Introduction à la programmation logique

Prolog

Jacques Savoy

Bibliographie

[Clocksin 03] William F. Clocksin, Christopher S. Mellish : Programming in Prolog. 5th Ed., Springer-Verlag, Berlin, 2003. (G6-1999).

[Bratko 01] Ivan Bratko : Prolog Programming for the Artificial Intelligence. 3rd Ed., Addison-Wesley, Harlow (UK), 2001. (G6-1919).

[O'Keefe 90] Richard A. O'Keefe : The Craft of Prolog. The MIT Press, Cambridge (MA), 1990. (G6-1415)

[Sterling 86] Leon Sterling, Ehud Shapiro : The Art of Prolog: Advanced Programming Techniques. The MIT Press, Cambridge (MA), 1986. (G6-139).

[Sethi 96] Ravi Sethi : Programming Languages : Concepts & Constructs. Addison-Wesley, Reading (MA), 1996. (G6-1554).

Plan

Chapitre 1 : Un survol de Prolog

- Prolog (PROgrammer en LOGique, 1970, Robert A. Kowalski (Edinburgh) & Alain Colmerauer (Marseille)) est né du besoin de pouvoir traiter la langue naturelle par ordinateur et, en particulier, la grammaire. Autres domaine d'applications :
    - Base de données relationnelles ;
    - Logique (et mathématiques) ;
    - Résolution de problème abstrait ;
    - Traitement de la langue naturelle ;
    - Résolution symbolique d'équations ;
    - Intelligence artificielle.
- On présente aussi Prolog comme un outil de programmation déclarative en l'opposant à une approche procédurale (ou prescriptive). Cette affirmation est vraie dans certaines limites et vous verrez que l'on peut aussi voir le Prolog procédural (et il est important de le connaître).

  Diverses implémentations disponibles :
    - SWI-Prolog (www.swi-prolog.org) ;
    - SICStus Prolog (www.sics.se/sicstus) ;
    - Open Prolog (www.cs.tcd.ie/open-prolog)

Un survol de Prolog (suite)

- Programmer en Prolog signifie identifier des objets et leurs relations.

  On doit déduire objets et relations entre objets d'un monde connu, comme par exemple des phrases comme « Jean est le père d'Anne » ou « Pierre est riche » ou encore « Les objets rares sont chers ». Les objets Prolog ne correspondent pas aux objets des langages de programmation par objets !

- Programmer signifie donc :
  - Spécifier des *faits* (vérifiés) sur les objets et leurs relations ;
  - Définir des *règles* sur les objets et leurs relations ;
  - Poser des *questions* sur les objets et leurs relations.

  Et si, à première vue, la réponse se limite à un *oui* ou *non*, Prolog peut faire plus.

Sicstus-Prolog

- Version SICStus (Swedish Institute of Computer Science) disponible pour Linux, MS-Windows, MacOS X. (pour environ € 132).

  Un bon manuel est à disposition.

  Possède de nombreuses extensions et aussi interfaces (avec d'autres languages ou avec des bases de données relationnelles).

```
SICStus 3.12.3 (x86-win-nt-4): Thu Oct 27 17:58:10 WEST
2005
Licensed to unine.ch.
|?- help.
(ou utiliser le menu Help – Manual)


(ou utiliser le menu File – Consult)


|?- load_files('C:/Documents and
Settings/sesavoy/MyDocuments/PrologFull/examples/famille.p
l').
% compiling c:/documents and settings/sesavoy/my
documents/prologfull/examples/famille.pl...
% compiled c:/documents and settings/sesavoy/my
documents/prologfull/examples/famille.pl in module user, 0
msec 768 bytes
| ?-


|?- halt.
(fenêtre disparaît)
```

SWI-Prolog

- Version disponible pour Linux, MS-Windows, MacOS X (under GNU Public License).

```
unix% /usr/local/bin/swipl
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.0)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free
software, and you are welcome to redistribute it under
certain conditions.
Please visit http://www.swi-prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
?- help.


?- help(consult).


?- [nom_fichier].


?- consult('Geographie.pl').
% Geographie.pl compiled 0.00 sec, 1,952 bytes
Yes
?-
?- halt.
unix%
```

Les faits

- Les faits : « Jean aime Marie » ou « Anne aime Jean » sont traduits en Prolog par :

```
aime(jean, marie).        % car Jean aime Marie
aime   (paul  ,   marie). % Paul est amoureux de Marie
aime   (marie  ,   paul). % et Marie aime Paul
```

  Avec

  - d'abord, le nom de la relation ou prédicat (débutant par une *minuscule*),
  - puis le/les arguments (ici « jean » et « marie ») séparés par une *virgule*, dans un ordre qui possède un sens (qui est le sujet et le complément de l'action décrite),
  - enfin, le fait se termine par un *point* « . ».

  Les espaces ne jouent pas de rôle et le % indique le début du commentaire. (Le commentaire peut aussi être placé entre /* … */). Le nom de la relation (ici « aime ») dépend du programmeur / analyste.

  Autres exemples :
  « Socrate est un homme »
  « Socrate est un Grec »
  « Aristote est le disciple de Socrate »
  « L'éléphant est un grand animal gris »
  « Tous les hommes sont faillibles » % C'est parfois trop compliqué

Les faits (suite)

Autres faits (dans la base de connaissance car ce n'est pas une base de données ou une simple collection de données).

```
aime(jean,peche).
homme(tintin).
pere(henri, paul).
livre(marx, leCapital).
donne(tintin, milou, os).
etudiant(tintin,20,informatique,1).
roi(tintin,belgique).
```

Si une relation comprend un seul argument, on parlera plutôt de propriété que de relation (par exemple, `homme(tintin), grand(babar)`).

L'interprétation de l'objet « tintin ».

L'interprétation « etudiant(tintin,20,informatique,1) » ou « e(t,20,i,1) » et la véracité du fait « roi (tintin, belgique) ».

Ecrivez : « Paul aime la bière », « Anne s'aime elle-même ».

- Une représentation graphique à l'aide d'un arbre est possible avec le nom de la relation comme racine et les arguments (ordonnés) comme branches.

Les questions

- En Prolog, les questions débutent par le point d'interrogation suivi du nom de la relation et du/des argument(s) (objet). On parle également de but pour désigner une question.

  Par exemple, on peut se poser la question de savoir si « Est-ce que Jean aime Marie ? ». Cette question se traduit par :

```
?- aime(jean, marie).
```

  Pour y répondre, l'interprète Prolog va essayer d'*unifier* la question posée avec un des faits de sa base de connaissance. S'il réussit, il répond « Yes » et « No » dans le cas contraire. Sur la base des connaissances suivantes :

```
aime(jean, marie).
aime(paul, marie).
aime(marie, paul).
aime(jean, peche).
aime(paul, biere).
roi(tintin,belgique).
?- aime(jean, biere).
No
?- aime(jean, marie).
Yes
?- roi(tintin,belgique).
Yes
?- homme(tintin).
ERROR: Undefined procedure: homme/1
```

  La réponse « *No* » ne signifie pas que c'est faux mais que sur la base des connaissances dont dispose Prolog, la question n'a pas pu être unifiée. De même, la réponse « *Yes* » n'indique pas que cela est vrai dans le monde réel mais que la question a pu être unifiée (peut-être qu'un jour Tintin sera le roi de Belgique …). Finalement, on a utilisé un prédicat non défini `homme` avec un seul argument.

Les variables (inconnues)

- Jusqu'à présent, nous pouvons écrire et interroger Prolog sur des questions fermées (ou des faits précis) dont la réponse est binaire (« oui / non »).

  Mais on aimerait savoir ce qu'aime Jean (tout ce qu'il aime), sans connaître a priori l'ensemble des choses que Jean aime. On a alors besoin d'une variable pour énumérer toutes ces choses. (Je préfère appeler ceci une inconnue mais de facto, on a choisit le mot de variable). On peut écrire quelque chose comme

  ```
  ?- aime(jean, «ce qui rend vrai cette relation »).
  ```

  Dans ce but, Prolog permet l'introduction de variables (dont le nom commence par une majuscule et dont la portée lexicale se limite à un énoncé).

  ```
  ?- aime(jean, CeQueJeanAime).
  ```

  ou (plus simplement)

  ```
  ?- aime(jean, X).
  X = marie
  Yes
  ```

  L'interpréteur doit alors *unifier* la question (qui comprend une variable libre) avec ses connaissances. Unifier ne signifie pas seulement voir s'il y a une égalité entre deux faits mais « rendre égaux » les deux faits. On peut trouver un (ou dans ce cas plusieurs) fait(s) qui s'unifie(nt) avec la question. La force de Prolog est de vous donner la possibilité de vous arrêter à la première réponse, de continuer pour voir la deuxième, … ou toutes les réponses possibles.

  Suivez le travail de l'interpréteur Prolog pour répondre à cette question et pour vous donnez toutes les réponses (introduisez un « ; » après chaque réponse pour indiquer « OU »).

  ```
  ?- aime(jean, X).
  X = marie ;
  X = peche ;
  No
  ```

  L'ordre des réponses fournies par l'interpréteur est donc dépendant de l'ordre des faits de la base de connaissance.

  Mais les variables peuvent aussi être introduites dans des faits (et cela est parfois très commode). Par exemple, pour indique qu' « Anne aime tout ».

  ```
  aime(anne, Z).
  aime(anne, _).  % car le _ indique aussi une variable
  ```

Les conjonctions

- En Prolog, on n'est pas limité à interroger un seul fait. On peut se demander si deux (ou plus) sont vérifiés simultanément. Voici notre base de connaissances :

```
aime(jean, peche).
aime(jean, marie).
aime(jean, paris).
aime(marie, paul).
aime(marie, paris).
aime(paul, marie).
aime(paul, biere).
aime(anne, bretagne).
roi(tintin, belgique).
```

Ainsi, on peut se poser la question de savoir « Si Jean aime Marie et si Marie aime Jean ».

```
?- aime(jean, marie).
Yes
?- aime(marie, jean).
No
```

Mais cela est un peu laborieux et on peut grouper ces deux questions en une seule (conjonction de deux faits) par l'introduction d'un « , » (pour un « ET » logique).

```
?- aime(jean, marie) , aime(marie, jean).
No
```

D'autres questions peuvent être posées :
« Existe-t-il quelque chose que Jean et Paul aiment ? ».

Les règles

- Les faits sont toujours vrais (dans le monde que Prolog connaît). On ne limite d'aucune manière la véracité de ces faits. Par exemple, on aimerait dire que « Benjamin aime toutes les boissons sucrées ».

  D'autre part, on aimerait généraliser des faits (au moyen d'une règle) afin d'éviter d'introduire tous les faits (qui sont vrais). Par exemple, on aimerait dire que « Paul aime toutes les femmes » ou que « Tous les oiseaux ont des plumes », que « deux personnes appartiennent à la même fratrie s'ils ont les mêmes parents », etc.

  Une règle correspond à une affirmation générale sur les objets et leurs relations. Par exemple, on sait que « Paul aime tous ceux qui aiment la bière » que l'on écrit en Prolog comme suit :

  ```
  aime(paul, X) :- aime(X, biere).
  ```

  Et cette règle se compose :

  d'une tête (`aime(paul, X)`);

  du symbole « :- » pour indiquer le « SI » ;

  d'un corps (`aime(X, biere)`);

  et d'un « . » final.

  Et dans cette règle, la portée lexicale de la variable X est la règle ; le X désigné dans n'importe quelle partie de la règle est toujours le même objet.

  La règle peut s'analyser comme un fait conditionnellement vrai, c'est-à-dire vrai si le corps de la règle est vérifié. En d'autres termes, la tête peut être inférée si l'ensemble du corps est vérifié.

- L'ensemble des règles possédant le même nom (foncteur) et le même nombre d'arguments (arité) doivent se suivre dans votre programme (elles doivent former un *paquet de clauses*).

Les règles (suite)

D'autres règles peuvent être introduites :
« Anne aime tous les rois ».
« Arthur aime ceux qui l'aiment ».
« Jean aime toutes les femmes ».
« Paul aime les gens qui aiment la biere et Londres ».

« Il faut taxer les riches ».

« Marie aime toutes les villes ».

```
aime(anne, UnRoi) :- roi(UnRoi, Pays).
aime(arthur, X) :- aime(X, arthur).
aime(jean, X) :- femme(X).
aime(paul, X) :- aime(X, biere), aime(X, londres).
taxer(UnePersonne) :- riche(UnePersonne).
```

Les règles (suite)

- Voici notre base de connaissances :

```
masculin(hubert).
masculin(denis).
masculin(robert).
masculin(joseph).
masculin(georges).
masculin(henri).
feminin(nelly).
feminin(martine).
feminin(anne).
feminin(jeanne).

% parent(X,Y) est vrai si Y est le pere/mere de X
parent(robert, hubert).
parent(robert, georges).
parent(robert, anne).
parent(joseph, nelly).

parent(hubert, denis).
parent(hubert, martine).
parent(nelly, denis).
parent(nelly, martine).
parent(georges, jeanne).
parent(georges, henri).
```

Définissez une règle `pere(Pere, Enfant)` qui est vrai si Pere est le père de Enfant. Faites de même avec la relation `mere(Mere, Enfant).`

Définissez une règle `fils(Fils, Parent)` qui est vrai si Fils est le fils du parent Parent. Faites de même avec la relation `fille(Fille, Parent).`

Définissez une règle

`frere(Frere, X)` qui est vrai si Frere est le frère de X.

`oncle(Oncle, X)` qui est vrai si Oncle est l'oncle de X.

`neveu(Neveu, X)` qui est vrai si Neveu est le neveu de X.

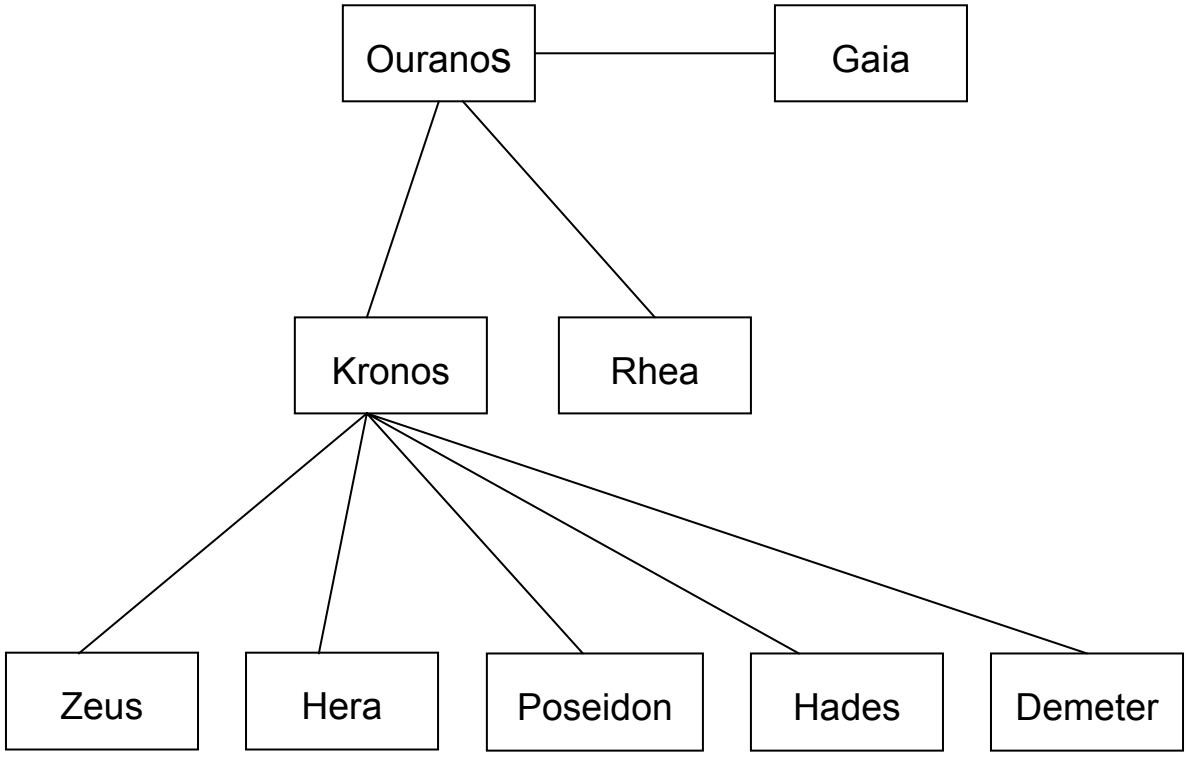
`grand-pere(GP, X)` qui est vrai si GP est le grand-père de X.

Les règles (suite)

- Regardez bien que l'on peut interroger une règle de plusieurs manières. Par exemple avec `frere(Frere,Personne) :`
  - savoir si Frere est le frère de Personne ;
  - trouver le (les) frère(s) de Personne ;
  - savoir si Frere a des frères ou sœurs ;
- Voir des exemples dans famille.pl ou dieux.pl

Les règles (les dieux grecs)

Voici l'arbre de filiation de la mythologie grecque.



Chapitre 2 : Les détails

- Une syntaxe très simple (notation BNF – *Backus-Naur Form* -) qui repose sur la notion de *terme* (servant de base pour décrire les données et les programmes).

< fait >	::=	< terme > .
< règle >	::=	< terme > :- < termes > .
< requête >	::=	< termes > .
< terme >	::=	< nombre > | < atome > | < variable >
		| < atome > (< termes >)
< termes >	::=	< terme > | < terme > , < termes >

avec

::= peut être
| ou

< >pour les symboles non terminaux

- Un terme est donc soit une constante (qui se subdivise en nombre ou atome), soit une variable soit une structure (ou terme composé).
- A l'aide des faits et des règles, on crée donc des programmes en Prolog qui est une séquence (donc l'ordre a de l'importance) de clauses. Une clause s'est :

Une version brève …

< fait >	::=	foncteur(arg1, arg2, …, argn) .
< règle >	::=	tête :- corps .
< requête >	::=	?-fait

- Il est assez simple de faire tourner en rond l'interpréteur Prolog. Par exemple avec :

```
p :- p.
?- p.
```

Un peu plus subtil ; en principe on peut appeler une règle de plusieurs façons (e.g, `frere(jean,V)` ou `frere(B,anne)`) mais il arrive que l'interpréteur réponde que sous une seule forme et que pour l'autre il tourne en rond.

Constantes, variables, structures

- Prolog possède des constantes qui se subdivisent en atomes et nombres.

  Les atomes comprennent les noms (`roi, jules, marie05, nil`), les symboles (comme « `:-` » ou « `+` ») et les chaînes de caractères (délimités par « ‘ »).

  Les nombres peuvent êtres des entiers (`34, -45, +12`) ou des réels (`-34.14, 0.987, -0.4e+2`).

- Les variables qui débutent par une majuscule (`Fils`, `X`, `Z67`, `U_02`) ou « _ » (qui correspondent à une inconnue anonyme ou muette comme `_, _X1`).

- Les structures (ou terme composé) permettent de regrouper des éléments reliés logiquement. Elles débutent par un atome (dénommé foncteur) puis entre parenthèses, les arguments séparés d’un virgule (« , »).

  Exemple :

```
parent(anne,hubert).
roi(louisXV, france, regne(1715, 1774), bourbon).
employe(nom,numero,date(jour,mois,an)).
point(x,y).
segment(point(x1,y1),point(x2,y2)).
triangle(point(x1,y1),point(x2,y2),point(x3,y3)).
Ouvrage (auteur(‘Adam Smith’, annee(1723, 1790)),
         economie,
         ouvrage(‘la Richesse des Nations', 1776),
         cote(yl8,2345)) .
```

  Le foncteur est le nom de la relation et l’arité le nombre d’arguments d’une relation.

Opérateurs

- Les opérateurs en Prolog sont des atomes (composés de symbole(s)) qui sont des foncteurs. On les rencontre essentiellement pour le calcul arithmétique. Prolog permet d'écrire les opérations de base comme :

```
+(3,4)
*(3,4)
+(4,(5,5))
```

Mais la notation préfixée n'est pas toujours la meilleure présentation et l'on préfère écrire : `3+4`, `3 * 4` ou `4 + 5 * 5` ou, pour certains opérateurs, soit en notation préfixée (-3, √n), soit en notation postfixée (n !).

En tout cas, l'emploi de parenthèses permet de préciser l'ordre de l'évaluation. Mais, en l'absence de celles-ci, une expression comme « 4 + 5 * 5 » possède deux évaluations possibles. Il faut donc préciser :

- la préséance des opérateurs ;
- l'associativité.

La préséance se définit habituellement par un ordre de priorité (avec 1 la plus forte priorité). Habituellement, les opérateurs binaires « * » et « / » ont une priorité plus forte que « + » et « - » (mais attention l'opérateur unitaire « + » et « - » aura une plus forte priorité que « * » et « / »).

Opérateurs (suite)

L'associativité permet de dire si l'expression « 16/4/2 » doit être évaluée comme « (16/4)/2 » (associatif à gauche) ou comme « 16/(4/2) » (associatif à droite).

En Prolog, les opérateurs peuvent être définis par le prédicat

```
op(ClasseDePriorité, Notation, Nom).
```

Dans laquelle, la notation est fx (préfixée), xfy (infixée) ou xf (postfixée) pour « f » l'opérateur, « x », « y » les arguments et « y » pour un argument de même ou de priorité supérieure (dont `ClasseDePriorité` plus petit), et « x » pour un argument de priorité supérieure. Les opérations arithmétiques se définissent ainsi :

```
op(500,yfx,'+').
op(500,yfx,'-').
op(400,yfx,'/').
op(400,yfx,'//').
op(400,yfx,'*').
op(400,yfx,'mod').        % nom pour un opérateur
op(200,fy,'-')
```

A vous d'écrire l'opérateurs de puissance (par exemple « ** » ou « ^ »).

```
op(200,xfx,'**').
op(200,xfy,'^').
```

Ainsi l'expression « 2^3^4 » (`xfy`, associativité à droite) sera évaluée comme « 2^(3^4) » tandis que l'expression « 16/4/2 » (`yfx`, associativité à gauche) sera évaluée comme « (16/4)/2 ».

L'unification (« = »)

- L'unification n'est pas un simple « *pattern matching* » (appariement exact de modèles) qui retourne vrai si deux termes sont égaux. L'unification rend égaux les deux termes. Et il faut bien le comprendre pour programmer en Prolog.

  Appel

  ```
  ?-  X = Y
  ```

  Dans ce cas, l'interprète utilise son algorithme d'unification pour rendre égaux X et Y. Trois cas sont possibles pour une unification réussie, à savoir :

  1. X et Y sont des constantes et elles sont égales ;
  2. X ou Y est (sont) une variable libre ; dans ce cas il y a succès et la variable libre est liée au terme représenté par la seconde variable (si les deux sont des variables libres, elles co-référencent alors sur le même objet) ;
  3. X et Y sont des termes composés. Il y a succès si les deux ont le même foncteur, le même nombre d'arguments et si les arguments (dans l'ordre) s'unifient l'un avec l'autre.

  Appel

  ```
  ?-  alpha = alpha
  ?-  23 = 24
  ?-  23 = alpha
  ?-  alpha = X
  ?-  N = 24
  ?-  N = M
  ?-  N = date(6,juin,1944).
  ?-  lettre(C) = mot(toto)
  ?-  syntagme(D,Nom,Adj) = syntagme(X,Y,Z)
  ?-  auteur(X, action) = auteur(smith, action)
  ?-  auteur(dijsktra, routing) = auteur(X, Y, A)
  ?-  foo(X, X) = foo(a, b)
  ?-  foo(X, a(b,c)) = foo(Z, a(Z,c))
  ?-  a(b,C,d(e,F,g(h,i,J))) = a(B,C,d(E,f,g(H,i,j)))
  ```

Arithmétique

- L'arithmétique en Prolog se base sur les opérateurs suivants :

```
X + Y
X - Y
X * Y
X ^ Y   % X a la puissance Y
X / Y
X // Y  % division entière
X mod Y % modulo (reste de la division)
```

- Que donne la question suivante :

```
?- 2 + 3 = 5
```

  Prolog est basé sur une évaluation paresseuse des expressions arithmétiques. Il faut utiliser l'opérateur `is` pour forcer l'évaluation numérique (avec l'absence de toute inconnue dans la partie droite du `is`) !

- Voici quelques exemples

```
X is 22/5
X = 4.4
X is 22//5
X = 4
X is 14 mod 3
X = 2
```

- Et rien ne vous empêche de chercher tous les nombres …

```
entier(0).
entier(N) :- entier(N1), N is N1+1.
```

Arithmétique (suite)

- Pour les comparaisons de *nombres* (mais pas de variables ou de termes composés), on peut utiliser les opérateurs suivants :

```
X =:= Y % vrai si X et Y représente le même nombre
X =\= Y % vrai si X et Y représente des nombres différents
X < Y   % vrai si X est strictement plus petit que Y
X > Y
X =< Y  % et non pas <=
X >= Y  % et non pas =>
```

- Avec un petit peu d'arithmétique, on peut donner les règles (foncteur `gcd`) pour trouver le plus grand diviseur commun (soit D) de deux nombres (soit X et Y), on peut utiliser l'algorithme d'Euclide qui correspond aux règles suivantes :

  1. si X = Y, alors D = X ;
  2. si X < Y, alors D est le `gcd` de X et de la différence Y – X ;
  3. si Y < X, alors utiliser la règle 2 en changeant X et Y.

  Il s'écrit de la manière suivante :

```
gcd(X,X,X).
gcd(X,Y,D) :- X < Y, Y1 is Y-X, gcd(X,Y1,D).
gcd(X,Y,D) :- Y < X, gcd(Y,X,D).
```

  Autre exemple, le nombre de Fibonacci :

```
fibo(1,1).
fibo(2,1).
fibo(N,F) :- N > 2, N1 is N-1, fibo(N1,F1),
                    N2 is N-2, fibo(N2,F2), F is F1+F2.
```

- Voir des exemples dans geographie.pl ou roi.pl.

Chapitre 3 : Structures

- La liste (séquence ordonnée d'éléments de longueur variable) est la principale structure en Prolog. Les éléments peuvent être des constantes (atomes ou nombres) ou des termes composés. Pour l'interpréteur Prolog, la liste peut être :
  - une liste vide notée `[]` ou `.()` ;
  - une liste composée d'un élément de tête et d'un reste comprenant la liste sans le premier élément ; cette idée se note `[T|R]`.

  Les éléments d'une liste sont séparés (comme les arguments) par une virgule « , ». Exemple de listes :

```
[a]     % ou .(a, []).
[a,b,c] % ou .(a, .(b, .(c, []))).
[la,souris,grise,trottine].
[jean,aime,[les,vins,canadiens]].
[la,souris,grise,trottine].
[jan,fev,mar,avr,mia,juin,juil,aout,sept,oct,nov,dec].
[tennis, ski, musique].
[bach, mozart, beethoven, beatles].
```

  Les deux notations sont possibles mais évidemment la notation avec des parenthèses droites est plus simple. Mais les formes suivantes sont aussi équivalentes :

```
[a,b,c] = [a|[b,c]] = [a,b|[c]] = [a,b,c|[]]
```

  On utilise aussi des modèles (*pattern*) pour imposer un certain type de listes, comme, par exemple :

```
[X].        % liste composée d'un élément
[X,Y].      % liste composée de deux éléments
[X,Y,Z].    % liste composée de trois éléments
```

  Les unifications suivantes sont possibles :

```
[X] = [tintin].
[X] = [123].
[X] = [roi(tintin,Belgique)].
[X] = [[la, souris, grise]].
```

Les listes

- Grâce à la notation `[T|R]`, on peut traiter des listes de longueur variable car on dit simplement qu'une liste est composée d'un élément en tête (ou d'une tête) et d'un reste (éventuellement vide) qui est la liste sans son premier élément.

Liste	Tête	Reste
[a,b,c]	a	[b,c]
[a]	a	[]
[a,b]	a	[b]
[[le,chat],mange,S]	[le,chat]	[mange,S]
[[X+Y],2+5]	[X+Y]	[2+5]
[le,chat,mange,[S]]	le	[chat,mange,[S]]
[]	échec	échec

- Les listes suivantes peuvent s'unifier si :

Liste1	Liste2	sous consitions
[a,b,c]	[X,Y,Z]	X=a, Y=b, Z=c
[tintin]	[T|R]	T=tintin, R =[]
[[le,chat],mange,S]	[T|R]	T=[le,chat], R =[mange,S]
[a,b,c]	[T1,T2|R]	T1=a, T2=b, R=[c]
[souris|R]	[souris,grise]	R=[grise]

- Quelques prédicats sont bien utiles sur les listes

  mais d'abord pour les amoureux de Scheme / Lisp :

```
car([X|_],X).      % extrait le 1er élément
cdr([T|R],R).      % la liste sans le 1er élément
cons(T,R,[T|R]).   % le constructeur de liste
```

Prédicats (récursifs) sur les listes

- Il est très utile de savoir si un élément apparaît (au moins une fois) dans une liste. Habituellement, ce prédicat se nomme `member` et correspond aux règles suivantes :

    1. c'est vérifié si la tête de liste est l'élément recherché ;
    2. si l'élément recherché est quelque part dans le reste de la liste.

  Il s'écrit de la manière suivante :

```
member(X,[X|_]).      % ou member(X,[Y|_]) :- X=Y.
member(X,[_|R]) :- member(X,R).
```

  Un élément appartient à une liste si c'est le premier élément (cas simple) ou bien s'il apparaît dans le reste (soit la liste sans le premier élément, via un appel récursif).

- Ecrivez le prédicat `islist(L)` qui retourne true si L est une liste. C'est vrai si :

    1. c'est une liste vide ;
    2. le reste est une liste, quelque soit le premier élément.

```
islist([]).         % une liste vide est une liste
islist([_|R]) :- islist(R). % ou islist([T|R]):-islist(R).
```

  Avez-vous le comportement attendu, par exemple avec `islist(X)` ? Mais on peut s'appuyer simplement sur l'unification.

- Ecrivez le prédicat `length(L,N)` qui retourne true si L est une liste composée de N éléments. Les règles sont les suivantes :

    1. la liste vide a une longueur de 0 ;
    2. la longueur est 1 de plus que la liste sans l'élément de tête.

```
length([],0).         % si la liste est vide, c'est 0
length([_|R],N) :- length(R,Nr), N is Nr + 1.
```

Prédicats sur les listes (suite)

- Ecrivez le prédicat `append(L1,L2,L)` qui retourne true si L est une liste composé des éléments de L1 puis des éléments de L2.

```
append([],L,L).        % si la 2e est vide, c'est simple
append([T|R1],L,[T|R2]) :- append(R1,L,R2).
```

  Comment est-ce possible que cela fonctionne ?

  Est-ce vraiment efficace ?

- Grâce à ce prédicat, on peut définir très simplement les deux prédicats suivants :

```
prefix(P,L) :- append(P,X,L).
suffix(S,L) :- append(X,S,L).
```

- Ecrivez un paquet de clauses (nommé `add`) permettant d'ajouter un élément en tête d'une liste.

  Par exemple :

```
add(X,L,[X|L]).
```

- Ecrivez un paquet de clauses (nommé `del`) permettant de supprimer un élément (une seule fois s'il apparaît plusieurs fois) d'une liste.

  Une solution :

```
del(X,[X|R],R).
del(X,[T|R1],[T|R2]) :-  del(X,R1,R2).
```

Prédicats sur les listes (suite)

- Ecrivez un paquet de clauses (nommé `permutation`) permettant de proposer les éléments d'une liste dans un autre ordre.

  Une solution :

```
permutation([],[]).
permutation(L, [T|R]) :- del(T,L,L1), permutation(L1,R).
```

- Autre exemple :

```
reverse(List,Reverse).

reverse([],[]).
reverse([T|R],L) :- reverse(R,L1), append(L1,[T],L).

%  palindrome(List).

palindrome([]).
palindrome([_]).
palindrome(L) :- append([T|R],[T],L), palindrome(R).
```

- Autre exemple avec des nombres :

```
%  sumList(List,Sum).

sumList([],0).
sumList([Tete|Reste],Sum) :- sumList(Reste,N), Sum is N+T.

%  maxList(List, Max).

maxList([X],X).
maxList([X,Y|Reste],Max) :- maxList([Y|Reste],MaxReste),
                            max(X,MaxReste,Max).

% ordered(List). si List est une liste ordonnée de nombres

ordered([X]).
ordered([X,Y|Reste]) :- X =< Y, ordered([Y|Reste]).
```

Mapping

- Parfois, on désire parcourir une structure pour en générer une nouvelle. Par exemple, on peut écrire une phrase en français et vouloir la traduire en anglais. Une phrase s'écrit comme une liste comme, par exemple :

```
[le, chat, court].
[le, chat, court, et, la, chatte, mange].
```

Voici notre dictionnaire bilingue :

```
translate(le, the).
translate(la, the).
translate(les, the).
translate(et, and).
translate(chat, cat).
translate(chatte, cat).
translate(chats, cats).
translate(chattes, cats).
translate(court, run).
translate(souris, mouse).
translate(souris, mice).
translate(courrent, run).
translate(mange, eats).
translate(mangent, eat).
translate(Other, Other).  % pour les autres cas
```

Et le mécanisme de traduction :

```
transSentence([], []).
transSentence([T|R], [X|Y]) :- translate(T,X),
                               transSentence (R,Y).
```

Et un exemple :

```
transSentence([le,chat,mange,la,souris], E).
E = [the, cat, eats, the, mouse] ;
E = [the, cat, eats, the, mice]
Yes
```

Faites de même avec :

```
enToutes lettres([1,5,7], Ex).
Ex = [un, cinq, sept] ;
Yes
```

Arbres

- Les arbres syntaxiques (par exemple).

  On peut définir un lexique (`prédicat determinant(), adjectif(), nom(), verbe()`) et une morphologie minimale (`genre(), nombre()`). On y ajoute un brin de syntaxe avec le prédicat `accordGenre()`.

```
determinant(le).
determinant(la).

adjectif(verte).
adjectif(blanc).
adjectif(blanche).

nom(chat).
nom(souris).

verbe(mange).
verbe(trotine).

genre(le,masculin).
genre(la,feminin).
genre(rouge,_).
genre(blanc,masculin).
genre(blanche,feminin).
genre(chat,masculin).
genre(souris,feminin).

accord(X,Y) :- genre(X,G), genre(Y,G).
```

  Et quelques exemples :

```
?- nom(N).

N = chat;
N = souris ;
No

?- nom(chien).

No

?- genre(souris,G).

G = feminin ;
No

?- genre(Mot,masculin).

Mot = le;
Mot = rouge;
Mot = chat;
No
```

Arbres (suite)

```
?- nom(chien).
No
?- accord(la,souris).
Yes
?- genre(Nom,feminin), nom(Nom).
Nom = souris ;
No
```

- On peut alors définir des éléments (sous-arbres syntaxiques) en définissant le syntagme nominal (`sn()`), le syntagme verbal (`sv()`) et la phrase (`p()`).

```
sn(D,N) :- determinant(D), nom(N), accord(D,N).
sn(N,A) :- nom(N), adjectif(A), accord(N,A).
sn(D,N,A) :- determinant(D), nom(N), adjectif(A),
             accord(D,N), accord(N,A).
p(snm(det(D),nom(N),G),D,N) :- sn(D,N), accord(D,G).
p(snm(nom(N),adj(A),G),N,A) :- sn(N,A), adjectif(A),
                               accord(N,A).
p(snm(det(D),nom(N),adj(A),G),D,N,A) :- sn(D,N,A),
                     determinant(D), adjectif(A),
                     accord(N,A).
```

On vérifie le syntagme nominal.

```
?- sn(M1,M2).
M1 = le
M2 = chat ;
M1 = la
M2 = souris ;
M1 = chat
M2 = blanc ;
M1 = souris
M2 = blanche ;
No
```

Arbres (suite)

```
?- P(S,le,chat).
S = snm(det(le), nom(chat), masculin) ;
No
?- P(S,N,rouge).
P = snm(nom(chat), adj(rouge), masculin)
N = chat ;
P = snm(nom(souris), adj(rouge), feminin)
N = souris ;
No
?- P(S,D,N,A).
S = snm(det(le), nom(chat), adj(rouge), masculin)
D = le
N = chat
A = rouge ;
S = snm(det(le), nom(chat), adj(blanc), masculin)
D = le
N = chat
A = blanc ;
No
```

On ajoute le verbe et la structure que l'on veut est un arbre plus complexe.

```
?- p(PH,le,chat,blanc,mange).
PH = p(s(snm(det(le), nom(chat), adj(blanc), masculin)),
v(verb(mange))) ;
No
```

Pensez-vous que cette approche soit vraiment la bonne ?

Et les règles à ajouter …

```
% Relation pour une structure de phrase
p(p(s(snm(det(D),nom(N),G)),v(verb(V))) ,D,N,V) :-
     sn(D,N), determinant(D), accord(D,N), genre(N,G),
     verbe(V).
p(p(s(snm(nom(N),adj(A),G)),v(verb(V))) ,N,A,V) :-
     sn(N,A), adjectif(A), accord(N,A), genre(N,G),
     verbe(V).
p(p(s(snm(det(D),nom(N),adj(A),G)),v(verb(V))),D,N,A,V) :-
     sn(D,N,A), determinant(D), adjectif(A), accord(N,A),
     genre(N,G), verbe(V).
```

Exemples de graphe

- Les graphes : l'exemple d'une compagnie aérienne.
  Le prédicat `vol(N,Ville_1,Ville_2,T,D)` est vrai si le numéro du vol `N` relie `ville_1` à `ville_2` dans un temps de T heures et pour une distance `D`.

  Actuellement la base de connaissances est la suivante :

```
vol(1,toronto,chicago,1,550).
vol(2,montreal,paris,8,4500).
vol(3,londres,paris,1,500).
vol(4,montreal,new_york,1,500).
vol(5,chicago,new_york,2,700).
vol(6,new_york,tampa,4,1200).
vol(7,tampa,cancun,4,1100).
vol(8,paris,rome,2,800).
vol(9,toronto,londres,9,5300).
vol(10,rome,new_york,8,4800).
vol(11,toronto,montreal,1,500).
```

  Le prédicat `relie(V1,V2)` est vrai s'il existe un vol direct entre les deux villes, ou bien si l'on peut joindre (avec une ou plusieurs villes intermédiaires) les villes `V1` et `V2`.

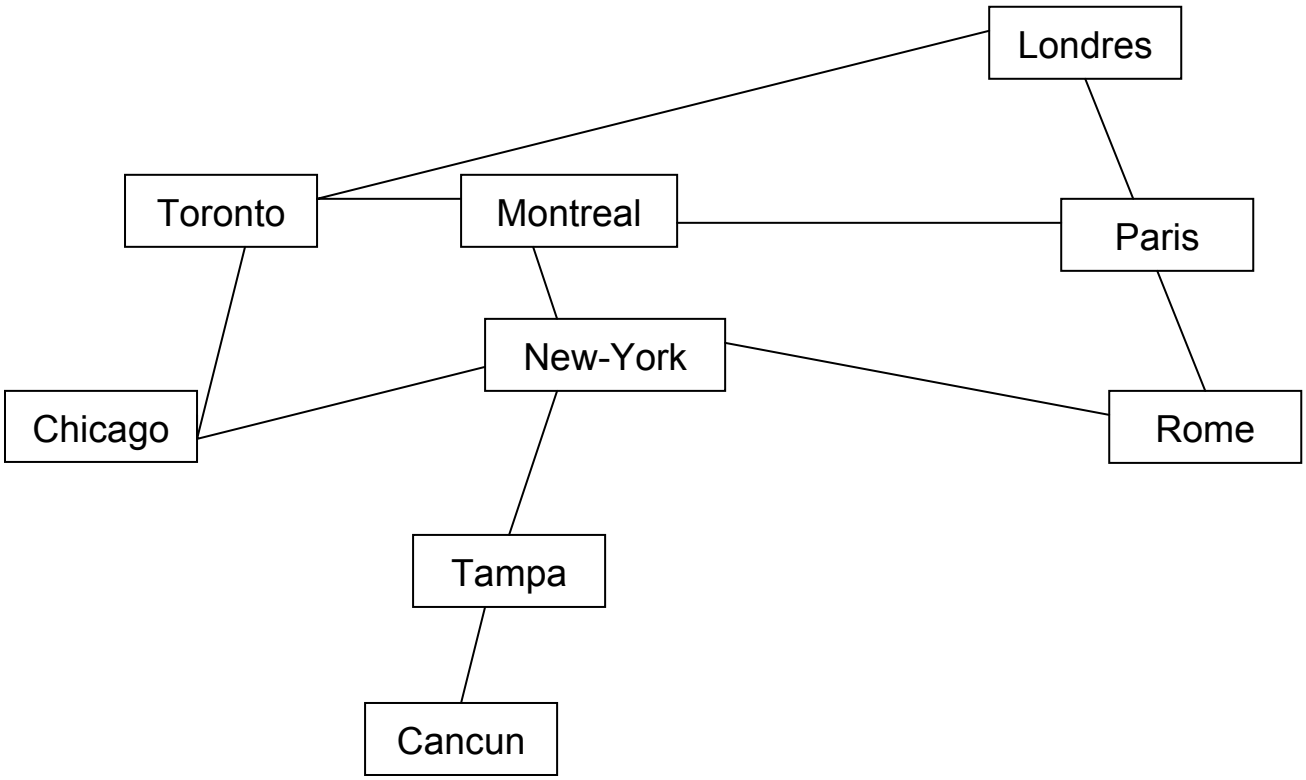
```
relie(V1,V2) :- vol(_,V1,V2,_,_).
relie(V1,V2) :- vol(_,V2,V1,_,_).
relie(V1,V2) :- relie(V1,V3), (vol(_,V3,V2,_,_);
                               vol(_,V2,V3,_,_))
```

```
?- relie(V,montreal).
```

```
V = toronto ;
V = paris ;
V = new_york ;
V = montreal ;
Yes
```

  Attention, la logique peut vous faire faire le tour de la planète plusieurs fois pour trouver un autre chemin de `V1` vers `V2` !

Vue graphique de la compagnie aérienne



Exemples de graphe (suite)

Le prédicat `temps_voyage(V1,V2,T)` est vrai s'il existe un vol (direct ou non) entre les deux villes `V1` et `V2` avec un temps de vol `T`.

```
temps_voyage(V1,V2,Temps) :- vol(_,V1,V2,Temps,_).
temps_voyage(V1,V2,Temps) :- vol(_,V2,V1,Temps,_).
temps_voyage(V1,V2,Temps) :- temps_voyage(V1,V3,T1),
                             (vol(_,V3,V2,T2,_);
                              vol(_,V2,V3,T2,_)),
                             Temps is T1 + T2.
```

```
?- temps_voyage(paris,montreal,T).
```

```
T = 8 ;
T = 11 ;
T = 12 ;
T = 24 ;
T = 10 ;
Yes
```

Le prédicat `distance_voyage(V1,V2,D)` est vrai s'il existe un vol (direct ou non) entre les deux villes `V1` et `V2` avec une distance de vol `D`.

```
distance_voyage(V1,V2,D) :- vol(_,V1,V2,_,D).
distance_voyage(V1,V2,D) :- vol(_,V2,V1,_,D).
distance_voyage(V1,V2,D) :- distance_voyage(V1,V3,D1),
                            (vol(_,V3,V2,_,D2);
                             vol(_,V2,V3,_,D2)),
                            D is D1 + D2.
```

```
?- distance_voyage(paris,montreal,D).
```

```
D = 4500 ;
D = 6100 ;
D = 6100 ;
D = 13500 ;
D = 5500 ;
Yes
```

A vous d'améliorer ces prédicats pour éviter de faire des aller-retour (en utilisant une liste pour mémoriser les villes intermédiaires, par exemple).

Chapitre 4 : Techniques de programmation

- Pour bien programmer en Prolog, un modèle à suivre est le « *generate and test* » c'est-à-dire que l'on génère une solution candidate que l'on teste ensuite pour vérifier que cette solution candidate soit acceptable. Le modèle est le suivant :

```
conclusion(S) :- generer(S), verifier(S).
```

- Une bonne programmation (en Prolog) cherche à atteindre les buts suivants :

  1. correct
  2. convivilal
  3. efficace
  4. lisible
  5. facile à modifier
  6. robuste
  7. documenté

- Une stratégie d'analyser et de programmation repose sur le *stepwise refinement*, une approche *top-down*.

  Pour vous aider, on peut imaginer une solution basée sur

  - la récursivité avec en premier le/les cas simple(s) puis l'appel récursif sur une taille plus faible
  - généraliser le problème posé et cette généralisation est parfois plus simple à implanter.

- Quelques idées pour écrire vos paquets de clauses.

  - faites les courtes ;
  - utiliser des noms parlants (ou mémotechniques) ;
  - la mise en page (*pretty printing*) s'avère utile ;
  - utiliser l'opérateur coupe-choix (« ! ») avec circonspection ;
  - l'emploi du « ; » doit être limité ;
  - les procédures `assert()` et `retract()` sont à limiter au maximum ;
  - introduisez des commentaires.

Travailler avec un accumulateur

- Nous avons proposé un prédicat `length(L,N)` qui retourne true si L est une liste composé de N éléments avec les règles suivantes :

```
length([],0).
length([_|R],N) :- length(R,Nr), N is Nr + 1.
```

Cette solution s'appuie sur un appel récursif jusqu'à rencontrer la liste vide. Ensuite on dépile en ajoutant un à la variable N.

Au lieu d'aller vers le cas simple (ici, la liste vide) sans accumuler d'information, on peut tenir compte de ce dépillement pour accumuler le résultat (ici un entier via la variable A).

```
listlen(L,N) :- listlenacc(L,0,N).
listlenacc([],A,A).
listlenacc([T|R],A,N) :- A1 is A + 1, listlenacc(R,A1,N).


?- listlen([a,b,c,d],N).

listlenacc([a,b,c,d],0,N).
listlenacc([b,c,d],1,N).
listlenacc([c,d],2,N).
listlenacc([d],3,N).
listlenacc([],4,N).
```

Automate

- En informatique, les automates sont importants et on les utilise sous plusieurs formes. Dans le cas présent, on désire savoir si une chaîne présentée est correcte (respecte la petite grammaire que l'on a définit) ou non.

  Dans un tel cas (automate non-déterministe fini) on modélise un état initial et un état final. On peut passer d'un état vers un autre selon les liens qui relient ces états et sur chaque lien on peut imposer la présence (consommation) d'un élément (un caractère dans le cas présent).

  Si, en partant de l'état initial avec une chaîne (de caractères dans le cas présent), on arrive à trouver un chemin pour aboutir à l'état final avec une chaîne vide, alors la chaîne est correcte (ou respecte les conditions de notre automate).

- Imaginons un automate qui reconnaît les nombres réels introduits par l'usager (par exemple, « 345 », « -12 », « +3.15 »).

  Graphiquement, on a

QuickTime™ and a
TIFF (Uncompressed) decompressor
are needed to see this picture.

Automate

Et en Prolog,

```
final(s3).

trans(s0,'+',s1).      % le signe + ou - devant le nombre
trans(s0,'-',s1).
trans(s1,N, s1) :- digit(N).
trans(s1,'.',s2).     % le point décimal
trans(s2, N, s2) :- digit(N).

silent(s0,s1).    % +- optionel
silent(s1,s3).    % partie décimale optionel
silent(s2,s3).    % +- optionel

digit(0).
digit(1).
digit(2).
digit(3).
digit(4).
digit(5).
digit(6).
digit(7).
digit(8).
digit(9).
```

Il reste à définir en Prolog le mécanisme de notre automate (un mécanisme général qui fonctionnera pour tout automate).

Automate (suite)

- Le mécanisme général d'un automate est le suivant :

```
accept(State,[])    :- final(State).
accept(State,[H|T]):- trans(State,H,StateF),
                         accept(StateF,T).
accept(State,L)     :- silent(State,StateF),
                         accept(StateF,L).
```

Application. En partant de l'état S0, et avec la chaîne « 123.45 », est-ce que j'arrive à l'état final (S3) et donc la chaîne respecte la syntaxe.

```
?- accept(s0,[1,2,3,'.',4,5]).
Yes
```

Quelles sont les chaînes acceptables composées de 3 éléments ?

```
?- accept(s0,[A,B,C]).
A = +, B = 0, C = 0 ;
A = +, B = 0, C = 1 ;
…
A = +, B = ., C = 1 ;
…
A = +, B = ., C = 9 ;
…
A = -, B = 0, C = 0 ;
A = -, B = 0, C = 1 ;
…
A = 0, B = 0, C = 0 ;
A = 0, B = 0, C = 1 ;
```

Logique floue (*fuzzy logic*)

- Prolog se base sur l'hypothèse du monde clos (*closed world assumption*) selon laquelle tous les faits qui sont connus (ou dérivables) sont inclus dans les clauses. Selon cette hypothèse, si l'interpréteur échoue, il annonce « faux ».
- Rien ne nous empêche d'utiliser Prolog pour étendre la logique vers une approche de la logique non classique comme, par exemple la logique floue ou une logique à trois valeurs. Dans le cas de la logique floue, une proposition n'est pas vraie ou fausse mais on lui associe une valeur qui varie entre 0 (faux) et 1 (vrai). Voici quelques exemples :

```
celebre(napoleon,1).
celebre(godel,1).
celebre(turing,1).
celebre(bush,0.5).
celebre(jean,0.6).
celebre(van_gogh,1).

riche(bush,1).
riche(napoleon,0.6).
riche(turing,0.1).
riche(jean,0.3).
riche(van_gogh,0).
```

La modélisation gagne en précision et on peut combiner ces affirmations avec les opérateurs « et », « ou » et « non », par exemple pour répondre à la question « Est-ce que Turing fait partie des gens riches et célèbres ? ».

Opérateur	Degré d'appartenance
A **et** B	min $\{D_a, D_b\}$
A **ou** B	max $\{D_a, D_b\}$
non A	1- D_a

```
celebre(napoleon,1) , riche(napoleon,0.6) -> 0.6.
celebre(van_gogh,1) , riche(van_gogh,0) -> 0.0

celebre(jean,0.6) ; riche(jean,0.3) -> 0.6.


not(riche(turing,0.3)) -> 0.7.
```

Logique floue (suite)

- Une manière de faire fonctionner Prolog avec une autre logique est de faire ceci :

```
?- question([aime(jean,marie), aime(marie,jean)], Valeur).
```

Ou de manière générale :

```
?- question([ButA, ButB], Valeur).

question([ButA, ButB],Verite) :- demontrer(ButA, Va),
                                 demontrer(ButB, Vb),
                                 combin(Va, Vb, Verite),
                                 write(Verite).

combin(Va, Vb, Vb) :- Va => Vb, !.
combin(Va, Vb, Va).

demontrer(ButA, Va) :- ButA,
                       ButA =..L,
                       renverse(L),
                       valeur(L, Va).
demontrer(ButA, 0).  % si ButA échoue

notFuzzy(P, V) :- P, !, P = ..L, renverse(L, L1),
                  valeur(L1, Vf), V is 1-Vf.
notFuzzy(P, 0).        % je ne sais pas

renverse(L, R) :- renverse(L,[],R).
renverse([H|T], L1, L) :- renverse(T, [H|L1], L).
renverse([], L, L).

valeur([Tete|_], Tete).
```

Logique floue (suite)

Avec la base de connaissances :

```
aime(jean, marie, 0.8).
aime(paul, marie, 0.9).
aime(marie, paul, 0.7).
aime(marie, jean, 0.2).
aime(jean, biere, 0.3).
aime(marie, biere, 0).
```

```
?- aime(jean, marie, V).
```

```
V = 0.8 ;
No
```

```
demontrer(aime(jean,marie,V), Vf).
```

```
V = 0.8, Vf = 0.8 ;
No
```

```
?- question([aime(jean,marie,Va),aime(marie,jean,Vb)], V).
```

```
Va = 0.8, Vb = 0.2, V = 0.2 ;
No
```

```
notFuzzy(aime(jean,marie,Va), V).
```

```
Va = 0.8, V = 0.2 ;
No
```

Logique à trois valeurs

- On peut imaginer une autre logique dans laquelle une proposition peut être fausse 0, vraie 1 ou inconnue (0.5). Chaque fait aura alors une de ces trois valeurs et il faut définir la valeur de vérité pour les opérateurs et, ou et non.

fait A	fait B	A et B	A ou B
		min	max
1	1	1	1
1	0	0	1
1	0.5	0.5	1
0	1	0	1
0	0	0	0
0	0.5	0	0.5
0.5	1	0.5	1
0.5	0	0	0.5
0.5	0.5	0.5	0.5

fait A	not A
1	0
0	1
0.5	0.5

Pour l'opérateur `not`, on peut écrire :

```
not3val(P,0) :- P, P=..L, renverse(L,L1), valeur(L1,1), !.
not3val(P,1) :- P, P=..L, renverse(L,L1), valeur(L1,0), !.
not3val(P,0.5).  % je ne sais pas
```

Le problème des reines

- Le problème de placer huit reines sur un échiquier sans qu'elles s'attaquent est un problème classique en IA. L'espace des solutions à considérer est relativement important, soit $64 \cdot 63 \cdot 62 \cdot \ldots \cdot 57 = 1{,}7 \cdot 10^{14}$ possibilités.

  Mais une astuce pour réduire immédiatement cet espace de recherche est prendre en compte le fait que chacune des huit reines doivent être placés sur une colonne différentes.

- Le principal problème à résoudre est le choix de la représentation de l'échiquier et des reines. Nous avons opté pour une paire d'entiers `X/Y` pour donner les coordonnées de l'abscisse et ordonnée sur l'échiquier. Une solution (possible ou non) se représente par une liste de huit paires d'entiers comme, par exemple `[1/4,2/2,3/7,4/3,5/6,6/8,7/5,8/1]` et une autre est `[1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]`.

  Pour contrôler si une solution est acceptable, il ne doit pas y avoir d'attaque possible entre les huit reines.

- On peut résoudre ce problème en tenant compte de l'argumentation suivante. On considère la position d'une reine et on regarde si elle menace les autres. Dans ce cas, on a les règles suivantes à contrôler dans le prédicat `solution(Sol)`.

  Si la liste des autres positions de reines est vide, il n'y a pas de menace et c'est une solution acceptable.

  Si la liste des autres reines n'est pas vide (situation `[X/Y|Others]`), alors on a une solution acceptable si :
  - il n'y a pas d'attaques dans les positions `Others`;
  - `X` et `y` sont des entiers entre 1 et 8 ;
  - la reine en position `X/Y` ne menace aucunes des reines dans les positions `Others` que l'on contrôle via le prédicat `noattack`.

  Mais comment savoir si deux positions (`X/Y`) et (`X1/Y1`) sont sur la même diagonale ou non ?

Le problème des reines (suite)

Le prédicat `solution(Sol)` peut être écrit de manière récursive et donc on peut admettre que Others contient une solution partielle correcte.

Pour que les valeurs `X` et `Y` soient comprises entre 1 et 8, on peut générer les X (en imposant une solution de départ avec une reine dans chaque colonne) et imposer que `Y` soit membre de la liste `[1,2,3,4,5,6,7,8]`.

- Pour le prédicat `noattack(X/Y,Others)` doit respecter les règles suivantes :

  Si la liste `Others` est vide, la position `X/Y` est toujours acceptable.

  Sinon, on contrôle avec la première position de `Others` avec la position `X/Y`. Il y a menace si les deux positions sont sur la même ligne (même `Y`), ou sur la même colonne (même `X`) ou la même diagonale.

Le problème des reines (suite)

Le prédicat `solution([X/Y|Others])` possède la position de la première reine en `X/Y` et les autres reines dans `Others`.

```
solution([]).
solution([X/Y|Others]) :- solution(Others),
   member(Y, [1,2,3,4,5,6,7,8]),
   noattack(X/Y, Others).
```

Le prédicat `noattack(Pos,ListePos)` retourne vrai s'il n'y a pas de conflits.

```
noattack(_,[]).           % Si vide, OK pas de menace
noattack(X/Y,[X1/Y1|Others]) :-
  Y =\= Y1,               % Different Y-coordinates
  Y1-Y =\= X1-X,          % Different diagonals
  Y1-Y =\= X-X1,
 noattack(X/Y, Others).
```

Pour faciliter l'appel, on crée le prédicat `solve(Sol)` et on donne une solution (générée par `template(Sol)`) au prédicat `solution(Sol)`.

```
solve(S) :- template(S), solution(S).

template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

L'appel se réalise de la manière suivante :

```
?- solve(S).

S = [1/4, 2/2, 3/7, 4/3, 5/6, 6/8, 7/5, 8/1] ;
S = [1/5, 2/2, 3/4, 4/7, 5/3, 6/8, 7/6, 8/1] ;
S = [1/3, 2/5, 3/2, 4/8, 5/6, 6/4, 7/7, 8/1] ;
S = [1/3, 2/6, 3/4, 4/2, 5/8, 6/5, 7/7, 8/1] ;
S = [1/5, 2/7, 3/1, 4/3, 5/8, 6/6, 7/4, 8/2]
Yes
```

Le *cut* (« ! »)

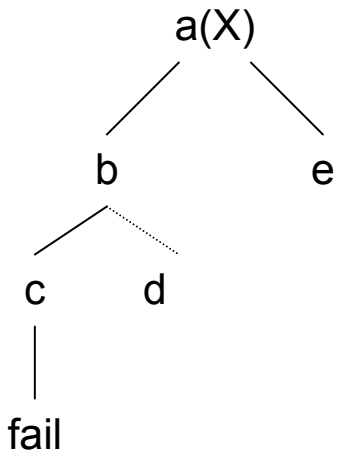
- L'opérateur *cut* (« ! ») empêche le retour arrière (*backtracking*) et on le nomme également le coupe-choix. Dès que l'interpréteur Prolog le rencontre, il y a une garantie de succès (le *cut* réussit toujours) mais il bloque les choix faits sur la règle. Voyons un exemple.

  Pour démontrer `a(N)`, j'ai deux choix (`N=1` avec `b`, `N=2` avec `e`), comme pour démontrer `b`. Pour `d` et `e`, j'ai une seule démonstration et `c` est toujours faux.

```
a(1) :- b.
a(2) :- e.
b :- c.
b :- d.
c :- fail.
d.
e.
```

```
?- a(N).
```

```
N = 1 ;
N = 2 ;
No.
```



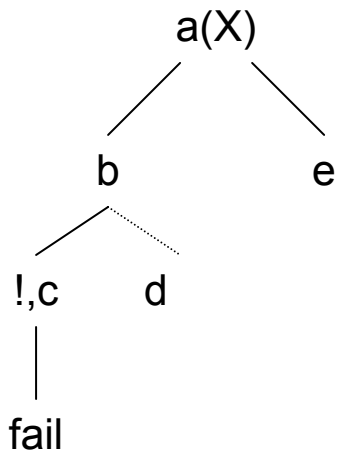
Le *cut* (« ! ») (suite)

- Avec un *cut*, la deuxième démonstration de `b` (avec `d`) ne sera jamais utilisée.

```
a(1) :- b.
a(2) :- e.
b :- !, c.
b :- d.
c :- fail.
d.
e.
```

```
?- a(N).
```

```
N = 2 ;
No.
```



Le *cut* (suite)

- L'opérateur *cut* (« ! ») s'utilise essentiellement dans trois cas :
  - bloquer le choix d'une règle ;
  - le *cut and fail* ;
  - générer et tester
- Prenez le prédicat `max(N,M,Max)` qui est vérifié si `Max` est le nombre maximum entre `N` et `M`. Une écriture possible est :

```
max(N,M,N) :- N >= M.
max(N,M,M) :- M > N.
```

  Mais une *solution plus efficace* est la suivante car dès que le premier but de la première règle est vérifiée, il n'y a plus d'autres solutions.

```
max(N,M,N) :- N >= M, !.
max(N,M,M).
```

  Le modèle de cet usage est le suivant. Le but `B` peut être démontré si `X` est vrai (règle R1) ou, en alternative, si `X` est faux (règle R2).

```
/* R1 */   B :- X , Z                B :- X , ! , Z
/* R2 */   B :- \+X , Y.             B :- Y.
```

- Autre exemple. Nous disposons du prédicat `name()` qui retourne les codes ASCII d'une constante. Par exemple :

```
?- name(aime,L).
L = [97, 105, 109, 101] ;
No
```

Le *cut* (suite)

On peut l'utiliser pour former une liste de lettres sur la base des codes ASCII de la manière suivante : D'abord l'explosion et l'implosion des noms.

```
nom(X,L) :- var(L),!, name(X,Codes), nom1(Codes,L).
nom(X,L) :- var(X), !, nom1(Codes,L), name(X,Codes).

nom1([],[]).
nom1([X|Xs],[X1|X1s]):- name(X1,[X]), nom1(Xs,X1s).

?- nom(aime,L).

L = [a, i, m, e] ;
No

?- nom(X,[a,i,m,e,r]).

X = aimer ;
No
```

Dans le premier cas, on sait que L est une variable, alors on empêche le retour arrière (cette règle s'avère le bon choix, il n'y en a pas d'autre). Dans le second cas (implosion), on sait que l'on a une séquence de lettres. On bloque le choix sur cette règle.

- Parfois on désire seulement *une seule solution*. En reprenant le prédicat `member()`, on constate aisément qu'il s'arrête à toutes les occurrences du terme X.

```
member(X,[X|_]).
member(X,[_|R]) :- member(X,R).
```

Et avec notre nouvelle version, on s'arrête dès qu'une solution a été trouvée.

```
member(X,[X|_]) :- !.
member(X,[_|R]) :- member(X,R).
```

Le *cut* (suite)

- Le deuxième cas où l'usage du *cut* s'avère opportun est l'introduction de la *négation* et sa fameuse séquence « *cut and fail* » ou « `!, fail` ».

  Par exemple, on sait que Heidi aime tous les animaux. On écrit :

```
aime(heidi,A) :- animal(A).
```

  mais il faudrait exclure les serpents …

```
aime(heidi,A) :- serpent(A), !, fail.
aime(heidi,A) :- animal(A).
```

  Deux autres exemples de l'emploi judicieux du *cut*.

  Ecrivez un prédicat `different(X,Y)` qui est vérifié si `X` est différent de `Y`.

```
different(X,X) :- !, fail.
different(X,Y).
```

  Ecrivez un prédicat `not(P)` qui est vérifié si `P` n'est pas vrai.

```
not(P) :- P, !, fail.  % si P réussi, stop et échec
not(_).                % si P échoue, alors not() réussit.
```

  Ce prédicat est disponible et il se note `\+(P)`.

Le *cut* (suite)

- Dans le troisième cas, il s'agit de bloquer la génération des solutions (souvent pour des raisons d'efficacité). Le modèle général est le suivant :

```
conclusion(S) :- generer(S), !, verifier(S).
```

Problème avec le *cut*

- Mais l'opérateur *cut* pose parfois des problèmes et il faut bien réfléchir avant de l'utiliser. Voici un exemple.

```
nombre_parents(adam,0) :- !.
nombre_parents(eve,0) :- !.
nombre_parents(Autre,2).
```

Et si on l'utilise :

```
?- nombre_parents(adam,N).
N=0 ;
No
?- nombre_parents(tintin,N).
N=2 ;
No
```

Cela semble parfait mais …

```
?- nombre_parents(eve,2).
Yes
```

Pourquoi ?

Une version corrigée :

```
nombre_parents(adam,N) :- !, N=0.
nombre_parents(eve,N) :- !, N=0.
nombre_parents(Autre,2).
```

Alternative :

```
nombre_parents(adam,0).
nombre_parents(eve,0).
nombre_parents(Autre,2) :- \+(Autre=adam), \+(Autre=eve).
```

Mais si …

```
?- nombre_parents(Personne,N).
```

Les prédicats prédéfinis utiles

- Contrôle de la base de connaissances :

```
consult(File)       %
asserta(Clause)     % ajoute au début la Clause
assertz(Clause)     % ajoute à la fin la Clause
retract(Clause)     % retire une clause

listing(Foncteur)   % pour voir les clauses du foncteur
```

- Contrôle le succès ou l'échec :

```
true    % succès assuré
fail    % échoue toujours, e.g., !, fail.
```

Exemple d'emploi :

```
?- question(X), write(X), nl, fail.
```

- Types de termes :

```
atom(X)     % vrai si X est une chaîne ou un nom
number(X)   %
atomic(X)   % atomic(2) est true
var(X)
nonvar(X)   % e.g nonvar(_) est toujours true
```

- Pour vous aider dans le déverminage (*debugging*) de vos programmes :

```
trace
notrace
spy P
nospy
debugging
nodebug
```

Les prédicats prédéfinis utiles (suites)

- Autres prédicats divers :

```
call(Goal)          %  prouve Goal
functor(T,F,N)      %  functor(foo(a,b),F,N)  F=foo, N=2
X =.. L             %  foo(a,b,c)=..L   (L=[foo,a,b,c])
arg(N,T,A)          %  arg(2,foo(a,b,c),X)   X=b
bagof(X, P, L)      %  trouve toutes les réponses
setof(X, P, L)      %  trouve toutes les réponses
findall(X, P, L)    %  trouve toutes les réponses
```

Que l'on peut utiliser dans le prédicat `maplist(P,Lin,Lout)` qui applique le prédicat `P` sur chaque élément de la liste `Lin` pour former `Lout`.

```
maplist(_,[],[]).
maplist(P,[T|R],[T1|R1]) :- Q=..[P,T,T1], call(Q),
                            maplist(P,R,R1)
```

Une simplification de `maplist` car `applist(P,Liste)` admet un prédicat P unaire qu'il va appliquer à tous les éléments de la liste.

```
applist(_,[]).
applist(P,[Tete|Reste) :- Q=..[P,Tete], call(Q),
                          applist(P,Reste).
```

Le prédicat `bagof(X,P,L)` donne dans `L` tous les objets `X` pour lesquelles `P` est vérifié.

```
age(pierre,7).
age(anne,5).
age(patricia,8).
age(tom,5).

?- bagof(C,age(C,5),L).
L = [anne, tom]
```

Le prédicat `setof(X,P,L)` fonctionne de la même manière mais la liste est trié par ordre croissant (opérateur `@<`) et sans duplicata.

Le prédicat `findall(X,P,L)` fonctionne de la même manière que `bagof()` mais tous les objets `X` sont collectés sans tenir compte des différentes solutions possibles à `P` (souvent très similaire à `bagof()`)..

Entrée - sortie

- Quelques fonctions utiles pour les entrées – sorties en Prolog. D'abord l'explosion et l'implosion des noms.

```
nom(X,Xl) :- var(Xl),!, name(X,Codes), nom1(Codes,Xl).
nom(X,Xl) :- var(X), !, nom1(Codes,Xl), name(X,Codes).

nom1([],[]).
nom1([X|Xs],[X1|X1s]):- name(X1,[X]), nom1(Xs,X1s).
```

```
?- nom(aime,L).

L = [a, i, m, e] ;

No

?- nom(X,[a,i,m,e,r]).

X = aimer ;

No
```

Utile car le prédicat `name()` travaille avec les codes Ascii.

```
?- name(aime,L).

L = [97, 105, 109, 101] ;

No
```

Ensuite la classification des caractères lus.

Le prédicat `carDansMot()` est vérifié si le caractère est une lettre ou « ' ».

```
carDansMot(C):- a @=< C, C @=< z.
carDansMot(C):- 'A' @=< C,C @=< 'Z'.
carDansMot('''').
```

Le prédicat `carFinMot()` est vérifié si le caractère marque la fin d'une phrase (et on ignore le newline).

```
carFinMot('.') :- skip(10).
carFinMot('?') :- skip(10).
carFinMot(X)   :- name(X,[10]).
```

Entrée - sortie

Ensuite, on lit un caractère que l'on ajoute au terme :

```
lireCar(X) :- get_char(Car), name(X,[Car]).
```

Enfin, on lit une séquence de caractères (phrase) que l'on ajoute au terme :

```
% transformation de la ligne lue en une liste d'atomes
lirePhrase(L) :- lireCar(C), lirePhrase1(C,L).
% ajoute un mot
lirePhrase1(C,[W|Ws]) :- carDansMot (C), !, lireMot
(C,W,C1), lirePhrase1(C1,Ws).

% fin de phrase
lirePhrase1(C,[]) :- carFinMot(C), !.

% sinon on oublie ce caractère
lirePhrase1(C,Ws):- lireCar(C1), lirePhrase1(C1,Ws).


% construit un mot
lireMot(C,W,C1):- carsDansMot(C,Cs,C1), nom(W,Cs).


carsDansMot(C,[C|Cs],C0) :- carDansMot(C), !,
                    lireCar(C1), carsDansMot(C1,Cs,C0).
carsDansMot(C,[],C) :- not(carDansMot(C)).
```

On l'utilise ainsi :

```
?- lirePhrase(Ph).
|: le chat mange la souris.
Ph = [le, chat, mange, la, souris] ;
No
```

De même :

```
?- ecrirePhrase([le,chat,mange]).
le chat mange
Yes
```

Avec la définition suivante :

```
ecrirePhrase([H|T]) :-write(H),write(' ') ecrirePhrase(T).
ecrirePhrase([]) :- nl.
```

Conclusion

- La programmation logique c'est (Kowalski, 1979)

  Algorithm = logic + control

  le contrôle est le mécanisme sous-jacent de l'interpréteur Prolog et la logique est introduite par le programmeur. Ce dernier écrit :

  `P :- C1 , C2 , … Cn`

  pour indiquer

  pour déduire P, il faut déduire C1 et C2 et … et Cn, avec n >= 0.

  On peut indiquer des faits par

  `P.`

  Et si l'on abandonne les arguments, on retombe sur le calcul des prédicats.

  Le contrôle est donné par l'ordre d'évaluation des buts (l'interpréteur débute par le sous-but le plus à gauche) et l'ordre des règles (prendre la première au début).

- Prolog et logique

  Prolog est basé sur la logique des prédicats du 1er ordre (vous ne pouvez pas interroger `N(jean,marie)` pour savoir les relations qui relient les objets `jean` et `marie` ou, par exemple, imposer que les prédicats soient tous d'arité 2).

  Prolog se limite aux clauses de Horn (un seul littéral positif, le but). Mais on peut transformer toute clause en clause de Horn. (voir appendix B [Clocksin 03])

  `P :- C1 , C2 , … , Cn`

  $P \Leftarrow C1 \wedge C2 \ \ldots \ \wedge Cn$

  $P \vee \neg C1 \vee C2 \ \ldots \ \vee \neg Cn$

  L'interpréteur Prolog utilise le principe de résolution proposé par Robinson (1965).

Conclusion (suite)

Pour des raisons d'efficacité, l'interpréteur choisit de résoudre

a) en débutant par le premier sous-but ;

b) de faire l'évaluation en profondeur.

Prolog inclut des prédicats extra-logiques avec des effets de bord (imprimer les résultats, lire les informations, `assert()`, `retract()`, etc., et le coupe-choix « ! » qui évite l'explosion combinatoire).

Prolog est semi-décidable (s'il trouve une réponse, c'est OK, mais il peut tourner en rond et, dans ce cas, on ne sait rien).

Annexe 1: L'unification écrit en Prolog

```
unify(X,Y) :- var(X), var(Y), X==Y, !.
unify(X,Y) :- var(X), var(Y), !, lier(X,Y).
unify(X,Y) :- var(X), !, lier(X,Y).
unify(X,Y) :- var(Y), !, lier(Y,X).
unify(X,Y) :- term_unify(X,Y).

term_unify(X,Y) :- functor(X,F,N),
                   functor(Y,F,N),
                   unify_args(N,X,Y).

unify_args(0,_,_) :- !.

unify_args(N,X,Y) :- arg(N,X,ArgX),
                     arg(N,Y,ArgY),
                     unify(ArgX,ArgY),
                     N1 is N-1,
                     unify_args(N1,X,Y).

lier(X,X).
```

Annexe 2: Opérations sur les ensembles

- Opérations sur les ensembles (*set*)

```
?- member(2,[1,2,3]).
Yes
?- member(C,[1,2,3]).
C = 1 ;
C = 2 ;
C = 3 ;
No
?- subset([a,e],[a,o,i,u,e,y]).
Yes
?- intersection([a,e,i],[a,b,c,d],I).
I = [a] ;
No
?- union([a,e,i],[a,b,c,d],U).
U = [e, i, a, b, c, d] ;
No
```

- Relation member(E,S) `true` si E appartient à au set S

```
member(X, [X|_]).
member(X, [_|R]) :- member(X,R).
```

- Relation member(E,S) true si E appartient à au set S

```
subset([],_).
subset([T|R],S) :- member(T,S), subset(R,S).
```

- Relation intersection(S1,S2,Inter) `true` si Inter est intersection de S1 et S2

```
intersection([],_,[]).
intersection([T|R],L,[T,R1]) :- member(T,L), !,
                                intersection(R,L,R1).
intersection([T|R],L,Inter) :- intersection(R,L,Inter).
```

- Relation union(S1,S2,Inter) `true` si Inter est intersection de S1 et S2

```
union([],S,S).
union([T|R],L,U) :- member(T,L), !, union(R,L,U).
union([T|R],L,[T|R1]) :- union(R,L,R1).
```

Annexe 3 : Exemple de Cut

- Soit la base de connaissances suivante :

```
p(1).
p(2) :- !.
p(3).
```

Que sera la réponse de Prolog à :

```
?- p(X).
```

```
X = 1 ;
X = 2 ;
No
```

Que sera la réponse de Prolog à :

```
?- p(X),P(Y).
```

```
X = 1, Y = 1;
X = 1, Y = 2;
X = 2, Y = 1;
X = 2, Y = 2;
No
```

Que sera la réponse de Prolog à :

```
?- p(X), !, P(Y).
```

```
X = 1, Y = 1;
X = 1, Y = 2;
No
```

Annexe 4 : Liste de travaux pratiques

- Listes des TP

  Introduction à Prolog

  Famille des dieux grecs et romains (faits et questions, voir `Dieux.pl`)

  Famille des dieux grecs et romains (règles)

  Le singe et la banane (Bratko, p. 46-50)

  Nombres : `Geographie.pl`, `Roi.pl`

  Graphes : `Vol.pl` (règles récursives, nombres, avec distance et temps maximum)

  Liste : exemples avec des listes, `Vol.pl` (avec itinéraire)

  Liste : assemblage (p. 64-67).

  La dérivation symbolique.

  Modèles écologiques (`Rabbit.pl`, `treeGrow.pl`).