Prolog

Un programme en Prolog peut être vu comme :

programme = base de données + requêtes

La base de données (appelée *base de connaissance*) contient des *clauses* qui sont des *faits* ou des *règles*. Un programme n'est pas une séquence d'actions mais un ensemble de faits avec des règles pour inférer de nouveaux faits à partir de ceux existants.

L'exécution interactive d'un programme se traduit par la demande d'une question, et si le résultat est vrai (`yes`), l'engin de recherche de Prolog identifie les variables satisfaisant la question. Autrement, Prolog répond par `no`.

Pour les exécutions compilée, le programme démarre par la démonstration d'une clause.

Historique

1972 : création de Prolog par A. Colmerauer et P. Roussel à U de Marseille.

1980 : reconnaissance de Prolog comme langage de développement en intelligence artificielle.

Depuis : plusieurs dialectes

plusieurs extensions vers la programmation par contraintes : Gödel, Oz, etc.

Éléments fondamentaux du *Prolog* : Faits

Les faits sont des affirmations qui décrivent des relations ou des propriétés.

Exemples

```
masculin(jean).
feminin(catherine).
pere(paul,jean).     % paul est le père de jean
mere(isabelle,jean). % isabelle est la mère de jean
```

La forme générale d'un fait est :

```
prédicat(argument1,argument2, …).
```

Un prédicat est un symbole qui traduit une relation. L'*arité* est le nombre de ses arguments. On identifie un prédicat par son nom et son arité : `prédicat/arité`, par exemple `mere/2`.

De manière générale :

- Un prédicat unaire donne une propriété de l'argument :
  Fifi est un chien. `chien(fifi).`
  Le soleil est chaud. `chaud(soleil).`
  Le ciel est bleu. `bleu(ciel).`
- Un prédicat binaire exprime une relation entre 2 objets :
  Fido est plus grand que Fifi. `plusGrand(fido,fifi).`
- Un prédicat d'arité supérieur à 2 établit un lien entre ses arguments :
  `compositeur(beethoven,1770,1827).`

Questions ou requêtes

En mode interactif, on peut poser des questions sur les faits :

```
?- masculin(jean).
Yes

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

On peut aussi utiliser des *variables*. Elles peuvent s'identifier à toutes les autres valeurs : aux *constantes*, aux *termes composés*, aux variables elles-mêmes.

Les constantes commencent par une minuscule, les variables commencent par une majuscule.

```
?- masculin(X).
X = jean ;
X = luc ;
No
```

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

Types en *Prolog*

En *Prolog*, tout est un terme :

- Les *constantes* ou les *termes atomiques* :

  Les *atomes* sont des chaînes alphanumériques qui commencent par une minuscule : `jean`, `paul`, `pas0.`

  On peut transformer une chaîne contenant des caractères spéciaux (point, espaces, etc.) dans un atome en l'entourant de caractères « ' » ou « " ». Ainsi `'vin de Lavaux', 'Lavaux'` sont des atomes.

  Les *nombres* : 19, -25, -3.14, 23E-5

- Les *variables* commencent par une majuscule ou le signe _.

  _ tout seul est une variable anonyme et peut désigner n'importe quelle variable.
  Ainsi `X, XYZ, Xyz, _x _3 _` sont des variables.
  Le système renomme en interne ses variables et utilise la convention `_nombre`, comme `_127` ou `_G127`.

Remarque :

Certains Prolog ont introduit le concept de *domaine* pour distinguer les symboles (variables) les entiers (long, ulong), les réels, etc. C'est le cas de Visual Prolog®.

Quelques définitions

Une *formule* est une application d'un prédicat à ses arguments.

L'*unification* est le procédé par lequel on essaie de rendre 2 formules identiques en donnant des valeurs aux variables qu'elles contiennent.

Une unification peut réussir ou échouer :

`pere(X,X)` et `pere(jean,paul)`

ne peuvent pas s'unifiées.

Lors d'une question, si elle réussit, les variables s'*unifient*.

Exemple :

```
?- pere(X,Y)
X = paul
Y = jean
Yes
```

Le Prolog unifie le terme de la question au terme contenu dans la base de données. Pour ceci, il réalise la *substitution* des variables `X` et `Y` par des termes, ici des constantes. On note cette substitution par :

`{X = paul, Y = jean}`

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

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

En *Prolog*, une variable ne représente pas une location contenant une valeur modifiable.

Une variable instanciée ne peut pas changer de valeur.

Types en *Prolog* : Variables partagées

On utilise une même variable pour contraindre deux arguments à avoir la même valeur.

Exemple

Pour chercher un père qui a le même nom que son fils (exemple un peu idiot) :

```
?- pere(X,X).
```

Les questions peuvent être des conjonctions et on peut partager des variables entre les buts.

Exemple

Pour chercher tous les fils d'un père, on partage la variable `X` entre `pere` et `masculin` :

```
?- pere(X,Y), masculin(Y).
```

Règles

Les *règles* permettent d'exprimer des *conjonctions de buts*.

Leur forme générale est :

<tête> :- C_1, C_2,... ,C_n.

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

Les éléments du corps sont aussi appelés des *sous-buts*, car pour démontrer la tête de la règle, il faut démonter tous ses sous-buts.

Exemples

« Si X est un homme alors X boit de la bière. »

```
boit(X,biere) :- homme(X)
```

« Il n'y a pas de fumée sans feu. »

```
fumee :- feu
```

« Jean étranglera le prof s'il échoue ce cours. »

```
etrangle(jean,X) :-
  echoue(jean,'Langages et paradigmes de
                                programmation'),
  enseigne(X,'Langages et paradigmes de
                                programmation').
```

Arbre généalogique :

```
fils(A,B):- père(B,A), masculin(A).
fils(A,B):- mère(B,A), masculin(A).
parent(X,Y):- père(X,Y).
parent(X,Y):- mère(X,Y).
grandParent(X,Y):- parent(X,Z), parent(Z,Y).
```

Règles

- Les faits sont une forme particulière de règles qui sont toujours vraies. Un fait est une clause dont la queue est vide.

  Exemples :

  `fait.` est en effet équivalent à `fait :- true.`

  `boit(X,biere).` signifie que tout le monde boit de la bière ou « Pour tout X, X boit de la bière.»

- On peut aussi utiliser une variable anonyme :

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

  peut s'écrire

  ```
  enfant(Y) :- parent(_,Y).
  ```

- Une *requête* est une clause sans tête

  ?- <*corps*>.

  signifiant : Détermine si <*corps*> est vrai.

- Les règles peuvent aussi être récursives.

  Exemple : Définissons un nouveau prédicat déterminant l'ancêtre X de Y par récursivité :

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

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

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

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

Exécution des requêtes

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

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

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

est équivalent à

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

Un prédicat correspond donc à un ensemble de règles ou de faits de même nom et de même arité : les clauses du prédicat.

Beaucoup de Prolog demandent que toutes les règles et tous les faits d'un même prédicat soient contigus et groupés ensembles dans le fichier du programme. On note un prédicat par son nom et son arité, par exemple `fils/2`, `parent/2`, `grandParent/2`.

Cas de Visual Prolog®

Un environnement de développement intéressant est Visual Prolog®.

En Visual Prolog®, un programme comporte 4 sections :

- La section des *clauses* décrivant les faits et les règles.
- La section des *prédicats* pour déclarer les prédicats et le domaine (types) de ses arguments.
- La section des *domaines* pour déclarer tous les nouveaux types pas prédéfinies.
- La section de l'*objectif* qui est le but principal ou l'amorçage du programme.

Exemple

```
DOMAINS
  nom, sport = symbol
PREDICATES
  pratique(nom,sport)
CLAUSES
  pratique(charles,tennis).
  pratique(jean,football).
  pratique(paul,hockey).
  pratique(pierre,natation).
  pratique(marc,tennis).
  pratique(jules,Activite) :- pratique(paul,Activite).
GOAL
  pratique(jules,hockey).
```

Exercices

Exercice 1 : Arbre généalogique

Pierre a 3 enfants: Adam, Paul et Chloé.

Les faits du programme *Prolog* sont :

```
homme(pierre).
homme(adam).
homme(paul).
femme(chloe).
pere(pierre,adam).
pere(pierre,paul).
pere(pierre,chloe).
```

A partir de ces assertions, donnez les règles générales :

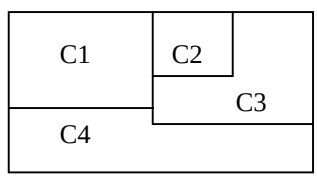
1. enfant(X,Y) qui exprime que X est un enfant de Y
2. fils(X,Y) qui exprime que X est un fils de Y
3. fille(X,Y) qui exprime que X est une fille de Y
4. frereOuSoeur(X,Y) qui exprime que X est frère ou sœur de Y
5. frere(X,Y) qui exprime que X est un frère de Y
6. soeur(X,Y) qui exprime que X est une sœur de Y

Attention : un individu n'est pas son propre frère ou sa propre sœur.

Utilisez le prédicat X \= Y pour distinguer 2 variables.

Exercice 2

On se propose de définir un prédicat permettant de colorier la carte suivante :



Les règles sont les suivantes :

- On dispose de trois couleurs qui sont : vert, jaune et rouge ;
- Deux zones contiguës doivent avoir des couleurs différentes.

Ecrivez un prédicat *coloriage(C1, C2, C3, C4)* qui comporte deux parties : la première partie génère toutes les valeurs possibles de C1, C2, C3 et C4. La seconde vérifie si les colorations obtenues sont conformes à la carte par l'utilisation du prédicat *X \= Y* sur les couleurs des zones contiguës.

Solutions

Exercice 1

1. `enfant(X,Y) :- pere(Y,X).`
2. `fils(X,Y) :- enfant(X,Y), homme(X).`
3. `fille(X,Y) :- enfant(X,Y), femme(X).`
4. Si on admet que X est le frère ou la sœur de Y si X et Y ont le même père Z, la relation *frereOuSoeur* s'écrit :
   `frereOuSoeur(X,Y) :- pere(Z,X), pere(Z,Y), X \= Y.`
5. X est le frère (respectivement la sœur) de Y à condition que X soit frère ou sœur de Y et que X soit de sexe masculin (respectivement féminin). Ce qui nous donne :
   `frere(X,Y) :- frereOuSoeur(X,Y), homme(X).`
6. `soeur(X,Y) :- frereOuSoeur(X,Y), femme(X).`

Exercice 2

```
couleur(vert).
couleur(jaune).
couleur(rouge).
coloriage(C1,C2,C3,C4) :- couleur(C1), couleur(C2),
        couleur(C3), couleur(C4), C1 \= C2, C1 \= C3, C1 \= C4,
        C2 \= C3, C3 \= C4.
```

Ici, on génère tout d'abord les couleurs, et on effectue les tests de conformité ensuite.

```
coloriage(C1,C2,C3,C4) :- couleur(C1), couleur(C2),
        C1 \= C2, couleur(C3), C1 \= C3, C2 \= C3,
        couleur(C4), C1 \= C4, C3 \= C4.
```

Ici, on effectue les tests de conformité à la volée, ce qui réduit grandement la taille de l'arbre de recherche.

Solutions

Exercice 2 (suite)

En Visual Prolog®, le programme correspondant à la première solution est

```
PREDICATES
  nondeterm couleur(symbol)
  nondeterm coloriage(symbol,symbol,symbol,symbol)
CLAUSES
  couleur(vert).
  couleur(jaune).
  couleur(rouge).
  coloriage(C1,C2,C3,C4) :- couleur(C1), couleur(C2),
                couleur(C3), couleur(C4),C1 <> C2, C1 <> C3,
                C1 <> C4, C2 <> C3, C3 <> C4.
GOAL
  coloriage(C1,C2,C3,C4).
```

et donne les unifications suivantes :

```
  C1=vert, C2=jaune, C3=rouge, C4=jaune
  C1=vert, C2=rouge, C3=jaune, C4=rouge
  C1=jaune, C2=vert, C3=rouge, C4=vert
  C1=jaune, C2=rouge, C3=vert, C4=rouge
  C1=rouge, C2=vert, C3=jaune, C4=vert
  C1=rouge, C2=jaune, C3=vert, C4=jaune
  6 Solutions
```

Opérateurs

Chaque *Prolog* dispose d'opérateurs binaires infixés (`X+Y`), unaire préfixés (`-X`) et unaire suffixés (`X^2`). Le langage considère les opérateurs comme des foncteurs et transforme les expressions en termes : `2 * 3 + 4 * 2` est un terme identique à `+(*(2,3),*(4,2))`.

Règles de précédence et d'associativité

Les opérateurs sont définis par leurs priorités 0 à 1200 (0 étant la plus haute) et leurs associativités.

L'associativité détermine le parenthésage de `A op B op C` :

- Si elle est à gauche, on a `(A op B) op C`
- Si elle est à droite, on a `A op (B op C)`
- Sinon elle est non associative : les parenthèses sont obligatoires, et la syntaxe `A op B op C` est interdite.

Un type d'associativité est associé à chaque opérateur :

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

Règles de précédence et d'associativité

Certains opérateurs sont prédéfinis en Prolog Standard.

Priorité	Associativité	Opérateurs
1200	xfx	:-
1100	xfy	;
1000	xfy	,
700	xfx	= \=
700	xfx	== \==
700	xfx	is =:= =\= < <= > >=
500	yfx	+ -
400	yfx	* / (flottant) // (entier)
200	xfy	& ^
200	fy	-

On peut définir de nouveaux opérateurs par :

`op(Précédence,Associativité,NomOpérateur).`

Exemple

`op(1000,xfx,aime)` définit `aime` comme un opérateur infixé non associatif.

Par la suite, on peut écrire l'expression `Tarzan aime Jane` au lieu de `aime(Tarzan,Jane)`.

Opérateurs logiques d'unification

- `=` dans `X = Y` signifie « `X` s'unifie avec `Y` »

```
?- a(b,X,c) = a(b,Y,c).
X = Y
?- p(a,[A,[B,C]],25) = p(C,[B,[D,E]],25).
A = B = D, C = E = a.
```

- `\=` dans `X \= Y` signifie « `X` ne s'unifie pas avec `Y` »

```
?- a(b,X,c) \= a(b,Y,c).
No
?- [A,[B,C]] \= [A,B,C].
Yes
```

Opérateurs arithmétiques

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

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

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

?- X is 1 + 1 + a.      % erreur car a n'est pas un nombre

?- X is 1 + 1 + Z.      % erreur car Z n'est pas instancié à un nombre

?- 5 is (3*7+1) // 4.
Yes

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

Opérateurs numériques vs littérales

Dans l'expression

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

Il y a évaluation des 2 additions (celle de gauche puis celle de droite) avant d'évaluer la comparaison.

Il est important de distinguer les opérateurs numériques des opérateurs littéraux, ainsi que d'unification.

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

Exemples

```
?- 1 + 2 =:= 2 + 1.
Yes
?- 1 + 2 = 2 + 1.
No
?- 1 + 2 = 1 + 2.
Yes
?- 1 + X = 1 + 2.
X = 2.
```

Opérateurs numériques vs littérales

```
?- 1 + X =:= 1 + 2.          % X n'est pas un nombre

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

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

?- 1 + X == 1 + 2.           % « est déjà instancié ?»
No

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

?- a(b,X,c) == a(b,Y,c).
No

?- a(b,X,c) == a(b,X,c).
Yes

?- A \== hello.              % « n'est pas déjà instancié »
Yes

?- a(b,X,c) \== a(b,Y,c).
Yes
```

Listes

Motivation

Comment pourrait-on représenter le contenu d'un frigidaire ?

```
frigidaire(lait,eau,creme)
frigidaire(beurre,chat)
frigidaire(lait,beurre,fromage,bière)
```

Ceci donne 3 faits différents représentés sous la forme de structures.

Pour représenter un nombre variable d'arguments avec la même structure, on pourrait utiliser un fait d'arité 2 : `frigidaire`/2

```
frigidaire(lait,frigidaire(eau,frigidaire(creme,nil)))
frigidaire(beurre,frigidaire(chat,nil))
frigidaire(lait,frigidaire(beurre,frigidaire(fromage,
   frigidaire(bière,nil))))
```

Cette notation est trop encombrante. Le plus simple serait d'introduire une liste.

```
.(lait,.(eau,.(creme,nil)))
.(beurre,.(chat,nil))
.(lait,.(beurre,.(fromage,.(bière,nil))))
```

Listes

Le foncteur « `.` » est le foncteur de liste : `.(a,.(b,[]))`. Il est équivalent à `[a,b]`.

Exemples

```
.(a,.(b,.(c,[])) = [a,b,c]
.(1,.(2,.(3,.))) = [1,2,3]
[a]
[a,X,pere(X,jean)]
[[a,b],[[[pere(X,jean)]]]]
```

`[]` est la liste vide.

L'opérateur « | » est un constructeur de liste et peut servir pour extraire la tête de la liste.

Exemples

```
.(a,.(b,.(c,[])) = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]] = [a,b,c]
.(1,.(2,.(3,.))) = [1|[2|[3]]] = [1,2,3]

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

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

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

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

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

Quelques prédicats de manipulation de listes

Certains prédicats de manipulation de listes sont prédéfinis suivant les variantes de *Prolog*.

1. Prédicat `member/2`

Appartenance d'un élément à une liste.

Première façon d'écrire le prédicat :

```
member(X,[Tete|Queue]) :- X = Tete
member(X,[Tete|Queue]) :- X \= Tete, member(X,Queue).
```

Mais puisque *Prolog* utilise des gabarits pour identifier les termes d'une expression et que les règles sont exécutées dans l'ordre qu'elles sont écrites, on peut simplifier la définition du prédicat :

```
member(Tete,[Tete|_]).
member(Tete,[_|Queue]) :- member(Tete,Queue).
```

Exemples

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

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

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

Exécution de `member(X,[a,b,c]).`

Trouver une unification :

- unifie `member(X,[a,b,c])` avec `member(Tete,[Tete|_])`
- instancie `X` à a
- Une règle sans corps réussit, affichage de `X = a`

On presse « `;` » pour demander la solution suivante.

Trouver une seconde unification

- unifie `member(X,[a,b,c])` avec `member(Tete,[_|Queue])`
- instancie `X` à `Tete`, instancie `Queue` à `[b,c]`
- replace la tête de la règle par son corps et essaie de satisfaire le but `member(X,[b,c])`

Trouver une unification

- unifie `member(X,[b,c])` avec `member(Tete,[Tete|_])`
- instancie `X` à b
- Une règle sans corps réussit, affichage de `X = b`

etc.

Quelques prédicats de manipulation de listes

2. Prédicat append/3

Concatène deux listes.

```
append([],L,L).
append([X|XS],YS,[X|Liste]) :- append(XS,YS,Liste).
```

Exemples

```
?- append([a,b,c],[d,e,f],[a,b,c,d,e,f]).
Yes
?- append([a,b],[c,d],[e,f]).
No
?- append([a,b],[c,d],L).
L = [a,b,c,d]
?- append(L,[c,d],[a,b,c,d]).
L = [a,b]
?- append(L1,L2,[a,b,c]).
L1 = [], L2 = [a,b,c] ;
L1 = [a], L2 = [b,c] ;
```

etc., avec toutes les combinaisons

Si nous souhaitons afficher toutes les combinaisons possibles sans faire intervenir l'usager,

```
combinaisonListe(Liste) :-
   append(L1,L2,Liste),
   write(L1), write(L2), nl,
   fail.            % Le prédicat fail échoue toujours.
```

`append(L1,L2,[a,b]).` donne

```
[][a,b]
[a][b]
[a,b][]
No
```

Quelques prédicats de manipulation de listes

3. Prédicat `length/2`

Détermine la longueur d'une liste :

```
length([],0).
length([Tete|Queue],N) :-
   length(Queue,N1), N is N1 + 1.
```

Exemples

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

4. Prédicat `reverse/2`

Inverse l'ordre d'une liste.

Première solution coûteuse :

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

Deuxième solution par une liste intermédiaire : On passe d'une arité 2 à une arité 3 :

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

Coupure

Le prédicat prédéfini « `!` » appelé « le cut » permet d'empêcher le retour en arrière. La règle

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

satisfait d'abord le premier sous-but `a(X)`, puis vérifie ensuite le second sous-but `b(X)`. Si le second sous-but échoue, il n'y a pas de retour en arrière à `a(X)`.

- La coupure permet d'indiquer à Prolog qu'on ne désire pas conserver les *points de choix* en attente.
- Le but recherché est d'améliorer l'efficacité et d'exprimer le déterminisme.
- Quand une coupure est franchie :
  - ! coupe toutes les clauses en dessous de lui,
  - ! coupe tous les buts à sa gauche,
  - Par contre, il est possible de retourner arrière sur les sous-buts à droite de la coupure.

Exemples

```
factorielle(0,1) :- !.
factorielle(1,1) :- !.
factorielle(N,Resultat) :-
   N1 is N-1,
   factorial(N1,ResultatPrecedent),
   Resultat is ResultatPrecedent*N.

estMembre(X,[X|_]) :- !.
estMembre(X,[_|Queue]) :- estMembre(X,Queue).
```

Coupure

Règles pratiques d'utilisation

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

Ce prédicat conduit correctement au calcul du minimum pour la première solution. La coupure est nécessaire pour éviter de produire une seconde solution au cours d'un retour en arrière. Mais si nous avons la règle :

P :- C_1, …, C_{i-1}, min(2,3,Z), C_i, …, C_n.

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

Autre coupure

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

Coupure : Danger

Tous les enfants ont 2 parents à l'exception de Robin.

```
parents(batman,2).
parents(robin,0) :- !.
parents(X,2).

?- parents(robin,N).
N = 0
```

mais

```
?- parents(robin,2).
Yes
```

Pour corriger le problème, il faudrait écrire :

```
parents(batman,2).
parents(robin,N) :- !, N=0.
parents(X,2).
```

Négation

Un programme logique exprime ce qui est vrai. La négation, c'est ce que l'on ne peut pas prouver.

Le prédicat `not` se définit par :

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

Exemple et précautions

Il vaut mieux que les variables soient instanciées avant d'utiliser une négation :

```
?- not(member(X,[a,b])).
No
```

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

Négation

Dans une règle, il faut veiller à l'instanciation des variables.

Exemple

```
marie(francois).
etudiant(rene).

etudiantCelibataire(X) :-
    not(marie(X)),
    etudiant(X).
```

La requête suivante échoue :

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

car `X` s'associe à `francois` puis le `not` fait échouer le sous-but.

Alors que :

```
?- etudiantCelibataire(rene).
Yes
```

réussit car `X = rene` donc `not(marie(rene))` est vrai.

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

```
etudiantCelibataire(X) :-
    etudiant(X),
    not(marié(X)).
```

Ainsi `X` sera instancié avant d'être soumis à la négation.

Exercice

Un paysan se dirige vers un marché vendre son fromage. Il est accompagné du chat de sa femme, de sa souris porte-bonheur. Au bord d'une rivière qu'ils doivent traverser, il y a un petit bateau ne pouvant transporter que 2 parmi les 4 objets et le paysan est le seul à pouvoir ramer.

Tout semble aller pour le mieux, si ce n'est que le chat mangerait la souris s'ils sont laissés seuls, et si la souris reste seul avec le fromage, elle le grignote le fromage et le rend invendable.

Trouver une suite de traversée permettant au paysan de vendre son fromage et de conserver son porte-bonheur.

Le problème peut se modéliser par une série d'états (paysan,chat,souris,fromage) qui peuvent prendre 2 valeurs : gauche et droite.

Le but revient à transformer l'état de départ (gauche,gauche,gauche, gauche) à l'état final (droite,droite,droite,droite) par une suite d'états.

Solution

```
DOMAINS
% Position des objets par rapport à la rivière
POSITION = gauche; droite
% Un état est le tuple (paysan,chat,souris,fromage) donnant la position de
% chaque objet
ETAT = état(POSITION,POSITION,POSITION,POSITION)
% Un chemin est une liste d'états
CHEMIN = ETAT*

PREDICATES
% Débute la recherche et affiche le résultat s'il existe.
% premier argument: état initial;
% second argument: état final.
nondeterm démarreRecherche(ETAT,ETAT)

% Trouve un chemin reliant 2 états.
% premier argument: état initial du chemin;
% second argument: état final du chemin;
% troisième argument: liste intermédiaire d'états visités formant le
%                     chemin courant;
% dernier argument: liste d'états formant le chemin final (c'est le but à
% satisfaire).
nondeterm trouveChemin(ETAT,ETAT,CHEMIN,CHEMIN)

% Complète le sens de déplacement pour relier les 2 états.
% premier argument: premier état;
% second argument: état voisin du premier sinon il y a échec.
nondeterm déplace(ETAT,ETAT)

% Donne l'inverse d'une position si elle existe. Permet de passer de gauche
% à droite et vice versa.
% premier argument: première position;
% second argument: inverse de la première position sinon il y a échec.
donneInverse(POSITION,POSITION)
```

Solution

```
% Détermine si l'état ne respecte pas les contraintes. Un état (P,C,S,F)
% est invalide si C=S ou si S=F et que le paysan ne se trouve pas sur le même
% bord de la rivière.
% argument: état à vérifier.
nondeterm vérifieContraintes(ETAT)

% Vérifie si un état fait partie d'un chemin ou d'une liste, nécessaire pour
% trouver une solution sans cycle.
% premier argument: état à vérifier;
% second argument: chemin ou liste où l'état peut figurer.
nondeterm estMembre(ETAT,CHEMIN)

% Affiche toutes les étapes (états) d'une solution
% argument: liste des états à afficher.
afficheChemin(CHEMIN)

% Affiche les objets qui traversent la rivière. Ceux-ci sont donnés par la
% différence de 2 états
% ajacents d'une solution
% premier argument: état de départ;
% second argument: état d'arrivé.
afficheTransition(ETAT,ETAT)

CLAUSES
démarreRecherche(EtatInitial,EtatFinal):-
  trouveChemin(EtatInitial,EtatFinal,[EtatInitial],Solution),
  write("Une solution du problème est :\n"),
  afficheChemin(Solution).

trouveChemin(EtatInitial,EtatFinal,EtatsEmpruntés,Solution):-
  déplace(EtatInitial,EtatSuivant),           % Généner le prochain état
  not(vérifieContraintes(EtatSuivant)),       % Vérifier s'il est sage de
                                              % faire cette transition
  not(estMembre(EtatSuivant,EtatsEmpruntés)), % Eviter les cycles
  trouveChemin(EtatSuivant,EtatFinal,[EtatSuivant|EtatsEmpruntés],Solution)
  .                     % Pour avoir une seule solution, remplacer . par ,!.
trouveChemin(EtatFinal,EtatFinal,Solution,Solution).   % Unifier les chemins
```

Solution

```
déplace(état(X,X,Souris,Fromage),état(Y,Y,Souris,Fromage)):-
  donneInverse(X,Y).                    % Déplacer le paysan et le chat
déplace(état(X,Chat,X,Fromage),état(Y,Chat,Y,Fromage)):-
  donneInverse(X,Y).                    % Déplacer le paysan et sa souris
déplace(état(X,Chat,Souris,X),état(Y,Chat,Souris,Y)):-
  donneInverse(X,Y).                    % Déplacer le paysan et le fromage
déplace(état(X,Chat,Souris,Fromage),état(Y,Chat,Souris,Fromage)):-
  donneInverse(X,Y).                    % Déplacer le paysan

donneInverse(gauche,droite).
donneInverse(droite,gauche).

vérifieContraintes(état(Paysan,X,X,_)):-  % Le chat mange la souris si le
  donneInverse(Paysan,X), !.              % paysan n'est pas présent
vérifieContraintes(état(Paysan,_,X,X)):-  % La souris mange le fromage si
  donneInverse(Paysan,X), !.              % paysan n'est pas présent

estMembre(X,[X|_]):- !.
estMembre(X,[_|Liste]):- estMembre(X,Liste).

afficheChemin([EtatDebut,EtatFinal|QueueChemin]):-
  afficheTransition(EtatDebut,EtatFinal),
afficheChemin([EtatFinal|QueueChemin]).
afficheChemin([]).

afficheTransition(état(X,Chat,Souris,Fromage),état(Y,Chat,Souris,Fromage)):-
  !, write("Le paysan traverse la rivière de ",X," à ",Y,"\n").
afficheTransition(état(X,X,Souris,Fromage),état(Y,Y,Souris,Fromage)):- !,
  write("Le paysan et le chat traversent la rivière de ",X," à ",Y,"\n").
afficheTransition(état(X,Chat,X,Fromage),état(Y,Chat,Y,Fromage)):- !,
  write("Le paysan et sa souris traversent la rivière de ",X," à ",Y,"\n").
afficheTransition(état(X,Chat,Souris,X),état(Y,Chat,Souris,Y)):- !,
  write("Le paysan et son fromage traversent la rivière de ",X," à ",Y,"\n").

GOAL
démarreRecherche(état(gauche,gauche,gauche,gauche),
                 état(droite,droite,droite,droite)).
```

Solution : exécution

Une solution du problème est :
Le paysan et sa souris traversent la rivière de droite à gauche
Le paysan traverse la rivière de gauche à droite
Le paysan et son fromage traversent la rivière de droite à gauche
Le paysan et sa souris traversent la rivière de gauche à droite
Le paysan et le chat traversent la rivière de droite à gauche
Le paysan traverse la rivière de gauche à droite
Le paysan et sa souris traversent la rivière de droite à gauche

Une solution du problème est :
Le paysan et sa souris traversent la rivière de droite à gauche
Le paysan traverse la rivière de gauche à droite
Le paysan et le chat traversent la rivière de droite à gauche
Le paysan et sa souris traversent la rivière de gauche à droite
Le paysan et son fromage traversent la rivière de droite à gauche
Le paysan traverse la rivière de gauche à droite
Le paysan et sa souris traversent la rivière de droite à gauche
no