Chapitre 5

Expressions structurées

Déjà évoquées, les expressions structurées ou termes sont tous les mots que l'on peut former grâce à des symboles fonctionnels appelés « foncteurs ». Certains sont déjà utilisés et ont une signification sémantique connue de Prolog comme les symboles arithmétiques notés de manière infixe X + 7 ou préfixe +(X, 7) ou bien celui de construction de listes comme [X | L], mais d'autres peuvent être imaginés sans même qu'il y ait une intention opératoire de la part du programmeur, simplement ne serait-ce que pour regrouper des données comme on va le voir dans ce chapitre.

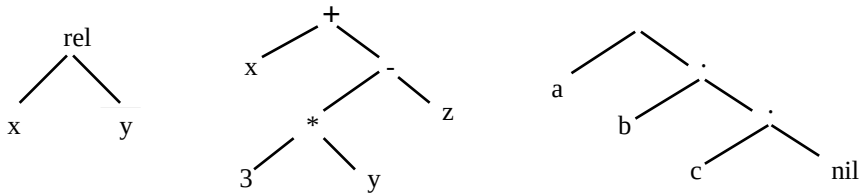
L'algorithme d'unification

Rappelons que Prolog passe son temps à chercher à « unifier » des expressions logiques. Par exemple, lorsqu'il est en présence d'une tête de règle comme *pred(X, Y, Z)* et d'un but comme *pred(fonc(X, Y), X + 7, [X | L])*, il commencera par s'assurer qu'il s'agit du même nom de relation *pred*, ce qui est le cas ici, puis que le nombre d'arguments 3 est respecté. (C'est pourquoi il est possible d'utiliser le même nom pour des relations ayant des nombres d'arguments différents, Prolog ne mélange rien, le programmeur, si.) Après quoi, il renommera ses variables afin de restituer à l'utilisateur ce que celui-ci a demandé, autrement dit, il y aura un remplacement ou « renommage » du *X* de la règle par un *X*1 = *fonc*(*X*, *Y*) et du *Y* par un *Y*1 = *X* + 7, enfin du *Z* par un *Z*1 = [*X* | *L*]. L'unification réussit ici, alors qu'elle aurait échoué entre *pred(g(X))* et *pred(fonc(X, Y))* ainsi qu'entre *pred(X, X – Y)* et *pred(fonc(X, Y), X + 7).*
Les symboles fonctionnels ou foncteurs débutant par des minuscules sont à considérer comme des objets constants au même titre que les opérateurs prédéfinis +, − ou |.

Le traitement des arbres en Prolog

Rappelons que les termes de Prolog sont les atomes (constantes débutant par une minuscule ou valeurs numériques et variables débutant par une majuscule) et les termes structurés avec des « foncteurs ». Par exemple, le prédicat

personne établit dans l'exercice 2 plus loin une relation entre un nom, un âge, une adresse et une profession qui sont des termes structurés. 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 composés par un foncteur et ses arguments peuvent être représentés par des arbres ; ainsi pour une relation à deux paramètres écrite *rel(x, y).* en Prolog, il s'agit de l'arbre de gauche sur le schéma.
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).*
Enfin, la liste [a, b, c] correspond à l'arbre de droite où l'opérateur est la construction . ou |.



Remarque sur un exemple, longueur d'une liste :

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
Il n'y a donc pas de « calcul numérique », mais substitution de l'inconnue *N* par une expression.
Les listes sont des arbres binaires particuliers dont le foncteur a été noté par l'opérateur 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 sur la figure.

On dispose en Prolog d'un prédicat *functor(A, R, N)* établissant une relation entre un arbre *A*, sa racine *R* et son arité *N*.
On a par ailleurs 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].

Le prédicat *arg* donne l'argument de rang spécifié au sein d'un terme, ainsi :

functor(comp(f, g), *F*, *A*). → *F* = comp *A* = 2
arg(2, fonc(f, g), *A*). → *A* = g
functor(+(*X*, 7), *F*, *A*). → *X* = _922 *F* = + *A* = 2
Ici, *X* est une variable, le foncteur *F* est le symbole + et le nombre d'arguments *A* est 2.
Et pour bien comprendre qu'une liste est une construction à deux arguments :
functor([a, b, c], *F*, *A*). → *F* = . *A* = 2
arg(2, [a, b, c], *A*). → *A* = [b, c]
arg(2, [a, b, c], *A*), arg(2, *A*, *X*). → *A* = [b, c] *X* = [c]

X =.. réalise l'inverse :
X =.. [+, a, b]. → *X* = a + b
X =.. [f, a, b, c]. → *X* = f(a, b, c)

et f(a, b, c) =.. *X*. → *X* = [f, a, b, c]

Attention une constante est elle-même un terme d'arité zéro :
a =.. [*X* | *Q*]. → *X* = a *Q* = []

1 Substitution
Un terme *T1* est substitué par un autre terme *T2* au sein d'un troisième devant donner le résultat *R*.

La première clause indique bien le terme à remplacer de façon impérative, c'est-à-dire que la coupure empêche de faire autre chose les deux dernières clauses disent ce qu'il faut faire récursivement pour explorer un terme en passant par la liste associée, et la seconde clause comporte la condition *atom* qui signifie qu'on ne doit rien faire en cas d'un terme atomique différent de ce que l'on a spécifié à substituer, et ceci pour la raison déjà dite qu'une constante est elle-même un terme d'arité zéro.

```
sub(T1, T2, T1, T2) :- !.
sub(_, _, T, T) :- atom(T), !.
sub(T1, T2, [T | Q], [TS | QS]) :- !, sub(T1, T2, T, TS), sub(T1, T2, Q, QS).
sub(T1, T2, T, R) :- T =.. [F | A], sub(T1, T2, A, RA), R =.. [F | RA].
```

sub(cos(x), exp(y), 3*cos(2*x) + 5*cos(x) − 2*log(cos(x)), *R*).
→ *R* = 3*cos(2*x) + 5*exp(y) − 2*log(exp(y))

2 Base de données
Écrire une petite base ou des personnes ont une adresse et une profession, elles-mêmes structurées. Si la base contient un certain nombre de personnes, on doit pouvoir faire des requêtes diverses.

personne(luc, 37, adresse(4, republique, pontarlier), profession(fonc, justice)).
personne(eve, 19, adresse(13, gagarine, privas), profession(prive, employe)).
personne(max, 64, adresse(28, lilas, mende), profession(lib, agriculteur)).
personne(irma, 41, adresse(7, victor-hugo, romorantin),
profession(fonc, educ)).

Exemples

Pour obtenir tout le monde :
personne(*X*, *N*, _, _).
→ *X* = luc *N* = 37 ; *X* = irma *N* = 41 ; *X* = eve *N* = 19 ;
X = max *N* = 64 ; no
Ceux de Privas, les personnels de l'éducation de plus de 40 ans, les agriculteurs du Var, etc. :
personne(*X*, _, adresse(_, _, privas), _). → *X* = eve
personne(*X*, *N*, _, profession(_, educ)), *N* > 40. → *X* = irma *N* = 41

3 Appartenance en profondeur
Construire le prédicat d'appartenance à un arbre binaire étiqueté (mais cet arbre *arb* ne peut être un paramètre, ce serait de la logique du second ordre).

appa(X, arb(X, _, _)).
appa(X, arb(_, G, D)) :- appa(X, G).
appa(X, arb(_, G, D)) :- appa(X, D).
appa(X, X) :- atom(X).

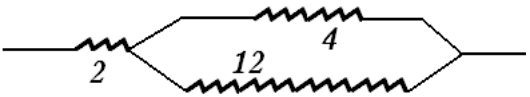
Exemples

appa(a, arb(b, arb(b, c, d), arb(d, e, arb(c, a, d)))). → yes

appa(X, arb(b, arb(b, c, d), arb(d, e, arb(c, a, d)))).
→ *X* = b ; *X* = b ; *X* = c ; *X* = d ; *X* = d ; *X* = e ; *X* = c ; *X* = a ; *X* = d ; no

4 Circuits de résistances électriques
On sait que la résistance équivalente à un circuit en séquence est la somme des résistances et qu'en parallèle, c'est la somme divisée par le produit.

On va décrire un circuit par un terme structuré avec les foncteurs *série* et *parallèle*.



Pour l'exemple ci-contre, on a donc : *res*(*seq*(2, *par*(12, 4)), *R*). → *R* = 5

```
res(seq(X, Y), R) :- !, res(X, RX), res(Y, RY), R is RX + RY.
res(par(X, Y), R) :- !, res(X, RX), res(Y, RY), R is RX*RY / (RX + RY).
res(X, X).
```

5 Exemple du calcul propositionnel
À 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).
sat(non(E)) :- invalide(E).
sat(et(E, F)) :- sat(E), sat(F).
sat(ou(E, F)) :- sat(E).
sat(ou(E, F)) :- sat(F).

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

6 Exemple de l'axiomatique de Peano

L'axiomatique de Peano (1899) consiste à donner les axiomes de base 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. Formellement, soit V un ensemble dénombrable de symboles x, y, … et l'alphabet $L = V \cup \{0, s, +, *, =\}$ avec les axiomes :
$0 + x = x$,
$sx + y = s(x + y)$,
$0*x = 0$,
$sx*y = (x*y) + y$
et le schéma d'axiome de récurrence, pour toute propriété P à une variable libre :
$P(0)$ et $[\forall x\ P(x) \rightarrow P(s(x))] \rightarrow \forall x\ P(x)$

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

7 Dérivation formelle
Donner quelques règles pour que le prédicat *d(E, X, D)* soit satisfait si *D* est l'expression dérivée de *E* suivant la variable *X*.

On décrit ci-dessous un certain nombre de règles de dérivation, il est facile d'en rajouter pour des fonctions particulières telles que sin, cos...
Les coupures permettent à Prolog de gagner du temps puisqu'une seule règle ne s'applique à chaque fois suivant le foncteur racine (une addition, soustraction, exponentielle, etc.). Mais surtout, elles sont indispensables pour que la dernière clause ne s'applique que pour les constantes dont la dérivée est 0.

```
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).
```

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)

Cet exercice soulève un problème lié à la logique du premier ordre.
Dans le cas, par exemple, de la dérivation, on serait tenté, pour la composition de fonctions, de donner une règle comme une de ces deux suivantes :

```
deriv(F(G), X, DF*DG) :- deriv(F, G, DF), deriv(G, X, DG).
d(comp(F, G), X, DF*DG) :- d(F, G, DF), d(G, X, DG).
```

Mais *G* n'est pas un atome et cela ne peut fonctionner non plus.
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 par exemple :

```
deriv(sin(U), X, cos(U) * DU) :- deriv(U, X, DU). ...
```

8 Simplification des expressions arithmétiques
En écrivant le plus possible de 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 fonctionne 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'ordonnancement pour ranger par exemple les polynômes dans l'ordre décroissant $7x^3 + 3x^2 - 8x + 2$, faire disparaître 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.

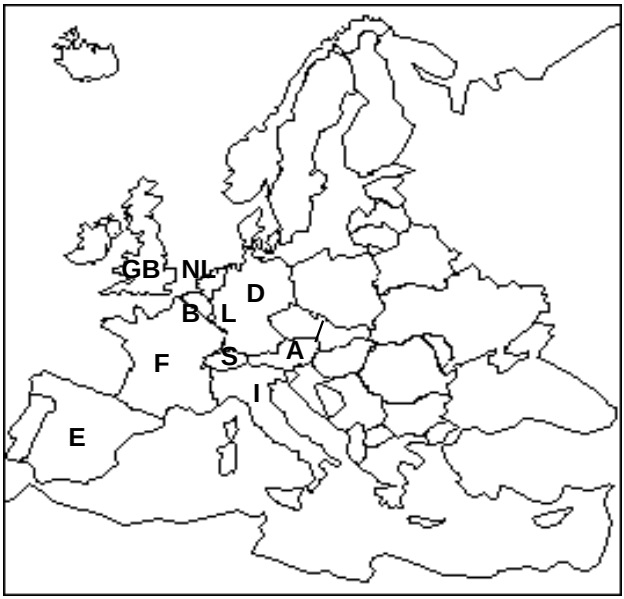
9 Exemple d'arbres, le jeu des pions
Initialement on a B B _ N N N N, on doit avoir N N N N _ B B en déplaçant le trou ou bien en déplaçant symétriquement N B _ pour avoir _ N B ou le contraire.

Contrairement aux grenouilles du chapitre précédent, 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) orientées 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
reso(E, [E | CH]) :-  deplacer(E, NE), reso(NE, CH). % CH = chemin
reso(E, [E]) :- final(E).
ecrire([X | L]) :-  write(X), nl, ecrire(L).
ecrire([]).
```

```
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
```

10 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* portant sur une liste de pays et une 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 inclus 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). % clause principale

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

11 Logigram II

Un exemple des années 1960 consiste en 5 maisons alignées ayant 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. On s'attache à ne prendre que des identificateurs évidents tels que *J* = japonais ou *UT* = ukrainien buvant du thé.

```
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)]).
        % ass = association
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).
```

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

12 Logigram III
Quatre 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 caravane, 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, caravane],
        [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, caravane, 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, caravane, coq, toubib), ass(irma, chateau, cheval, etud)]
;
D = [ass(max, pavillon, chien, toubib), ass(luc, studio, poisson, instit), ass(eve, caravane, coq, gardien), ass(irma, chateau, cheval, etud)]
;
no (il y a donc deux solutions)

13 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* et 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 successeur :

```
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 avancer de nouveau
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]

14 Analyseur syntaxique
Ce problème est excessivement vaste. Il faudrait dans un premier temps, faire l'analyse syntaxique d'une phrase avec subordonnées, en reconnaissant le caractère compréhensible, plutôt qu'attesté, de la phrase : trouver le verbe principal et son groupe verbal, le sujet, qui peut être un groupe nominal avec subordonnée. On ne considère pas les conjonctions.

On ne considère que quelques déterminants *det*, pronoms *pro*, verbes *v*… puis quelques constructions de groupes nominaux *gn* et verbaux *gv* et quelques constructions de phrases grâce à des inflexions *inf*.

```
det(le).
det(la).
det(les).
det(son).
pro(celui).
pro(le).
prep(dans).
v(vit).
v(lit).
v(est).
v(habite).
v(eleve).
v(mange).
inf(qui).
inf(que).
inf(dont).
adj(noir).
adj(petit).
adj(grande).
adj(rouge).
adj(blanches).
adj(mort).
adj(belle).
nom(belle).
nom(soir).
nom(lit).
nom(maison).
nom(chat).
nom(souris).
 nom(eleve).
 adv(toujours).
 adv(soigneusement).
```

C'est le minimum de mots pour traiter les exemples suivants. On a besoin d'une « dissociation » en deux parties non vides.

```
dis([X], L, [X | L]) :- L \== [].
dis([X | L], M, [X | R]) :- dis(L, M, R).
```

```
dis(X, Y, [a, b, c, d]).
→ X = [a]   Y = [b, c, d] ; X = [a, b]  Y = [c, d] ; X = [a, b, c]  Y = [d] ; no
```

```
fraz(P, phrase(sujet(X), gv(Y))) :- dis(GN, GV, P), gn(GN, X), gv(GV, Y).
gn([N], pronom(N)) :- pro(N).
gn([N], nom(N)) :- nom(N).
gn([D | GN], gn(det(D), X)) :- det(D), !, gn(GN, X).
gn([P | GN], gn(prep(P), X)) :- prep(P), !, gn(GN, X).
gn([A | GN], gn(adj(A), X)) :- adj(A), !, gn(GN, X).
gn([N, A], gn(nom(N), adj(A))) :- nom(N), adj(A).

gv([V], verbe(V)) :- v(V), !.
gv([A | GV], gv(adv(A), X)) :- adv(A), !, gv(GV, X).
gv([V, A], gv(verbe(V), adv(A))) :- v(V), adv(A), !.
gv([V, A], gv(verbe(V), adj(A))) :- v(V), adj(A), !.
gv(GV, gv(X, Y)) :- dis(V, N, GV), gv(V, X), gn(N, Y).
```

Ci-dessous, quelques décompositions de groupes nominaux et verbaux. On oblige les groupes non vides afin que la récursion converge toujours. La subordonnée ouverte par une inflexion peut être une véritable sous-phrase, d'où les clauses supplémentaires :

```
gn(GN, gn(X, sub(I, Y)))
        :- dis(N, SUB, GN), gn(N, X), inf(I), dis([I], V, SUB), gv(V, Y), !.
gn(GN, gn(X, sub(I, Y)))
        :- dis(N, SUB, GN), gn(N, X), inf(I), dis([I], P, SUB), fraz(P, Y), !.
```

```
gv([toujours, mange, soigneusement], X).
→ X = gv(adv(toujours), gv(verbe(mange), adv(soigneusement)))
gn([le, petit, chat, noir], X).
→ X = gn(det(le), gn(adj(petit), gn(nom(chat), adj(noir))))
gn([dans, son, petit, lit], X).
→ X = gn(prep(dans), gn(det(son), gn(adj(petit), nom(lit))))
```

Maintenant des exemples de phrases :

fraz([le, petit, chat, mange], *X*).
→ *X* = phrase(sujet(gn(det(le), gn(adj(petit), nom(chat)))), gv(verbe(mange)))
fraz([la, belle, lit, toujours, dans, son, lit], *X*).
→ *X* = phrase(sujet(gn(det(la), nom(belle))), gv(gv(gv(verbe(lit), adv(toujours)), gn(prep(dans), gn(det(son), nom(lit))))))
% On voit qu'on a levé une ambiguïté entre deux homonymes

gn([le, petit, chat, noir, qui, mange, la, souris], *X*).
→ *X* = gn(det(le), gn(adj(petit), gn(gn(nom(chat), adj(noir)), sub(qui, gv(verbe(mange), gn(det(la), nom(souris)))))))
% n'est qu'un gn, alors qu'on a une phrase :
fraz([le, chat, qui, mange, la, souris, est, noir], *X*).
→ *X* = phrase(sujet(gn(det(le), gn(nom(chat), sub(qui, gv(verbe(mange), gn(det(la), nom(souris)))))), gv(gv(verbe(est), adj(noir))))

fraz([le, chat, qui, mange, la, souris, qui, est, belle, est, mort], *X*).
→ *X* = phrase(sujet(gn(det(le), gn(nom(chat), sub(qui, gv(verbe(mange), gn(det(la), gn(nom(souris), sub(qui, gv(verbe(est), adj(belle))))))))), gv(gv(verbe(est), adj(mort))))
fraz([la, maison, dont, le, chat, est, noir, est, rouge], *X*).
→ *X* = phrase(sujet(gn(det(la), gn(nom(maison), sub(dont, phrase(sujet(gn(det(le), nom(chat))), gv(gv(verbe(est), adj(noir))))))), gv(gv(verbe(est), adj(rouge))))

Ci-dessous deux exemples ou « élève » est un nom puis un verbe et la « belle » un adjectif puis un nom :
fraz([la, belle, eleve, dont, la, maison, est, rouge, mange], *X*).
→ *X* = phrase(sujet(gn(det(la), gn(adj(belle), gn(nom(eleve), sub(dont, phrase(sujet(gn(det(la), nom(maison))), gv(gv(verbe(est), adj(rouge))))))))), gv(verbe(mange)))

fraz([la, belle, eleve, les, souris, que, son, chat, qui, est, noir, mange], *X*).
→ *X* = phrase(sujet(gn(det(la), nom(belle))), gv(gv(verbe(eleve), gn(det(les), gn(nom(souris), sub(que, phrase(sujet(gn(det(son), gn(nom(chat), sub(qui, gv(verbe(est), adj(noir))))))), gv(verbe(mange)))))))))

De plus en plus tourmenté, mais analysable :

fraz([celui, dont, le, chat, qui, mange, les, souris, est, noir, habite, dans, la, maison, dont, la, belle, est, toujours, dans, son, lit], *X*).
→ *X* = phrase(sujet(gn(pronom(celui), sub(dont, phrase(sujet(gn(det(le), gn(nom(chat), sub(qui, gv(verbe(mange), gn(det(les), nom(souris))))))), gv(gv(verbe(est), adj(noir))))))), gv(gv(verbe(habite), gn(prep(dans), gn(det(la), gn(nom(maison), sub(dont, phrase(sujet(gn(det(la),nom(belle))), gv(gv(gv(verbe(est), adv(toujours)), gn(prep(dans), gn(det(son), nom(lit))))))))))))

fraz([le, petit, chat, noir, qui, vit, dans, la, grande, maison, qui, est, rouge, mange, toujours, les, souris, blanches, que, celui, qui, habite, la, maison, eleve, soigneusement], *X*).
→ *X* = phrase(sujet(gn(det(le), gn(adj(petit), gn(gn(nom(chat), adj(noir)), sub(qui, gv(verb(vit), gn(prep(dans), gn(det(la), gn(adj(grande), gn(nom(maison), sub(qui, gv(verbe(est), adj(rouge))))))))))))), gv(gv(gv(verbe(mange), adv(toujours)), gn(det(les), gn(gn(nom(souris), adj(blanches)), sub(que, phrase(sujet(gn(pronom(celui), sub(qui, gv(verb(habite), gn(det(la), nom(maison))))))), gv(gv(verbe(eleve), adv(soigneusement))))))))))

Bien entendu, l'analyse syntaxique demande bien plus de règles, avec par exemple les conjonctions et une limitation aux tournures attestées (« le chat noir » et « le petit chat » sont attestés, « le noir chat » ne l'est pas vraiment.
Une partie des ambiguïtés peut d'ailleurs être levée par des juxtapositions interdites (Voir L. Gacôgne, *512 problèmes corrigés*, Ellipses, 1995) telles qu'un déterminant suivi d'une préposition etc. Puis viennent l'analyse morphologique des mots infléchis, des accords, et une analyse « pragmatique » comprenant les références des pronoms, le traitement des ellipses, des métaphores... qui nécessite une formalisation en graphes conceptuels pour arriver à une certaine « compréhension ».

15 Problème simplifié des loups et moutons
Ils sont *N* moutons sur la rive d'un fleuve en compagnie de *N* loups (variante des missionnaires et des cannibales). Avec les mêmes conditions que pour ce premier problème, ils doivent tous passer sur l'autre rive grâce à une barque contenant un nombre *K* d'animaux.

Grâce au foncteur *config(M, K, Rive)* où *M* est le nombre de couples en rive 1 (*Rive* = 1 pour le départ et –1 pour l'arrivée). Le prédicat *infer* n'est pas idéal car il faudrait débuter par les plus gros transports dans un sens et par les plus petits dans l'autre sens.

```
inv(L, LI) :- invbis(L, [], LI).
invbis([X | D], T, R) :- invbis(D, [X | T], R).
invbis([], R, R).
infer(K, K) :- 0 < K. % produit les entiers M non nuls inférieurs à K
infer(M, K) :- 1 < K, L is K – 1, infer(M, L).
vue([]).
vue([config(N, _, R) | S]) :- write(N), write(' rive '), write(R), nl, vue(S).
app(X, [X | _]) :- !.
app(X, [_ | L]) :- app(X, L).

mouv(config(M1, K, R1), config(M2, K, R2)) :- infer(M, K), R2 is (–R1),
                                              M2 is M1 − M*R1.
suite([config(0, K, –1) | S]) :- !, inv([config(0, K, –1) | S], R), vue(R).
suite([E | S]) :- mouv(E, F), not(app(F, S)), suite([F, E | S]).
jeu(N, K) :- suite([config(N, K, 1)]).
```

Exemple avec 8 moutons et une barque de 3 places

```
jeu(8, 3).
→
8 rive 1
5 rive –1
7 rive 1
4 rive –1
5 rive 1
2 rive –1
3 rive 1
0 rive –1
```

```
jeu(13, 7).
→
13 rive 1
6 rive –1
12 rive 1
5 rive –1
11 rive 1
4 rive –1
10 rive 1
3 rive –1
9 rive 1
2 rive –1
8 rive 1
1 rive –1
7 rive 1
0 rive –1
```

16 Le paysan, son chat et sa souris fétiche
Le paysan va vendre son fromage en traversant une rivière avec une barque qu'il est seul à pouvoir conduire. La barque ne peut contenir qu'un des 3 objets et il ne doit pas laisser sans surveillance le chat et la souris pas plus que la souris et le fromage.
Trouver la suite des déplacements à effectuer.

Ce problème est assez analogue au dernier et sera traité à peu près de la même façon grâce au foncteur *config(P, C, S, F)* ; on doit pouvoir passer de la situation (1, 1, 1, 1) à (–1, –1, –1, –1), en notant 1 la rive de départ et –1 celle d'arrivée pour les positions respectives des quatre acteurs. On aura compris que *P* = paysan, *C* = chat, *S* = souris, *F* = fromage.
Les 4 mouvements possibles sont ici dans le sens *mouv(arrivée, départ)*, il emporte la souris, le chat, le fromage, et enfin, il est seul.
Le *if* permet des affichages sélectifs.

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

if(1, M) :- write(M).
if(-1, _) :- write(' ').
```

```
interdit(config(P, C, S , F)) :- (P is –S, S = F); (P is –C, C = S).

vue([]).
vue([config(P, C, S, F) | R])
                    :- if(P, 'Paysan et bateau '), if(C,'chat '),
                     if(S, 'souris '), if(F, 'fromage '), nl, vue(R).

mouv(config(P1, C0, P1, F0), config(P0, C0, P0, F0))
                    :- P1 is (–P0), not(interdit(config(P1, C0, P1, F0))).
mouv(config(P1, P1, S0, F0), config(P0, P0, S0, F0))
                    :-  P1 is (–P0), not(interdit(config(P1, P1, S0, F0))).
mouv(config(P1, C0, S0, P1), config(P0, C0, S0, P0))
                    :-  P1 is (–P0), not(interdit(config(P1, C0, S0, P1))).
mouv(config(P1, C0, S0, F0), config(P0, C0, S0, F0))
                    :- P1 is (–P0), not(interdit(config(P1, C0, S0, F0))).

suite([config(–1, –1, –1, –1) | S])
                    :- !, inv([config(–1, –1, –1, –1) | S], R),
                    write('Rive initiale'), nl, vue(R).
suite([E | S]) :- mouv(F, E), not(app(F, S)), suite([F, E|S]).

jeu :- suite([config(1, 1, 1 ,1)]), !.  % le départ est [config(1, 1, 1, 1)]
```

Exécution

```
jeu.        % renvoie l'affichage de ce qui est sur la rive initiale
→
Paysan et bateau chat souris fromage
  chat  fromage
Paysan et bateau chat  fromage
   fromage
Paysan et bateau  souris fromage
   souris
Paysan et bateau  souris
```

17 Le singe et les bananes
Il faut décrire les opérations qu'un singe peut faire dans une pièce où se situent une caisse et des bananes suspendues. Le singe a besoin de déplacer la caisse pour monter dessus et attraper les bananes.

Au chapitre suivant, nous verrons que les relations « assert » et « retract » peuvent décrire la suite des états du problème, mais elles sont d'un maniement très difficile ; de plus, en cas de retour en arrière, rien ne modifie ces ajouts et suppressions qui ont été faits à tort. Il vaut bien mieux prendre une représentation des données, certes souvent embarrassante, mais qui permet de décrire correctement ce qu'est une suite d'actions pour arriver à une solution.

a) Voici une solution qui fonctionne vu la simplicité du problème, mais qui oblige à revenir à la racine de l'arbre de recherche à chaque échec, car une liste de mouvements est réécrite en une liste plus courte. Il faut (paragraphe suivant) au contraire que le sens de la réécriture de Prolog, aille dans le sens de la recherche d'une suite de mouvements. Un état du problème *E* est d'abord structuré comme un terme à quatre arguments : la position du singe, celle de la caisse, celle de la banane et l'état du singe au sol ou perché. Les positions dans la pièce ne sont que *s* (entrée-sortie), *m* (le milieu) et *c* (position initiale de la caisse).
Notons encore une fois que le même nom « *inv* » peut sans dommage être utilisé pour deux prédicats distincts à deux ou trois arguments, l'unification faisant correctement son office.

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

inv(L, LI) :- inv( L, [], LI).
inv([X | L], T, R) :- inv(L, [X | T], R).
inv([], R, R).

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

vue([X | L]) :- write(X), sep, vue(L).
vue([]).

sep :- write(', ').          % fait l'affichage d'un séparateur

but(etat(m, m, m, haut)).
```

```
mouv([etat(s, c, m, sol)]).
mouv([F, E | L]) :- mouv([E | L]), cht(E, F).

cht(E, F) :- marcher(E, F); deplacer(E, F); monter(E, F).

marcher(etat(A1, B, C, D), etat(A2, B, C, D)) :- distincts(A1, A2, [s, m, c]).

deplacer(etat(A1, A1, C, D), etat(A2, A2, C, D)) :- distincts(A1, A2, [s, m, c]).
monter(etat(A, A, A, sol), etat(A, A, A, haut)).

jeu :- mouv([X | L]), but(X), inv([X | L], I), vue(I).
```

```
distincts(X, c, [a, b, c, d, e]). → X = a ; X = b ; X = d ; X = e ; no
distincts(X, Y, [a, b, c]).  → X = a   Y = b ; X = a   Y = c ; X = b   Y = a ;
X = c   Y = a ; X = b   Y = c ; X = c   Y = b ; no
jeu. →
etat(s, c, m, sol), etat(c, c, m, sol), etat(m, m, m, sol), etat(m, m, m, haut),
yes
```

b) Maintenant, une meilleure solution, car permettant de mettre les actions dans une liste, de plus lorsqu'on cherche successivement les actions *A* appartenant à cette liste, on l'exécute ensuite de façon à instancier ses variables, puis, comme on le fait au paragraphe suivant, on teste si l'état *F* ainsi obtenu n'a pas déjà été rencontré dans la liste des états déjà visités *LE*.

```
jeu :- mouv([F | L], R), but(F), inv(R, RI), vue(RI).
but(etat(m, m, m, haut)).

mouv([etat(s, c, m, sol)], []).
mouv([F, E | L], [A | LA]) :- mouv([E | L], LA),
                    app(A, [marcher(E, F), deplacer(E, F), monter(E, F)]), A.

marcher(etat(A1, B, C, D), etat(A2, B, C, D)) :- distincts(A1, A2, [s, m, c]).

deplacer(etat(A1, A1, C, D), etat(A2, A2, C, D)) :- distincts(A1, A2, [s, m, c]).

monter(etat(A, A, A, sol), etat(A, A, A, haut)).
```

```
jeu. →
marcher(etat(s, c, m, sol), etat(c, c, m, sol)),
deplacer(etat(c, c, m, sol), etat(m, m, m, sol)),
monter(etat(m, m, m, sol), etat(m, m, m, haut)),
yes
```

c) On peut rajouter des actions comme descendre, prendre en main, manger, etc. Si on demande à présent au singe de sortir en ayant mangé la banane et remis la caisse à sa place. On représente alors un état du problème par un quintuplet (position du singe, état du singe au sol ou grimpé, position de la caisse, position de la banane, état de la banane suspendue, en main, ou mangée). L'intérêt du Prolog est manifestement présent dans l'écriture de certaines actions comme « monter » où les contraintes sur la position sont inscrites dans l'écriture même des états, l'unification ne pouvant se faire que si les positions du singe et de la caisse sont identiques. On remarque bien des déplacements inutiles, mais s'arrêtant à la première solution rencontrée, on ne cherche pas les autres.

```
jeu :- init(E), mouv([E], []).
init(etat(s, sol, c, m, susp)).
but(etat(s, sol, c, _, estomac)).
mouv([F | LE], LA) :- but(F), inv(LA, R), vue(R).
mouv([E | LE], LA) :- actions(E, F, RA), app(A, RA), A, not(app(F, LE)),
                      mouv([F, E | LE], [A | LA]).
actions(E, F, [marcher(E, F), deplacer(E, F), monter(E, F), descendre(E, F),
                      prendre(E, F), manger(E, F)]).
marcher(etat(S1, sol, C, B, EB), etat(S2, sol, C, B, EB))
                      :- distincts(S1, S2, [c, s, m]).
deplacer(etat(S1, sol, S1, B, EB), etat(S2, sol, S2, B, EB))
                      :- distincts(S1, S2, [c, s, m]).
monter(etat(C, sol, C, C, EB), etat(C, haut, C, C, EB)).
descendre(etat(C, haut, C, B, EB), etat(C, sol, C, B, EB)).
prendre(etat(C, haut, C, C, susp), etat(C, haut, C, C, main)).
manger(etat(S, ES, C, B, main), etat(S, ES, C, B, estomac)).
```

```
jeu. →
marcher(etat(s, sol, c, m, susp), etat(c, sol, c, m, susp))
deplacer(etat(c, sol, c, m, susp), etat(s, sol, s, m, susp))
deplacer(etat(s, sol, s, m, susp), etat(m, sol, m, m, susp))
monter(etat(m, sol, m, m, susp), etat(m, haut, m, m, susp))
prendre(etat(m, haut, m, m, susp), etat(m, haut, m, m, main))
descendre(etat(m, haut, m, m, main), etat(m, sol, m, m, main))
marcher(etat(m, sol, m, m, main), etat(c, sol, m, m, main))
marcher(etat(c, sol, m, m, main), etat(s, sol, m, m, main))
manger(etat(s, sol, m, m, main), etat(s, sol, m, m, estomac))
marcher(etat(s, sol, m, m, estomac), etat(c, sol, m, m, estomac))
marcher(etat(c, sol, m, m, estomac), etat(m, sol, m, m, estomac))
deplacer(etat(m, sol, m, m, estomac), etat(c, sol, c, m, estomac))
marcher(etat(c, sol, c, m, estomac), etat(s, sol, c, m, estomac))   yes
```

18 Robot aspirateur
Le salon, la cuisine et la chambre d'un appartement sont à nettoyer par un robot-aspirateur qui doit vider son sac après chaque pièce nettoyée, en retournant dans un placard où se trouve le vide-ordures et où il doit être rangé après son travail.
Un état du problème sera représenté par la position de l'aspirateur, son état, vide ou plein, et l'état propre ou sale de chaque pièce en prévoyant une liste de pièces. On peut aussi nettoyer le placard ! On restreint les déplacements d'une pièce à une autre que si celle-ci vient d'être nettoyée. Le placard porte le numéro 0, les trois pièces 1, 2, 3. Le prédicat « distincts » est le même que précédemment.

```
jeu :- init(E), mouv([E], []).

mouv([F | LE], LA) :- but(F), inv(LA, R), vue(R).
mouv([E | LE], LA)
        :- actions(E, F, RA), app(A, RA), A, not(app(F, LE)),
        mouv([F, E | LE], [A | LA]).

init(etat(0, vide, [sale, sale, sale])).
but(etat(0, vide, [propre, propre, propre])).

actions(E, F, [nettoyer(E, F), vider(E, F), passer(E, F)]).

nettoyer(etat(P, vide, LPS), etat(P, plein, LPP)) :- nettoyage(LPS, P, LPP).

nettoyage([sale | LP], 1, [propre | LP]).
        % les pièces sont numérotées à partir de 1
nettoyage([E | LP], N, [E | LPP]) :- N > 1, K is N − 1, nettoyage(LP, K, LPP).

vider(etat(0, plein, LP), etat(0, vide, LP)).   % le placard est la pièce numéro 0
passer(etat(P1, E, LP), etat(P2, E, LP)) :- distincts(P1, P2, [0, 1, 2, 3]).
```

```
jeu. →
passer(etat(0, vide, [sale, sale, sale]), etat(1, vide, [sale, sale, sale]))
nettoyer(etat(1, vide, [sale, sale, sale]), etat(1, plein, [propre, sale, sale]))
passer(etat(1, plein, [propre, sale, sale]),
   etat(0, plein, [propre, sale, sale]))
vider(etat(0, plein, [propre, sale, sale]), etat(0, vide, [propre, sale, sale]))
passer(etat(0, vide, [propre, sale, sale]), etat(1, vide, [propre, sale, sale]))
passer(etat(1, vide, [propre, sale, sale]), etat(2, vide, [propre, sale, sale]))
nettoyer(etat(2, vide, [propre, sale, sale]),
```

```
    etat(2, plein, [propre, propre, sale]))
passer(etat(2, plein, [propre, propre, sale]),
    etat(0, plein, [propre, propre, sale]))
vider(etat(0, plein, [propre, propre, sale]),
    etat(0, vide, [propre, propre, sale]))
passer(etat(0, vide, [propre, propre, sale]),
    etat(1, vide, [propre, propre, sale]))
passer(etat(1, vide, [propre, propre, sale]),
    etat(2, vide, [propre, propre, sale]))
passer(etat(2, vide, [propre, propre, sale]),
    etat(3, vide, [propre, propre, sale]))
nettoyer(etat(3, vide, [propre, propre, sale]),
    etat(3, plein, [propre, propre, propre]))
passer(etat(3, plein, [propre, propre, propre]),
    etat(0, plein, [propre, propre, propre]))
vider(etat(0, plein, [propre, propre, propre]),
    etat(0, vide, [propre, propre, propre]))
yes
```

On peut, bien sûr, perfectionner en limitant les déplacements que vers une pièce sale ou le placard de numéro 0, puis affecter les pièces d'un coefficient, et la capacité de l'aspirateur telle qu'il n'ira vider son sac que lorsqu'il est vraiment plein. On peut toujours tout compliquer.

19 Les grenouilles alignées
Un certain nombre *N* de grenouilles vertes sont alignées et séparées par un blanc de *N* grenouilles grises. Les règles sont qu'une grenouille voisine de la case libre peut s'y déplacer et qu'une grenouille peut sauter par-dessus une autre à condition qu'elle soit d'une autre couleur et qu'elle arrive dans la case vide. Programmer d'abord pour *N* = 3, ainsi il faut, en notant b la case vide, passer de *vvvbggg* à *gggbvvv*.

a) La représentation en liste s'impose assez naturellement, quoiqu'un exemple analogue figure avec la représentation en arbre au chapitre suivant.
Nous reprenons quasiment la même solution que celle de l'exercice précédent (aspirateur) en changeant les actions possibles.

```
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).
inv(L, LI) :- inv( L, [], LI).
inv([X | L], T, R) :- inv(L, [X | T], R).
inv([], R, R).
```

```
dif(X, X) :- !, fail.
dif(_, _).

deplacer([b, X | L], [X, b | L]).
deplacer([X, b | L], [b, X | L]).
deplacer([X | L], [X | LD]) :- deplacer(L, LD).
sauter([X, Y, b | L], [b, Y, X | L]) :- dif(X, Y).
sauter([b, X, Y | L], [Y, X, b | L]) :- dif(X, Y).
sauter([X | L], [X | LS]) :- sauter(L, LS).
```

> Exemple
> deplacer([v, v, v, v, v, b, g, g, g, g, g, g], R).
> → R = [v, v, v, v, b, v, g, g, g, g, g, g] ; R = [v, v, v, v, v, g, b, g, g, g, g, g] ; no
> sauter([v, v, v, v, v, g, b, g, g, g, g, g], R).
> → R = [v, v, v, v, b, g, v, g, g, g, g, g] ; no

```
suite([[g, g, g, b, v, v, v] | S], LA) :- !, inv(LA, R), vue(R). %  test d'arrêt
suite([E | S], LA) :- app(A, [sauter(E, F), deplacer(E, F)]), A, not(app(F, S)),
                      suite([F, E | S], [A | LA]).
```

Ainsi la réécriture de gauche à droite produit une suite de mouvements de plus en plus longue.

```
jeu1 :- suite([[v, v, v, b, g, g, g]], []).
```

b) Pour généraliser, mais solution assez longue dès $N = 5$.

```
conc([X | L], M, [X | R]) :- conc(L, M, R).
conc([], L, L).
construc(0, _, []) :- !.
construc(N, X, [X | L]) :- M is N − 1, construc(M, X, L).

init(N, Dep, Arr) :- construc(N, v, LV), construc(N, g, LG),
                     conc(LV, [b | LG], Dep), conc(LG, [b | LV], Arr).

jeu2(N) :- init(N, Dep, Arr), suite(Arr, [Dep], []).

suite(But, [But | S], LA) :- !, inv(LA, R), vue(R).
suite(But, [E | S], LA) :- app(A, [sauter(E, F), deplacer(E, F)]), A,
                     not(app(F, S)), suite(But, [F, E | S], [A | LA]).
```

```
jeu1. →
deplacer([v, v, v, b, g, g, g], [v, v, b, v, g, g, g])
sauter([v, v, b, v, g, g, g], [v, v, g, v, b, g, g])
deplacer([v, v, g, v, b, g, g], [v, v, g, b, v, g, g])
sauter([v, v, g, b, v, g, g], [v, b, g, v, v, g, g])
deplacer([v, b, g, v, v, g, g], [b, v, g, v, v, g, g])
sauter([b, v, g, v, v, g, g], [g, v, b, v, v, g, g])
deplacer([g, v, b, v, v, g, g], [g, b, v, v, v, g, g])
deplacer([g, b, v, v, v, g, g], [b, g, v, v, v, g, g])
sauter([b, g, v, v, v, g, g], [v, g, b, v, v, g, g])
deplacer([v, g, b, v, v, g, g], [v, g, v, b, v, g, g])
sauter([v, g, v, b, v, g, g], [v, g, v, g, v, b, g])
deplacer([v, g, v, g, v, b, g], [v, g, v, g, b, v, g])
sauter([v, g, v, g, b, v, g], [v, g, b, g, v, v, g])
sauter([v, g, b, g, v, v, g], [b, g, v, g, v, v, g])
deplacer([b, g, v, g, v, v, g], [g, b, v, g, v, v, g])
sauter([g, b, v, g, v, v, g], [g, g, v, b, v, v, g])
deplacer([g, g, v, b, v, v, g], [g, g, b, v, v, v, g])
deplacer([g, g, b, v, v, v, g], [g, b, g, v, v, v, g])
sauter([g, b, g, v, v, v, g], [g, v, g, b, v, v, g])
deplacer([g, v, g, b, v, v, g], [g, v, b, g, v, v, g])
sauter([g, v, b, g, v, v, g], [g, v, v, g, b, v, g])
sauter([g, v, v, g, b, v, g], [g, v, v, g, g, v, b])
deplacer([g, v, v, g, g, v, b], [g, v, v, g, g, b, v])
deplacer([g, v, v, g, g, b, v], [g, v, v, g, b, g, v])
sauter([g, v, v, g, b, g, v], [g, v, b, g, v, g, v])
sauter([g, v, b, g, v, g, v], [b, v, g, g, v, g, v])
deplacer([b, v, g, g, v, g, v], [v, b, g, g, v, g, v])
deplacer([v, b, g, g, v, g, v], [v, g, b, g, v, g, v])
sauter([v, g, b, g, v, g, v], [b, g, v, g, v, g, v])
deplacer([b, g, v, g, v, g, v], [g, b, v, g, v, g, v])
sauter([g, b, v, g, v, g, v], [g, g, v, b, v, g, v])
sauter([g, g, v, b, v, g, v], [g, g, v, g, v, b, v])
deplacer([g, g, v, g, v, b, v], [g, g, v, g, b, v, v])
sauter([g, g, v, g, b, v, v], [g, g, b, g, v, v, v])
deplacer([g, g, b, g, v, v, v], [g, g, g, b, v, v, v])
yes
```

```
jeu2(2). →
deplacer([v, v, b, g, g], [v, b, v, g, g])
sauter([v, b, v, g, g], [v, g, v, b, g])
deplacer([v, g, v, b, g], [v, g, b, v, g])
sauter([v, g, b, v, g], [b, g, v, v, g])
deplacer([b, g, v, v, g], [g, b, v, v, g])
deplacer([g, b, v, v, g], [g, v, b, v, g])
sauter([g, v, b, v, g], [g, v, g, v, b])
deplacer([g, v, g, v, b], [g, v, g, b, v])
sauter([g, v, g, b, v], [g, b, g, v, v])
deplacer([g, b, g, v, v], [g, g, b, v, v])
yes
```

20 Planification, empilage et dépilage de cubes
Connu sous le nom de problème de Winograd, il s'agit, pour trois cubes *A, B, C,* de décrire les actions permettant de passer d'une pile de cubes *C* sur *A* et *B* à côté, à la pile *A* sur *B* sur *C*.

Dans le but d'écrire une planification à la STRIPS (Stanford Research Institute Problem Solver) [Nilssen 1971], on donne des spécifications d'actions permettant de passer d'un état à un autre. Comme dans tous ces problèmes de planification, la difficulté est d'abord de représenter le problème, définir l'état initial et l'état final, enfin le décrire en terme de prédicats, chacune des actions possibles étant associée à des contraintes et faisant toujours passer d'un état à un autre. Nous définissons d'abord les prédicats :

```
app(X, [X | _]).
app(X, [_ | L]) :- app(X, L).
inclus([], _).
inclus([X | L], M) :- app(X, M), inclus(L, M).
conc([], L, L).
conc([X | L], M, [X | R]) :- conc(L, M, R).
diff([], M, []).     % diff est la différence ensembliste
diff([X | L], M, R) :- app(X, M) , !, diff(L, M, R).
diff([X | L], M, [X | R]) :- diff(L, M, R).
ret(_, [], []).
ret(X, [X | L], R) :- !, ret(X, L, R).
ret(X, [Y | L], [Y | M]) :- ret(X, L, M).
disjoints([], _).
disjoints([X | M], L) :- app(X, L), !, fail.
disjoints([_ | M], L) :- disjoints(M, L).

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).
```

inclus([lib(*B*), lib(*Y*), sur(*B, X*)], [lib(2), lib(4), lib(b), lib(c), sur(b, 3), sur(c, a), sur(a, 1)]).
→ *B* = b *Y* = 2 *X* = 3 ; *B* = b *Y* = 4 *X* = 3 yes
tri([sur(b, c), lib(3), lib(2), lib(4), lib(b), sur(c, a), sur(a, 1)], *R*).
→ *R* = [lib(2), lib(3), lib(4), lib(b), sur(a, 1), sur(b, c), sur(c, a)]

a) Une première structuration pourrait être :

```
bloc(X) :- app(X, [a, b, c]).
init([surtable(a), surtable(b), sur(c, a), lib(c), lib(b), mainvide]).
final([surtable(c), sur(b, c), sur(a, b), mainvide]).
```

En déclarant toutes les actions par leur nom entre deux cubes *X* et *Y*, avec les listes *LC* de conditions, *LR* à retirer et *LA* à ajouter :

```
depiler(X, Y, [lib(X), mainvide, sur(X, Y)], [sur(X, Y), lib(Y)], [tenir(X), lib(Y)])
        :- bloc(X), bloc(Y), X \== Y.
empiler(X, Y, [lib(Y), tenir(X)], [tenir(X), lib(Y)], [mainvide, sur(X, Y)])
        :- bloc(X), bloc(Y), X \== Y.
prendre(X, [lib(X), mainvide, surtable(X)], [surtable(X), mainvide], [tenir(X)])
        :- bloc(X).
poser(X, [tenir(X)], [tenir(X)], [surtable(X), mainvide]) :- bloc(X).
applique(E, LA, LR, F) :- diff(E, LR, L1), conc(LA, L1, L2), tri(L2, F).
```

Appliquer permet donc, quand c'est possible, de passer d'un état *E* à un état *F*. Puis tous les mouvements sont déclarés :

```
mouv(E, F, M, _) :- inclus(E, F), write(M). % solution trouvée
mouv(E, F, M, T) :- empiler(X, Y, LC, LR, LA), inclus(LC, E),
                    applique(E, LA, LR, ES),
                    not(app(F, T)), mouv(ES, F, [empiler(X, Y) | M], [ES | T]).
mouv(E, F, M, T) :- poser(X, LC, LR, LA), inclus(LC, E),
                    applique(E, LA, LR, ES),
                    not(app(F, T)), mouv(ES, F, [poser(X, Y) | M], [ES | T]).
mouv(E, F, M, T) :- depiler(X, Y, LC, LR, LA), inclus(LC, E),
                     applique(E, LA, LR, ES),
                    not(app(F, T)), mouv(ES, F, [depiler(X, Y) | M], [ES | T]).
mouv(E, F, M, T) :- prendre(X, LC, LR, LA), inclus(LC, E),
                    applique(E, LA, LR, ES),
                    not(app(F, T)), mouv(ES, F, [prendre(X) | M], [ES | T]).
jeu :- init(E), final(F), mouv(E, F, [], [E]).
```

b) Mais une seconde structuration (il y en a bien d'autres) serait de définir le terme « mouv », réservant alors le nom *plan* pour un prédicat, en numérotant les emplacements. Un état est une liste comme les listes de condition ci-dessus, un mouvement est un terme clos comme *mouv(a, 2, 3)*. Le prédicat *applique* est vrai si une action *A* (un mouvement) fait passer de l'état *E* à l'état *F* par un jeu de suppressions et d'ajouts de conditions.

```
objet(X) :- place(X); bloc(X).
bloc(X) :- app(X, [a, b, c]).
place(X) :- app(X, [1, 2, 3, 4]).
init([lib(2), lib(4), lib(b), lib(c), sur(b, 3), sur(c, a), sur(a, 1)]).
final([sur(c, 2), sur(b, c), sur(a, b)]).

applique(E, A, LA, LR, F) :- diff(E, LR, L1), conc(LA, L1, L2), tri(L2, F).
action(mouv(B, X, Y), [lib(B), lib(Y), sur(B, X)], [sur(B, Y), lib(X)],
                                    [sur(B, X), lib(Y)]).
```

On a ici un cas intéressant d'axiome non clos, où on a encore *LC* = liste des conditions, *LA* = liste des ajouts, *LR* = retraits, quant au prédicat *plan*, il est vrai si et seulement si *R* est une suite d'actions permettant de passer de *E* à *F* en respectant les *Buts*.

```
plan(E, Buts, _, R) :- inclus(Buts, E), !.  % cas où les buts satisfaits en état E
plan(E, Buts, T, M, R) :- % T est la liste « tabou » des états antérieurs
        action(A, LC, LA, LR), inclus(LC, E),
        % On cherche une action A possible
        A =.. [mouv, B, X, Y], B \== Y,
        % Elle est de la forme mouv(B, X, Y)
        applique(E, A, LA, LR, ES), not(app(ES, T)),
        % A passe à l'état suivant ES
        plan(ES, Buts, [ES | T], [A | M], R).
         % cherche plan suivant de ES à F

jeu(R) :- init(E), final(F), plan(E, F, [], [], R).
```

> jeu(*X*). → *X* = [mouv(a, 3, b), mouv(a, 1, 3), mouv(a, 4, 1),
> mouv(b, a, c), mouv(b, 3, a), mouv(a, 1, 4), mouv(a, b, 1), mouv(c, 1, 2),
> mouv(a, 4, b), mouv(a, 2, 4), mouv(a, c, 2), mouv(b, 2, 3), mouv(a, 4, c),
> mouv(a, 3, 4), mouv(a, b, 3), mouv(c, 3, 1), mouv(a, 4, b), mouv(a, 1, 4),
> mouv(a, c, 1), mouv(b, 1, 2), mouv(a, 4, c), mouv(a, 2, 4), mouv(a, b, 2),
> mouv(c, 2, 3), mouv(a, 4, b), mouv(a, 3, 4), mouv(a, c, 3), mouv(b, 4, 1),
> mouv(a,b,c), mouv(a, 3, b), mouv(a, 1, 3), mouv(c, a, 2), mouv(b, 3, 4),
> mouv(b, 2, 3), mouv(b, 3, 2)]

c) Avec les mêmes petits prédicats *app*, *inclus*, mais une structuration des données nettement plus simple, mais éloignée de la description « strips ». Ici, on ne considère que les cubes *a*, *b*, *c* et la table *t* sans aucune numérotation d'emplacements. Dans ce qui suit, la table *t* doit être partie intégrante dans les relations de superpositions. On décrit le prédicat *dejavu* qui teste si un état est

déjà inclus dans un autre état de la liste *T*. *T* est la liste « tabou », liste des états antérieurs. Le prédicat *poss* indique si l'action en premier argument est possible à l'état *E*. Le prédicat *applique(E, A, F)* est vrai si et seulement si *F* est l'état produit par l'action *A* à partir de l'état *E*.

```
dejavu(E, [F | L]) :- inclus(E, F); dejavu(E, L).
init([lib(b), lib(c), lib(t), sur(a, t), sur(b, t), sur(c, a)]).
final([sur(a, b), sur(b, c)]).
poss(mouv(B, X, Y), E) :- app(sur(B, X), E), app(lib(B), E), app(lib(Y), E),
                          B \== Y.
applique(E, mouv(B, X, Y), [sur(B, Y), lib(X) | EI])
        :- diff(E, [sur(B, X), lib(Y)], EI).
plan(E, _, _, R, R) :- final(F), inclus(F, E), !.
plan(E, ES, T, M, R) :- poss(A, E), applique(E, A, ES), not(dejavu(ES, T)),
         plan(ES, F, [ES | T], [A | M], R).   % ici ES est l'état suivant

jeu(R) :- init(E), plan(E, _, [E], [], R).
        % donne les mouvements (à l'envers)
```

> jeu(*R*). → *R* = [mouv(a, t, b), mouv(b, a, c), mouv(b, t, a), mouv(c, b, t), mouv(c, a, b)]

Il s'agit d'un pur backtrack où c'est l'inclusion qui cherche toutes les actions possibles, méthodiquement sans aucune intelligence, car, on peut constater qu'on était loin du plan minimal et qu'en cette dernière version, il y a encore deux mouvements de trop à cause de l'ordre dans lequel sont donnés les faits. Ce qu'il faudrait, c'est opérer un choix dans les actions à essayer. En dirigeant la recherche par les buts à satisfaire, on peut évidemment améliorer, on peut aussi résoudre, mais de façon très périlleuse, le problème, par une recherche en largeur suivie d'une recherche en profondeur, c'est-à-dire sélectionner un but *B* dans les buts à réaliser, puis une action *A* amenant à ce but, et en enfin un double appel à *plan* pour trouver un *plan* passant de l'état *E* à *E1*, appliquer *A* sur *E1* pour arriver à *E2*, et enfin chercher un plan (en profondeur) passant de *E2* à l'état final. Voir [I. Bratko, *Prolog*, Addisson-Wesley, 1990] et [W. F. Clocksinn, C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 1981].

21 Ensemble des solutions vérifiant une propriété
Cet exercice montre qu'il est possible d'extraire facilement d'une base les données vérifiant une proposition logique structurée.
Le prédicat prédéfini *setof* permet, lorsqu'on demande *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 la propriété *p(X)* soit vérifiée.
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* réalise la même chose mais avec d'éventuelles répétitions des solutions puisqu'il s'agit d'un « sac » (concept formalisé par une fonction de répétition de *E* dans *N*) et non d'un ensemble *E*. Créer un exemple à partir d'une petite base de faits généalogiques.

```
homme(adam).
femme(eve).
homme(cain).
homme(abel).
femme(sarah).
mere(eve, abel).
mere(eve, seth).
epoux(adam, eve).
epoux(cain, sarah).
pere(adam, cain).
pere(adam, sarah).
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).
```

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. Exemples d'interrogations :

setof(*X*, homme(*X*), *E*). → *X* = _530 *E* = [abel, adam, cain]
En effet, *X* est aussi une variable

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]
On remarque qu'Adam et Eve n'ont pas de parents.

bagof(*X*, *P*^parent(*P*, *X*), *E*).
→ *E* = [cain, sarah, henosh, pierre, henok, joseph, irad, mehuyael, abel, seth, abel, seth]

22 Fibonacci avec mémorisation

Nous explorons maintenant des programmes qui se transforment eux-mêmes en cours d'exécution. La fameuse suite de Fibonacci peut être programmée avec la formule de récurrence qui la définit, mais en mémorisant les valeurs acquises au fur et à mesure.
En utilisant le prédicat *asserta(P)* qui rajoute la clause *P* en tête du programme en cours d'exécution ou *assertz(P*) pour la rajouter en queue, emmagasiner les différentes valeurs calculées en cours d'utilisation.

```
fib(0, 1).
fib(1, 1).
fib(N, F) :- N1 is N − 1, N2 is N − 2, fib(N2, F2), fib(N1, F1), !,
        F is F1 + F2, asserta(fib(N, F)).
```

fib(5, *F*). → *F* = 8
fib(10, *F*). → *F* = 89
fib(20, *F*). → *F* = 10946

On peut voir, après cela, ce qu'il en est du programme grâce au prédicat prédéfini *listing*. Comme on peut le constater, un certain nombre de données ont été rajoutées.

```
listing. → fib(20, 10946). fib(19, 6765). fib(18, 4181). fib(17, 2584).
fib(16, 1597). fib(15, 987). fib(14, 610). fib(13, 377). fib(12, 233).
fib(11, 144). fib(10, 89). fib(9, 55). fib(8, 34). fib(7, 21). fib(6, 13).
fib(5, 8). fib(4, 5). fib(3, 3). fib(2, 2). fib(0, 1). fib(1, 1).
fib(A, B) :- C is A − 1, D is A − 2, fib(D, E), fib(C, F), !,
            B is F + E, asserta(fib(A, B)).
```

23 Nombres premiers avec mémorisation de ceux obtenus

Voilà un exemple assez acrobatique de mémorisation des nombres premiers au fur et à mesure des demandes de l'utilisateur. Au départ, on donne simplement la liste des 10 premiers nombres premiers, et chaque fois que cette liste est agrandie, la première clause du prédicat *prem* est modifiée. On doit se servir pour cela du prédicat *retract(P)* permettant de retirer la clause *P* du programme en cours d'exécution.

Dans la recherche des nombres premiers, pour savoir si *N* l'est, on se limite aux tentatives de divisibilité de *N* par des diviseurs déjà reconnus comme premier (donc on peut aller de deux en deux) et inférieurs à la racine carrée entière *Q* de *N*.

Avec les prédicats définis ci-dessous, on peut demander la liste *P* des *N* premiers nombres premiers par *prem(P, N).*, si un entier est premier par *premier(N).* ou encore le nombre premier de rang R par *premier(R, N)*. En déduire, pour finir, la décomposition en facteurs premiers.

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

queue(0, L, L).
queue(K, [_ | L], R) :- KS is K – 1, queue(KS, L, R).

sqrt(N, Q) :- sqrbis(N, 1, Q).          % déjà vu au chapitre 3
sqrbis(N, M, Q) :- P is M*M, N < P, !, Q is M – 1.
sqrbis(N, M, Q) :- MS is M + 1, sqrbis(N, MS, Q).
divise(D, N) :- 0 is N mod D.

prem([29, 23, 19, 17, 13, 11, 7, 5, 3, 2], 10).

prem(P, N) :- nonvar(N), prem([D | PP], M), % D est le dernier premier trouvé
        M < N, !, K is N – M,
        % K = nb de nouveaux premiers à trouver
        recherche(K, D), prem(P, N).
        % cherche K nouveaux premiers après D puis délivre le résultat P
```

```
prem(P, N) :- nonvar(N), prem(PP, M), !,    % D est le dernier premier trouvé
        K is M – N, queue(K, PP, P). % on retire les K premiers de PP

recherche(0, _ ) :- !.
recherche(K, D) :- DS is D + 2, premier(DS), !, retract(prem(P, M)),
        MS is M + 1, asserta(prem([DS | P], MS)),
        KS is K – 1, recherche(KS, DS).
recherche(K, D) :- DS is D + 2, recherche(K, DS).

premier(N) :- prem(P, _), nonvar(P), app(N, P), !.
        % cas où N a déjà été trouvé premier
premier(N) :- prem([D | P], _), sqrt(N, Q), prembis(N, D, Q, P).
        % D est diviseur potentiel de N depuis 2 jusqu'à Q

prembis(N, D, Q, [DS | P]) :- Q < D, !, prembis(N, DS, Q, P).
        % Si D est plus grand que Q, on passe au nombre premier précédent
prembis(N, D, _, _) :- divise(D, N), !, fail. % D divise N, il ne sera pas premier
prembis(N, D, Q, [DS | P]) :- prembis(N, DS, Q, P).
prembis(_, _, _, []). % On peut aussi encombrer avec :- asserta(premier(N)).

premier(R, N) :- prem([N | _], R).
      % Pour avoir le nombre premier N de rang donné R
```

queue(3, [a, b, c, d, e, f, g, h], *R*). → *R* = [d, e, f, g, h]

prem(*P*, 8). → *P* = [19, 17, 13, 11, 7, 5, 3, 2]
prem(*P*, 16).
→ *P* = [53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2]
premier(37). → yes
premier(101). → yes

prem(*P*, *N*).
→ *P* = [53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2] *N* = 16

premier(49, N). → *N* = 227 ; *N* = 227

prem(*P*, *N*). → *P* = [227, 223, 211, 199, 197, 193, 191, 181, 179, 173, 167, 163, 157, 151, 149, 139, 137, 131, 127, 113, 109, 107, 103, 101, 97, 89, 83, 79, 73, 71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2] *N* = 49

prem(*P*, 100). → *P* = [541, 523, 521, 509, 503, 499, 491, 487, 479, 467, 463, 461, 457, 449, 443, 439, 433, 431, 421, 419, 409, 401, 397, 389, 383, 379, 373, 367, 359, 353, 349, 347, 337, 331, 317, 313, 311, 307, 293, 283, 281, 277, 271, 269, 263, 257, 251, 241, 239, 233, 229, 227, 223, 211, 199, 197, 193, 191, 181, 179, 173, 167, 163, 157, 151, 149, 139, 137, 131, 127, 113, 109, 107, 103, 101, 97, 89, 83, 79, 73, 71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2]

À présent, il est possible d'avoir les nombres premiers bornés par un entier *X* en les cherchant un à un.

```
preminf(P, X) :- prem([D | PP], _), D < X, !, recherche(1, D), preminf(P, X).
preminf(P, X) :- prem([D | PP], _), X < D, !, tronque(PP, X, P).
preminf(P, X) :- prem([D | P], _).
tronque(X, [Y | L], R) :- X =< Y, !, tronque(X, L, R).
tronque(X, R, R).
```

tronque(5, [7, 6, 5, 4, 3, 2, 1, 0], *R*). → *R* = [4, 3, 2, 1, 0]
preminf(*P*, 49).
→ *P* = [47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5,3, 2]
prem(*P*, *N*).
→ *P* = [53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11, 7, 5, 3, 2] *N* = 16
Pour constater que la liste s'est allongée.
preminf(*P*, 12). → *P* = [11, 7, 5, 3, 2]

Et pour finir, la décomposition en facteurs premiers.

```
decomp(N, [N]) :- premier(N), !.
decomp(N, R) :- preminf(P, N), reduire(P, N, R).

reduire([D | P], N, [D | R]) :- divise(D, N), !, NS is N / D, decomp(NS, R).
reduire([_ | P], N, R) :- reduire(P, N, R).
```

decomp(17, *X*). → *X* = [17]
decomp(12, *X*). → *X* = [3, 2, 2]
decomp(30, *X*). → *X* = [5, 3, 2]
decomp(49, *X*). → *X* = [7, 7]
decomp(113, *X*). → *X* = [113]
decomp(91, *X*). → *X* = [13, 7]

24 Fonction « act »
Il s'agit de la fonction *act(x)* = racine *x*-ième du produit des nombres premiers inférieurs à *x*. Programmer cette fonction $act(x) = (\Pi p)^{1/x}{}_{p<x,\ p \text{ premier}}$, discontinue, qui entre chaque valeur première est très légèrement décroissante. Elle admet un maximum voisin de 3 pour *x* = 113 et tend vers *e* à l'infini. Il est nécessaire de chercher le produit des nombres premiers en tant que grands nombres (listes de leurs chiffres).
act(3) = 1.26, *act*(4) = 1.56, *act*(5) = 1.43, *act*(6) = 1.76, *act*(7) = 1.62, *act*(8) = 1.95, *act*(9) = 1.81, *act*(10) = 1.7, *act*(11) = 1.62, *act*(12) = 1.9, *act*(13) = 1.81…

On reprend tous les prédicats définis au chapitre 3 sur les grands nombres, en rajoutant les suivants et on définit notamment le produit des éléments d'une liste de grands nombres.

```
maptrans([], []). % réalise la transcription des nombres en grands nombres
maptrans([N | L], [GN | GL]) :- trans(N, GN), maptrans(L, GL).

prod([X], X).      % réalise le produit des grands nombres situés dans la liste
prod([X | L], R) :- prod(L, K), mul(X, K, R).

act(X, F) :- preminf(P, X), maptrans(P, GP),
        prod(GP, M), rac(X, M, GQ), trans(F, GQ).
```

```
prod([[2, 0], [2, 0, 0, 0, 0], [3, 0, 0], [1, 0, 0]], P).
→ P = [1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0]

maptrans([23, 16, 49, 77, 84, 41, 512, 65536, 0], GN).
→ GN = [[2, 3], [1, 6], [4, 9], [7, 7], [8, 4], [4, 1], [5, 1, 2],
          [6, 5, 5, 3, 6], [0]]
act(49, Y). → Y = 2
act(64, Y). → Y = 2 % c'est à faire en réels
```

25 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'exécution. *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.
Un exemple qui date des premiers temps du Prolog, consiste à créer 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.
[P. St Dizier, *Initiation à la programmation en Prolog*, Eyrolles, 1987]

```
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
```

26 Minimorpion
Soit le problème suivant :

Etat initial $\begin{bmatrix} ® & ® & \\ ® & & \\ & & \end{bmatrix}$ Etat final $\begin{bmatrix} & & \\ & & ® \\ & ® & ® \end{bmatrix}$ Numérotation adoptée $\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$

Il faut jouer une partie sachant que l'on peut bouger un pion sur une case adjacente (un côté commun) ou faire sauter un pion par dessus un autre horizontalement ou verticalement à condition que la case d'arrivée soit libre. *retractall(P).* permet de retirer la clause *P* de la liste des clauses.
Attention, suivant les versions de Prolog, *retract* peut être évalué à vrai ou à faux lorsqu'il n'a rien retiré.

Dans le but d'écrire une planification (chapitre précédent), on donne des spécifications d'opérations de « glisser » et de « sauter ». On peut donc affirmer que chacune de ces deux actions est un prédicat à deux arguments notés *E* et *F* dans ce qui suit.

glisser de *X* vers *Y* : condition *X* occupé, *Y* libre, *X* et *Y* sont adjacents
ajout : *Y* occupé
retrait : *X* occupé

sauter de *X* vers *Y* : condition *X* occupé, *Y* libre,
et *X* et *Y* colinéaires séparés par *Z* occupé
ajout : *Y* occupé
retrait : *X* occupé

On peut aussi écrire un prédicat d'alignement de trois cases :

```
ligne(X, Y, Z) :- adj(X, Y), adj(Y, Z), Y is (X + Y)/2.

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

adj(N, M) :- N < 6, M is N + 3.
adj(N, M) :- M is N + 1, not(app(N, [2, 5, 8])).

saut(X, Y) :- Y is X + 6, Y < 9.
saut(0, 2).
saut(3, 5).
saut(6, 8).
```

```
glisser(E, F) :- pris(E), adj(E, FY), not(pris(F)), write(glisser(E, F)),
        retract(pris(E)), asserta(pris(F)), nl.
sauter(E, F) :- pris(E), saut(E, FY), not(pris(F)), M is (E + F)//2,
        pris(M), write(sauter(E, F)), retract(pris(E)), asserta(pris(F)), nl.

init :- retractall(pris(_)), asserta(pris(0)), asserta(pris(1)), asserta(pris(3)).

final :- pris(5), pris(7), pris(8).

mouv.
mouv :- app(E, [0, 1, 2, 3, 4, 5, 6, 7]), sauter(E, F), mouv.
mouv :- app(E, [0, 1, 2, 3, 4, 5, 6, 7]), glisser(E, F), mouv.
jeu :- init, mouv, final.
```

```
jeu.        % utilisation renvoyant la suite de déplacements :
sauter(0, 6)
glisser(1, 4)
sauter(3, 5)
glisser(4, 7)
sauter(6, 8)
yes
```

27 Robot envoyé sur Pluton
Au camp de base, un robot dispose de trois charges de carburant et il est capable d'effectuer les actions *aller(X, Y), prendre(C, X), poser(C, X), faireleplein(C, X)* où *X, Y* sont des lieux consécutifs : camp de base, étape, montagne. Il lui faut une charge de carburant pour parcourir une de ces distances et il peut en prendre une seule autre dans sa « remorque ».

```
consec(X, Y, [X, Y | _]).
consec(X, Y, [_ | L]) :- consec(X, Y, L).
consec(X, Y, [Y, X | _]).

init :- retractall(place(_, _)), retractall(roboten(_)), asserta(roboten(base)),
                asserta(place(c1, base)), asserta(place(c2, base)),
                asserta(place(c3, base)).

trajets(X, [X, Z | L], Y) :- aller(X, Z), trajets(Z, [Z | L], Y).
trajets(X, [X], X).

aller(X, Y) :- roboten(X), consec(X, Y, [base, etape, montagne]), place(C, X),
                retract(place(C, X)), prendre(D, X),
                write(alleravecplein(X, Y, C)), nl, retract(roboten(X)),
                asserta(roboten(Y)), poser(D, Y).

prendre(C, L) :- place(C, L), retract(place(C, L)), asserta(pris(C)),
write(prendre(C)), nl.
prendre(_, _).      % cas où on ne prend rien

poser(C, L) :- roboten(L), pris(C), asserta(place(C, L)), retract(pris(C)),
        write(poser(C)), nl.
poser(_, _).        % cas où rien n'est pris donc rien à poser

jeu(L) :- init, trajets(base, L, montagne).
```

```
jeu(X). →
prendre(c2)
alleravecplein(base, etape, c3)
poser(c2)
alleravecplein(etape, montagne, c2)
X = [base, etape, montagne]
```

Ces deux derniers exercices sont des exemples non généralisables, ce sont plutôt de mauvais exemples. Mais pour continuer, on peut généraliser à un

voyage interplanétaire : calculer le coût d'un voyage dans le système solaire sachant que si *u* est l'unité de base égale au 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, etc.

28 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)).
```