, | L], [X | M]) :- equ(L, M).
equ([| L], M) :- equ(L, M).*
equ(L, [|M]) :- equ(L, M).*

equ([a, *, *, b, *, c], [*, a, b, *, c, *]). → yes
equ([a, *, b, c], [a, b, c, *, c]). → no

equ(*X*, [a, b]).
→ *X* = [a, b] ; *X* = [a, b, *] ; *X* = [a, b, *, *] ; *X* = [a, b, *, *, *] ; *X* = [a, b, *, *, *, *] ; *X* = [a, b, *, *, *, *, *] ; *X* = [a, b, *, *, *, *, *, *] ; *X* = [a, b, *, *, *, *, *, *, *] [Execution aborted]
Ce qui est normal, puisqu’il y a une infinité de solutions.

17 Filtre où * dans *L* filtre une sous-liste consécutive non vide de *M*
Par exemple *filtre([a, a, *, b, *, c, *], [a, a, a, b, c, b, i, i, e, c, a, b, c]).* renverra vrai. Il se peut qu'il y ait plusieurs solutions, ce qui explique que cette valeur de vérité vraie soit renvoyée plusieurs fois.

La première clause est la condition d'un bon arrêt de la recherche pour Prolog, la seconde permet d'avancer simultanément dans les deux listes lorsqu'elles possèdent la même tête.

La troisième clause indique que l'étoile peut filtrer un seul élément de la seconde liste, mais la dernière clause indique qu'elle peut filtrer plus qu'un élément. Les deux seront utilisées dans une exploration complète.

```
filtre([], []).
filtre([X | L], [X | M]) :- filtre(L, M).
filtre([* | L], [_ | M]) :- filtre(L, M).
filtre([* | L], [_ | M]) :- filtre([* | L], M).
```

Exemples

filtre([*, c], [a, b]). → no
filtre([a, a, *, b, *, c, *], [a, a, a, b, c, b, i, i, e, c, a, b, c]). → yes

La question suivante montre parfaitement toutes les possibilités de filtrage d'une étoile à une sous-suite consécutive :

filtre(*X*, [a, b, c]).
→ *X* = [a, b, c] ; *X* = [a, b, *] ; *X* = [a, *, c] ; *X* = [a, *, *] ; *X* = [a, *] ;
X = [*, b, c] ; *X* = [*, b, *] ; *X* = [*, *, c] ; *X* = [*, *, *] ; *X* = [*, *] ;
X = [*, c] ; *X* = [*, *] ; *X* = [*] ; no

18 Moyenne d'une liste de nombres

Le prédicat *moy* reliant une liste *L* de nombres et sa moyenne *M* va appeler le prédicat *moyenne* qui va, de réécriture en réécriture, balayer la liste *L* en sommant à chaque fois la somme partielle *S* et en comptabilisant le nombre *N* d'éléments parcourus.

C'est l'exemple type d'une suite d'appels, donc de réécritures, de complexité linéaire, obtenue en donnant comme paramètres de la relation *moyenne*, tous ceux dont on a besoin en cours d'exécution du calcul.

```
moy(L, N) :- moyenne(L, 0, 0, N).
moyenne([], N, S, M) :- M is S / N.
moyenne([X | L], N, S, M) :- SS is S + X, NS is N + 1, moyenne(L, NS, SS, M).
```

> moy([12, 14], *M*). → *M* = 13
> moy([11, 12], *M*). → *M* = 11 % car il s'agit ici de la division entière
> moy([3, 0, 5, 8, 4, 9, 1, 7, 2, 6, 10], *M*). → *M* = 5

19 Inversion d'une liste

C'est l'opération miroir devant donner [*c*, *b*, *a*] pour [*a*, *b*, *c*]. Une première solution grossière est possible en se servant de la concaténation « conc ».

```
inverse([], []).
inverse([X | L], M) :- inverse(L, N), conc(N, [X], M).
```

On peut assez facilement calculer que pour chaque appel d'inverse une concaténation doit avoir lieu à la fin du résultat de l'inverse, ce qui fait une complexité quadratique des calculs, c'est dire que le temps de réponse sera quadruplé si on double la longueur de la liste, et multiplié par 9 si on la triple.
Une meilleure programmation utilise un troisième paramètre tampon et deux prédicats.
Faire l'essai de *inv* en partant de *inv([a, b, c], [], R).* Cette fois, la complexité du programme est linéaire, c'est-à-dire que le temps d'exécution est de l'ordre de la taille de la liste.
On pourrait même ici utiliser le même nom *inv* puisque l'unification se fera avec deux ou avec trois arguments de manière exclusive.

```
inv([], R, R).
inv([X | D], T, R) :- inv(D, [X | T], R).
inverse(L, R) :- inv(L, [], R).
```

Exemple (en suivant à la trace, on voit Prolog renommer les variables, entrer sur les différents appels et en sortir avec échec ou succès) :

> trace(inverse([a, b, c, d], *R*)).
> →
> • 1 | 1 call inverse([a, b, c, d], _568)
> • 2 | 2 call inv([a, b, c, d], [], _568)
> • 3 | 3 call inv([b, c, d], [a], _568)
> • 4 | 4 call inv([c, d], [b, a], _568)
> • 5 | 5 call inv([d],[c, b, a], _568)
> • 6 | 6 call inv([],[d, c, b, a], _568)
> • 6 | 6 exit inv([],[d, c, b, a], [d, c, b, a])
> • 5 | 5 exit inv([d], [c, b, a], [d, c, b, a])
> • 4 | 4 exit inv([c, d], [b, a], [d, c, b, a])
> • 3 | 3 exit inv([b, c, d], [a], [d, c, b, a])
> • 2 | 2 exit inv([a, b, c, d], [], [d, c, b, a])
> • 1 | 1 exit inverse([a, b, c, d], [d, c, b, a])
> → R = [d, c, b, a] yes

Comme la relation est symétrique, on peut être tenté de poser la question inverse :
inverse(*R*, [*a, b, c, d*]). → *R* = [*d, c, b, a*]
mais c'est un arbre infini et une pile pleine car le début de la trace est :

> • 1 | 1 call inverse(_60, [a, b, c, d])
> • 2 | 2 call inv(_60, [], [a, b, c, d])
> • 3 | 3 call inv(_1158, [_1157], [a, b, c, d])
> • 4 | 4 call inv(_1461, [_1460, _1157], [a, b, c, d])
> • 5 | 5 call inv(_1773, [_1772, _1460, _1157], [a, b, c, d])
> • 6 | 6 call inv(_2094, [_2093, _1772, _1460, _1157], [a, b, c, d])
> • 6 | 6 exit inv([], [a, b, c, d], [a, b, c, d])
> • 5 | 5 exit inv([a], [b, c, d], [a, b, c, d])
> • 4 | 4 exit inv([b, a], [c, d], [a, b, c, d])
> • 3 | 3 exit inv([c, b, a], [d], [a, b, c, d])
> • 2 | 2 exit inv([d, c, b, a], [], [a, b, c, d])
> • 1 | 1 exit inverse([d, c, b, a], [a, b, c, d])

Mais ensuite, Prolog va chercher des seconds arguments (le tampon *T*) de 5 puis 6 ... éléments.

20 Former un mot avec des lettres données
Avec un dictionnaire représenté par une liste de listes, par exemple *D = [[e, n, s, i, i, e], [s, u, p, e, l, e, c]…]* et une liste *L* de lettres, construire la relation indiquant s'il est possible de former un mot de *D* avec un sous-ensemble de *L*, ainsi : *former([b, a, c], [[a, a], [a, b, c, d], [c, e, a]], X). → no*

```
former(L, [T | Q], T) :- partie(T, L).
former(L, [_ | Q], X) :- former(L, Q, X).

partie([X | L], E) :- retir(X, E, F), partie(L, F).
partie([], _).
retir(X, [X | L], L).
retir(X, [Y | L], [Y | R]) :- retir(X, L, R).
% tous les retraits possibles et toutes les permutations de parties possibles
```

former([b, a, c], [[a, a], [a, b, c, d], [c, e, a]], X). → no
former([a, e, i, i, n, e, s, t], [[e, n, s, i, i, e], [s, u, p, e, l, e, c]] , X).
→ X = [e, n, s, i, i, e]

21 Représentation creuse d'un polynôme
Un polynôme peut se représenter comme liste de couples (coefficient, degré) ; ainsi le polynôme $3x^9 + 7x^2 - 4x^4 + 3$ sera représenté par [[3, 9], [7, 2], [–4, 4], [3, 0]]. Ecrire la relation *puissance(X, N, V)* vérifiée si $X^N = V$. Ecrire la relation *valeur(X, P, Y)* vérifiée si et seulement si *Y = P(X), P* étant une représentation de polynôme.

```
puiss(X, 0, 1).
puiss(X, N, V) :- K is N – 1, 0 =< K, puiss(X, K, U), V is X*U.

valeur(_, [], 0).
valeur(X, [[A, N] | P], Y) :- puiss(X, N, V), valeur(X, P, U), Y is U + A*V.
```

Exemples

puiss(_, 0, *P*) → *P* = 1
puiss(2, 9, *P*). → *P* = 512
puiss(7, 2, *P*). → *P* = 49
valeur(3, [[1, 2], [–5, 1], [6, 0]], *V*). → *V* = 0
valeur(2, [[3, 9], [7, 2], [–4, 4], [3, 0]], *V*). → *V* = 1503
valeur(2, [[1, 4], [1, 3], [1, 2], [1, 1], [1, 0]], *V*). → *V* = 31

22 Liste des éléments de rang pair ou impair d'une liste
Si *LI* est la liste des éléments de rangs impairs de *L*, il faut par exemple obtenir *LI* = [*a, c, e*] pour *L* = [*a, b, c, d, e*].

```
impairs([], []).
impairs([X], [X]).
impairs([X, Y | L], [X | M]) :- impairs(L, M).

pairs([], []).
pairs([_], []).
pairs([_, Y | L], [Y | M]) :- pairs(L, M).
```

> impairs([a, b, c, d, e, f, g, h, i, j, h, k], *R*).
> → *R* = [a, c, e, g, i, h] ; no
>
> Inversement, deux possibilités pour :
>
> impairs(*R*, [a, b, c, d]).
> → *R* = [a, _1706, b, _1711, c, _1716, d] ;
> R = [a, _1706, b, _1711, c, _1716, d, _1721] ; no
>
> Où on voit que Prolog invente des noms de variables pour constituer une liste dont la liste des éléments de rangs impairs est donnée.

23 Somme des éléments de *N* en *N* d'une liste de nombres

Dans ce qui suit, *SP* désigne la somme partielle des éléments déjà comptabilisés, *R* désigne le résultat final et *K* est un compteur qui indique une addition pour *SP* s'il est à *N* avec retour à 1.

```
som([], _, _, R, R). % liste épuisée
som([X | LR], N, N, SP, R) :- SS is SP + X, som(LR, 1, N, SS, R).
% ajout si rang N
som([X | LR], K, N, SP, R) :- K < N, KS is K + 1, som(LR, KS, N, SP, R).
% avancée si K < N
```

> som([1, 2, 3, 0, 4, 5, 3, 2, 5, 1, 1, 7, 0, 3, 8, 5], 5, 5, 0, R). → *R* = 12
> som([1, 2, 3, 0, 4, 5, 3, 2, 5, 1, 1, 7, 0, 3, 8, 5], 3, 3, 0, R). → *R* = 10
> som([1, 2, 3, 0, 4, 5, 3, 2, 5, 1, 1, 7, 0, 3, 8, 5], 2, 2, 0, R). → *R* = 25

24 Automates finis

On représente les mots sur un alphabet A par des listes, ainsi le mot « *abaa* » par $[a, b, a, a]$ sur $A = \{a, b\}$.
Un automate sur A est un ensemble d'états $Q = \{q_0, q_1, ...\}$ ayant un état initial q_0, un (ou plusieurs) état final, et une relation de transition donnée par des triplets (q, x, q') où $x \in A$ et $q, q' \in Q$. Un mot m est reconnu par l'automate s'il existe une suite de transitions de l'état initial à un état final au moyen des lettres de ce mot.
a) Ecrire les clauses générales pour cette reconnaissance.
b) Ecrire les clauses particulières décrivant l'automate à deux états q_0, q_1, et la transition définie par $tr(q_0, a) = q_1$, $tr(q_1, b) = q_0$ avec q_0 à la fois initial et final.
c) Décrire l'automate reconnaissant les mots contenant le sous-mot « *iie* » sur l'alphabet $A = \{a, e, i, o, u\}$.

```
reconnu(M) :- init(E), continue(M, E).

continue([], E) :- final(E).
continue([A | M], E) :- trans(E, A, F), continue(M, F).

init(q0).            % automate du petit b

final(q0).
trans(q0, a, q1).
trans(q1, b, q0).    % il reconnaît les mots de la forme (ab)^n
```

Exemples

> reconnu([a, b, a, b, a, b, a, b]). → yes
> reconnu([a, b, b, b, a, a]). → no
>
> reconnu(*M*). → *M* = [] ; *M* = [a, b] ; *M* = [a, b, a, b] ;
> *M* = [a, b, a, b, a, b] ; *M* = [a, b, a, b, a, b, a, b] ;
> *M* = [a, b, a, b, a, b, a, b, a, b] ; *M* = [a, b, a, b, a, b, a, b, a, b, a, b] ; etc.

```
init(q0).            % maintenant l'automate du petit c

trans(q0, i, q1).
trans(q1, i, q2).
trans(q2, e, q3).
trans(q2, i, q2).
```

*trans(q*0*, V, q*0*) :- app(V, [a, e, o, u]).*
*trans(q*1*, V, q*0*) :- app(V, [a, e, o, u]).*
*trans(q*2*, V, q*0*) :- app(V, [a, o, u]).*
*trans(q*3*, V, q*3*) :- app(V, [a, e, i, o, u]).*
*% ou trans(q*3*, _, q*3*). mais autres lettres*
*final(q*3*).*

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

Exemples

> reconnu([a, a, i , i, i, o, i, i, e, e, a, u, u, o, i, a]). → yes
> reconnu([a, a, i, e, i, i, o, i, a]). → no

25 Produit cartésien
Pour deux listes, le résultat *R* doit être l'ensemble de tous les couples possibles d'un élément de la première avec un élément de la seconde.
Ainsi par exemple la demande *pc(*[a, b], [0, 1, 2], *R).* doit renvoyer « l'ensemble produit » *R* = [[a, 0], [a, 1], [a, 2], [b, 0], [b, 1], [b, 2]]).

pc([], _, []).
pc(_, [], []).

pc([X | Y], Z, U) :- distrib(X, Z, V), pc(Y, Z, W), conc(V, W, U).

distrib(X, [], []).
distrib(X, [A | B], [[X, A] | U]) :- distrib(X, B, U).

Ce n'est pas le « dist » de l'exercice qui suit sur les parties, ici, on construit des couples en distribuant chaque *X* du premier ensemble sur le second argument. Noter que cet exercice utilise la concaténation.

> pc([a, b], [0, 1, 2], [[a, 0], [a, 1], [a, 2], [b, 0], [b, 1], [b, 2]]). → yes
>
> pc([a, b, c, d], [0, 1, 2, 3], *R*).
> → *R* = [[a, 0], [a, 1], [a, 2], [a, 3], [b, 0], [b, 1], [b, 2], [b, 3], [c, 0],
> [c, 1], [c, 2], [c, 3], [d, 0], [d, 1], [d, 2], [d, 3]]

26 Ensemble des parties de *p* éléments
Les parties de *{x}* ∪ *L* à *p* éléments sont celles de *L* à *p* éléments et aussi celles de *p* – 1 éléments de *L* auxquelles on rajoute *x*.

```
partie([X | E], P, [X | L]) :- K is P – 1, partie(E, K, L).
partie(E, P, [_ | L]) :- partie(E, P, L).
partie([], 0, _).
```

> partie(*E*, 3, [a, b, c, d]).
> → *E* = [a, b, c] ; *E* = [a, b, c] ; *E* = [a, b, d] ; *E* = [a, c, d] ; *E* = [b, c, d]
> % Il faudra une coupure (chapitre suivant), si on ne veut pas de répétitions

27 Ensemble des parties d'un ensemble
Les parties de *{x}* ∪ *L* sont celles de *L* et aussi les mêmes auxquelles on rajoute l'élément *x*.

Les deux clauses de « parties » suivent exactement la démonstration par récurrence sur le cardinal de l'ensemble des parties. Si *P* constitue la liste des parties de *L*, le fait de rajouter un élément *X* conduit à considérer ces parties (celles n'ayant pas *X*) et toutes celles obtenues en « distribuant » *X* à chacune de ces parties pour former la liste *Q*. « dist » réalise ce qui est connu en langage Lisp comme un « mapcar cons *X* ».

```
parties([], [[]]).
parties([X | L], R) :- parties(L, P), dist(X, P, Q), concat(P, Q, R).

dist(X, [Y], [[X | Y]]).
dist(X, [Y | L], [[X | Y] | P]) :- dist(X, L, P).

concat([], L, L).
concat([X | L], M, [X | R]) :- concat(L, M, R).
```

> parties([a, b, c], *R*). → *R* = [[], [c], [b], [b, c], [a], [a, c], [a, b], [a, b, c]]
>
> parties([a, b], [[], [a], [b], [a, b]]). → yes
>
> parties(*X*, [[], [b], [a], [a, b]]). → *X* = [a, b] ; puis, la conception ne permet l'interrogation que dans l'autre sens, alors pile pleine !!!

28 Liste des diviseurs d'un entier
Pour les avoir dans l'ordre on commence par l'entier *N* lui-même puis, en décroissant, on teste l'éventuel suivant qui serait *N* / 2 (// est la division entière en Prolog) puis on descend de 1 en 1 jusqu'à 1 qui est nécessairement le plus petit.

La liste *LD* des diviseurs de *N* est remplie petit à petit de *N* jusqu'à 1. C'est ce que fait *divbis* en plaçant 1 au début de la liste lorsqu'il a fini son office. Les deux clauses principales rajoutent ou ne rajoutent pas l'éventuel diviseur *D* à *LD* et passent au nouveau diviseur potentiel *ND*.

Voir au chapitre 3 à peu près la même chose avec les nombres parfaits.
Noter que pour le quotient dénommé *Q*, *Q* = 0 est une vérification que l'on peut racourcir en 0 *is N mod Q*.

```
diviseurs(N, LD) :- D is N // 2, divbis(N, D, [N], LD).

divbis(N, D, LD, R) :- Q is N mod D, Q = 0, DS is D – 1, 0 < DS,
                       divbis(N, DS, [D | LD], R).

divbis(N, D, LD, R) :- Q is N mod D, 0 < Q, DS is D – 1, 0 < DS,
                       divbis(N, DS, LD, R).

divbis(_, _, R, [1 | R]).
```

Exemples

diviseurs(91, *R*). → *R* = [1, 7, 13, 91]

diviseurs(24, *R*). → *R* = [1, 2, 3, 4, 6, 8, 12, 24]

diviseurs(48, *R*). → *R* = [1, 2, 3, 4, 6, 8, 12, 16, 24, 48]

diviseurs(5040, *R*). → *R* = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 28, 30, 35, 36, 40, 42, 45, 48, 56, 60, 63, 70, 72, 80, 84, 90, 105, 112, 120, 126, 140, 144, 168, 180, 210, 240, 252, 280, 315, 336, 360, 420, 504, 560, 630, 720, 840, 1008, 1260, 1680, 2520, 5040]

29 L'argot des bouchers des Halles
Cette façon de parler était construite par la règle suivante : si la première lettre d'un mot est une consonne, cette consonne est remplacée par la lettre « l » et se place au début d'un suffixe choisi au hasard dans le cas où ce dernier commence par une voyelle.
Si le suffixe commence par une consonne, la consonne débutant le mot est perdue. Les suffixes sont *em, ji, oc, muche…*
Ainsi « boucher » va donner loucherbem (abrégé en « louchebem » mais ne pas considérer ce genre d'inflexion), « patron » va donner « latronmuche », jargon, « largonji » etc. Construire le prédicat *argot(M, A).* renvoyant une traduction *A* en argot du mot *M*.
On représente les mots par la liste de leurs lettres, ainsi :
*argot([*b, o, u, c, h, e, r*], A).* → *A* = *[*l, o, u, c, h, e, r, b, e, m*]*.
On vérifiera que *random(X)* est un prédicat qui affecte *X* par un réel au hasard entre 0 compris et 1 non compris, et qu'à trois arguments dont les deux premiers sont donnés, *X* est unifié à un entier au hasard entre le premier et le deuxième argument entier non compris, par exemple : *random(*3, 25*, X).* → *X* = 19
On peut commencer par construire le prédicat *hasard(L, X)* retournant un élément *X* pris au hasard dans une liste *L*.

Grâce aux prédicats déjà vus de rang, longueur et concaténation, on peut écrire.

hasard(L, X) :- *len(L, N), random(*0*, N, R), rg(X, R, L).*
voyelle(X) :- *app(X, [a, e, i, o, u, y]).*
consonne(X) :- *app(X, [b, c, d, f, g, h, j, k, l, m, n, p, q, r, s, t, v, w, x, z).*

On verra au chapitre suivant qu'il est plus simple d'écrire par une négation le prédicat *consonne(X)* :- *not(voyelle(X)).*

suffixe(S) :- *hasard([[e, m], [j, i], [o, c], [m, u, c, h, e]], S).*

argot([C | M], [l | A]) :- *consonne(C), suffixe([V | S]), voyelle(V),*
conc(M, [C, V | S], A).
argot([C1 | M], [l | A]) :- *consonne(C1), suffixe([C2 | S]), consonne(C2),*
conc(M, [C2 | S], A).

argot([b, o, u, c, h, e, r], *A*). → *A* = [l, o, u, c, h, e, r, b, e, m]
argot([j, a, r, g, o, n], *A*). → *A* = [l, a, r, g, o, n, m, u, c, h, e]
argot([p, a, t, r, o, n], *A*). → *A* = [l, a, t, r, o, n, b, e, m]
argot([p, a, t, r, o, n], *A*). → *A* = [l, a, t, r, o, n, m, u, c, h, e]

30 Plus longue sous-séquence entre deux listes
On cherche les suites extraites communes à deux listes et leur longueur.
Ainsi *ls([a, b, c, a], [a, a, b, b, c], R, N).* doit renvoyer *R = [a, b, c]* et *N* = 3,
ls([a, a, b, a, a, c, e, f], [b, b, c, d, e], R, N). → *N* = 3 et *R = [b, c, e]*
ou encore *ls([a, t, a, a, c, g, g, a, t, t], [t, a, c, c, g, c, g, a, t, a, c], R, N).* → *[t, a, c, g, g, a, t]* de longueur 7.

Dans ce qui suit *NP* est la longueur provisoire de la plus longue sous-séquence commune *LP* entre les deux listes *L* et *M*. Mais en fait, nous avons toutes les solutions de nombreuses fois.

```
ls(_, [], [], 0). % Avec la coupure (chapitre suivant) il est possible de réduire
ls([], _, [], 0).
ls(L, M, R, N) :- lcs(L, M, [], 0, R, N).

lcs([X | L], [X | M], LP, NP, [X | R], K) :- ls(L, M, R, N), K is N + 1, NP =< K.
lcs([_ | L], M, LP, NP, R, N) :- ls(L, M, R, N), NP < N.
lcs(L, [_ | M], LP, NP, R, N) :- ls(L, M, R, N), NP < N.
```

31 Tout entier se décompose en somme de quatre carrés
C'est vrai s'il y a répétitions et si zéro peut être présent, mais on demande ici la décomposition d'un entier *N* en quatre carrés distincts non nuls et en utilisant l'appartenance *app*.

```
intervalle(1, [1]).
intervalle(N, [N | L]) :- 1 < N, M is N – 1, intervalle(M, L).

decomp(N, A, B, C, D) :- K is round(sqrt(N)), intervalle(K, I), app(A, I),
        app(B, I), A > B, app(C, I), B > C, app(D, I), C > D,
        N is A*A + B*B + C*C + D*D.
```

> intervalle(7, *I*). → *I* = [7, 6, 5, 4, 3, 2, 1]
> decomp(12, *A*, *B*, *C*, *D*). → no
> decomp(30, *A*, *B*, *C*, *D*). → *A* = 4 *B* = 3 *C* = 2 *D* = 1 ; no
> decomp(50, *A*, *B*, *C*, *D*). → *A* = 6 *B* = 3 *C* = 2 *D* = 1 ; no
> decomp(290, *A*, *B*, *C*, *D*). → *A* = 15 *B* = 6 *C* = 5 *D* = 2 ; *A* = 14 *B* = 9 *C* = 3 *D* = 2 ; *A* = 14 *B* = 7 *C* = 6 *D* = 3 ; *A* = 13 *B* = 9 *C* = 6 *D* = 2 ; *A* = 12 *B* = 11 *C* = 4 *D* = 3 ; *A* = 12 *B* = 9 *C* = 8 *D* = 1 ; *A* = 12 *B* = 9 *C* = 7 *D* = 4 ; no

MCours.com