Prolog

Notions de base :

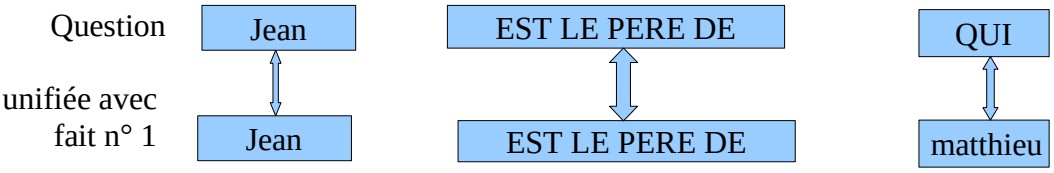
- Un programme logique est un ensemble d'axiomes, ou de règles définissant des relations entre obets.
  Règles : les règles sont des énoncés de la forme :
  $A \leftarrow B_1, B_2, \ldots B_n$ où $n \geq 0$
- Un calcul d'un programme logique est une déduction de conséquences à partir du programme.
- Le type d'énoncé le plus simple est appelé un **fait**. Les faits sont un moyen d'affirmer qu'une relation a lieu entre les objets.
  Ex.: `père(jean, matthieu).`

Ce fait affirme que jean est le père de `matthieu`, ou il y a la relation (ou **prédicat**) `père` entre les individus `jean` et `matthieu` (les noms des individus sont appelés les **atomes**).

- Un ensemble fini de faits constitue un **programme**.

Déclarer, chercher et comparer (notion d'unification) :

fait n° 1 : pour signifier que jean est père de matthieu — `pere(jean, matthieu).`
fait n° 2 : ---------- jean est père de thomas — `pere(jean, thomas).`
fait n° 3 : ---------- thomas est père de sylvie — `pere(thomas, sylvie).`



Cette façon de faire, soit chercher l'égalité entre deux phrases, se nomme en Prolog « **l'unification** ».
Supposons que nous voulons de qui jean est père. Il nous suffit de chercher dans la base des faits (qui constitue la base de données), d'une manière très naturelle càd en comparant la question :

- jean est père de QUI ? à chacune des lignes de la base de données, ce qui fera :
  1. comparaison de « jean est père de QUI » avec
     « jean est père de matthieu »

Donc QUI = matthieu

  2. comparaison de « jean est père de QUI » avec
     «jean est père de thomas»

Donc QUI = thomas

  3. comparaison de « jean est père de QUI » avec
     «thomas est père de sylvie»

Donc la comparaison ne donne <u>rien</u>

L'**unification** est un processus qui fait coïncider les formules dans la définition formelle de la **Résolution**. La Résolution est une **Règle d'Inférence** càd qu'elle indique comment une proposition peut découler d'un Ensemble d'autres propositions.
L'unification est l'activité qui consiste à dériver des conséquences intéressantes d'un Ensemble donné de propositions ou en d'autres mots : la démonstration des théorèmes.

Au début des années '60 lorsque les chercheurs ont commencé à étudier la possibilité de programmer des calculateurs numériques pour démontrer automatiquement des théorèmes il y a eu beaucoup d'agitation. L'une de principales découvertes de cette époque est celle du Principe de Résolution par J. Alan Robinson est ses applications à la démonstration automatique de théorèmes. La Résolution est conçue pour travailler avec les formules en forme clausale. En partant de 2 clauses reliées de façon adéquate, elle générera une nouvelle clause qui en est la conséquence.
Exemple : à partir de :

triste(jean) ; fache(jean) :- travaille(aujourdhui), pluie(aujourdhui).

et de :

desagreable(jean) :- fache(jean), fatigue(jean).

on déduit :

triste(jean) ; desagreable(jean) :- travaille(aujourdhui), pluie(aujourdhui), fatigue(jean).

En termes de Prolog, si nous avons deux clauses sous forme de structures et si nous faisons coïncider les sous-structures correspondantes, le résultat de leur combinaison serait la représentation de la nouvelle clause.

Définition :

Deux termes A et B sont unifiables s'il existe une substitution $^{\upsilon}$ telle que $A^{\upsilon} = B^{\upsilon}$. Une telle substitution est un unificateur de A et B.
Exemple: si A = point(X, Y, 12).
et B = point(U, 13, V).
alors $^{\upsilon}1$ = { X = U, Y = 13, V = 12} est un unificateur de A et B puisque $A^{\upsilon}1 = B^{\upsilon}1$ = point(U, 13, 12), ou un autre :
$^{\upsilon}2$ = {X = 15, Y= 13, U = 15, V = 12} est un autre unificateur de A et B puisque $A^{\upsilon}2 = B^{\upsilon}2$ = point(15,13,12=.

Attention : on ne peut unifier une variable X à une structure la contenant par exemple X et pere(X) ne sont pas unifiables.
Dans la plupart des interprétations en Prolog on unifie toujours une variable à une structure. Car on dit que l'on n'affecte pas le test de *Occur-Check*, qu'il est nécessaire pour que l'opération d'unification soit correcte.

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 (nomfichier.pl), puis d'appeler pl sur la ligne de commande, charger le fichier par "*consult(nomfichier)*." (n'oubliez pas le . « point »).

On peut alors "exécuter" le programme, en interrogeant l'interprète :

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

Prolog cherche à prouver que le but (prédicat) 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 *l'unification* :

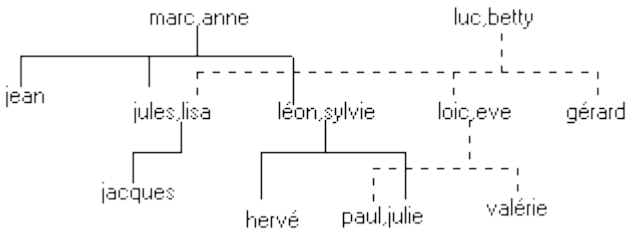
```
?- est_un_homme(X).
X=marc (entrez ; pour demander la solution suivante)
X=jean
?- est_le_pere_de(Pere,jean).
Pere=marc
?- est_le_mari(Pere,Mere), est_le_pere_de(Pere,jean).
Pere=marc, Mere=anne
```

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 grande différence 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 : compléter les données pour respecter :



On définira également les règles définissant : *est_le_fils*, *est_la_fille*, *est_la_belle_mere*,....

Pour effectuer un ou, on peut utiliser le OU (;) ou utiliser plusieurs règles:

```
est_la_belle_mere_de(BelleMere,Gendre) :-
      est_le_mari_de(Gendre,Epouse),
      est_la_mere_de(BelleMere,Epouse). /*Belle-mère d'un homme*/
est_la_belle_mere_de(BelleMere,Bru) :-
      est_le_mari_de(Epoux,Bru),
      est_la_mere_de(BelleMere,Epoux). /*Belle-mère d'une femme*/
```

On peut également définir les femmes par :

```
est_une_femme(F) :- not(est_un_homme(F)).
```

Cette règle marche pour des variables liées (dira Yes s'il ne l'a pas trouvé parmi les hommes) mais pas pour une variable libre (en effet, toute combinaison de caractères pourrait répondre à la requête). Pour qu'une règle puisse s'appliquer à une variable libre, il faut que l'on permette à Prolog

de trouver toutes les possibilités (et donc qu'elles soient en nombre limité). Exemple:

```
est_une_femme(F) :-
      est_le_pere_de(_,F), /* _ est le nom (imposé) d'une variable
                  dont la valeur ne nous intéresse pas, Prolog n'a
                  donc pas besoin de la lier */
      not (est_un_homme(F)).
```

Ici, on cherche recherche d'abord ceux et celles qui ont un père, puis on élimine les hommes. On peut remarquer qu'anne et betty ne seront pas considérées comme femme. Il suffit pour résoudre ce problème de rajouter une seconde règle, traitant des épouses. Mais pour qu'il ne nous affiche pas deux fois celles qui sont à la fois épouses et filles, on pourra l'écrire ainsi :

```
est_une_femme(F) :-
      est_le_mari_de(_,F)
      and not (est_un_homme(F))
      and not (est_le_pere_de(_,F)) /* pour ne pas reprendre les cas
            traités par la règle précédente */
```

le but : *est_une_femme(X)* nous donne maintenant toutes les femmes que nous avons définies.

Exercice : rajouter tantes, cousins, grand mères...

Pour simplifier la gestion des hommes par exemple, on peut utiliser une liste (voir détails plus loin) :

```
hommes([marc, luc, jean, jules, léon, loic, gerard, hervé, jacques,
paul]).
appartient_à(X,[X|_]).
appartient_à(X,[_|Queue]) :- appartient_à(X,Queue).
est_un_homme(X) :-
      hommes(Liste),
      appartient_à(X,Liste).
```

Variables numériques

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

- Si X et Y sont liées, répondra Yes si elles sont égales, No sinon.
- Si X est libre et Y est lié, proposera comme solution X valant la valeur de Y.
- Si X est lié et Y est libre, proposera comme solution Y valant la valeur de X.
- Si X et Y sont 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).

- Regardez les questions suivantes et les réponses que l'interprète fournit :

```
?- X = 1+1+1+1.

X = 1+1+1+1

Yes

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

X = 1+1+1+1
Y = 4

Yes

?- X = 1+1+1+a.

X = 1+1+1+a

Yes

?- X is 1+1+a.
ERROR: Arithmetic: `a/0' is not a function

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

Z = 2
X = 5

Yes

?- 1+2 < 3+4.

Yes
```

Il y a évaluation des 2 termes de gauche et de droite avant la comparaison (idem pour les : <<, >>, ^, v). Pour numérique on a le symbole « =:= » et pour littérale : « == » , « \== », =<, >=
Testez les questions suivantes :

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

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

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

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

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

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

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

No
?- 1+X =:= 1+2.
ERROR: Arguments are not sufficiently instantiated
?- 1+2 == 1+2.

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

No
?- 1 + a == 1 + a.

Yes
?-
```

Exemple : écrire un programme qui pour un but *"affiche(10)."* affiche les nombres de 10 à 0 : (on utilise le prédicat prédéfini write) :

```
affiche(X) :-
      write(X), write(' '),
      X>0,   /* condition de fin */
      Y is X-1,
      affiche(Y).
```

Rq1 : il faut obligatoirement utiliser une variable intermédiaire (Y ici).

Rq2 : évidement, là je triche quand même : c'est plus procédural que déclaratif.

Exo : afficher désormais dans l'ordre croissant .

Truc : il suffit d'inverser l'ordre des instructions, mais il répond false (comme d'ailleurs aussi dans le cas précédent) mais avant les affichages. Il suffit de donner le fait : affiche(0).

Pour les autres comparaisons, les opérateurs (qui évaluent à droite et à gauche) sont : =:= =\= (différent) < > =< >= , les opérateurs arithmétiques : + - * / mod ** , sin, cos, log,....

exercices amusants

chaque lettre représente un chiffre et un seul. Trouver la ou les solutions à :

MOT	FORTY	NEUF		
+ MOT	+ TEN	+ UN	UNE	CHIEN
+ MOT	+ TEN	+ UN	+ UNE	+ CHASSE
= BLA	= SIXTY	= ONZE	=DEUX	= GIBIER

proposition de solution (la plus simple possible), pour le premier :

```
/* je décris l'ensemble des possibilités envisageables */
chiffre(0).
chiffre(1).
chiffre(2).
chiffre(3).
chiffre(4).
chiffre(5).
chiffre(6).
chiffre(7).
chiffre(8).
chiffre(9).

/* je note ce que je veux */
solution([M,O,T,B,L,A]) if
      chiffre(M),M=\=0,
      chiffre(O),O=\=M,
      chiffre(T),T=\=O,T=\=M,
      chiffre(B),B=\=0,B=\=T,B=\=O,B=\=M,
      chiffre(L),L=\=B,L=\=T,L=\=O,L=\=M,
      chiffre(A),A=\=L,A=\=B,A=\=T,A=\=O,A=\=M,
      3*(M*100+O*10+T)=:=B*100+L*10+A.
```

Remarque (importante) : on détaille ici exactement et précisément ce que l'on veut (le "quoi"), mais on ne donne aucune indication sur la méthode de résolution du problème, pas d'algorithme (le "comment"), on laisse Prolog se débrouiller.