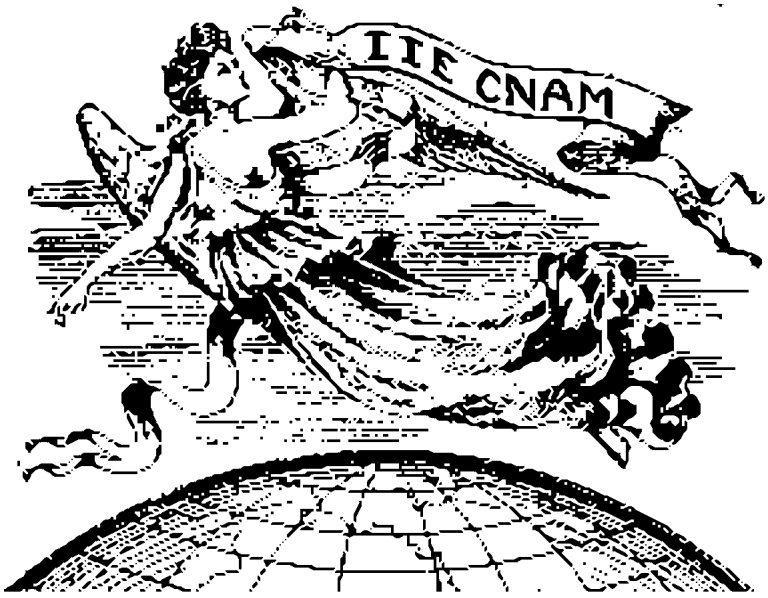
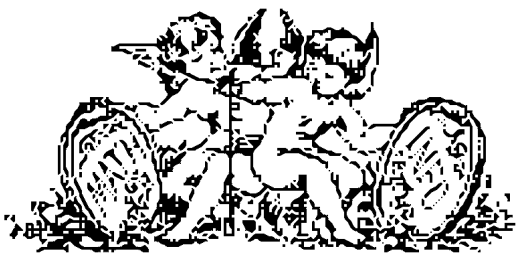
Institut d'Informatique d'Entreprise



PROGRAMMATION LOGIQUE



L.Gacôgne

décembre 2001

CHAPITRE 15

LE CHAINAGE-ARRIERE : PROLOG

Le Prolog est né vers 1980 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, et qui a influencé le domaine de l'interrogation des bases de données déductives.

Fonctionnement d'un interpréteur prolog, le principe de résolution

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 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) ou non nécessairement clos comme : (égal X X), et d'autre part de règles sous forme de "clauses de Horn" comme : (père X Y) et (père Y Z) $\Rightarrow$ (grand-père X Z). On écrit ces règles dans le sens de la réécriture, la conclusion en premier, celle-ci devant s'effacer afin d'être remplacée par les hypothèses : 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. Avec la syntaxe du Turbo-Prolog C if H_1 and H_2 and ... and H_n. ou bien avec celle du C-Prolog ou de "Eclipse" C :- H_1, H_2, ... , H_n.

On pose un but 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. Si par un jeu de substitutions de variables, les deux termes peuvent être rendus égaux, une telle unification est possible, alors avec ces mêmes substitutions partout dans Q, cette conclusion est remplacée par les hypothèses qui pourraient l'entraîner. C'est "l'effacement" et la constitution d'une "résolvante".

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 le "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, c'est en cela que l'on parle de non-déterminisme. Le prédicat trace(but). permet de voir tous les appels et leur résultat logique lors de la résolution d'un but.

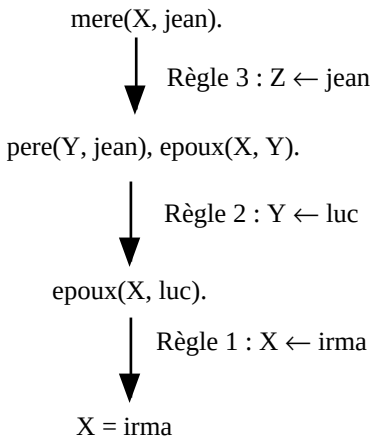
Exemple

La base de règles est formée de deux faits, et d'une règle :

```
epoux(irma, luc).
pere(luc, jean).
mere(X, Z) :- pere(Y, Z), epoux(X, Y).
```

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 Y est le père de Z et si X a pour époux ce même Y, alors X est la mère de Z". Une variable débute toujours par une majuscule, les constantes telles que jean, luc... par des minuscules.

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



Constantes et variables
Remarque, toutes les clauses (règles ou faits) se terminent par un point.
Dans la plupart des Prolog, les variables débutent par une majuscule ou un tiret, par exemple A, A1, Xa, _8, _A ... le symbole _ désignant une variable anonyme. Les atomes sont des chaînes repérées avec quotes comme 'abc' ou des minuscules a, abc, ...
Les commentaires sont encadrés par /* ... */ ou bien précédés de % en Open-Prolog.
En C-Prolog on dispose de quelques prédicats prédéfinis tels que : var(X) et nonvar(X), number(X), integer(X), atom(X), les prédicats infixes <, =<, @< pour les chaînes... L'affectation "X is E" s'emploie avec une variable et une expression E, par exemple : X is 2*exp(Y) (le signe = en turbo-prolog). L'expression E de droite est alors calculée avant d'être assignée à X. Aux questions 4 is 3+1. et 4 is 3. Prolog répond oui, non comme pour le prédicat d'égalité, cependant is fait un effet de bord : une affectation.

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". 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 cet exemple.
La stratégie standard est l'ordre de parcours racine-gauche-droite de l'arbre de recherche, 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.
L'ordre des clauses a également son importance en cas de définition récursive par exemple :

```
fac(N, R)  :-  K is N-1, fac(K, R'), R is R'*N.
fac(0, 1).
```
Ce programme bouclerait indéfiniment.

L'ordre des prémisses est plus délicat à appréhender, car pour les définitions :

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

On aura beaucoup de recherches inutiles lors de la question "ascendant(X, max).", l'ordre "ascendant(Z, X), parent(X, Z)." serait plus efficace, mais par contre la situation est inversée pour la question "ascendant(max, X).", puisque Prolog résoud de gauche à droite.
Malgré ces défauts, Prolog resoud magistralement les exercices ordinaires de backtrack, voir par exemple plus loin la reconnaissance de mots par automates finis et les fameuses tours de Hanoï. Il s'agit 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 prédicat "mouv" traduit récursivement ce qu'il faut faire comme mouvements. Le prédicat "hanoi" associé au nombre de disques, permet de démarrer. L'explication est transparente, pour déplacer les N disques de A sur C, il faut déplacer les N-1 premiers sur B, puis le dernier sur C, et enfin les N-1 de B sur C. (2^{N-1} transports). "write" n'accepte qu'un seul argument en C-prolog, il faut donc en écrire plusieurs, et se servir de "nl" (retour à la ligne).

```
mouv(0, A, _ , C).
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).
hanoi(3). →     transport de gauche sur droite          % le faire à la main pour N = 2
                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
```

Vision logique et vision opérationnelle

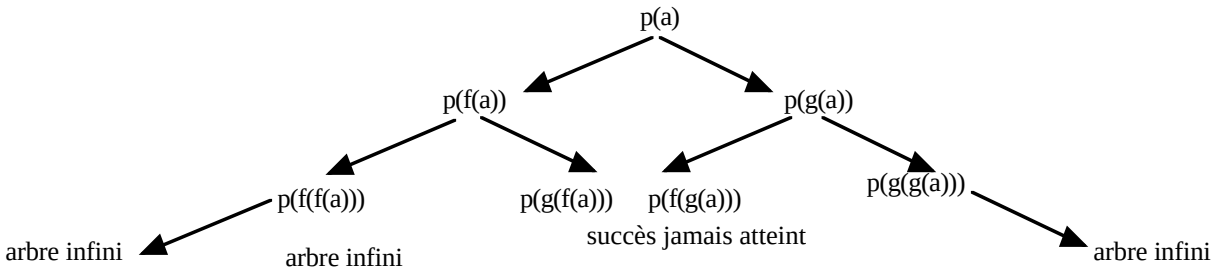
Plus généralement, voir Prolog avec une vision logique sémantique plutôt que comme un démonstrateur est malheureusement insuffisant, en effet, Prolog suit une stratégie, et c'est cette vision "opérationnelle" qui compte, ce qui fait que certaines preuves évidentes ne pourront jamais être détectées par Prolog. Par exemple dans le programme constitué des deux clauses ordonnées p :- p. p. la seconde clause ne sera jamais examinée.

Ou encore, 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 $p(f(g(a))) \Rightarrow p(g(a)) \Rightarrow p(a)$

Mais l'ordre des clauses empêchera d'y arriver.



La stratégie "en largeur d'abord" explorant l'arbre étage par étage, assurerai le succès, mais serait bien trop inefficace.

Problèmes liés à la transitivité et à la symétrie des relations

Si 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 une branche infinie, il faut donc remplacer la clause "chemin" 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 "mariés(X, Y) :- mariés(Y, X)." est logiquement correct mais absoluement 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 la réponse sinon :

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

Le traitement des listes en Prolog

La liste est une structure de données des plus pratiques. 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, et [a | X] si a en est le premier élément et X la liste de ce qui suit, ou encore [a, b, c | X] si l'on distingue les trois premiers éléments.

Prédicat prédéfini : name(X, L) qui relie un atome X à la liste L de ses codes ascii.

En Prolog II, la liste [a, b, c] est notée a.b.c.nil ou encore a.L si L est une autre liste, et une lettre unique est toujours une variable dans cette version.

Exemple du prédicat "prefixe(P, L)" où P est une sous liste commençante de L, de l'appartenance à une liste et de l'affichage des éléments d'une liste :

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

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

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

Le prédicat "être une liste" se définit avec les deux clauses :

```
liste([]).
liste([ _ | _ ]).
```

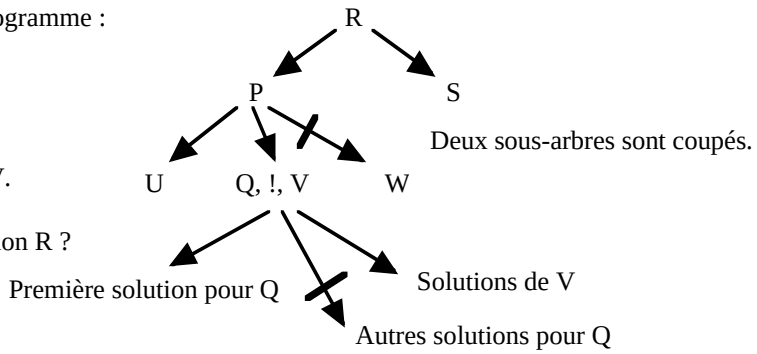
Longueur : long([ ], 0). long([X | L], N) :- long(L, M), N is M+1.

Si la seconde clause est remplacée par "long([X | L], N) :- long(L, N-1)." il ne peut y avoir de solution en demandant N, par contre en la remplaçant par " long([X | L], M+1) :- long(L, M)." alors la variable N cherchée s'unifie à un arbre +(M, 1) qui à son tour va se complexifier pour donner une solution syntaxique : long([a, z, e, r, t, y], N). → N = 0+1+1+1+1+1+1

Contôle de l'exploration de l'arbre par les primitives "impasse" et "coupure"
Lorsque c'est le tour de "impasse" (fail) d'être effacée alors la recherche continue en remontant au noeud supérieur.
Outre le prédicat prédéfini "impasse" un autre prédicat permet de contrôler l'exploration de l'arbre de recherche. Lorsque "coupure" (noté par "cut", / ou !) s'efface, alors tous les choix en attente sont supprimés entre ce noeud et celui où est apparue la coupure. ! signifie donc que la première solution trouvée sur sa gauche suffira, mais que Prolog cherchera toutes les solutions aux prédicats sur la droite de !.

Pour le programme :

R :- P.
R :- S;
P :- U.
P :- Q, !, V.
P :- W.
et la question R ?

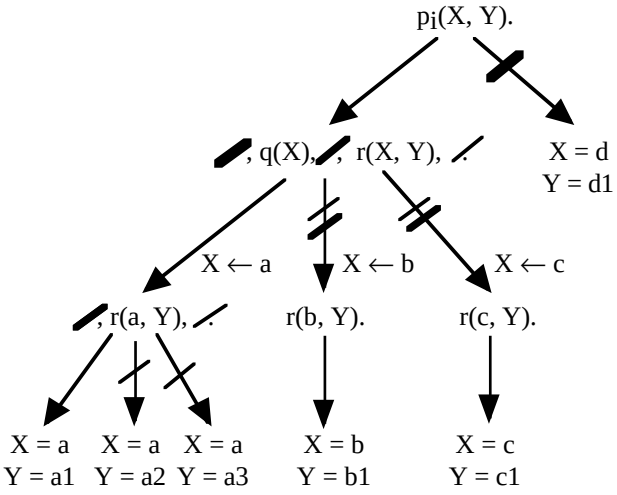


Exemple, on considère quatre programmes ne variant que par la dernière clause p_i mise à la place de p(x,y) si ... :
Soient les 4 programmes constitués avec les clauses relatives aux prédicats q, r, p avec 3 possibilités de positions de la coupure (vues par différentes épaisseurs) pour p(X, Y).

```
q(a).
q(b).
q(c).
r(b, b1).
r(c, c1).
r(a, a1).
r(a, a2).
r(a, a3).

p0(X, Y) :- q(X), r(X, Y).
p1(X, Y) :- q(X), r(X, Y), /.
p2(X, Y) :- q(X), //, r(X, Y).
p3(X, Y) :- ///, q(X), r(X, Y).
pi(d, d1).      % avec i successivement 0, 1, 2, 3
```

Suivant le but demandé :



Pour p_0 les 6 solutions sont données alors qu'il n'y en a qu'une (a, a_1) pour p_1, les 3 (a, a_i) pour p_2, et 5 pour p_3, (d, d_1) n'étant pas regardée.

Négation
Le problème qui se pose pour la négation, est que le "faux" est remplacé par le concept de "non prouvable". La négation n'est donc qu'une "négation par l'absence", elle peut être simulée par les deux clauses :

```
not(P) :- P, !, fail.
not(P).                 % ordre des deux clauses indispensable
```

En effet, la première clause signifie que si P est attesté, alors la coupure cessera toute recherche ultérieure, et l'impasse "fail" forcera la tête not (P) à être fausse. La seconde clause indique que si P n'est pas présent dans la base de connaissance, alors not(P) est vrai.

Par exemple "différent" (prédéfini par \==) peut être redéfini par :
```
dif(X, X) :- !, fail.
dif(X, Y).
```

Par ailleurs, not(P) n'instancie rien, P peut être résolu en donnant des solutions s'il contient des variables, alors que not(P) sera un succès sans donner d'instanciation. Un exemple extrêmement réduit aidera à comprendre ce mécanisme.
```
etudiant(max).   etudiant(luc).   mineur(jean).
etudiantmajeur(X) :- not(mineur(X)), etudiant(X).
```

La question etudiantmajeur(luc). donnera vrai dans la mesure où le fait mineur(luc) n'existe pas dans la base, mais la question "etudiantmajeur(X)." ne donnera rien (pas d'instanciation pour X), sauf si on inverse l'ordre des hypothèse de "etudiantmajeur", car alors X = max puis X = luc prouveront d'abord "etudiant(X)".

Utilisation de la coupure pour réduire l'espace de recherche
Lorsque l'on est certain de l'unicité d'une solution, la coupure empêche les retours en arrière inutiles, voire néfaste, par exemple dans :
```
min(X, Y, X) :- X <= Y, !.   min(X, Y, Y) :- X > Y.
```

On pourrait supprimer X > Y, si le troisième argument (le résultat du min) est inconnu, mais alors "min(3, 5, 5)." réussirait, ce qu'il vaut mieux éviter.

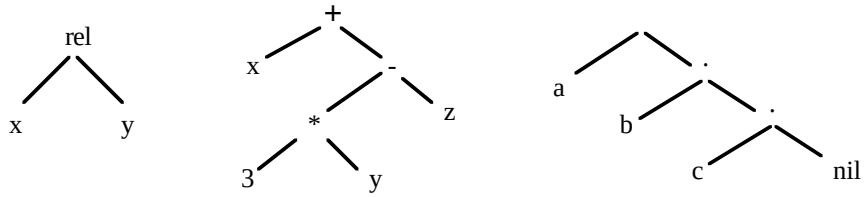
Utilisation pour l'appartenance :
```
membre(X, [X|_]) :- !.   membre(X, [_|L]) :- membre(X, L).
```

Dès qu'on trouve un élément commun, "disjoints" échouera.
```
disjoints(L, M) :- not(elcommun(X, L, M)).
elcommun(X, L, M) :- membre(X, L), membre(X, M).
```

Ou bien :
```
disjoints(X, Y) :- membre(E, X), membre(E, Y), !, fail.   disjoints(X, Y).
```

Après avoir défini le prédicat "liste" on peut écrire not(liste(X)) mais aussi définir :
```
noliste ([]) :-  !, fail.
noliste ([_ | _]) :-  ! , fail.
noliste(_).
```

Le traitement des arbres en Prolog
Les arbres constituent une structure de données plus générale que les listes. Un foncteur est en Prolog, un constructeur d'expressions, c'est à dire d'arbres.
Les termes composées par un "foncteur" et ses arguments sont représentés par des arbres ainsi pour une relation à deux paramètres écrite rel(x, y). en turbo-prolog ou en C-prolog ou écrite <rel, x, y > en Prolog II, il s'agit de l'arbre de gauche.
Un terme composé avec des foncteurs élémentaires en notation infixe tel que x + (3*y -z) sera celui du milieu, x + y. étant un terme prolog identique à +(x, y).



Les listes sont des arbres binaires particuliers dont le foncteur a été noté par le point dans les débuts du Prolog, comme dans les premiers Lisp. La liste [a, b, c] par exemple, est en fait le terme .(a, .(b, .(c, []))), c'est à dire l'arbre de droite.
On dispose en Prolog d'un prédicat "functor" :
functor(A, R, n) établissant une relation entre un arbre A, sa racine R, et son arité n,
ainsi que d'un prédicat noté infixement "=.." établissant une relation entre un arbre et une liste, exemple : f (x, y, z) =.. [f, x, y, z]. est une clause vraie, ou encore x + 3 =.. [+, x, 3].
Exemple de l'appartenance à un arbre binaire "arb" (mais cet arbre "arb" ne peut être un paramètre, ce serait de la logique du second ordre).

```
app(X, arb(X, _, _).
app(X, arb(_, G, D) :- app(X, G).
app(X, arb(_, G, D) :- app(X, D).
```

Problème lié à la logique du premier ordre
Dans la cas par exemple de la dérivation, on serait tenté par exemple pour la composition de fonctions, de donner une règle "deriv(F(G), X, DF*DG) :- deriv(F, G, DF), deriv(G, X, DG)." Voir plus loin l'exercice sur la dérivation où deriv(E, X, D) est satisfait dès lors que D est l'expression dérivée de E par rapport à X.

Pour, Prolog, F n'est pas une variable, mais un foncteur bien défini, ainsi en est-on réduit à écrire toutes les règles telles que "deriv(sin(U), X, cos(U) * DU) :- deriv(U, X, DU)." ...

Exemple du calcul propositionnel
A partir de deux constantes "vrai" et "faux", on redéfinit la syntaxe des expressions de la logique élémentaire avec les connecteurs "non", "et", "ou". Le prédicat de "satisfiabilité" sat(E) réussit si et seulement si une substitution de constantes aux variables de l'expression E est telle que E vaut "vrai". Le prédicat "invalide" fait la même chose avec le "faux".

```
sat(vrai).                          invalide(faux).
sat(non(E)) :- invalide(E).         invalide(non(E)) :- sat(E).
sat(et(E, F)) :- sat(E), sat(F).    invalide(et(E, F)) :- invalide(E).
sat(ou(E, F)) :- sat(E).            invalide(et(E, F)) :- invalide(F).
sat(ou(E, F)) :- sat(F).            invalide(ou(E, F)) :- invalide(E), invalide(F).
```

Malheureusement, on retrouve pour la question ci-dessous une branche infinie dans l'arbre de recherche, mais jamais la réponse Y = faux qui ne pourrait être vue qu'après l'exploration de cette branche. Ce programme ne réalise pas la "complétude" : toute formule valide syntaxiquement (tautologie) est sémantiquement vraie et réciproquement. Ici l'aspect opérationnel ou constructif (d'une preuve) ne peut rendre compte de l'aspect sémantique.

```
sat(ou(et(X, vrai), non(Y))).
→       X = vrai                        (et Y quelconque, constitue la première réponse, puis...)
        X = non(faux)                   X = non(non(vrai))
        X = non(non(non(faux)))         X = non(non(non(non(vrai))))
        X = non(non(non(non(non(faux)))))   X = non(non(non(non(non(non(vrai)))))) .....
```

Ensemble des solutions vérifiant une propriété
Le prédicat "setof" permet, lorsqu'on demande par exemple "setof(X, p(X), E)." de délivrer l'ensemble E de toutes les solutions X vérifiant la propriété p(X). On peut même demander le but "setof(f(X), p(X), E)." qui donnera l'ensemble E des f(X), si f est une fonction, tels que p(X) soit vérifié. Par ailleurs la proposition peut être structurée comme (p(X), q(X)) pour la conjonction, (p(X); q(X)) pour la disjonction et Y^p(X, Y) pour "il existe Y tel que p(X, Y)".
Le prédicat "bagof" fait la même chose mais avec d'éventuelles répétitions des solutions puisqu'il s'agit d'un "sac" (concept formalisé par une fonction "répétition" de E dans N) et non d'un ensemble E. Exemple à partir d'une petite base de faits généalogiques :

```
homme(adam).                        pere(adam, cain).
femme(eve).                         pere(adam, sarah).
homme(cain).                        pere(seth, henosh).
homme(abel).                        pere(cain, pierre).
femme(sarah).                       pere(cain, henok).
mere(eve, abel).                    pere(cain, joseph).
mere(eve, seth).                    pere(henok, irad).
epoux(adam, eve).                   pere(irad, mehuyael).
epoux(cain, sarah).
parent(P, X) :- pere(P, X).
parent(M, X) :- mere(M, X).
pere(P, E) :- epoux(P, M), mere(M, E).
```

On demande l'ensemble des hommes, des individus homme ou femme, des hommes ayant épousé une femme, des "papa" des femmes, de ceux qui ont au moins un parent. Pour cette dernière question, on demande le "sac" qui donne des répétitions à cause de la règle entre père et mère.

```
setof(X, homme(X), E). → X = _530  E = [abel, adam, cain]          % X et P sont aussi des variables
setof(X, (homme(X); femme(X)), I). → X = _1136 I = [abel, adam, cain,eve, sarah]
setof(X, (homme(X), F^epoux(X, F)), E). → X = _883 F = _890 E = [adam, cain]
setof(papa(X), femme(X), P). → X = _1021 P = [papa(eve), papa(sarah)]
setof(X, P^parent(P, X), E). →
        E = [abel, cain, henok, henosh, irad, joseph, mehuyael, pierre, sarah, seth]
bagof(X, P^parent(P, X), E). →
        E = [cain, sarah, henosh, pierre, henok, joseph, irad, mehuyael, abel, seth, abel, seth]
```

EXERCICES TD 1

1° L'inspecteur Maigret 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 s'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 au bar mercredi, Eric au bar mardi. et qu'Eve était à l'hippodrome lundi.. (on ne prends pas en compte la présence des victimes comme possibilité qu'ils aient été aussi voleurs ce jour là). Ecrire le programme Prolog qui, a la question suspect(X), renverra toutes les réponses possibles et représenter l'arbre de recherche de Prolog.

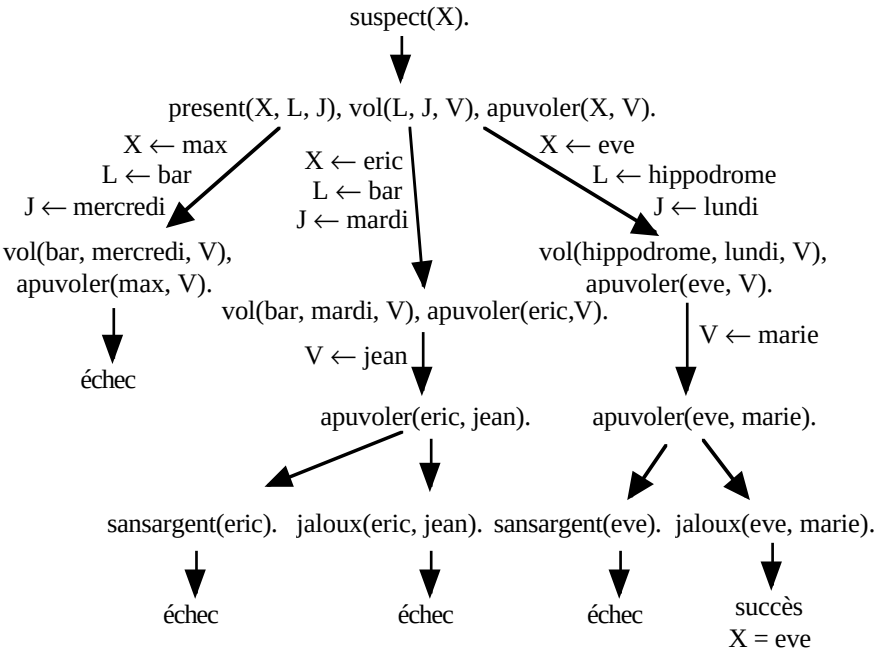
```
suspect(X) :- present(X, L, J), vol(L, J, V), apuvoler(X, V).
apuvoler(X, _) :- sansargent(X).
apuvoler(X, Y) :- jaloux(X, Y).
vol(hipp, lundi, marie).
vol(bar, mardi, jean).
vol(stade, jeudi, luc).
sansargent(max).
jaloux(eve, marie).
present(max, bar, mercredi).
present(eric, bar, mardi).
present(eve, hipp, lundi).
```

La question :

suspect(X).

renvoie :

X = eve ;
no



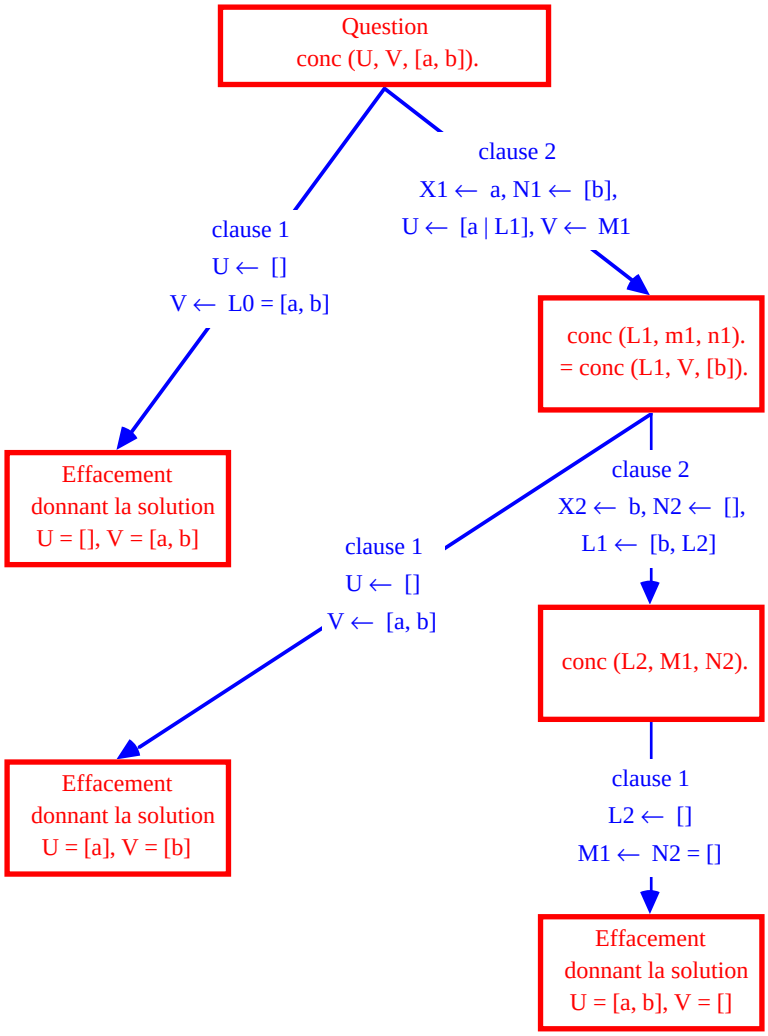
2° Opérations sur une base de données : étant donné une relation notée "rel" à 3 arguments, définir la relation pr "projection" de rel sur les 2 premiers arguments, puis la "sélection" sr des objets dont les deux premiers arguments vérifient une propriété prop. Si r et s sont deux relations à deux arguments, 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.

```
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).
```

3° Concaténation Définir les clauses nécessaires à la concaténation des listes, exemple "conc([a], [b, c], [a, b, c])." est vrai, "conc(X, [a], [b])." est impossible. Dessiner l'arbre de résolution de "conc(U, V, [a, b])." en marquant sur chaque arc les instanciations des variables.

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



4° Sous-liste consécutive 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, B, alors A et B sont notées avec "_" qui signifie "une liste" et D représente la concaténation de A avec SL.

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

On demandera les sous-listes de [a, b, c] par "souliste(X, [a, b, c]).".

5° Sous-liste pure par exemple sliste([a, c, d, e], [a, b, c, d, e, f]). est vrai.

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

EXERCICES TD 2

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).
```

Ou bien
```
last(X, L) :-  conc(_, [X], L).
```

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

Ou bien
```
zajouter(X, L, R) :- conc(L, [X], R).
```

7° Eléménts consécutifs d'une liste "consec(X, Y, L)" où X et Y sont consécutifs dans cet ordre dans 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).
```

8° Rang d'un élément dans une liste

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

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

9° Retraits eff(X, L, R) où R est la liste L dans laquelle la première occurence de X est effacée, eff2 pour effacer toutes les occurences, puis eff3 pour effacer à tous les niveaux de L.

```
eff(_, [], []).
eff(A, [A | L], L) .
eff(A, [B | L], [B | M]) :- eff(A, L, M).
```

Mais, en cherchant toutes les solutions :

eff(a, [f, a, d, a], X). → X = [f, d, a] ; X = [f, a, d] ; X = [f, a, d, a] ; no

Le ";" donne les autres solutions. Il faut rajouter la prémisse "coupure" dans la seconde clause.

```
eff(X, [X | L], L) :- !.
```

Alors : eff(a, [f, a, d, a], X). → X = [f, d, a] ; no

Pour eff2, on remplace la seconde clause par "eff2 (A, [A | L], M) :- eff2(A, L, M)." Mais là aussi "eff2 (a, [b, a, b, a], X)." donnerait X = [b, b] puis X = [b, a, b] puis X = [b, b, a] d'où la nécessité de la coupure.

```
eff2(_, [], []).
eff2(A, [A | L], M) :- eff2(A, L, M), !.
eff2(A, [B | L], [B | M]) :- eff2(A, L, M).

eff3(_, [], []).
eff3(X, [X | L], M) :- eff3(X, L, M), !.
eff3(X, [Y | L], [Z | M]) :- eff3(X, Y, Z), eff3(X, L, M), !.
eff3(_, X, X).
```

Exemple eff3(a, [a, b, [b, a, c], [b, [a, c], b], a], R). → R = [b, [b, c], [b, [c], b]] ; no

10° Substitution subst(X, Y, L, R) où R est le résultat de la liste L dans laquelle X a été substitué par Y. Construire de même le prédicat sb de substitution à toutes les profondeurs.

```
subst(_, _, [], []).
subst(X, A, [X | L], [A | M]) :- subst(X, A, L, M).
subst(X, A, [Y | L], [Y | M]) :- subst(X, A, L, M).
```

Substitutions à toutes les profondeurs

```
sb(_, _, [], []).
sb(X, A, [X | L], [A | M]) :- sb(X, A, L, M).
sb(X, A, [Y | L], [Z | M]) :- sb(X, A, Y, Z), sb(X, A, L, M).
```

11° Filtre suivant une étoile, on veut par exemple equ([a, *, *, b, *, c], [*, a, b, *, c, *]).

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

12° Filtre où * dans L filtre une sous-liste consécutive non vide de M

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

13° Inversion d'une liste

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

Mais une meilleure programmation utilise un troisième paramètre tampon et deux prédicats : Faire l'essai de inv en partant de inv([a, b, c], [], R).

```
inv([], L, L).
inv([X | Y], Z, R) :-  inv(Y, [X | Z], R).
inverse(L, R) :-  inv(L, [], R).
```

EXERCICES TD 3

14° Tri par insertion Le premier élément d'une liste étant mis de côté, le reste est trié (suivant la même méthode), puis l'élément est inséré à sa place.

```
tri([X | L], R) :- tri(L, LT), insert(X, LT, R).
tri([], []).
insert(X, [], [X]).
insert(X, [Y | L], [X, Y | L]) :- X < Y, !.
insert(X, [Y | L], [Y | M]) :- insert(X, L, M).
```

tri([1,5,6,2,3,8,0,4,1,9,2,5], L). renvoie L = [0,1,1,2,2,3,4,5,5,6,8,9]

15° Tri par fusion La liste à trier est séparée en deux, ces deux parties sont triées suivant le même algorithme, puis fusionnées. On peut construire et utiliser un prédicat impairs(LI, L) où LI est la liste des éléments de rangs impairs de L. Exemple LI = [a, c, e] pour L = [a, b, c, d, e]

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

trifus([], []).
trifus([X], [X]).
trifus(L, R) :- scission(L, G, D), trifus(G, GT), trifus(D, DT), fusion(GT, DT, R).
scission(L, G, D) :- impairs(L, G), pairs(L, D).        % pair étant à définir

fusion([], L, L).
fusion(L, [], L).
fusion([X | P], [Y | Q], [X | R]) :- X < Y, fusion(P, [Y | Q], R), !.
fusion([X | P], [Y | Q], [Y | R]) :- fusion([X | P], Q, R), !.
```

Ou plus directement, on peut définir :

```
scission([], [], []).
scission([X], [], [X]).
scission([X, Y | L], [Y | P], [X | I]) :- scission(L, P, I).
```

trifus([6, 5, 8, 0, 2, 3, 7, 5], L). → L = [0, 2, 3, 5, 5, 6, 7, 8] % est un exemple

16° Tri par segmentation Un élément (pivot) est choisi dans la liste, puis tous les éléments sont versés dans deux listes G, D suivant qu'ils sont inférieurs ou supérieurs au pivot. Ces deux listes sont triées de la même façon puis concaténées de part et d'autre du pivot.

```
partition([X | L], P, [X | G], D) :- X =< P,!, partition(L, P, G, D).
partition([X | L], P, G, [X | D]) :- partition(L, P, G, D).        % retirer la coupure, oblige à rajouter X > P
partition([], _, [], []).
triseg([], []).
triseg([X | L], R) :- partition(L, X, G, D), triseg(G, GT), triseg(D, DT), conc(GT, [X | DT], R).
Ou mieux :      tri(LD, LT) :- tribis(LD, LT, []). % le second argument étant un tampon, le troisième, le résultat
                tribis([], L, L).
                tribis([X | R], LT, LR) :- partition(R, X, G, D), tribis(G, LT, [X | Q]), tribis(D, Q, LR).
tri([2,5,0,3,9,4,7,6,3,5,1,2,8,0], X). → X = [0,0,1,2,2,3,3,4,5,5,6,7,8,9]
```

17° Définir les relations de parenté frere(X, Y) vérifiée si X est un frère de Y, oncle(O, X), demifrere(X, Y), puis les relations de parenté au degré n, sachant qu'il s'agit de la distance dans l'arbre, généalogique, ainsi père et fils sont au degré 1, frères et soeurs au degré 2, oncle et neveu au degré 3 etc...

```
homme(adam).                pere(adam, cain).           pere(adam, sarah).
femme(eve).                 epoux(adam, eve).           epoux(cain, sarah).
mere(eve, abel).            mere(eve, seth).            pere(seth, henosh).
pere(cain, pierre).         pere(cain, henok).          pere(cain, joseph).
pere(henok, irad).          pere(irad, mehuyael).
parent(P, X) :- pere(P, X).     parent(M, X) :- mere(M, X).
pere(P, E) :- epoux(P, M), mere(M, E).
degre(X, Y, D) :- branche(A, X, N), branche(A, Y, M),!, D is N+M.
/* la coupure est nécessaire pour ne pas aller chercher d'ancêtre commun plus haut que le premier */
branche(X, X, 0).
branche(A, X, N) :- parent(P, X), branche(A, P, K), N is K+1. % A est un ascendant de X
branche(X, irad, Y). → X = irad Y = 0 ; X = henok Y = 1 ; X = cain Y = 2 ; X = adam Y = 3 ; no
degre(pierre, joseph, X). → X = 2          degre(mehuyael, joseph, X). → X = 4
degre(adam, irad, X). → X = 3              degre(mehuyael, adam, X). → X = 4
degre(irad, henosh, X). → X = 5
```

18° 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 final q_0, un (ou plusieurs) état final, et une relation de transition donnée par des triplets (q, x, q') où x ∈ A et q, q' ∈ Q. Un mot m est reconnu par l'automate ssi il existe une suite de transition 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 q0, q1, et la transition définie par tr(q1, a) = q2, tr(q2, b) = q1 avec q0 à la fois initial et final.
c) Décrire l'automate reconnaissant les mots contenant le sous-mot "iie" sur A = {a, e, i, o, u}.

```
reconnu(M) :- init(E), continue(M, E).
continue([A | M], E) :- trans(E, A, F), continue(M, F).
continue([], E) :- final(E).
init(q0).                   % automate du petit b
final(q0).
trans(q0, a, q1).
trans(q1, b, q0).           % il reconnait les mots de la forme (ab)n
reconnu([a, b, a, b, a, b, a, b]). → yes
reconnu([a, b, b, b, a, a]). → no
init(q0).                   % automate du petit c
trans(q0, i, q1).
trans(q1, i, q2).
trans(q2, e, q3).
trans(q2, i, q2).
trans(q0, V, q0) :- membre(V, [a, e, o, u]).
```

```
trans(q1, V, q0) :- membre(V, [a, e, o, u]).
trans(q2, V, q1) :- membre(V, [a, o, u]).
trans(q3, V, q3) :- membre(V, [a, e, i, o, u]).        % ou trans(q3, _, q3). mais autres lettres
final(q3).
membre(X, [X | _ ]).
membre(X, [_ | L]) :- membre(X, L).
```

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

EXERCICES TP

VMS -> ED par <ed nom.pro> et <c> qui vous ouvre un fichier en mode texte
VMS -> PROLOG par la commande <prolog>, retour par <halt.> Dans Prolog, la clause <reconsult('nom.pro').> charge les clauses du fichier écrit préalablement dans l'éditeur en écrasant les anciennes. La clause <listing.> permet de le vérifier.
PROLOG -> ED par la clause <system("ed nom.pro").>
ED -> VMS par exit et ctrlZ.
UNIX -> PROLOG par <eclipse> chargement par compile(nom). ou [nom]. Ce nom est sans extension ni majuscule. Attention à grouper les clauses par paquets concernant le même prédicat. Sortie par ^D.

19° Problème du voyageur de commerce

a) Des villes Paris, Belmopan (sortez votre atlas), Ouagadougou,... sont reliées par des "routes" à sens unique pour l'instant.
b) Définir le prédicat "chemin" puis, en introduisant des distances, le prédicat "route" de X à Y par le chemin L de longueur N.
c) Introduire des boucles et définir des chemins sans points doubles.

On aura bien sûr un programme composé d'abord de 10 faits du type :

```
arc(paris, tirana).                arc(oulan-bator , kuala-Lumpur).   arc(paris, reykjavik).
arc(tirana, oulan-bator).          arc(oulan-bator, kuala-lumpur).    arc(paris, oulan-bator).
arc(belmopan, reykjavik).          arc(belmopan, paris).              arc(belmopan, ouagadougou).
arc(paris, ouagadougou).           arc(ouagadougou, kuala-lumpur).
```

En fait, ne pas oublier de mettre des minuscules à ces villes qui sont des constantes, puis on définira un chemin comme une liste par :

```
chemin(X, Y, [X,Y]) :-  arc(X, Y).
chemin(X, Y, [X| L]) :-  arc(X, Z), chemin(Z, Y, L).
```

On pourra alors poser les buts :
Comment se rendre de Belmopan à Kuala-Lumpur ?

```
chemin(belmopan, kuala-lumpur, X). →   X = [belmopan, paris, tirana, oulan-bator, kuala-lumpur] ;
                                       X = [belmopan, paris, oulan-bator, kuala-lumpur] ;
                                       X = [belmopan, paris, ouagadougou, kuala-lumpur] ;  no
```

Quels sont les chemins arrivant à Reykjavik ?

```
chemin(V, reykjavik, L).  →    V = paris  L = [paris, reykjavik] ;
                               V = belmopan  L = [belmopan, reykjavik] ;
                               V = belmopan  L = [belmopan, paris, reykjavik] ; no
```

b) On peut introduire des distances ou des coûts en remplaçant les clauses "arc" par une relation donnant en plus la distance, dist (A, B, 5). dist (C, F, 12) où A, B, C, ... sont des villes, par exemple et en remplaçant "chemin" par :

```
dist(paris, tirana, 10).          dist(oulan-bator , kuala-lumpur, 25).   dist(paris, reykjavik, 15).
dist(paris, oulan-bator, 30).     dist(tirana, oulan-bator, 25).          dist(belmopan, reykjavik, 20).
dist(belmopan, paris, 30).        dist(belmopan, ouagadougou, 35).        dist(paris, ouagadougou, 15).
dist(ouagadougou, kuala-lumpur, 20).
dist(oulan-bator, kuala-lumpur, 15).
        route(X, Y, [X,Y], N) :-  dist(X, Y, N).
        route(X, Y, [X|L], N) :-  dist(X, Z, K), route(Z, Y, L, M) , N is K + M.
```

Les chemins issus de A de longueur < 22 seront demandés par :

route(belmopan, V, L, N), N < 22. → V = reykjavik L = [belmopan, reykjavik] N = 20 ; no

Les chemins de paris à F ne passant pas en Tirana, par :

```
app(X, [X|_]).  app(X, [_|L]) :- app(X, L).
route(paris, F, L, N) ,  not(app(tirana, L)). → F = reykjavik L = [paris, reykjavik] N = 15 ;
                                                F = oulan-bator L = [paris, oulan-bator] N = 30 ;
                                                F = ouagadougou L = [paris, ouagadougou] N = 15 ;
                                                F = kuala-lumpur L = [paris, oulan-bator, kuala-lumpur] N = 45 ;
                                                F = kuala-lumpur L = [paris,ouagadougou,kuala-lumpur] N = 35 ; no
```

Mais ceci a l'inconvénient de construire L pour ensuite vérifier si D y est présent ou non, il vaut mieux construire un autre prédicat route-sans (X, Y, Z, L, N).

c) S'il y a des boucles comme par exemple un arc de Ouagadougou vers Belmopan, on peut définir :

```
arc(ouagadougou, belmopan).
chemin-injectif(X, Y, [X,Y], _) :- arc(X, Y).          % L sert de tampon, R sera le résultat
chemin-injectif(X, Y, [X|R], L) :- arc(X, Z), not(app(Z, L)), chemin-injectif(Z, Y, R, [Z|L]).
```

La question "chemin(belmopan, tirana, C)." donne une infinité de solutions :

C = [belmopan, paris, tirana] ; C = [belmopan, paris, ouagadougou, belmopan, paris, tirana] ; et ainsi de suite ...

Par contre :

```
chemin-injectif(belmopan, tirana, C, [belmopan]). → C = [belmopan, paris, tirana] ; no
chemin-injectif(paris, reykjavik, C, [paris]).
```

Ne donne que deux solutions : C = [paris, reykjavik] ; C = [paris, ouagadougou, belmopan, reykjavik] ; no

20° Utilisation de foncteurs : dérivation formelle Donner quelques règles pour le prédicat d(E, X, D) satisfait si D est l'expression dérivée de E suivant la variable X

```
d(U+V, X, DU+DV) :- !, d(U,X,DU), d(V,X,DV).
d(U-V, X, DU-DV) :- !, d(U,X,DU), d(V,X,DV).
d(U*V, X, DU*V+U*DV) :- !, d(U,X,DU), d(V,X,DV).
d(U/V, X, (DU*V-U*DV)/V^2) :- !, d(U,X,DU), d(V,X,DV).
d(U^N, X, DU*N*U^N1) :- integer(N), N1 is N-1, d(U,X,DU).
d(-U, X, -DU) :- !, d(U, X, DU).
d(exp(U), X, exp(U)*DU) :- !, d(U, X, DU).
d(log(U), X, DU/U) :- !, d(U, X, DU).
d(X, X, 1) :- !.     % coupure essentielle pour empêcher d'aller voir la règle suivante en cas de succès
d(C, X, 0).
```

Par exemple :
d(x*x*x,x, R). → R = (1*x+x*1)*x+x*x*1
d(2*x+log(x), x, R). → R = 0*x+2*1+1/x ;
d(exp(3*x+1), x, R). → R = exp(3*x+1)*(0*x+3*1+0) ;
d(3*x+5*y, x, R). → R = 0*x+3*1+(0*y+5*0) ;
d(3*x+5*y, y, R). → R = 0*x+3*0+(0*y+5*1)

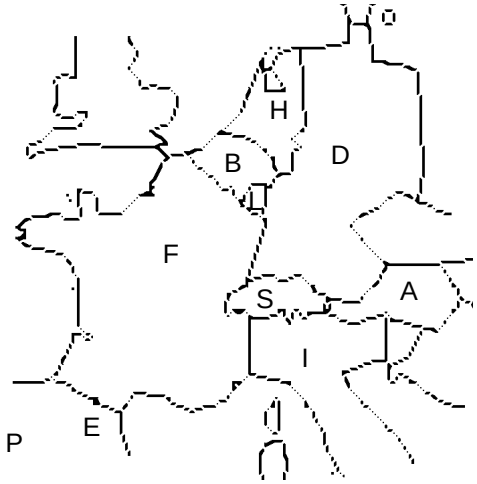
21° Arithmétique de Peano L'axiomatique de Peano, consiste à donner les axiomes de bases de l'arithmétique. On se contentera de l'addition ou la multiplication des nombres jusqu'à huit, mais cela montrera néanmoins la variété des requêtes que l'on peut faire.

```
nom(0, zero).                    nom(s(0), un).                  nom(s(s(0)), deux).
nom(s(s(s(0))), trois).          nom(s(s(s(s(0)))), quatre).     nom(s(s(s(s(s(0))))), cinq).
nom(s(s(s(s(s(s(0)))))), six).   nom(s(s(s(s(s(s(s(0))))))), sept).  nom(s(s(s(s(s(s(s(s(0)))))))), huit).
Ou mieux avec   nom(X, D) :- donnees(D), aux(X, D, N).
                aux(s(X), [_ | D], N) :-  aux(X, D, N).          aux(0, [N], N).
                donnees([un, deux, trois, ..., cent]).
plus(0, X, X).
plus(s(X), Y, s(Z)) :- plus(X, Y, Z).
add(X, Y, Z) :- nom(Xe, X), nom(Ye, Y), nom(Ze, Z), plus(Xe, Ye, Ze).
mult(0, X, 0).
mult(s(X), Y, Z) :- mult(X, Y, P), plus(P, Y, Z).
prod(X, Y, Z) :- nom(Xe, X), nom(Ye, Y), nom(Ze, Z), mult(Xe, Ye, Ze).

add(un, deux, trois).→ yes
add(trois, quatre, huit).→ no
add(trois, quatre, X).→ X = sept
add(X, deux, cinq). → X = trois
add(X, Y, deux).→ X = zero Y = deux ; X = un Y = un ; X = deux Y = zero
prod(trois, deux, X). → X = six
prod(X, Y, huit). → X = un Y = huit ; X = deux Y = quatre ; X = quatre Y = deux ; X = huit Y = un
prod(X, Y, trois), add(A, B, Y). → X = un Y = trois A = zero B = trois ; X = un Y = trois A = un B = deux ; X =
un Y = trois A = deux B = un ; X = un Y = trois A = trois B = zero ; X = trois Y = un A = zero B = un ; X =
trois Y = un A = un B = zero
```

22° Coloriage d'une carte On impose les prédicats couleurs([vert, rouge, jaune, bleu]). et carte(LP). où LP est une liste de termes construits avec un "foncteur" pays(N, C, LV) tel que N soit le nom du pays, C sa couleur et LV la liste des couleurs des voisins.

On définit un prédicat coloriage (liste de pays, liste de couleurs) qui sera vrai si les couleurs disponibles peuvent être attribuées aux pays sans que deux pays voisins aient la même. En se limitant à quelques pays, on représentera C la couleur choisie pour le pays de nom N, LC la liste de toutes les couleurs (qui ne doit jamais être diminuée), et LCV la liste des couleurs des voisins inclue dans la liste AC des autres couleurs que C :




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

ret(X, [X|L], L).
ret(X, [Y|L], [Y|R]) :- ret(X, L, R). % ainsi ret(a,l,X) échoue si a n'est pas dans l, sinon il faut la règle ret(_,[],[]).
                                      % mais alors ret(a,l,X) donne l en dernière solution au cas où a est dans l.
vue([]).
vue([pays(N, C, _ ) | L]) :- write(N), write(' -> '), write(C), nl, vue(L).
coloriage([], LC).
coloriage([pays(N, C, LCV)|PR], LC) :- ret(C, LC, AC), inclus(LCV, AC), coloriage(PR, LC).
carte([pays(france, F, [B, D, L, S, I]), pays(belgique, B, [F, H, D, L]), pays(allemagne, D, [F, B, H, L]),
       pays(hollande, H, [B, D]), pays(autriche, A, [D, S, I]), pays(espagne, E, [F, P]),
       pays(luxembourg, L, [F, B, D]), pays(suisse, S, [D, F, I]), pays(italie, I, [F, S, A]),
       pays(portugal, P, [E])]).
jeu :- carte(LP), coloriage(LP, [vert, rouge, jaune, bleu]), vue(LP).
```

La seule question à poser : jeu.

france -> vert, belgique -> rouge, allemagne -> jaune, hollande -> vert, autriche -> vert, espagne -> rouge, luxembourg -> bleu, suisse -> rouge, italie -> jaune, portugal -> vert

15-23° Ecrire les clauses Prolog correspondant au faits que les animaux sont herbivores ou carnivores, les carnivores mangent de la viande et des herbivores, lesquels mangent de l'herbe. Tous boivent de l'eau. Les animaux féroces sont carnivores, le lion est féroce alors que l'antilope est un herbivore. Qui consomme quoi ?

```
animal(X) :- herbivore(X) ; carnivore(X).        % résume deux règles grâce au ;
herbivore(antilope).
feroce(lion).
carnivore(X) :- feroce(X).
mange(X, Y) :- carnivore(X) , herbivore(Y).
mange(X, viande) :- carnivore(X).
boit(X, eau) :- animal(X).
mange(X, herbe) :- herbivore(X).
consomme(X, Y) :- mange(X, Y).
consomme(X, Y) :- boit(X, Y).
```

On posera le but "consomme(X, Y)." qui donne 5 solutions.

15-24° 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 naquit 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 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, cousin etc ...

15-25° 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.

```
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)."

15-26° Que se passe-t-il si l'on remplace les clauses p, successivement par les clauses p4 p5 p6 correspondant à p1 p2 p3 dans lesquelles "impasse" prend la place de "coupure" ?

15-27° **Animaux mutants** grâce à une base d'animaux tels que cheval, lapin, pintade, ... , pour lesquels on construira par concaténation des mutants "lapintade", ... Se servir d'un prédicat que l'on peut définir par nonvide ([_ | _]).

15-28° **Ecrire une liste** Affichages successifs des éléments d'une liste.

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

15-29° **Listes disjointes**

```
disjoints(X, Y) :- app(E, X), app(E, Y), !, fail.
disjoints(X, Y).
```

15-30° Prédicat impairs(LI, L) où LI est la liste des éléments de rangs impairs de L. Par exemple LI = [a, c, e] pour L = [a, b, c, d, e]

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

15-31° **Insérer** X à la N-ième place dans une liste L

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

15-32° **Un tri stupide** le tri de L doit être une permutation de L qui est ordonnée

```
tri(L, LT) :- permut(L, LT), ordre(LT), !.
permut([], []).      % indique que les deux arguments sont des permutations
permut([X|L], [Y|M]) :- ret(Y, [X | L], N), permut(N, M).
ordre([]).           % indique si la liste est ordonnée
ordre([_]).
ordre([X, Y | L]) :- X =< Y, ordre([Y | L]).
ret(X, [X|L], L).
ret(X, [Y|L], [Y| R]) :- ret(X, L, R).
% tri([5, 7, 5, 1, 4, 6, 8], L). renvoie L = [1,4,5,5,6,7,8]
```

15-33° **Représentation creuse d'un polynôme** comme liste de couples (coefficient, degré), ainsi $3x^9 + 7x^2 - 4x^4 + 3$ sera représenté par [[3, 9], [7, 2], [-4, 4], [3, 0]]. Ecrire la relation puissance(X, N, V) vérifiée ssi $X^N = V$. Ecrire la relation valeur(X, P, Y) vérifiée ssi Y = P(X), P étant une représentation de polynôme.

```
puiss(X, 0, 1).
puiss(X, N, V) :- K is N-1, 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.
```

15-34° **Ensemble des parties.**Exemple parties ([a, b], [[], [a], [b], [a, b]]).
Les parties de $\{x\} \cup L$ sont celles de L et aussi les mêmes auxquelles on rajoute 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).      /* "dist" réalisant un "mapcar cons X" */
```

15-35° **Différence ensembliste**
diff(A, B, R) est vrai si la liste R est A privée des éléments présents dans B.

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

15-36° **Produit cartésien pc (A, B, R).**
Exemple pc([a, b], [0, 1, 2], [[a, 0], [a, 1], [a, 2], [b, 0], [b, 1], [b, 2]]) est vrai.

```
pc([], L, []).
pc(L, [], []).
pc([X | Y], Z, U) :- pcbis(X, Z, V), pc(Y, Z, W), concat(V, W, U).
pcbis(X, [], []).
pcbis(X, [A | B], [[X, A] | U]) :- pcbis(X, B, U).
```

15-37° Automate fini reconnaissant les mots (ab)*
Avec 2 états dont l'un est à la fois initial et final, écrire les transitions.

```
init(q0).
terminal(q0).
trans(q0, a, q1).
trans(q1, b, q0).
reconnu(L) :- init(Q), accessible(Q, L), fini(L, S), terminal(S).
accessible(_, []).
accessible(Q, [X | L]) :- trans(Q, X, S), accessible(S, L).
fini([X], S) :- trans(_, X, S), !.
fini([_ | L], S) :- fini(L, S).
```

reconnu(X). donne X = [a,b] ; X = [a,b,a,b] ; X = [a,b,a,b,a,b] ; X = [a,b,a,b,a,b,a,b] ; etc

15-38° Aplatir une liste On veut, par exemple, que aplatir([[a,z,e,r],t,y,[u,i,[o,p],q,s],d,f], R). renvoie R = [a,z,e,r,t,y,u,i,o,p,q,s,d,f]

```
liste([]).
liste([_|_]).
inv([], L, L).
inv([X | L], T, R) :- inv(L, [X | T], R).
aplatir(L, R) :- apbis(L, [], LA), inv(LA, [], R).
apbis(X, T, [X | T]) :- not(liste(X)).
apbis([X | L], T, R) :- apbis(X, T, XA), apbis(L, XA, R).
apbis([], L, L).
```

15-39° Suite de Fibonacci Programmer de manière astucieuse pour ne pas refaire les mêmes calculs (l'arbre de recherche est réduit à une branche). ex. fib(5, X). -> X = 8

```
fib(0, 1).          fib(1, 1).          fib(N, R) :- fibo(R, 1, 1, N).
fibo(U, U, V, 0):-!.
fibo(R, U, V, N):- M is N-1, S is U+V, fibo(R, V, S, M).
```

15-40° Combinaisons faire de même.

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

15-41° Algorithme à retour assez simple Construire un nombre avec une fois chacun des chiffres de 1 à 9 tel que les k premiers chiffres de ce nombre soit divisible par k, pour k de 1 à 9. Exemple: essaidiv("123456789", 1, [], X). X = 381654729
Détail technique "ab+12" est identique à [97,98,43,49,50], et name(S, LA) établit la relation entre la chaîne de caractères S et la liste de ses codes Ascii.

```
essaidiv([], D, CC, R) :- name(R,CC). % cas où c'est fini, D=diviseur initialement 1, R=résultat
        % CC=liste des ascii des chiffres choisis, LC = liste des ascii des chiffres disponibles

essaidiv(LC, D, CC, R) :-               % CC liste des ascii des chiffres choisis, initialement []
        ret(C, LC, RC),                 % choisir un chiffre C dans la chaine LC, il reste les chiffres RC
        append(CC, [C], NCC),           % NCC=nouvelle liste de chiffre choisis
        name(N, NCC),                   % qui correspond à un nombre N
        0 is N mod D,                   % il faut que N soit divisible par D
        ND is D+1,                      % ND=nouveau diviseur à essayer
        essaidiv(RC, ND, NCC, R).       % faire la suite RC= reste des chiffres
append([], X, X).
append([X|Y], Z, [X|R]) :- append(Y, Z, R).
ret(X, [X|R], R).
ret(X, [Y|R], [Y|Z]) :- ret(X, R, Z).
essaidiv("123456789", 1, [], R). → R = 381654729
```

15-42° Logigram I Un exemple des années 1960 est que 5 maisons alignées ont des attributs mutuellement distincts. On sait par exemple que l'Anglais habite la maison verte, l'Espagnol possède un chien, on boit du café dans la maison rouge, l'Ukrainien boit du thé, la maison blanche suit la rouge, le fumeur de Craven élève des escargots, celui qui habite la maison jaune fume des Gauloises, on boit du lait dans la troisième maison, la première est habitée par un Norvégien, le fumeur de pipe est voisin de là où il y a le renard, le fumeur de Gauloises est voisin du cheval, celui qui fume des Lucky-Strike, boit du jus d'orange, le Japonais fume des gitanes et le Norvégien est voisin de la maison bleue. Qui boit de l'eau et qui a le zèbre ?

On construit des termes structurés associant une nationalité, une couleur, un animal, une boisson et un type de tabac, et D sera la liste de ces associations :

```
app(X, [X|_]).
app(X, [_|L]) :- app(X, L).
consec(X, Y, [X, Y|_]).
consec(X, Y, [_|L]) :- consec(X, Y, L).
voisins(X, Y, L) :- consec(X, Y, L).
voisins(X, Y, L) :- consec(Y, X, L).
premier(X, [X | _]).
trois(T, [_, _, T | _]).

donnees([ass(N1, C1, A1, B1, T1), ass(N2, C2, A2, B2, T2), ass(N3, C3, A3, B3, T3), ass(N4, C4, A4, B4, T4),
        ass(N5, C5, A5, B5, T5)]).

nation(ass(N, _, _, _, _), N).
couleur(ass(_, C, _, _, _), C).
animal(ass(_, _, A, _, _), A).
boisson(ass(_, _, _, B, _), B).
fume(ass(_, _, _, _, T), T).

jeu(D) :- donnees(D), app(A, D), nation(A, anglais), couleur(A, verte),
        app(ES, D), nation(ES, espagnol), animal(ES, chien), app(R, D), couleur(R, rouge), boisson(R, cafe),
        app(UT, D), nation(UT, ukrainien), boisson(UT, the), consec(BL, R, D), couleur(BL, blanche),
        app(EC, D), animal(EC, escargot), fume(EC, craven),
        app(GJ, D), fume(GJ, gauloises), couleur(GJ, jaune), trois(TL, D), boisson(TL, lait),
        premier(N, D), nation(N, norvegien), voisins(MP, NR, D), fume(MP, pipe), animal(NR, renard),
        voisins(G, CH, D), fume(G, gauloises), animal(CH, cheval),
        app(LO, D), fume(LO, luckystrike), boisson(LO, judorange),
        app(J, D), nation(J, japonais), fume(J, gitanes), voisins(B, N, D), couleur(B, bleue),
        app(E, D), boisson(E, eau), app(ZE, D), animal(ZE, zebre).
```

Exemple : jeu(D). → D = [ass(norvegien, jaune, renard, eau, gauloises), ass(ukrainien, bleue, cheval, the,pipe), ass(anglais,verte, escargot, lait, craven), ass(espagnol, blanche, chien, judorange, luckystrike), ass(japonais, rouge, zebre, cafe, gitanes)] ; no

15-43° Logigram II Max a un chat, Eve n'est pas en pavillon, Luc habite un studio mais le cheval n'y est pas. Chacun habite une maison différente et possède un animal distinct, qui habite le chateau et qui a le poisson ?

```
maison(chateau).  maison(studio).  maison(pavillon). animal(chat).  animal(poisson).  animal(cheval).
rel(max, M, chat) :- maison(M).
rel(luc, studio, A) :- animal(A), A \== cheval.
rel(eve, M, A) :- maison(M), M \== pavillon, animal(A).

dif(X, X, _) :- !, fail.  dif(X, _, X) :- !, fail.  dif(_, X, X) :- !, fail.  dif(_, _, _).
reso(MM, ME, AE, AL) :- rel(max, MM, chat),rel(eve, ME, AE), dif(MM, ME, studio),
                        rel(luc, studio, AL), dif(AE, AL, chat).
```

Exemple : reso(MM, ME, AE, AL). → MM = pavillon ME = chateau AE = cheval AL = poisson

15-44° Logigram III 4 personnes de professions distinctes habitent 4 logements distincts et possèdent chacun un animal distinct des autres. On sait que Max a un chien, que Luc habite un studio et n'est pas gardien, qu'Eve n'est pas en pavillon et qu'Irma n'est ni gardienne, ni médecin. De plus, l'étudiant habite le château, le coq est dans la caravanne, le cheval n'est pas dans le studio et ce n'est pas le médecin qui a le poisson. Qui est instituteur ? Il y a beaucoup de façon de prendre le problème, on impose ici de représenter les données comme une liste de termes structurés "ass(I, M, A, P)" (ass = association) et de considérer dans le prédicat principal qu'on nommera "assigner", les paires obligatoires (ex. I = luc, M = studio) et celles qui sont impossibles (ex. A = poisson, P = medecin).

```
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).
ret(X, [X|L], L).
ret(X, [Y|L], [Y| R]) :- ret(X, L, R).
assoc(U, V, U, V).                    % en fait la seconde clause est inutile
assoc(X, Y, U, V) :- X \== U, Y \== V.
imp(U, V, U, V) :- !, fail.
imp(_, _, _, _).

Autre solution    assoc(X, Y, A, B) :- (X == A, Y == B); (X \== A, Y \== B).
                  imp(X, Y, A, B) :- X \== A ; Y \== B.

donnees([ass(max, _, chien, _), ass(luc, studio, _, _), ass(eve, _, _, _), ass(irma, _, _, _)]).
jeu(D) :- donnees(D), assigner(D, [pavillon, chateau, studio, caravanne], [chien, cheval, poisson, coq], [etud,
gardien, instit, toubib]).

assigner([], _, _, _).
assigner([ass(I, M, A, P) | LR], LM, LA, LP) :- app(M, LM), app(A, LA), app(P, LP),
        assoc(M, P, chateau, etud), assoc(M, A, caravanne, coq), imp(A, P, poisson, toubib),
        imp(I, P, irma, toubib), imp(I, P, irma, gardien),
        imp(I, P, luc, gardien), imp(I, M, eve, pavillon), imp(M, A, studio, cheval), ret(M, LM, LMR),
        ret(A, LA, LAR), ret(P, LP, LPR), assigner(LR, LMR, LAR, LPR).
jeu(D). -> D = [ass(max,pavillon,chien,gardien), ass(luc,studio,poisson,instit), ass(eve,caravanne,coq,toubib),
ass(irma,chateau,cheval,etud)] ; -> D = [ass(max,pavillon,chien,toubib), ass(luc,studio,poisson,instit),
ass(eve,caravanne,coq,gardien), ass(irma,chateau,cheval,etud)] ; -> no (deux solutions)
```

15-45° Automate de Turing Sur l'alphabet "unaire" A = {blanc, 1} où "blanc" représente un espace séparateur, on simule une mémoire non bornée organisée en "ruban". Une transition (q, x, q', y, action) signifie qu'à l'état q, pointant sur le symbole x, on passe à l'état q' en remplaçant x par y ret en se déplaçant à droite, à gauche ou pas. En notant G, D les listes de gauche et droite, et en structurant une configuration avec (état, gauche, tête, droite), programmer l'automate et trouver des transitions pour calculer les fonctions successeur et addition.

```
suivant(config(Q1, G1, X, D1), config(Q2, G2, Y, D2)) :- trans(Q1, X, Q2, Z, Dep),
                                        modif(config(Q1, G1, X, D1), Z, Dep, config(Q2, G2, Y, D2)).

modif(config(_, G, X, []), Y, droite, config(_, [Y | G], blanc, [])).
modif(config(_, [], X, D), Y, gauche, config(_, [], blanc, [Y | D])).
modif(config(_, G, X, [A | D]), Y, droite, config(_, [Y | G], A, D)).
modif(config(_, [A | G], X, D), Y, gauche, config(_, G, A, [Y | D])).
modif(config(_, _, X, D), Y, rien, config(_, _, Y, D)).
calcul(C1, C2) :- suivant(C1, C2), !.
calcul(C1, C2) :- suivant(C1, C3), calcul(C3, C2).
resultat([X | D], [Y | R]) :- calcul(config(init, [], X, D), config(final, _, Y, R)).
resultat([], [Y | R]) :- calcul(config(init, [], blanc, []), config(final, [], Y, R)).
```

Exemple de la fonction succeseur

```
trans(init, 1, raj, 1, gauche).
trans(raj, blanc, final, 1, rien).
resultat([1,1,1,1], R). -> R = [1,1,1,1,1] successeur de tout entier non nul
```

Exemple de l'addition

```
trans(init, 1, av, 1, droite).          % on démarre sur la tête du premier argument
trans(av, 1, av, 1, droite).            % avancée sur les 1
trans(av, blanc, rec, 1, gauche).       % arrivée au séparateur, on le remplace et recule
trans(rec, 1, rec, 1, gauche).          % retour au début
trans(rec, blanc, supp, blanc, droite). % on va supprimer le tout premier 1
trans(supp, 1, bf, blanc, droite).
trans(bf, 1, final, 1, rien).
resultat([1,1,blanc,1,1,1], R). -> R = [1,1,1,1,1] ; no
resultat([1,1, 1, 1,blanc,1,1,1], R). -> R = [1,1,1,1,1,1,1] ; no
```

Exemple de la soustraction partielle

```
trans(init, 1, av1, 1, droite).          % avancées dans le premier argument
trans(av1, 1, av1, 1, droite).
trans(av1, blanc, voir, blanc, droite).  % voir s'il y a encore qqch à soustraire
trans(voir, 1, av2, 1, droite).
trans(voir, blanc, recb, blanc, gauche).
trans(recb, blanc, bf, blanc, gauche).   % bf = bientôt-fini
trans(av2, 1, av2, 1, droite).
trans(av2, blanc, efd, blanc, gauche).   % efd doit effacer à droite
trans(efd, 1, rec2, blanc, gauche).      % efd avance donc tant qu'il y a des 1
trans(rec2, 1, rec2, 1, gauche).         % recule sur le second argument
trans(rec2, blanc, efg, blanc, gauche).  % efg va devoir effacer un 1 à gauche
trans(efg, 1, efg, 1, gauche).
trans(efg, blanc, blg, blanc, droite).   % blg va devoir mettre un blanc et ravancer
trans(blg, 1, av1, blanc, droite).
trans(bf, 1, bf, 1, gauche).
trans(bf, blanc, final, blanc, droite).

resultat([1,1,1,1,1,blanc,1,1,1], R). -> R = [1,1,blanc,blanc,blanc,blanc,blanc]
resultat([1,1,blanc,1,1], R). -> R = [blanc,blanc,blanc,blanc]
```

15-46° Numération anglaise Ecrire un prédicat nombre" établissant la relation entre une liste de mots anglais traduisant un nombre inférieur à 100, et la valeur de ce nombre

```
chiffre(one, 1). chiffre(two, 2). chiffre(three, 3). chiffre(four, 4). chiffre(five, 5). chiffre(six, 6). chiffre(seven, 7).
chiffre(eight, 8). chiffre(nine, 9). gdchif(ten,10). gdchif(eleven,11). gdchif(twelwe, 12). gdchif(thirteen, 13).
gdchif(forteen, 14). gdchif(fifteen, 15). gdchif(sixteen, 16). gdchif(seventeen, 17). gdchif(eighteen, 18).
gdchif(nineteen, 19). diz(twenty, 20). diz(thirty, 30). diz(forty, 40). diz(fifty, 50). diz(thirty, 30). diz(forty, 40).
diz(sixty, 60). diz(seventy, 70). diz(eighty, 80). diz(ninety, 90).
nombre(X, N) :- trois(X, N),!.
nombre(X, N) :- deux(X, N),!.
nombre([U], N) :- chiffre(U, N),!.
nombre([zero], 0).
trois([C, hundred |L], N) :- chiffre(C,V), restetrois(L,M), N is (100*V + M).
        restetrois([], 0).
        restetrois([and | L], N) :- deux(L, N).
deux([D|L], N) :- diz(D, V), restedeux(L, M), N is (V+M).
deux([C], N) :- gdchif(C, N).
deux([U], N) :- chiffre(U, N).
        restedeux([U], N) :- chiffre(U, N).
        restedeux([], 0).
```

Exemples :

```
nombre([three, hundred], X). → X = 300
nombre([sixty, five], X). → X = 65
nombre([one, hundred, four], X). → no
nombre([five, hundred, and, four], X). → X = 504
nombre([two, hundred, and, twenty, four], X). → X = 224
nombre([seven, hundred, and, ninety],X). → X = 790
nombre(L, 15). → L = [fifteen]
nombre(L, 243). → L = [two,hundred,and,forty,three]
nombre(L, 782). → L = [seven,hundred,and,eighty,two]
nombre(L, 999). → L = [nine,hundred,and,ninety,nine]
```

15-47° Numération japonaise Traduire en nombre une expression japonaise sachant que la numération est complètement régulière et que itchi=1, ni=2, san=3, shi=4, go=5,r oku=6, shitchi=7, hatchi=8, ku=9, giu=10, hyaku=100, sen=1000, man=10000

```
donnees([itchi,1,ni,2,san,3,shi,4,go,5,roku,6,shitchi,7, hatchi,8, ku,9, giu,10, hyaku,100, sen,1000, man,10000]).
present(X,U,[X,U|_]).
present(X,U,[_,_|L]) :- present(X,U,L).
val(X,U) :- donnees(D), present(X,U,D).
jap([X,Y|L],N) :- !, val(X,U), val(Y,V), jap(L,M), N is (M+(U*V)).
jap([X],N) :- !, val(X,N).
jap([], 0).                    % ne marche que dans un sens
                  jap([ni, sen, hatchi,hyaku,go,giu,san], N). → N = 2853
                  jap([ni,man,ku,sen,itchi,hyaku,roku,giu,shitchi], N). → N = 29167
```

15-48° Une crypt-addition SUEDE + SUISSE = BRESIL
Chaque lettre représente un chiffre, deux lettres distinctes sont associées à deux chiffres distincts, et S et B ne peuvent être zéro. Indication : div (ou // est la division entière), si ≠ n'existe pas, on écrira, comme cela a déjà été fait par le symbole \==.

```
app(X, [X|_]).
app(X, [_|L]) :- app(X, L).
chif(0).  chif(1). chif(2). chif(3). chif(4). chif(5). chif(6). chif(7). chif(8). chif(9).
retenue(0). retenue(1).
distincts([]).
distincts([X | L]) :-  not(app(X, L)), distincts(L).
suede :- som(0, E, E, L, R1), som(R1, D, S, I, R2), som(R2, E, S, S, R3),
         som(R3, U, I, E, R4), som(R4, S, U, R, R5), som(R5, 0, S, B, 0), distincts([S,E,D,L,I,B,U,R]),
         write( 'e='), write(E), write(' l='), write(L), write(' d='), write(D), write(' s='), write(S),
         write(' u='), write(U), write(' i='), write(I), write(' b='), write(B).
som(R, X, Y, Z, NR) :- retenue(R), chif(X), chif(Y), T is (X+Y+R), Z is (T mod 10), NR is (T / 10).
```

Par exemple : distincts([a, z, e,r ,t , y]). → yes distincts([f, a, d, a]). → no
Appel avec : suede. Réponse donnée : e=9 l=8 d=7 s=4 u=6 i=2 b=5 yes

15-49° Réaliser la simplification des expressions arithmétiques en écrivant toutes les règles de simplification. Exemple a*(b-b)+5-c+1 donne 6-c.
On devra écrire toutes sortes de clauses de réduction pour les différentes opérations.

On traite la commutativité en écrivant les termes comportant une valeur numérique de telle sorte qu'elle débute un produit et termine une somme.

```
simp(U+V, R) :- number(U), number(V), !, R is U+V.
simp(A+X, R) :- number(A), !, simp(X+A, R).
simp(U+V, R) :- simp(U, US), simp(V, VS), (U \== US ; V \== VS), !, simp(US+VS, R).
simp(U+(V+W), R) :- simp(U+V, PS), (PS \== U+V), !, simp(PS+W, R).      % associativité sélective
simp((U+V)+W, R) :- simp(V+W, PS), (PS \== V+W), !, simp(U+PS, R).
simp(X+0, R) :- !, simp(X, R).

simp(U*V, R) :- number(U), number(V), !, R is U*V.
simp(X*A, R) :- number(A), !, simp(A*X, R).
simp(U*V, R) :- simp(U, US), simp(V, VS), (U \== US ; V \== VS), !, simp(US*VS, R).
simp(U*(V*W), R) :- simp(U*V, PS), (PS \== U*V), !, simp(PS*W, R).      % associativité sélective
simp((U*V)*W, R) :- simp(V*W, PS), (PS \== V*W), !, simp(U*PS, R).
simp(1*X, R) :- !, simp(X, R).
simp(0*X, 0) :- !.

simp(U*X+V*X, R) :- !, simp((U+V)*X, R).
simp(U*X-V*X, R) :- !, simp((U-V)*X, R).
simp(X-0, R) :- !, simp(X, R).
simp(X-X, 0) :- !.
simp(U-V, R) :- number(U), number(V), !, R is U-V.
```

```
simp(A-X, R) :- number(A), !, simp(-X+A, R).
simp(U-V, R) :- simp(U, US), simp(V, VS), (U \== US ; V \== VS), !, simp(US-VS, R).
simp((U-V)+W, R) :- !, simp(U-(V-W), R). % et bien d'autres règles
simp(X/1, R) :- !, simp(X, R).
simp(X/X, 1) :- !.
simp(U/V, R) :- number(U), number(V), !, R is U/V.
simp(X^0, 1) :- !.
simp(X^1,R) :- !, simp(X, R).
simp(X*X, R^2) :- !, simp(X, R).
simp(X^N*X, R^M) :- !, simp(X, R), M is N+1.
simp(X^N*X, R^M) :- !, simp(X, R), M is N+1.
simp(X^N*X^M, R^S) :- !, simp(X, R), S is N+M.
simp(sin(X), Y) :- simp(X, R), number(R), !, Y is sin(R).
% Il est impossible de donner une règle du second ordre indiquant d'évaluer F(X) pour tout F
simp(X, X). % dernier cas obligatoire pour avoir par exemple simp(x+y, R). R = x+y

% simp(3+x+2 , R). → R = x+5 ;
% simp(x/x, R). → R = 1 ;
% simp(2+5+3, R). → R = 10 ;
% simp(2*3+4*5, R). → R = 26 ;
% simp(2*3*2*x + 2*2*3*x^1, R). → R = 24*x ;
% simp(2*3*2*x - 2*2*3*x^1, R). → R = 0 ;
% simp(5*3*x-3*4*x , R). → R = 3*x ;
% simp(x*x, R). → R = x^2
% simp(x^3*x^2*x, R). → R = x^6
% simp(3*x^3*x^2*x - 2*x^6, R). → R = x^6 ;
% simp(3*x^3*2*x^2*x*2 + 2*x^6*1*5, R). → R = 22*x^6 ;
% simp(x*2*3 + 5*x - 2*x, R). → R = 9*x ;
% simp(2+3*x*x+3*4+x*1, R). → R = 3*x^2+(x+14) ;
```

C'est encore très incomplet, simp(x*2*3 + 5*x - 1*x*1, R). ne marche pas sauf en rajoutant des règles fastidieuses telles que : simp(U*X+X, R) :- !, simp((U+1)*X, R).
L'étape suivante est de compléter par des règles d'ordonancement pour ranger par exemple les polynômes dans l'ordre décroissant $7x^3 + 3x^2 - 8x + 2$, faire disparaitre les signes de multiplication et d'exponentiation, séparer les monômes par des blancs et les ranger dans l'ordre alphabétique suivant l'usage $ab^2cx^3y^2z$ par exemple.

15-50° Cryptadditions Il s'agit de résoudre des équations (uniquement des sommes) où chaque lettre (10 au maximum) représente de façon injective un chiffre (non nul s'il est en début de mot). Le programme Prolog minimum demandé pourra prendre les mots comme liste de leurs lettres, ainsi "femme" sera [f,e,m,m,e], et devra bien sûr afficher les associations f=6, e=1, m=7... trouvée s'il y en a. Quelques exemples à tester : pere + mere = bebe, nuit + lit = reve, eve + eve = adam, homme + femme = amour, one + nine + fifty + twenty = eighty, roue + roue = velo, jupiter + saturne + uranus = neptune, un + peu + fou = fada, zwei + drei + vier = neun, bah + bah = rhum, cuire + en + poele = frire et aussi : ain + aisne + drome + marne = somme qui donne (1+2+26+51=80) e=5, n=0, i=9, m=3, r=6, o=8, s=7, a=1, d=2.
Extensions possibles :
a) Réaliser la transformation pere+mere -> [p,e,r,e], [m,e,r,e] pour avoir par exemple jeu(plein+soleil=bruler). → n=6, l=1, r=7, i=5, e=0, u=2, p=8, o=9, s=3, b=4
Détail technique "ab+12" est identique à [97,98,43,49,50], et encore, name(azerty, X). renvoie X = [97,122,101,114,116,121]. Cependant "name" ne veut pas d'un mélange de lettres et d'autres signes.
b) Chercher des multiplications comme 6*un=six, 2*oeil=yeux, 4*patte=chien, lyndon*b=johnson, fermat*s = last + theorem, ice*rips = titanic ou limacons * oo = blablabla sachant que limacons*o=nagnagna, une division : jam,es / bond = o,ox dans laquelle o = 0 et x = 7..... Réponses 286,51 / 4093 et 216,58 / 3094. (Problèmes issus du journal de Spirou.)

```
chif(0). chif(1). chif(2). chif(3). chif(4). chif(5). chif(6). chif(7). chif(8). chif(9).
% une clause chif(X) :- (-1 < X), (X < 10). ne peut convenir car il faut forcer X à être instancié.

inv(L, LR) :- invbis(L, [], LR).      % Inverse d'une liste
```

```
invbis([], R, R).
invbis([X|L], R, LR) :- invbis(L, [X|R], LR).
toutinv([],[]).      % réalise un mapcar de "inverse" sur une liste
toutinv([X|L], [IX| IL]) :- inv(X, IX), toutinv(L, IL).

ass(X, V, [], [[X, V]]).                % La lettre X est associée à la valeur V
ass(X, V, [[X, V]|R], [[X, V]|R]).
ass(X, V, [[X, W]|_], _) :- V \== W, fail.
ass(X, V, [[Y, V]|_], _) :- X \== Y, fail.
ass(X, V, [[Y, W]|R], [[Y, W]|NR]) :- X \== Y, V \== W, ass(X, V, R, NR).

reso(R, [], [], [Z], CH) :- chif(R), R \== 0, ass(Z, R, CH, NCH), write(NCH). % CH, NCH  listes des choix
reso(S, [[X]|RL], LR, T, CH) :- chif(V), V \== 0, ass(X, V, CH, NCH), % RL lignes restantes à sommer
        NS is (S + V), reso(NS, RL, LR, T, NCH).                % cas du premier chiffre X d'une ligne
% LR la liste de référence constituant ce qui restera à additionner, T le total, X le
% chiffre courant de la colonne dont on connait déjà la somme S, NS la nouvelle somme
% S+valeur(X) et Z est le chiffre du total correspondant à la colonne.
reso(S, [[X|L]|RL], LR, T, CH) :- chif(V), ass(X, V, CH, NCH),  % L est une ligne
        NS is (S + V), reso(NS, RL, [L|LR], T, NCH).            % RL lignes restantes à sommer
reso(S, [[]|RL], LR, T, CH) :- reso(S, RL, LR, T, CH).          % cas d'une ligne épuisée
reso(S, [], LR, [Z, Y|T], CH) :- U is (S mod 10), ass(Z, U, CH, NCH),
        NR is (S / 10), reso(NR, LR, [],[Y|T], NCH).            % cas de fin de colonne
solution(LT, R) :- inv(R, RR), toutinv(LT, RLT), reso(0, RLT, [], RR, []).

Mais on a mieux, sans "inv" ni "toutinv", tr(C, _, R, _, S) réalisant le décodage d'une chaîne C en la liste des
listes des opérandes R et de la somme S.
tr([61,Y|L],R,RO,[],S) :- tr(L,R,RO,[Y],S), !.          % signe = ascii 61
tr([43|L],R,RO,[],S) :- tr(L,[[]|R],RO,[],S), !.        % signe + ascii 43
tr([X|L],[T|R],RO,[],S) :- tr(L,[[X|T]|R],RO,[],S), !.
tr([X|L],R,RO,SP,S) :- tr(L,R,RO,[X|SP],S), !.          % SP=somme partielle S=somme
tr([],R,R,S,S).
jeu(CH) :- tr(CH,[[]],R,[],S), lettres(R, LR), lettres(S, LS), reso(0, LR, [], LS, []).
lettres([], []). % correspondance entre liste de code et liste de lettres
lettres([N|L], [A|LL]) :- number(N), name(A, [N]), lettres(L, LL).
lettres([LN|R], [LA|LR]) :- lettres(LN, LA), lettres(R, LR).

jeu("lit+nuit=reve"). → [[t,1],[e,2],[i,3],[v,6],[u,5],[l,7],[n,8],[r,9]]
jeu("li+li=cii"). → [[i,0],[l,5],[c,1]]
jeu("suede+suisse=bresil"). → [[e,9],[l,8],[s,4],[d,7],[i,2],[u,6],[r,0],[b,5]]
jeu("pere+mere=bebe"). → [[e,0],[r,2],[b,4],[m,1],[p,3]]
jeu("soleil+sable=bikini").→ [[l,6],[e,7],[i,3],[n,0],[b,5],[a,1],[k,8],[o,9],[s,4]]
jeu("homme+femme=amour"). → [[e,1],[r,2],[m,7],[u,4],[o,5],[h,3],[f,6],[a,9]]
jeu("eve+eve=adam"). → [[e,6],[m,2],[v,5],[a,1],[d,3]]
jeu("fort+erie=ville"). → [[t,0],[e,6],[i,4],[r,8],[l,2],[o,3],[f,7],[v,1]]
jeu("megot+megot=clopes"). → [[t,2],[s,4],[o,3],[e,6],[g,5],[p,0],[m,8],[l,7],[c,1]]
jeu("coup+d+epee=botte"). → [[p,2],[d,8],[e,4],[u,0],[t,5],[o,3],[c,9],[b,1]]
jeu("ne+au+cap=eban"). → [[e,1],[u,2],[p,3],[n,6],[a,4],[c,9],[b,0]]
jeu("jupiter+saturne+uranus=neptune"). →  [[r,8],[e,6],[s,2],[u,3],[n,4],[t,0],[a,9],[i,7],[p,5],[j,1]]
jeu("roue+roue=velo"). → [[e,3],[o,6],[u,5],[l,0],[r,4],[v,9]]
jeu("roue+roue+roue+roue=auto"). →  [[e,6],[o,4],[u,9],[t,8],[r,1],[a,5]]
jeu("rennes+nantes=angers"). → [[s,0],[e,3],[r,6],[n,2],[t,1],[g,4],[a,9]]
jeu("jean+aime=marie"). → [[n,0],[e,4],[m,1],[a,7],[i,8],[r,2],[j,9]]
jeu("six+cinq=onze"). → [[x,1],[q,2],[e,3],[n,4],[i,6],[z,0],[s,7],[c,8],[o,9]]
jeu("un+un+neuf=onze"). → [[n,1],[f,7],[e,9],[u,8],[z,4],[o,2]]
jeu("filo+filino+patro+patrino=familio"). →  [[o,0],[n,1],[r,2],[l,5],[i,9],[t,8],[a,3],[f,6],[p,4],[m,7]]
jeu("rosee+froid=givre"). → [[e,1],[d,0],[i,2],[r,3],[s,6],[o,8],[v,4],[f,5],[g,9]]
jeu("bah+bah=rhum"). → [[h,4],[m,8],[a,3],[u,6],[b,7],[r,1]] (bon test)
jeu("un+peu+fou=fada"). → [[n,2],[u,4],[a,0],[o,3],[e,7],[d,5],[p,8],[f,1]] (bon test, un peu long)
jeu("four+three=seven"). → [[r,2],[e,3],[n,5],[u,0],[o,9],[v,1],[h,4],[f,8],[t,6],[s,7]]
jeu("ten+ten+twenty+twenty=twenty=eighty"). → [[n,4],[y,6],[t,1],[e,3],[h,5],[g,0],[w,2],[i,7]]
jeu("tenia+du+chien=cenure"). → [[a,2],[u,3],[n,5],[e,0],[d,8],[i,7],[r,6],[h,4],[t,9],[c,1]
jeu("cinq+cinq+vingt=trente"). → [[q,3],[t,1],[e,7],[g,5],[n,8],[i,4],[c,6],[v,9],[r,0]]
jeu("cuire+en+poele=frire"). → [[e,9],[n,1],[l,0],[r,2],[i,7],[o,5],[u,6],[c,3],[p,4],[f,8]]
jeu("send+more=money"). → [[d,7],[e,5],[y,2],[r,8],[n,6],[o,0],[m,1],[s,9]]
```

jeu("belier+cancer+balance=taureau"). → [[r,1],[e,7],[u,9],[c,4],[a,8],[i,2],[n,0],[l,3],[b,5],[t,6]]
jeu("donald+gerald=robert"). → [[d,5],[t,0],[l,8],[r,7],[a,4],[e,9],[n,6],[b,3],[o,2],[g,1]] (10 chiffres)
jeu("fraise+citron+cerise=raisin"). → [[e,0],[n,5],[s,1],[o,4],[i,6],[r,9],[t,8],[a,7],[c,3],[f,2]]
jeu("zwei+drei+vier=neun"). → [[i,4],[r,0],[n,8],[e,9],[u,7],[w,3],[v,1],[d,2],[z,5]]
jeu("cuatro+cuatro+cuatro+cuatro+cuatro=veinte").→ [[o,9],[e,5],[r,6],[t,4],[n,3],[a,0],[i,2],[u,7],[c,1],[v,8]]
jeu("one+nine+fifty+twenty=eighty"). → [[e,2],[y,6],[t,1],[n,9],[o,0],[i,4],[f,8],[h,3],[g,7],[w,5]]
jeu("piano+sonata=fifteen"). → [[o,4],[a,8],[n,2],[t,3],[e,6],[i,0],[p,7],[f,1],[s,9]]
jeu("expense+causes=account"). → [[e,4],[s,6],[t,0],[n,1],[u,8],[o,2],[p,7],[a,5],[c,3],[x,9]]
jeu("nasa+has+stars+in=sight"). → [[a,0],[s,2],[n,1],[t,5],[i,6],[r,9],[h,7],[g,8]]
jeu("power+for=peace"). → [[r,2],[e,4],[o,3],[c,7],[w,1],[f,9],[a,0],[p,5]]
jeu("alaska+kansas+texas=states"). → [[a,4],[s,6],[k,1],[e,0],[x,9],[t,2],[n,8],[l,5]]
jeu("enigme+enigme=colles"). → [[e,4],[s,8],[m,7],[g,5],[l,1],[i,0],[n,6],[o,2],[c,9]]
jeu("une+demi+finale=facile"). → [[e,2],[i,8],[l,0],[m,4],[n,5],[u,6],[a,9],[d,7],[c,3],[f,1]]
jeu("seven+seven+six=twenty"). → [[n,2],[x,0],[y,4],[i,5],[e,8],[t,1],[v,7],[s,6],[w,3]]
jeu("pleine+lune=crimes"). → [[e,2],[s,4],[n,6],[u,3],[i,1],[m,5],[l,9],[r,0],[p,7],[c,8]]
jeu("clinton+monica=cancans"). → [[n,2],[a,9],[s,1],[c,4],[o,7],[t,3],[i,5],[m,8],[l,0]]
jeu("calcul+mental=eleves"). → [[l,7],[s,4],[a,8],[u,0],[e,9],[c,5],[t,1],[v,6],[n,2],[m,3]]
jeu("calcul+integral=ingenier"). → [[l,4],[r,8],[a,9],[u,7],[e,6],[c,3],[i,2],[g,5],[n,0],[t,1]]
jeu("gilles+cohen=genial"). → [[s,1],[n,3],[l,4],[e,8],[a,6],[h,0],[i,5],[o,9],[c,2],[g,7]]
jeu("un+one+uno=euro"). → [[o,8],[e,1],[n,9],[u,3],[r,2]]
jeu("bug+bug+bug+bug+bug=win"). → [[g,2],[n,0],[u,5],[i,6],[b,1],[w,7]]
jeu("gin+fizz=hips"). → [[z,2],[n,1],[s,3],[i,8],[p,0],[g,9],[f,4],[h,5]]
jeu("pierre+peter=pareil"). → [[r,6],[e,8],[l,4],[i,5],[t,1],[p,3],[a,9]]
jeu("jean+lisa=amis"). → [[a,3],[n,4],[s,7],[i,0],[e,5],[m,6],[j,1],[l,2]]
jeu("chien+chien=meute"). → [[n,3],[e,6],[t,2],[i,0],[u,1],[h,8],[c,4],[m,9]]
jeu("hasta+la+vista=salut"). → [[a,9],[t,7],[l,8],[u,4],[s,3],[i,0],[v,1],[h,2]]

15-51° Problème simplifié des missionnaires : ils sont N sur la rive d'un fleuve en compagnie de N cannibales (sur la même rive contrairement au problème du chapitre 15). Avec les mêmes conditions que pour le premier problème, ils doivent tous passer sur l'autre rive grâce à une barque contenant un nombre pair de personnes.

15-52° Voyage interplanétaire : calculer le coût d'un voyage dans le système solaire sachant que si u est l'unité de base : prix du voyage entre deux satellites consécutifs d'une planète, le prix d'une planète à l'autre sera proportionnel à leur distance augmenté du transfert à leur satellite le plus éloigné. Le prix Terre-Mercure est fixé à k*u, les distances entre planètes sont prises en moyenne avec $0,3(2^{n-2} - 2^{m-2})$ si elles sont de rang n et m. On donne le système :
Rang 1 : Mercure, 2 : Vénus, 3 : Terre (lune), 4 : Mars (Phobos, Deimos), 5 : Astéroïdes, 6 : Jupiter (Métis, Adrastée, Amaltée, Thébé, Io, Europe, Ganimède, Callisto, Léda, Himalia, Elara, Lysithéa, Ananka, Carme, Pasiphaé, Sinope), 7 : Saturne (Atlas, Prometheus, Pandora, Epiméthéus, Janus, Mimas, Encelade, Calypso, Thétys, Télesto, Dioné, Hélène, Rhéa, Titan, Hypérion, Japet, Phoebé ...), 8 : Uranus (Juliet, Puck, Bianca, Cordélia, Cressida, Desdemone, Ophélie, Rosalind, Portia, Bélinda, Miranda, Ariel, Umbriel, Titania, Obéron) 9 : Neptune (Naiad, Thalassa, Galatea, Larissa, Proteus, Triton, Néréide) 10 : Pluton (Charon).

15-53° Exemple d'arbres, le jeu des pions : initialement on a B B _ N N N N, on doit avoir N N N N _ B B en déplaçant le trou, ou bien en déplaçant symétriquement N B _ pour avoir _ N B ou le contraire.

On va représenter l'état du jeu par un arbre binaire trou(gauche, droite) et les listes de chaque côté comme noir (ce qui suit) ou blanc (ce qui suit) orientée à partir du trou. On désigne par nil la constante arbre sans feuilles. Les règles sont :

```
deplacer(trou(A, noir(B)), trou(noir(A), B)).
deplacer(trou(blanc(A), B), trou(A, blanc(B))).
deplacer(trou(noir(blanc(A)), B), trou(A, noir(blanc(B)))).
deplacer(trou(A, blanc(noir(B))), trou(blanc(noir(A)), B)).
init(trou(G, D)) :-  toutblanc(G), toutnoir(D).
final(trou(G, D)) :-  toutnoir(G), toutblanc(D).
toutnoir(noir(A)) :- toutnoir(A).          toutblanc(blanc(A)) :-  toutblanc(A).
```

```
toutnoir(nil).                          toutblanc(nil).
resoudre(E) :- init(E), reso(E, S), ecrire(S).  % E état, NE=nouvel état, CH=chemin
reso(E, [E | CH]) :-  deplacer(E, NE), reso(NE, CH).
reso(E, [E]) :-  final(E).
ecrire([X | L]) :-  write(X), nl, ecrire(L).      ecrire([]).
```

Exemple : resoudre(trou(blanc(blanc(nil)), noir(noir(noir(noir(nil)))))). →

```
trou(blanc(blanc(nil)),noir(noir(noir(noir(nil)))))
trou(noir(blanc(blanc(nil))),noir(noir(noir(nil))))
trou(blanc(nil),noir(blanc(noir(noir(noir(nil))))))
trou(nil,blanc(noir(blanc(noir(noir(noir(nil)))))))
trou(blanc(noir(nil)),blanc(noir(noir(noir(nil)))))
trou(blanc(noir(blanc(noir(nil)))),noir(noir(nil)))
trou(noir(blanc(noir(blanc(noir(nil))))),noir(nil))
trou(noir(blanc(noir(nil))),noir(blanc(noir(nil))))
trou(noir(nil),noir(blanc(noir(blanc(noir(nil))))))
trou(noir(noir(nil)),blanc(noir(blanc(noir(nil)))))
trou(blanc(noir(noir(noir(nil)))),blanc(noir(nil)))
trou(blanc(noir(blanc(noir(noir(noir(nil)))))),nil)
trou(noir(blanc(noir(noir(noir(nil))))),blanc(nil))
trou(noir(noir(noir(nil))),noir(blanc(blanc(nil))))
trou(noir(noir(noir(noir(nil)))),blanc(blanc(nil)))      yes
```

15-54° Exemple d'utilisation des primitives "retract" et "assert", le jeu des animaux
Deux primitives permettent de modifier l'état de la base de connaissance en cours d'éxecution. "assert(P)" permet de rajouter la clause P dans cette base, "asserta" la rajoute au début, et "assertz" à la fin. "retract(P)" permet de retirer la clause P de la liste des clauses.
Cet exemple qui se trouve dans beaucoup de manuels, consiste à créér un arbre binaire en posant des questions auxquelles on doit répondre par oui ou non, pour déterminer un animal. Mais si le joueur propose un animal qui n'est pas déjà dans l'arbre initial, on va lui demander son nom et ce qui le distingue d'un autre de façon à enrichir l'arbre au cours des sessions.

```
animal('il a des plumes '(poule, cochon)).          %  c'est simplement l'arbre initial
jeu :- repeat, animal(X), demander(X, Y), changer(X, Y), write('encore ? '), read(non).
demander(X, Y) :- functor(X,_ , 0), ! , write('c"est '), write(X), nl, verif(X, Y). % cas d'une feuille donc d'arité 0
demander(X, Y) :- functor(X, F, 2), write(F), write('? '), read(reponse), determiner(X, reponse, Y).
determiner(X, non, Y) :-   X =.. [F, U, V], demander(V, W), Y =.. [F, U, W].
determiner(X, oui, Y) :- X =.. [F, U, V], demander(U, W), Y =.. [F, W, V].
        % pose des questions dans le fils droit ou gauche
changer(X, X).
changer(X, Y) :- X \== Y, retract(animal(X)), asserta(animal(Y)).
verif(X, Y) :- write(' ok ? '), read(reponse), construire(X, reponse, Y).
construire(X, oui, X).
construire(X, non, Y) :-   write('quel est son nom ? '), read(nom), write('qu"est ce qui le distingue de '), write(X),
                           write('? '), read(phrase),Y =.. [phrase, nom, X].
% nom est un nouvel animal et phrase sera une nouvelle question du jeu
```

15-55° Commandes pour un robot Organiser des tâches pour apporter un objet à un endroit. En partant des connaissances suivantes : la clé de la chambre est à la cave, la chambre est fermée, le livre est dans la chambre, la clé de la cave est au grenier, la personne est dans la cuisine, la cave est fermée. Il faut apporter le livre au salon. Faire chercher la solution qui décrit toutes les opérations élémentaires dans l'ordre en se servant des prédicats "assert" et "retract".

```
etre(cle(chambre), cave).
etre(livre, chambre).
etre(cle(cave), grenier).
hommedans(cuisine).
ferme(chambre).
ferme(cave).                % sont les 6 données
sep :- write(', ').         % fait l'affichage d'un séparateur
apporter(Ob, Lieu) :- prendre(Ob), aller(Lieu), poser(Ob, Lieu).
```

```
poser(Ob, E) :- main(Ob), aller(E), assert(etre(Ob, E)),retract(main(Ob)),write(poser(Ob)),sep.

prendre(Ob) :- main(Ob).  % on peut tenir en main plusieurs choses
prendre(Ob) :- etre(Ob, E), hommedans(E), assert(main(Ob)), retract(etre(Ob, E)), write(prendre(Ob)), sep.
prendre(Ob) :- etre(Ob, E), aller(E), prendre(Ob), sortir(E).
aller(E) :- hommedans(E).
aller(E) :- hommedans(L), sortir(L), entrer(E).
aller(E) :- entrer(E).
sortir(E) :-  ferme(E),write(enfermé(E)), sep, ouvrir(E), sortir(E).
sortir(E) :-  hommedans(E), retract(hommedans(E)), write(sortir(E)), sep.
entrer(E) :-  hommedans(L), sortir(L), entrer(E).
entrer(E) :-  ferme(E), write(allerporte(E)),sep, write(porteclose(E)), sep, ouvrir(E), entrer(E).
entrer(E) :-  assert(hommedans(E)), write(entrer(E)), sep.      % assert est faux si l'assertion est déjà vérifiée
ouvrir(E) :-  not(ferme(E)).
ouvrir(E) :-  main(cle(E)), retract(ferme(E)), write(ouvrir(E)), sep.
ouvrir(E) :-  prendre(cle(E)).
```

On demande le but : apporter(livre, salon).

sortir(cuisine), allerporte(chambre), porteclose(chambre), allerporte(cave), porteclose(cave), entrer(grenier), prendre(cle(cave)), sortir(grenier), allerporte(cave), porteclose(cave), ouvrir(cave), entrer(cave), prendre(cle(chambre)), sortir(cave), allerporte(chambre), porteclose(chambre), ouvrir(chambre), entrer(chambre), prendre(livre), sortir(chambre), entrer(salon), poser(livre), yes

On pourra rajouter et raffiner pour donner d'autres ordres.

```
refermer(E) :- ferme(E).
refermer(E) :- main(cle(E)), not(ferme(E)), aller(E), write(fermer(E)), sep, assert(ferme(E)).
```

15-56° Construire un prolog sans variables ni symboles fonctionnels, en lisp. Les seuls mots réservés seront "impasse", "sortie" et "arrêt".

```
(de muprolog (BR Q) (cond ; la procédure principale
                          ((null BR) (muprolog (inbase) (inquestion)))
                          ((null Q) (muprolog BR (inquestion))))
                          (t (moteur1 nil Q Q BR BR))
(de xcons (X L) (append L (list X))) ; apposer un X à la fin d'une liste L
(de cdl (L) (if (null (cdr L)) nil (cons (car L) (cdl (cdr L))))) ; tout sauf le dernier de L
(de rg (E L) (if (equal E) (car L)) 0 (1+ (rg E (cdr L))))) ; rang à partir de 0 d'un E dans L
(de inbase (B) (if (null B)          (print '"Entrez vos règles comme listes de listes, la conclusion en dernier.")
                                     (print '"Exemple ((= A B) (= B C) (= A C))") )
                   (let ((L (read))) (if (null L) B (inbase (xcons (append (last L) (cdl L)) B)) )))
(de inquestion () (print '"Posez une question de la forme ((fils JEAN X) (sortie X))") (read))
(de gensym () (if (boundp 'CO) (concat 'X (incr CO)) (set 'CO 0) 'X0))
```

Est un générateur de symboles (que nous redéfinissons) qui construit une nouvelle variable à chaque appel, ce sont X0, X1, X2, ... CO étant un compteur de ces variables

```
(de var (X) (cond ; reconnaissance d'une variable (lettres ou lettres suivies d'un chiffre)
                  ((listp X) nil)
                  ((null (cdr (explode X))) t)
                  ((> 65 (cadr (explode X))) t)))
(de occ (X L) (cond ; teste la présence de X à un niveau quelconque de L
                  ((null L) nil)
                  ((atom (car L)) (or (eq X (car L)) (occ X (cdr L))))
                  (t (or (occ X (car L)) (occ X (cdr L))) )))
(de alpha (Q R) (cond ; renvoie Q sans variable libre commune avec R
                  ((null R) Q)
                  ((listp (car R)) (alpha Q (append (car R) (cdr R))))
                  ((var (car R)) (if          (occ (car R) Q) (alpha (subst (gensym) (car R) Q) (cdr R))
                                              (alpha Q (cdr R))))
                  (t (alpha Q (cdr R))) ))
(de ufp (E C) (cond ; reconnait s'il y a unification possible entre les listes E et C
```

```
                ((null C) (if (null E) t))
                ((null E) (if (null C) t))
                ((eq (car E) (car C)) (ufp (cdr E) (cdr C)))
                ((var (car E)) (ufp (cdr E) (cdr C)))
                ((var (car C)) (ufp (cdr E) (cdr C))) ))
(de eff (F H Q) (cond ; renvoie Q  dans laquelle F est remplacée par les composantes de H
                ((null Q) nil)
                ((equal (car Q) F) (append H (eff F H (cdr Q))))
                (t (cons (car Q) (eff F H (cdr Q)))) ))
```

A chaque appel de "unif", Q et R sont alpha-invariants, E est le littéral de Q à tester et RQ est le reste des littéraux de Q

```
(de unif (Q RQ E C H R) (cond ; donne le nouvel état obtenu à partir de Q grâce à  règle R
                ((null RQ) Q)
                ((ufp E C) (cond
                        ((null E) (eff (car RQ) H Q))
                        ((eq (car E) (car C)) (unif Q RQ (cdr E) (cdr C) H R))
                        ((var (car E)) (unif (subst (car C) (car E) Q) (subst (car C) (car E) RQ)
                                        (subst (car C) (car E) (cdr E)) (cdr C) H R))
                        ((var (car C)) (unif (subst (car E) (car C) Q) (subst (car E) (car C) RQ) (cdr E)
                                        (subst (car E) (car C) (cdr C)) (subst (car E) (car C) H) R))))
                (t (unif Q (cdr RQ) (cadr RQ) (car R) (cdr R) R))))
(de ufr (Q R) (unif Q (cdl Q) (car Q) (car R) (cdr R) R))
(de uf (Q R) (if (member t (mapcar (lambda (E) (ufp E (car R))) Q))
                (ufr (alpha Q R) R)
                Q)) ; uf sera appelée par le moteur, elle fait les substitutions de variables dans Q si  nécessaire
(de moteur1 (LQ Q QU B BR LR) (cond   ; fait en gros la descente dans l'arbre
                ((equal Q QU) (moteur2 LQ Q B BR LR))
                ((member '(impasse) QU) (moteur2 LQ Q nil BR LR)); on force la remontée
                ((eq (caar QU) 'sortie) (print 'réponse: (cadar QU)) (moteur2 LQ Q B BR LR))
                ((eq (caar QU 'arrêt) (print 'réponse: (cadar QU)) 'fini) ; on stoppe l'exploration
                (t      (print 'règle: (rg (car B) BR) '--> QU) ; édition de la règle
                        (moteur1 (cons Q LQ) QU (uf QU (car BR)) (cdr BR) BR (cons (car B) LR) ))))
(de moteur2 (LQ Q B BR LR) (if          ; fait les remontées, mutuellement récursive avec moteur1
                (null B) (if (null LQ) 'fini ; toutes les branches sont épuisées
                        (moteur1 (cdr LQ) (car LQ) (uf (car LQ) (car LR)) ; descente relancée
                                (cdr (member (car LR) BR)) BR (cdr LR)))
                (moteur1 LQ Q (uf Q (car B)) (cdr B) BR LR))) ; descente avec la règle suivante
```

15-57° Retour sur un logigram en muprolog, et résoudre ce logigramme : le flic nourrit un poulet en cabanne, l'épicier se paye un château mais se contente d'un colibri, le prof a un chat. Il y a aussi une grotte et un chien. Où habite le toubib et quel animal possède-t-il ?

```
(set 'BR ( ; On utilise un prédicat quaternaire :
((dif X X Y Z) (impasse))       ((dif X Y X Z) (impasse))       ((dif X Y Z X) (impasse))
((dif X Y Y Z) (impasse))       ((dif X Y Z Y) (impasse))       ((dif X Y Z Z) (impasse))
((dif I J K L))
((prof possede chat))           (flic habite cabanne))          ((epicier habite chateau))
((flic possede poulet))         ((epicier possede colibri))     ((maison grotte))
((maison masure))               ((maison chateau))              ((maison cabanne))
((animal poulet))               ((animal chien))                ((animal colibri))
((animal chat))
((X habite Y) (maison Y))
((X possede Y) (animal Y)) ))
```

On pose la question :

```
(muprolog BR '((toubib habite A) (flic habite B) (prof habite C) (epicier habite D) (toubib possede P) (prof possede Q) (flic possede R) (epicier possede S) (dif P Q R S) (dif A B C D) (arret (toubib A P))))
```

→ (toubib grotte chien)

15-58° Agence matrimoniale Donnez des faits indiquant simplement pour des prénoms, les sexes, tailles, chevelures et âges. Puis viennent des définitions de prédicats "goûts", "recherche", "convenir", "mêmes goûts" et "assortis".

```
((Alfred  homme grand brun mur))          ((VIictor homme moyen bmond jeune))
((Hector homme petit brun mur))           ((Irma femme moyenne blonde moyen))
((Rosa femme petite blonde jeune))        ((Olga femme petite brune mur))
((Alfred gouts class aventurevelo))       ((Victor gouts pop sf ski))
((Hector gouts jazz polar ski)            ((Irma gouts class aventure velo))
((Rosa gouts pop sf s) (s dif boxe))      ; Le sport est tout sauf de la boxe.
((Olga gouts m aventure velo))            ; m peut être instancié par n'importe quoi.
((Alfred recherche grande rousse jeune)) ((Victor recherche t blonde jeune) (t dif grande)
((Hector recherche petite blonde moyen)) ((Irma recherche grand brun moyen))
((Rosa recherche moyen blond jeune))      ((Olga recherche moyen brun mur)) ; C'est tout pour les faits bruts.
((X dif X) (impasse))      ; Ces deux clauses ne peuvent pas être permutées,
((X dif Y))                ; elles définissent la relation "être différents"
((X  convient  Y)          (X  homme  T1 C1  A1) (Y  femme  T2  C2  A2)
                           (X  recherche  T2  C2  A2) (Y  recherche  T1  C1  A1))
((X  meme-gouts  Y) (X  gouts  M  L  S) (Y gouts  M  L  S))
((X  assorti Y) (X  convient  Y) (X  meme-gouts  Y))
```

On lance une recherche par (On pourra vérifier à la main qu'il n'y a pas d'autres solutions.) :

```
(muprolog '((X  assorti Y) (arret (X et Y se-marièrent-et-eurent-beaucoup-d-enfants ))) BM)
→ (Victor et Rosa se-marièrent-et-eurent-beaucoup-d-enfants)
```

15-59° Retour sur Peano Dans la version rudimentaire dite "muprolog", ne comportant en tout et pour tout que trois mots prédéfinis "sortie" pour une édition à l'écran, "arret" de la recherche après un succès et "impasse" pour forcer la remontée. Contrairement aux autres Prolog, elle ne fait pas de différence a priori entre les noms de symboles et ceux de prédicats, cela pourrait présenter de grands dangers de tentatives d'unification entre les deux faits (X dif Y) et (superieur X Y) par exemple, ce qui serait absurde. Par contre, cela permet de choisir des noms de prédicats fractionnés tels que (X plus Y egal Z) ce qui est plus agréable à la vue que (plus X Y Z).

```
((zero suc un)) ((un suc deux)) ((deux suc trois)) ((trois suc quatre)) ((quatre suc cinq)) ((cinq suc six))
((X suc Y) (impasse))              ; il n'y a pas d'autres relations de succession, on se limite à l'intervalle de 0 à 6.
((X  plus zero egal  X))           ; expriment les axiomes de Peano x + 0 = x et x + (y + 1) = (x + y) + 1
((X  plusY  egal  Z) (U  suc  Y) (X  plus  U  egal V) (V  suc  Z))
```

On utilise maintenant cette base BP :

```
(muprolog '((un plus deux egal X) (arret  X))  BP)          → trois
(muprolog '((deux plus  X  egal cinq) (arret X))  BP)       → trois
(muprolog '((X  plus  Y egal deux) (sortie  (X  Y)))  BP)   → (deux zero), (un un), (zero deux)
(muprolog '((trois plus deux egal  X) (arret  X)) BP)       → cinq
```

15-60° Retour sur le coloriage d'une carte, on cherche à colorier une carte, des pays d'Europe avec 4 couleurs en définissant une relation de voisinage qui est symétrique et antiréflexive, une couleur ne pouvant être voisine d'elle-même.

```
((bleu vois rouge)) ((rouge vois vert)) ((vert vois bleu)) ((jaune vois bleu)) ((rouge vois jaune)) ((vert vois jaune))
((X  vois  X) (impasse))
((X voisY) (Y vois X))     ; pour la relation de voisinage, le seul prédicat utilisé est donc "voisin",
; pour colorier huit pays avec les couleurs inconnues A,B, etc... , on pose la longue question :
((F vois D) (F vois I) (A vois I) (H vois D) (B vois H) (S vois F) (A vois D) (D vois L) (D vois S) (I vois S)
(L vois F) (B vois F) (S vois A) (D vois B) (B vois L)
(arret (FRA: F ALL: D BEL: B NED: H LUX: L SUI: S ITA: I AUT: A)))
→ (FRA: bleu  ALL: rouge  BEL: jaune  NED: bleu  LUX: vert  SUI: vert  ITA: rouge  AUT: bleu)
```