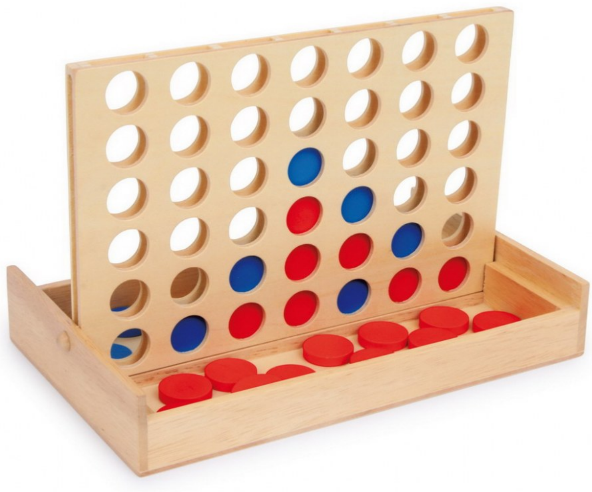
INSA Lyon
20, avenue Albert Einstein
69621 Villeurbanne Cedex

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 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 interogation $4 \in [1, 2, 3]$ 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 objects et/ou constantes. Par exemple :

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

Ce qui en Prolog donne :

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

Ainsi a l'éxécution, Prolog est capable d'évaluer les solutions de X grace a cette propriétée ainsi définie :

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

1.4 Recherche de solutions d'un système

Une propriétée sur une variable par exemple, peut être définit par plusieur 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 dangeureusitée de de la recherche de solution, est qu'il est possible qu'une infinitée de solutions vérifient une meme propriétée. Considerons par exemple le code suivant :

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

Cette est équivalent à $1 \in L$. Mais alors, combien de listes pourraient vérifié 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 éxiste 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 complement 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'abort, on remarque que $X \in [X, ...]$, 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 ailleur 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'implementer un predicat permetant 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'abort, 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 definir un predicat comme echouant (fail) revien a ne pas le definir. 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'idee, est de recursivement prouver *elementAtPos*/3 mais en retirant le premier element a chaque foie :

```
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 jeux

Le puissance 4 est un jeux de societé à deux joueurs. Chaque joueurs doivent, chacun leurs tour, inserer un jeton de leurs couleur dans une des sept colones cote à cote, chacunes aillant une capacité maximal de six jetons. Le but du jeu est d'aligner verticalement, horizontalement ou 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 precedament, puis prévoir de jouer le $4^{\text{ème}}$ au tour suivant. En effet, l'adversaire pourrait bloquer cet alignement, lorsque c'est a 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 realisé

En plus de l'implémentation du module de méchanisme de jeux et réalisation des tests unitaire, nous avons implementé 4 joueurs aillant des stratégies différentes :

— joueur aléatoire ;
— joueur aléatoire munis d'heuristiques ;
— joueur parcourant l'arbre des possibilités ;
— inteligence artificielle apprenant par **moteur d'inférence**, de ses échecs precédent.

Mais aussi :

— interface utilisateur en ligne de commende pour jouer une partie ;
— module de tournoit générant des statistiques ;
— module d'entrainement du moteur d'inférence ;
— module d'étude de l'apprantissage du moteur d'inférence ;
— sauvgarde et chargement de la base de connaissances du moteur d'inférence.