



INSA Lyon  
20, avenue Albert Einstein  
69621 Villeurbanne Cedex

LIVRABLE DE PROJET

---

# Prolog

## « Puissance 4 »

du 1<sup>er</sup> au 15 octobre 2013

---



*Hexanôme H4404 :*

Guillaume ABADIE  
Nicolas BUISSON  
Louise CRÉPET  
Rémi DOMINGUES  
Aline MARTIN  
Martin WETTERWALD

*Enseignants :*

Jean François BOULICAUT  
Mehdi KAYTOUE

Année scolaire 2013-2014

# 1. Bilan des exercices

## 1.1 Prédicats

La particularité de Prolog réside dans le fait qu'il n'y a plus d'itérations comme un langage de programmation conventionnel. Tout n'est que prédicat. Ainsi, la méthode de programmation est très différente. Fini le traitement de données dans des variables que l'on met à jour, car la programmation par prédicats permet simplement de lier des propriétés entre des variables et/ou constantes.

Considérons pour la suite, le prédicat :

$$(membre(X, L) \Leftrightarrow vrai) \Leftrightarrow X \in L$$

## 1.2 Vérification de propriétés

La vérification de propriétés permet de s'assurer qu'une (ou plusieurs) constantes vérifient un ensemble de prédicats. Considérons le code ci-dessus.

Alors on a à l'exécution :

```
?- membre(1, [1, 2, 3]).  
true  
  
?- membre(4, [1, 2, 3]).  
false
```

En effet, à la première interrogation, on vérifie le prédicat  $1 \in [1, 2, 3]$ , ce qui est vrai, d'où la réponse de Prolog « *true* ». La propriété est alors vérifiée, renvoyant ainsi vrai. Tandis que la seconde interrogation  $4 \in [1, 2, 3]$  est fausse car  $4 \notin [1, 2, 3]$ , d'où la réponse « *false* ».

## 1.3 Recherche de solutions

La recherche de solution consiste à définir des propriétés entre des objets et/ou constantes. Par exemple :

$$X \in [1, 2, 3]$$

Ce qui en Prolog donne :

```
?- membre(X, [1, 2, 3]).
```

Ainsi a l'exécution, Prolog est capable d'évaluer les solutions de  $X$  grâce a cette propriété ainsi définie :

```
?- membre(X, [1, 2, 3]).
X = 1;
X = 2;
X = 3;
false
```

## 1.4 Recherche de solutions d'un système

Une propriété sur une variable par exemple, peut être défini par plusieurs prédicats. Par exemple :

$$\begin{cases} L \in [1, 2, 3] \\ L \in [3, 4, 2] \end{cases}$$

Cela revient simplement a l'écriture en Prolog :

```
?- membre(X, [1, 2, 3]), membre(X, [3, 4, 2]).
X = 2;
X = 3;
false
```

## 1.5 Recherche de solutions non-déterministe

La dangerosité de la recherche de solution, est qu'il est possible qu'une infinité de solutions vérifient une même propriété. Considérons par exemple le code suivant :

```
?- membre(1, L).
```

Cette est équivalent à  $1 \in L$ . Mais alors, combien de listes pourraient vérifier cette propriété?

**Initialisation** : Une liste telle que  $[1, 2]$  vérifie cette propriété.

**Hérédité** : En notant  $cat(A, B)$  la concatenation de deux listes  $A$  et  $B$ ,

Soit une liste  $L$  telle que  $1 \in L$ ,

Alors  $\forall X \in \mathbb{N} / 1 \in cat([X], L)$

**Conclusion** : Il existe une infinité de solutions et Prolog va essayer de toutes les générer, causant une exception du au manque de mémoire de la machine.

```
?- membre(1, L).
L = [1|_G2214] ;
```

```

L = [ _G2213, 1|_G2217] ;
L = [ _G2213, _G2216, 1|_G2220] ;
L = [ _G2213, _G2216, _G2219, 1|_G2223] ;
L = [ _G2213, _G2216, _G2219, _G2222, 1|_G2226] ;
L = [ _G2213, _G2216, _G2219, _G2222, _G2225, 1|_G2229] ;
...

```

## 1.6 Programation de prédicats triviaux

Au paravant, nous fesions que utiliser des prédicats, mais bien entendu, l'objectif est de pouvoir coder les siens. C'est ici que la méthode de programmation diffère complètement de la programmation itérative. Nous devons proceder avec une démarche d'analyse mathématique. Pour cela, interessons nous à la ré-écriture de

$$(membre(X, L) \Leftrightarrow vrai) \Leftrightarrow X \in L$$

Tout d'abord, on remarque que  $X \in [X, \dots]$ , ou autrement dit :  $X \in (L = cat([X], L1))$  avec  $L1$  une autre liste. Alors on en deduit le premier prédicat :

```
membre(X, [X|_]) .
```

On remarque par ailleurs que :

$$\forall (X, Y) \in \mathbb{E}^2, X \in L \Rightarrow X \in cat([Y], L)$$

Ainsi on en déduit le second prédicat :

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

## 1.7 Prédicats avec calculs arithmétiques

On se propose d'implémenter un predicat permettant de verifier a un element a une position donnée dans une liste :  $X = L[I]$

Exemple :

```

?- elementAtPos(2, hello, [hi, hello, bye]).
true

?- elementAtPos(3, hi, [hi, hello, bye]).
false

```

Tout d'abord, le premier cas evidents survient lorsque la list  $L$  est vide, le predicat doit etre faut :

```
elementAtPos(_, _, []) :- fail.
```

Notez que cette ligne est facultative, car définir un predicat comme echouant (fail) revient à ne pas le définir. Nous le metons ici simplement pour illustrer la demarche.

Vien ensuite le cas evident ou nous voulons tester le premier element de la liste. Ainsi :

```
elementAtPos(1, X, [X|_]).
```

Enfin, il demeure le cas ou ce n'est pas le premier element. Alors, l'idée, est de recursivement prouver *elementAtPos/3* mais en retirant le premier element a chaque fois :

```
elementAtPos(I, X, [_|L]) :-  
    I1 is I-1, elementAtPos(I1, X, L).
```

## 1.8 Recherche avec calculs arithmétiques

Avec le prédicat *elementAtPos/3* défini ci dessus, il est alors facil de l'utiliser pour rechercher un element  $X$  à une position  $I$  dans une liste  $L$ .

```
elementAtPos(2, X, [10, 11, 12]).  
X = 11 ;  
false
```

Cependant il survient une erreur spécial si nous cherchons l'index  $I$  d'un élément  $X$  dans une liste  $L$ .

```
elementAtPos(I, 12, [10, 12]).  
ERROR: >/2: Arguments are not sufficiently instantiated
```

En effet si nous deroulons les predicat, nous optenons alors :

```
elementAtPos(I, 12, [10, 12]) :-  
    I1 is I - 1, elementAtPos(I1, 12, [12]).  
elementAtPos(I, 12, [10, 12]) :- elementAtPos(I + 1, 12, [12]).
```

Ainsi nous voyons l'expression  $I + 1$  a verifie une propriété lié par le prédicat *elementAtPos/3*, générant une erreur car étant une variable non lié (unbound variable).

En effet, cette erreur peut etre facilement reproduite par :

```
?- elementAtPos(I + 2, 11, [10, 11, 12]).  
ERROR: elementAtPos/3: Arguments are not sufficiently instantiated
```

## 2. Projet : Puissance 4

### 2.1 Règles du jeu

Le puissance 4 est un jeu de société à deux joueurs. Chaque joueur doit, à son tour, insérer un jeton de sa couleur dans une des sept colonnes côte à côte, chacune ayant une capacité maximale de six jetons. Le but du jeu est d'aligner verticalement, horizontalement ou en diagonal, 4 jetons de sa couleur avant l'adversaire.

### 2.2 But du joueur idéal

Dans le cas d'un joueur idéal, le but n'est simplement d'aligner 3 jetons précédemment, puis prévoir de jouer le 4<sup>ème</sup> au tour suivant. En effet, l'adversaire pourrait bloquer cet alignement, lorsque c'est à son tour de jouer. L'objectif du joueur idéal est donc de réaliser au moins deux alignements de 3 jetons en un coup. Laissant ainsi le joueur adverse contre l'inévitable fatalité : Il ne peut plus contrer ces alignements en un seul jeton.

### 2.3 Travail réalisé

En plus de l'implémentation du module de mécanisme de jeu et réalisation des tests unitaire, nous avons implémenté 4 joueurs ayant des stratégies différentes :

- joueur aléatoire ;
- joueur aléatoire munis d'heuristiques ;
- joueur parcourant l'arbre des possibilités ;
- intelligence artificielle apprenant par **moteur d'inférence**, de ses échecs précédents.

Mais aussi :

- interface utilisateur en ligne de commande pour jouer une partie ;
- module de tournoi générant des statistiques ;
- module d'entraînement du moteur d'inférence ;
- module d'étude de l'apprentissage du moteur d'inférence ;
- sauvegarde et chargement de la base de connaissances du moteur d'inférence.

## 3. Implémentation des exercices

### 3.1 Exercice 1

```
personne(pjeanmarie,h).
personne(mmathieu,h).
personne(mscuturici,f).
personne(mroland,h).
personne(schristine,f).
personne(rjeannine,f).
personne(brumpler,f).
personne(wmartin,h).
personne(aguillaume,h).
```

```
parent(aguillaume,mscuturici).
parent(aguillaume,schristine).
parent(wmartin,mmathieu).
parent(wmartin,mscuturici).
parent(mmathieu,brumpler).
parent(mmathieu,pjeanmarie).
parent(mroland,brumpler).
parent(mroland,pjeanmarie).
```

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

```
demifrere(X,Y) :- parent(Z,P1), parent(X,P1), not(X = Z).
frere(X,Z) :-
    parent(Z,P1), parent(X,P1), parent(X,P2),
    parent(Z,P2), not(P1 = P2), not(X = Z).
```

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

```
oncleDeX(X,Y) :- parent(X,Z), frere(Z,Y).
```

## 3.2 Exercice 2

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Text existence d'un membre dans une liste
% END
```

```
    membre(X, [X|_]).
```

```
% FOREACH
```

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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Soustraction ensembliste : arg2\arg1 = arg3
% END
```

```
    element1(X, [X|R], R).
```

```
% FOREACH
```

```
    element1(X, [T|Q1], [T|Q2]) :- element1(X, Q1, Q2).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Extraction des permutations
% END
```

```
    extract(List, [X]) :-
        membre(X, List).
```

```
% FOREACH
```

```
    extract(List, [X|L]) :-
        element1(X, List, R), extract(R, L).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Concatenation de listes
% END
```

```
    concatLists([], Result, Result).
```

```
% FOREACH
```

```
    concatLists([X|ListA], ListB, [X|Result]) :-
        concatLists(ListA, ListB, Result).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Inversion de la liste
% END
```

```
    invertList([], []).
```

```
% FOREACH
```

```
    invertList([X|List], Return) :-
        invertList(List, Result),
        concatLists(Result, [X], Return).
```



```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg4 = arg3.replaceAll(arg1 by arg2)
% END
    subsAll(_, _, [], []).

% FOREACH
    subsAll(Pattern, Replace, [Pattern|List], [Replace|Result]) :-
        subsAll(Pattern, Replace, List, Result).
    subsAll(Pattern, Replace, [X|List], [X|Result]) :-
        subsAll(Pattern, Replace, List, Result), not(X = Pattern).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg4 = arg3.replaceFirst(arg1 by arg2)
% END
    subsFirst(Pattern, Replace, [Pattern|List], [Replace|List]).

% FOREACH
    subsFirst(Pattern, Replace, [X|List], [X|Result]) :-
        subsFirst(Pattern, Replace, List, Result), not(X = Pattern).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg4 = arg3.replaceOnce(arg1 by arg2)
% END
    subsOnce(Pattern, Replace, [Pattern|List], [Replace|List]).

% FOREACH
    subsOnce(Pattern, Replace, [X|List], [X|Result]) :-
        subsOnce(Pattern, Replace, List, Result).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TESTS!
testsExo2 :-
    membre(a, [a,b]),
    not(membre(x, [a,b])),

    element1(b, [a,b,c], [a,c]),
    element1(a, [a,b,c], [b,c]),
    element1(a, [b,a,c], [b,c]),
    element1(a, [b,d,w,z,d,a], [b,d,w,z,d]),

    extract([a,b,c,d], [b,c]),
    extract([a,b,c,d], [c,b]),

    concatLists([], [], []),
    concatLists([a,b], [], [a,b]),

```

```

concatLists([], [a,b], [a,b]),
concatLists([a,b], [c,d], [a,b,c,d]),

invertList([], []),
invertList([a,b], [b,a]),

subsAll(x, y, [a,x,b,x], [a,y,b,y]),

subsFirst(x, y, [a,x,b,x], [a,y,b,x]),
not(subsFirst(x, y, [a,x,b,x], [a,x,b,y])),
not(subsFirst(x, y, [a,x,b,x], [a,y,b,y])),

subsOnce(x, y, [a,x,b,x], [a,y,b,x]),
subsOnce(x, y, [a,x,b,x], [a,x,b,y]),
not(subsOnce(x, y, [a,x,b,x], [a,y,b,y])).

```

### 3.3 Exercise 3

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg2 = arg3[arg1]
% END
    elementFind(1, Pattern, [Pattern|_]).

% FOREACH
    elementFind(Pos, Pattern, [_|ListEnd]) :-
        NextPos is Pos-1, elementFind(NextPos, Pattern, ListEnd).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TESTS!
testsExo3 :-
    elementFind(1, a, [a,b,c,a,b]),
    elementFind(4, a, [a,b,c,a,b]),
    elementFind(2, b, [a,b,c,a,b]),
    not(elementFind(1, a, [])).

```

### 3.4 Exercice 4

[illegible]

```

% END
    setInter ([], _, Result, Result).

% FOREACH
    setInter ([X|A], B, Result, Inter) :-
        member(X,B) ->
            setInter (A, B, Result, [X|Inter]) ;
        setInter (A, B, Result, Inter).

% INIT
    setInter (A, B, Result) :-
        setInter (A, B, Result, []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg3 = arg1 - arg2
% END
    setMinus ([], _, Result, Result).

% FOREACH
    setMinus ([X|A], B, Result, Inter) :-
        member(X,B) ->
            setMinus (A, B, Result, Inter) ;
        setMinus (A, B, Result, [X|Inter]).

% INIT
    setMinus (A, B, Result) :-
        setMinus (A, B, Result, []).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% (arg1 = arg2)
% CALL (A = B) <=> (A - B = B - A)
    areEqualSets (A, B) :-
        setMinus (A, B, Minus),
        setMinus (B, A, Minus).

% Actually, the areEqualSets time complexity is the same as
% setMinus O(len(A)*len(B)). But with sorted sets given into
% parameters, this complexity would fall down to
% O(min(len(A),len(B))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TESTS!
testsExo4 :-
    isSet ([]),
    isSet ([a,b,c]),
    not(isSet ([a,b,a,c])),

```

```

genSet([], []),
genSet([a,b], R21),
genSet([a,b,a,b,c], R22),

areEqualSets(R21,[a,b]),
areEqualSets(R22,[a,b,c]),

areEqualSets([], []),
areEqualSets([a,b], [a,b]),
areEqualSets([a,b], [b,a]),

setUnion([], [], []),
setUnion([a,b], [], R01),
setUnion([], [a,b], R02),
setUnion([a,b], [c,d], R03),
setUnion([a,b,c], [c,d], R04),

areEqualSets(R01, [a,b]),
areEqualSets(R02, [a,b]),
areEqualSets(R03, [a,b,c,d]),
areEqualSets(R04, [a,b,c,d]),

setInter([], [], []),
setInter([a,b], [], []),
setInter([], [a,b], []),
setInter([a,b], [b,c], [b]),

setMinus([], [], []),
setMinus([a], [], [a]),
setMinus([], [a,b], []),
setMinus([a,b,c,d], [b,d], R05), areEqualSets(R05, [a,c]).

```