# Travaux Pratiques 1

# Programmation Logique sous l'environnement SWI-Prolog

## 1. Introduction :

Prolog est un langage extraordinaire, pas tant par ses possibilités effectives, mais parce qu'il nous montre qu'il peut exister d'autres moyens de programmer un ordinateur. Prolog (PROgrammation LOGique) est né en France (Marseille) et a servi de base aux programmes de recherche japonais sur les ordinateurs de 5ème génération. Ce qui est phénoménal, c'est qu'en Prolog, il nous suffit de décrire ce que l'on sait sur le domaine étudié, en vrac, dans l'ordre où ça nous vient (en Intelligence Artificielle, on appelle cela *une base de connaissances*). Puis on décrit notre problème, Prolog va nous le résoudre, sans qu'on n'ait à lui dire comment faire !

Au cours de ce TP, vous allez travailler avec l'environnement SWI-Prolog sous Windows (disponible pour les plateformes Linux et MacOS). Vous pouvez écrire et enregistrer vos programmes à l'aide d'un éditeur de texte séparé (par exemple, bloc-notes sous Windows), puis vous appellerez l'interpréteur SWI-Prolog pour exécuter ces derniers. SWI-Prolog se réduit donc à une fenêtre d'interrogation de programmes Prolog.

Il y a également l'environnement visual prolog, disponible en version commerciale mais aussi gratuite pour un usage privé. Et il en existe d'autres, en particulier GNU-Prolog (par l'INRIA).

## 2. Règles sur constantes et variables symboliques

Le "programme" se compose d'un ensemble de **règles** (sous la forme : *conclusion* IF *conditions*) et de **faits** (en général des affirmations : on précise ce qui est vrai, tout ce qui n'est pas précisé est faux ou inconnu).

Le langage gère des **variables simples** (entiers, réels,...) et des **listes** (mais pas les tableaux).
Les variables simples sont les entiers, les réels, les chaînes de caractères (entre " ") et les **constantes symboliques** (chaînes sans espace ni caractères spéciaux, commençant par une minuscule, mais ne nécessitant pas de ").

Les noms de variables commencent obligatoirement par une **majuscule** (contrairement aux constantes symboliques), puis comme dans les autres langages est suivi de lettres, chiffres, et _ . Un variable "contenant" une valeur est dite "**liée**" ou "**instanciée**", une variable de valeur inconnue est dite "**libre**".

Exemple de programme simple :

```
est_un_homme(marc). /* l'argument est pour moi le sujet */
est_un_homme(jean).
est_le_mari_de(marc,anne)./* 1er arg: sujet, puis complément */
est_le_père_de(marc,jean).
```

*est_un_homme est un "prédicat"*. Il sera soit vrai, soit faux, soit inconnu. Dans le "programme", toutes les règles concernant le même prédicat doivent être regroupées ensemble.

Pour exécuter ce programme sous SWI-Prolog, il suffit de l'enregistrer dans un fichier texte (*nomfic.pl*), puis charger le fichier par "**consult(nomfic).**" (n'oubliez pas le .).

On peut alors "exécuter" le programme à travers des requêtes :

*?- est_un_homme(marc).*
*Yes*
*?- est_un_homme(anne).*
*No*
*?- est_un_homme(luc).*
*No*

Prolog cherche à prouver que le but (goal) demandé est vrai. Pour cela, il analyse les règles, et considère comme faux tout ce qu'il n'a pas pu prouver. Quand le but contient une variable libre, Prolog la liera à toutes les possibilités :

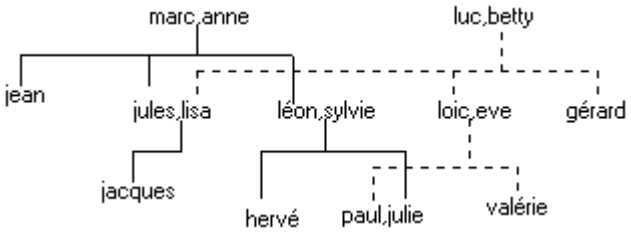
*?- est_un_homme(X).*
*X=marc (entrez ; pour demander la solution suivante)*
*X=jean*
*?- est_le_pere_de(Pere,jean).*
*Pere=marc*
*?- est_le_mari(Pere,Mere), est_le_pere_de(Pere,jean).*
*Pere=marc, Mere=anne*

On peut combiner les buts à l'aide des opérations logiques et (,) , ou (;) et non (not). Le but défini ci-dessus nous permet de trouver la mère de jean. Mais on peut également créer une règle en rajoutant :

```
est_la_mere_de(Mere,Enfant) :-
    est_le_mari(Pere,Mere),
    est_le_pere_de(Pere,Enfant).
```

Désormais le but : *est_la_mere_de(Mere,jean)* (réponse : Mere=anne) permet de rechercher la mère d'un individu, mais la même règle permet de répondre également à une recherche d'enfants : *est_la_mere_de(anne,X)* (réponse : X=jean). On peut aussi vérifier une affirmation (*est_la_mere_de(anne,jean)*) ou afficher tous les couples de solutions à la requête : *est_la_mere_de(M,X).* On ne trouvera ici qu'une solution, pour véritablement tester ceci il faut augmenter notre base de connaissances. Le comportement de Prolog est donc différent suivant que les variables soient liées (il cherche alors à prouver la véracité) ou libres (il essaie alors de les lier à des valeurs plausibles). Ceci démontre une différence capitale entre **un langage déclaratif** (on dit simplement ce que l'on sait) et **les langages procéduraux** classiques (on doit dire comment l'ordinateur doit faire, en prévoyant tous les cas).

**Exercice 1** : en se basant sur l'arbre généalogique suivant :




1. Représenter les données de cet arbre à travers les prédicats suivants :

   *est_un_homme(H) : H est un homme.*

   *est_une_femme(F) : F est une femme.*

   *est_le_mari_de(M,F) : M est le mari de F*

   *est_le_pere_de(P,E) : P est le père de E*

2. On désire compléter notre programme en définissant de nouvelles relations de parenté au sein de notre arbre généalogique. Ecrire les règles définissant les nouveaux prédicats suivants :

| | |
|---|---|
| *Est_parent_de (P,E)* | *qui réussit si P est le père ou la mère de E* |
| *est_le_fils(F, P)* | *qui réussit si P est le parent de F et F est un homme* |
| *est_la_fille(F, P)* | *qui réussit si P est le parent de F et F est une femme* |
| *est_le_frere(H,Y)* | *qui réussit si H est le frère de Y* |
| *est_la_soeur(F,Y)* | *qui réussit si F est la soeur de Y* |
| *gdparent(X,Y)* | *qui réussit si X est un grand parent de Y (grand-père ou grand-mère)* |
| *est_la_belle_mere(X,Y)* | *qui réussit si X est la belle mère de Y (Y peut être un homme ou une femme)* |

**Note** : appliquer les jeux d'essais nécessaires pour vérifier le bon fonctionnement de votre programme.

### 3. Entrées / sorties formatées

SWI-Prolog fournit des prédicats prédéfinis d'écriture formatée de termes. Ceux-ci sont très utiles pour afficher des messages, les valeurs de variables et lire des valeurs entrées au clavier par l'utilisateur que l'on unifiera ensuite à des variables.

Pour afficher un message ou la valeur d'une variable, vous utiliserez le prédicat write/1 qui permet d'afficher au choix la valeur d'une constante (un nombre, un atome ou encore une chaîne de caractères qui sera alors placée entre guillemets), ou encore celle d'une variable instanciée :

*write(3) :* affiche la constante " 3 ".

*write(martin ) :* affiche l'atome constant " martin ".

*write('le roi martin 3') :* affiche la chaîne constante de caractères " le roi martin 3 ".

*write(X ) :* affiche la valeur de la variable X (si celle-ci a été instanciée par unification).

L'ensemble de ces possibilités d'affichage peut être combiné par d'autres prédicats :

```
nl                  % saute une ligne
tab(N)              % affiche N espaces
get(C)              % lit un caractère et l'unifie avec C
get_single_char(C)  % idem get(C), mais n'attends pas de retour charriot
read(T)             % lit un terme (élémentaire ou composé) et l'unifie avec T
writef(Format,L)    % Format est une chaîne de caractères à afficher,
                    % L est une liste de termes à afficher
```

*Format* peut contenir des séquences particulières permettant d'afficher des caractères spéciaux:

| | |
|---|---|
| \n | <NL> est affiché |
| \t | <TAB> est affiché |
| \\ | '\' est affiché |
| \% | '%' est affiché |

*Format* peut également contenir des séquences particulières pour inclure des éléments de la liste *L*. La séquence dépend de la nature de l'élement à afficher:

| | |
|---|---|
| %t | si l'élément est un terme |
| %n | si l'élément est un code ASCII |
| %s | si l'élément est une chaîne de caractères |

Par exemple, l'exécution de

```
writef("ceci %t l'%s %t \n et j'affiche X=%t",[est,"exemple numero",45,X])
```

affichera la séquence:

```
ceci est l'exemple numero 45
et j'affiche X=toto
```

si la variable X a pour valeur toto.

**Exercice 2 :**

Modifiez votre programme pour obtenir des réponses formatées en langage naturel à vos questions concernant la relation de parenté est_le_fils/2. Par exemple on désire obtenir un affichage de ce type :

*?- est_le_fils(martin, X).*

*alain est le fils de martin*

## 4. Programmation récursive en Prolog : première approche

La puissance de Prolog réside avant tout dans son moteur d'inférence basé sur la résolution de Robinson. En particulier, celle-ci permet de déduire de nouvelles connaissances à l'aide de prédicats récursifs. Par exemple, la clause ci-dessous définit les amis d'une personne comme les amis de ses amis :

ami(X,Y) :- ami(X,Z), ami(Y,Z).

Bien entendu, cette récursivité doit être mise en oeuvre avec soin sous peine de voir la stratégie de résolution Prolog boucler indéfiniment (ce qui serait le cas dans l'exemple ci-dessus, d'ailleurs…). Nous n'avons pas le temps d'étudier en détail les situations de récursivité bloquantes, qui correspondent toutes à une récursivité à gauche. Nous allons nous contenter d'étudier ce problème expérimentalement sur quelques exemples.

**Exercice 3 :**

1. On demande de réaliser un prédicat d'arité 2 ancetre(X,Y) qui réussit si X est un ancêtre quelconque (parent, grand - parent, arrière grand parent, etc...) de Y.

2. Essayez d'écrire un prédicat d'arité 2 famille(H1,H2) qui réussit si H1 et H2 font partie d'une même famille, c'est à dire qu'ils ont au moins un ancêtre commun

## 5. variables numériques

En Prolog, l'écriture : **X=Y** peut avoir différents résultats :

- Si X et Y liées, répondra Yes si elles sont égales, No sinon.
- Si X libre et Y lié, proposera comme solution X valant la valeur de Y.
- Si X lié et Y libre, proposera comme solution Y valant la valeur de X.
- Si X et Y libres, erreur (car infinité de solutions)

Par contre l'écriture X=Y+1 (qui est équivalent à Y+1=X) va poser quelques problèmes. Il est tout d'abord obligatoire, dans cette écriture, qu'Y soit lié (Prolog ne sait pas inverser une équation, qu'elle soit simple comme ici ou plus compliquée). Mais ce n'est pas tout : **=** va simplement recopier l'équation (en ayant instancié les variables) :

Y=23, X=Y+1, write(X). affichera 23+1, et pas 24.

C'est pourquoi deux autres opérateurs sont définis :

- **is** calcule (on dit aussi "évalue") la valeur à droite (toujours liée) et instancie le résultat à la variable (libre) à gauche, ou la compare à la valeur liée de gauche. Par exemple X is 10+15 met 25 dans X, ou dit Yes si X vaut 25 (parce que avant on a dit X is 30-5 ou même X=25). Certains Prolog utilisent aussi l'opérateur :=
- **=:=** évalue les expressions à droite et à gauche et regarde si les résultats sont égaux (les deux arguments doivent être liés)
- Pour les autres comparaisons, les opérateurs (qui évaluent à droite et à gauche) sont :
    - =:= =\= (différent) < > =< >= ,
    - les opérateurs arithmétiques : + - * / mod ** , sin, cos, log,....

**Exercice 4** :

1. Ecrire un programme qui affiche les solutions (X1 et X2) d'une équation de 2$^{ème}$ degré ($A*X^2 + B*X +C = 0$) en entrant les valeurs des variables A, B et C.
2. Écrire un programme qui pour un but "affiche1(N)" affiche les nombres de N à 0 dans l'ordre décroissant.

3. Écrire un programme qui pour un but "affiche(N, Ordre)" affiche les nombres de N à 0 dans l'ordre défini par le paramètre Ordre (croissant ou décroissant).
4. Ecrire un programme qui pour un but "fact(N, X)" affiche les factoriels de 0 à N et enregistre la factoriel de N dans X.

   **Ex :** ? - fact(3, X).

   0 ! = 1

   1 ! = 1

   2 ! = 2

   3 ! = 6

   X = 6

**Notes :** Dans cet exercice, on utilise les prédicats prédéfinis write et *read*. En plus, on fait appel à la notion de récursivité.