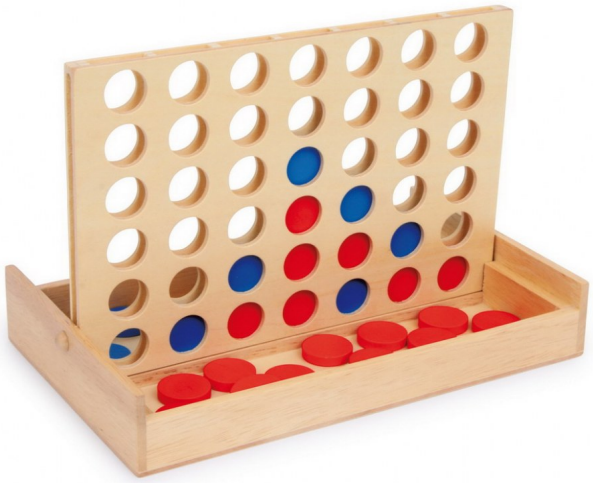
INSA Lyon
20, avenue Albert Einstein
69621 Villeurbanne Cedex

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

Sommaire

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 évident 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`) revient à ne pas le définir. Nous illustrons ici simplement la démarche que nous avons suivie.

Vient 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'idée est de récursivement prouver `elementAtPos/3`, mais en retirant le premier élément de la liste L à 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 fonctionne 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 égale 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

Il est alors facile d'utiliser le prédicat `elementAtPos/3` défini ci-dessus 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]).
```

Nous voyons que 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 être facilement reproduite par :

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

1.9 Conditions

En `Prolog`, il est possible de réaliser des conditions avec la syntaxe suivante :

```
(condition) -> (faire quelque chose) ; (faire autre chose)
```

Codons par exemple, un prédicat liant une liste et un ensemble associé. Trivialement, une liste vide est alors liée à un ensemble vide :

```
genSet([], []).
```

Soit L une liste, S son ensemble associé et X_n tel que $L_n = cat([X_n], L_{n-1})$,
Alors : $S_n = cat([X_n], S_{n-1}) \Leftrightarrow X_n \notin S_{n-1}$ car S_n est un ensemble.
D'où :

```
genSet([X|L], Set) :- member(X, L), genSet(L, Set).
genSet([X|L], [X|S]) :- not(member(X, L)), genSet(L, S).
```

Remarquez alors que les deux prédicats ci-dessus sont complémentaires. Mais si nous avions été dans un cas similaire mais plus complexe, la condition permet alors de factoriser le code, prévenant ainsi les problèmes de maintenabilité du code source :

```
genSet([X|L], S) :- member(X, L) -> genSet(L, S) ; genSet(L, R), S = [X|R].
```

1.10 Causalité

Considérons le code ci-dessous générant une liste ordonnée $[N, N-1, ..., 1]$:

```
generateRList([], 0).
generateRList([I|L], I) :- I1 is I - 1, generate(L, I1).
```

Ensuite essayons simplement :

```
?- generateRList(L, 7), sort(L, R).
L = [7, 6, 5, 4, 3, 2, 1],
R = [1, 2, 3, 4, 5, 6, 7] ;
ERROR: Out of global stack
```

En effet, `Prolog` va essayer de chercher toutes les solutions L et R vérifiant ce prédicat. Alors la première solution est évidente par l'ordre des prédicats. Mais `Prolog` va aussi essayer de chercher des solutions vérifiant cette propriété mais en commençant par évaluer `sort`. Ce dernier étant non déterministe, l'infinité de solutions provoque un dépassement de la pile. Mais en remplaçant un « `,` » (et) par un « `->` » (causalité), on obtient :

```
?- generateRList(L, 7) -> sort(L, R).
L = [7, 6, 5, 4, 3, 2, 1],
R = [1, 2, 3, 4, 5, 6, 7].
?-
```

Ici, nous précisons que si L ne vérifie pas `generateRList(L, 7)`, alors il n'est pas la peine d'aller plus loin.

2. Projet : Puissance 4

2.1 Règles du jeu

Le `Puissance 4` est un jeu de société à deux joueurs. Chaque joueur doit, chacun son tour, insérer un jeton de sa couleur dans une des sept colonnes du plateau de jeu, chacune ayant une capacité maximale de six jetons. On peut voir chaque colonne comme une pile de jetons, puisqu'on ne peut qu'empiler des jetons les uns sur les autres. 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, laissant ainsi le joueur adverse contre l'inévitable : il ne pourra plus contrer ces alignements en un seul jeton.

2.3 Travail réalisé

En plus de l'implémentation du module du mécanisme de jeu et de la réalisation des tests unitaires, nous avons implémenté 4 joueurs autonomes, dont une intelligence artificielle :

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

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.