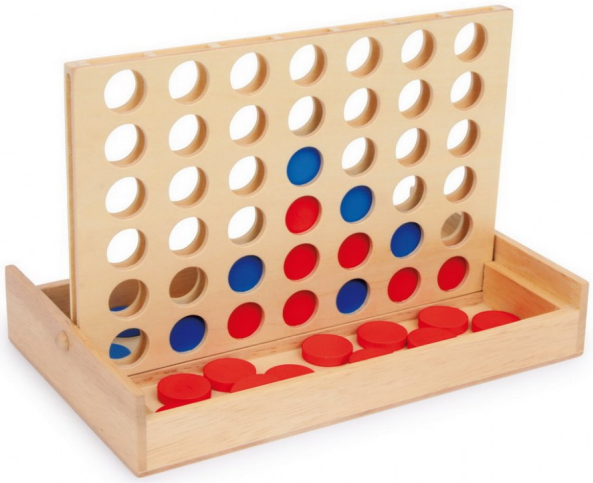
INSA Lyon
20, avenue Albert Einstein
69621 Villeurbanne Cedex

Livrable de projet

Prolog

« Puissance 4 »

du 1er 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 pas d'itérations comme dans un langage de programmation conventionnel : tout n'est que prédicat. Ainsi, la méthode de programmation est très différente. Il n'y a pas de traitement de données dans des variables que l'on met à jour, car la programmation par prédicat permet simplement de lier des propriétés entre des variables et/ou constantes.

1.2 Vérification de propriétés

Considérons pour la suite, le prédicat suivant :

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

La vérification de propriétés permet de s'assurer qu'une (ou plusieurs) constante vérifie un ensemble de prédicats.

Considérons le code énoncé précédemment. 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 $1 \in [1, 2, 3]$, cette propriété est vérifiée, d'où la réponse de `Prolog` « *true* ». 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 solutions 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, à l'exécution, `Prolog` est capable d'évaluer les solutions de X grâce à la définition précédente du prédicat membre.

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

On demande à `Prolog` s'il peut prouver le prédicat fourni. Ici, il existe plusieurs manières de le prouver. Taper le caractère «`;`» permet de lui demander de prouver le prédicat d'une autre façon. Cela explique le mot `false` à la fin, signifiant qu'il est impossible de prouver ce prédicat d'une autre manière que les trois précédemment données.

1.4 Recherche de solutions d'un système

Les propriétés d'une variable par exemple, peuvent être définies par plusieurs prédicats. Par exemple :

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

Peut revenir simplement en `Prolog` à

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

1.5 Recherche de solutions non-déterministes

Le risque encouru lors de la recherche de solution(s), est qu'il est possible qu'une infinité de solutions vérifie une même propriété. Considérons par exemple le code suivant :

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

Il est équivalent à $1 \in L$. Cherchons à déterminer combien de listes pourraient vérifier cette propriété :

Initialisation : La liste $[1, 2]$ par exemple vérifie cette propriété.

Hérédité : En notant $cat(A, B)$ la concaténation 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 due 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 Programmation de prédicats triviaux

Depuis le début de ce rapport, nous nous sommes contentés d'utiliser des prédicats, mais bien entendu, l'objectif est de pouvoir en programmer soi-même. C'est ici que la méthode de programmation diffère complètement de la programmation itérative. Nous devons procéder avec une démarche d'analyse mathématique. Pour cela, intéressons-nous à la réécriture de :

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

Tout d'abord, on peut remarquer que si X se trouve en tête de L, alors il appartient de manière évidente à cette liste. Autrement dit $X \in [X, ...]$, ou encore $X \in (L = cat([X], L1))$ avec $L1$ une autre liste. Alors on en déduit le premier prédicat :

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

On remarque par ailleurs que si X n'est pas en tête de la liste recherchée, alors il faut vérifier s'il n'est pas en tête de la sous-liste associée au retrait de cette tête,

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

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 prédicat permettant de vérifier un élement à 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 évidents survient lorsque la liste L est vide, le prédicat doit alors être faux puisqu'aucun élément ne peut exister dans une liste vide :

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

Notez que cette ligne est facultative, car définir un prédicat comme échouant (fail) reviens à ne pas le définir. Nous illustrons ici simplement la démarche que nous avons suivis.

Viens ensuite le cas évident où nous voulons tester le premier élément de la liste.

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

Enfin, il demeure le cas où X n'est pas le premier élément. Alors, l'idee, est de récursivement prouver $elementAtPos/3$ mais en retirant le premier élément de la liste L a chaque fois :

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

Vous aurez remarqué ici la présence d'un index I qui est décrémenté dans $I1$. En effet, le système de récursion marche de telle sorte que lorsqu'une solution sera trouvée via le cas évident, si elle est trouvée, alors la valeur $I1$ de l'avant dernier appel récursif sera égal à 1, et donc il pourra être vu comme

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

et ainsi de suite, la valeur de I dépendra donc de la profondeur de la récursion nécessaire, or comme la tête de la liste est enlevée à chaque récursion cela correspond bien à un index de la liste.

1.8 Recherche avec calculs arithmétiques

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

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

Cependant si nous cherchons l'index I d'un élément X donné et dans une liste L donnée, il survient une erreur spéciale.

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

En effet si nous déroulons les prédicats, nous obtenons 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 $I - 1$ tente de vérifier une propriété liée par le prédicat *elementAtPos*/3. Cela génère une erreur car elle est une variable non liée (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 jeux

Le puissance 4 est un jeu de societé à deux joueurs. Chaque joueur doit, chacun son tour, insérer un jeton de sa couleur dans une des sept colonnes côte à côte, chacune ayant une capacité maximale de six jetons. Ce pion doit, bien entendu, tomber jusqu'à la case la plus basse inocupée de sa colonne. Les cas d'un jeton ne tombant pas jusqu'en bas ou dans la colonne choisie ne comptent pas comme coups joués. Le but du jeu est d'aligner verticalement, horizontalement ou en diagonale 4 jetons de sa couleur avant l'adversaire.

2.2 But du joueur idéal

Dans le cas d'un joueur idéal, le but est simplement, après avoir aligné 3 jetons, de prévoir de jouer le $4^{\text{ème}}$ au tour suivant. Mais, comme l'adversaire pourrait casser la ligne en jouant à cet endroit lors de son tour. 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 pourra plus contrer ces tentatives en un seul jeton.

2.3 Travail realisé

En plus de l'implémentation du module de méchanisme de jeux et réalisation des tests unitaires, nous avons implementé 4 intelligences artificielles ayant des stratégies différentes :

- joueur aléatoire ;
- joueur aléatoire muni d'heuristiques ;
- joueur parcourant l'arbre des possibilités ;
- inteligence artificielle apprenant par **moteur d'inférence**, de ses échecs precédents.

Mais également :

- interface utilisateur en ligne de commande pour jouer une partie ;
- module de tournois générant des statistiques ;
- module d'entrainement 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 existance 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 Exercice 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

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% isSet(A)
% END
    isSet(_, []).

% FOREACH
    isSet(Tested, [X|Remainings]) :-
        not(member(X, Tested)),
        isSet([X|Tested], Remainings).

% INIT
    isSet(List) :-
        isSet([], List).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg2 = set(arg1)
% END
    genSet([], Result, Result).

% FOREACH
    genSet([X|L], Result, BuildedSet) :-
        member(X, BuildedSet) ->
        genSet(L, Result, BuildedSet) ;
        genSet(L, Result, [X|BuildedSet]).

% INIT
    genSet(L,Result) :-
        genSet(L, Result, []).


% all following function are assuming the parameters
% are (un)ordered sets!!!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg3 = arg1 U arg2
% END
    setUnion([], Result, Result).

% FOREACH
    setUnion([X|A], Union, Result) :-
        member(X,Union) ->
        setUnion(A,Union,Result) ;
        setUnion(A, [X|Union], Result).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% arg3 = arg1 inter arg2
```

```
% 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).

% Actualy, 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),                areEqualSets(R21,[a,b]),
genSet([a,b,a,b,c], R22),          areEqualSets(R22,[a,b,c]),

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

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