INF8215 - Intelligence Artificielle TP2
Utilisation de Prolog

Mohammed Najib Haouas & Aymane Timilla & Flora Ferreira

Septembre 2017

Introduction

Prolog est un langage de programmation fonctionnel logique. L'utilisation du programme fait intervenir une série d'interrogations sur une base de connaissance. Prolog parcourt elle-ci pour déduire les solutions qui vérifient l'ensemble des connaissances qui définissent le monde qu'on a programmé dans cette base. Pour cela, Prolog repose sur l'algorithme de chaînage arrière ainsi que sur la logique de prédicats. Notre travail ci-dessous explore certaines capacités de ce programme.

1 Exercices de départ (parties 3.3, 3.4, 3.5, 3.6)

1.1 Partie 3.3

```
?- possede(X,chat).
X = jean.

?- possede(pierre,X).
X = chien ;
X = cheval.
```

Jean possède un chat, Pierre possède un chien et un cheval.

1.2 Partie 3.4

```
?- amis(X,Y),amis(Y,X).
X = pierre,
Y = jean ;
X = jean,
Y = pierre ;
false.
```

Jean et Pierre sont amis mutuellement.

1.3 Partie 3.5 et 3.6

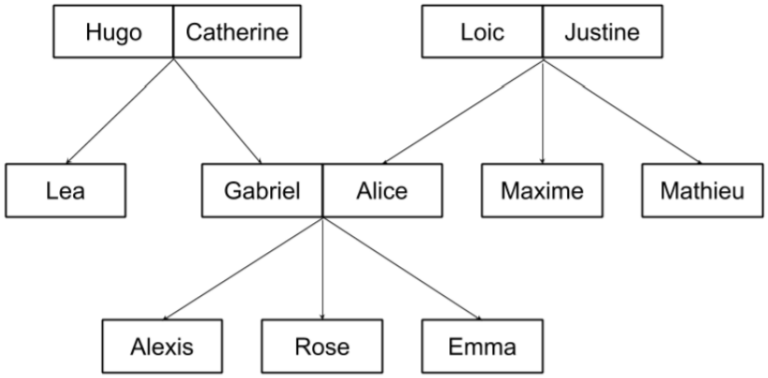
```
?- amis_2(pierre,X).
X = pierre ;
X = paul ;
X = jacques.
```

Paul et Jacques sont tous les deux des amis au deuxième degré de Pierre. Le `X = Pierre` est une aberration que l'on corrige ci-dessous avec le prédicat `X \== Z` :

```
?- amis_2(pierre,X).
X = paul ;
X = jacques.
```

2 Représentation d'un arbre généalogique (Partie 3.7)

On définit l'arbre généalogique suivant en Prolog :



Pour cela, on définit les prédicats `homme/1`, `femme/1` et `parent/2`.

On définit les relations qui peuvent lier 2 personnes à partir de ces trois propriétés (voir le code pour les prototypes) :

1. `enfant/2` : X est enfant de Y ssi Y est parent de X.
2. `fille/2` : X est enfant de Y et X est une femme.
3. `fils/2` : X est enfant de Y et X est un homme.
4. `mere/2` : X est parent de Y et X est une femme.
5. `pere/2` : X est parent de Y et X est un homme.
6. `grand_parent/2` : Il existe Z tel que X est parent de Z, et Z est parent de Y.
7. `grand_mere/2` : X est grand parent de Y et X est une femme.
8. `grand_pere/2` : X est grand parent de Y et X est un homme.
9. `petit_enfant/2` : Y est grand parent de X.
10. `petite_fille/2` : Y est grand parent de X et X est une femme.
11. `petit_fils/2` : Y est grand parent de X et X est un homme.

12. `soeur/2` : X est une femme, X a un parent commun Z avec Y, Z est une femme. On oblige Z a être une femme pour ne pas avoir la répétition des frères et soeur, ainsi on a une seule possibilité de trouver que Alice est soeur de Maxime, pour Z = Justine.
13. `frere/2` : X est un homme, X a un parent commun Z avec Y, Z est un homme. On impose le fait que Z est un homme cette fois par soucis d'égalité.
14. `tante/2` : il existe Z tel que X est soeur de Z et Z est parent de X.
15. `oncle/2` : il existe Z tel que X est frère de Z et Z est parent de X.
16. `niece/2` : X est une femme, et Y est tante ou oncle de X.
17. `neveu/2` : X est un homme, et Y est tante ou oncle de X.

Pour rendre le code plus flexible et plus simple, certaines de ces fonctions s'appellent entre-elles. Par exemple, `petit_fils/2` appelle `petit_enfant/2` ou encore `grand_mere/2` appelle `mere/2`.

Par ailleurs, il a fallu faire attention aux fonctions `frere/2` et `soeur/2`. En effet, pour déterminer les frères et soeurs, on a eu recourt à `parent/2` deux fois (cela est dû au fait que deux personnes sont frère et soeur si il ont le même parent. Avec seulement cette condition, il est possible qu'un personne soit son propre frère ou soeur. Il a fallu ajouter une condition supplémentaire pour éviter cela (individus avec le même parent différents l'un de l'autre). De plus, vu que chacun a deux parents, on s'est fié juste au parent de sexe masculin. Cela évite les doublons.

Voici quelques utilisations de ces fonctions :

```
?- neveu(X,maxime).
X = alexis ;
false.

?- neveu(alexis,X).
X = maxime ;
X = mathieu ;
X = lea ;
false.

?- frere(X,mathieu).
X = maxime ;
false.

?- frere(mathieu,X).
X = alice ;
X = maxime ;
false.
```

```
?- grand_parent(X,rose).
X = hugo ;
X = catherine ;
X = loic ;
X = justine ;
false.

?- grand_parent(X,alice).
false.

?- pere(loic,X).
X = alice ;
X = maxime ;
X = mathieu.

?- pere(X,gabriel).
X = hugo ;
false.
```

3 Calculs mathématiques (Partie 4.3)

On utilise Prolog ici pour définir plusieurs fonctions mathématiques.

3.1 Somme

On commence par définir la fonction `sum/3` qui prend 3 paramètres. Le dernier paramètre devient une évaluation de l'addition des deux premiers.

3.2 Maximum

On définit ici `max2/3` et `max2/4` qui retournent dans le dernier paramètre le plus grand élément parmi les 2 ou 3 premiers arguments, respectivement.
Pour `max/3`, on a deux prédicats distincts. Le premier prédicat retourne le premier élément dans le 3e argument seulement si le premier élément est plus grand ou égal au deuxième. Si cela s'avère faux, le deuxième prédicat est évalué et celui-ci place systématiquement le deuxième argument à la place du troisième. Un *cut* `!` assure que le deuxième prédicat de `max/3` ne sera pas évalué si le premier s'avère vrai.

De façon similaire, un premier prédicat de `max/4` détermine si le 3e élément indiqué est plus petit que l'un ou l'autre des deux autres arguments. Si tel est le cas, ce n'est pas un max. On appelle alors `max/3` sur les 3 autres arguments restants. Le second prédicat retourne quant à lui systématiquement le 3e argument à la place du 4e car si le premier prédicat échoue, c'est sûrement un max. Pareil encore une fois, un *cut* `!` assure que le deuxième prédicat ne sera pas évalué si le premier s'avère vrai.

3.3 Dérivation de polynômes

Afin de réaliser cette opération, il est nécessaire de diviser le travail en plusieurs tâches plus simples et de faire valoir la capacité de Prolog à réaliser des tâches récursives. A la fin, on veut qu'un simple appel à `d/3` puisse réaliser la dérivée du premier argument par la variable au deuxième argument. Notre méthode repose sur les hypothèses initiales suivantes :

1. Le premier argument est un polynôme composé de monômes simples de la forme $a \cdot x^n$ avec a et n des chiffres entiers ou décimaux (et non des expressions) et x la variable de dérivation.
2. Les variables indiquées doivent être nommées par une seule lettre.

On distingue ces prochains cas de figure de base pour un appel à `d/3` du type `d(Y,X,R)` :

1. Si `Y` est une constante, `R` est 0.
2. Si `Y` est `X`, `R` est 1.
3. Si `Y` est de la forme `A*X`, `R` est `A`.
4. Si `Y` est de la forme `X^B`, `R` est `B*X^(B-1)` (expression évaluée).
5. Si `Y` est de la forme `A*X^B`, `R` est `AB*X^(B-1)` (expression évaluée).

Finalement, on définit un prédicat supplémentaire qui sépare les polynômes donnés et qui appelle récursivement les précédents prédicats sur chaque monôme. Celui-ci unifie `Y` à `A+B` et appelle deux fois `d/3` sur `A` et sur `B` respectivement.

Pour rendre l'affichage plus concis, on enlève les monômes nuls, les puissances de 1 ainsi que les multiplications par 1. Ces prochains prédicats, intercalés entre ceux précédents, permettent de réaliser cela :

1. Si `Y` est de la forme `X^2`, `R` est `2*X`.
2. Si `Y` est de la forme `A*X^2`, `R` est `2A*X` (expression évaluée).
3. Si `Y` s'unifie à `A+B` et que `A` ne ressemble pas à un monôme comme définit plus haut, on appelle `d/3` uniquement sur `B`. La même opération est réalisée dans un autre prédicat subséquent si `B` n'est pas un monôme attendu.

A la fin, nous ajoutons un dernier prédicat par défaut qui retourne 0 pour n'importe quel X et Y. Celui-ci traite le cas où un monôme est non reconnu ou dont la variable n'est pas celle de dérivation.

3.4 Exemple d'exécution

```
?- sum(5,4,R).
R = 9.

?- max2(2,4,R).
R = 4.

?- max2(22,4,R).
R = 22.

?- max2(22,4,33,R).
R = 33.

?- max2(22,44,33,R).
R = 44.

?- max2(222,44,33,R).
R = 222.

?- d(5,x,R).
R = 0.

?- d(5*y,x,R).
R = 0.

?- d(2*y^5+x^3+3*y+7*x^4+x^2+5,x,R).
R = 3*x^2+28*x^3+2*x.

?- d(2*y^5+x^3+3*y+7*x^4+x^2+5,y,R).
R = 10*y^4+3.
```

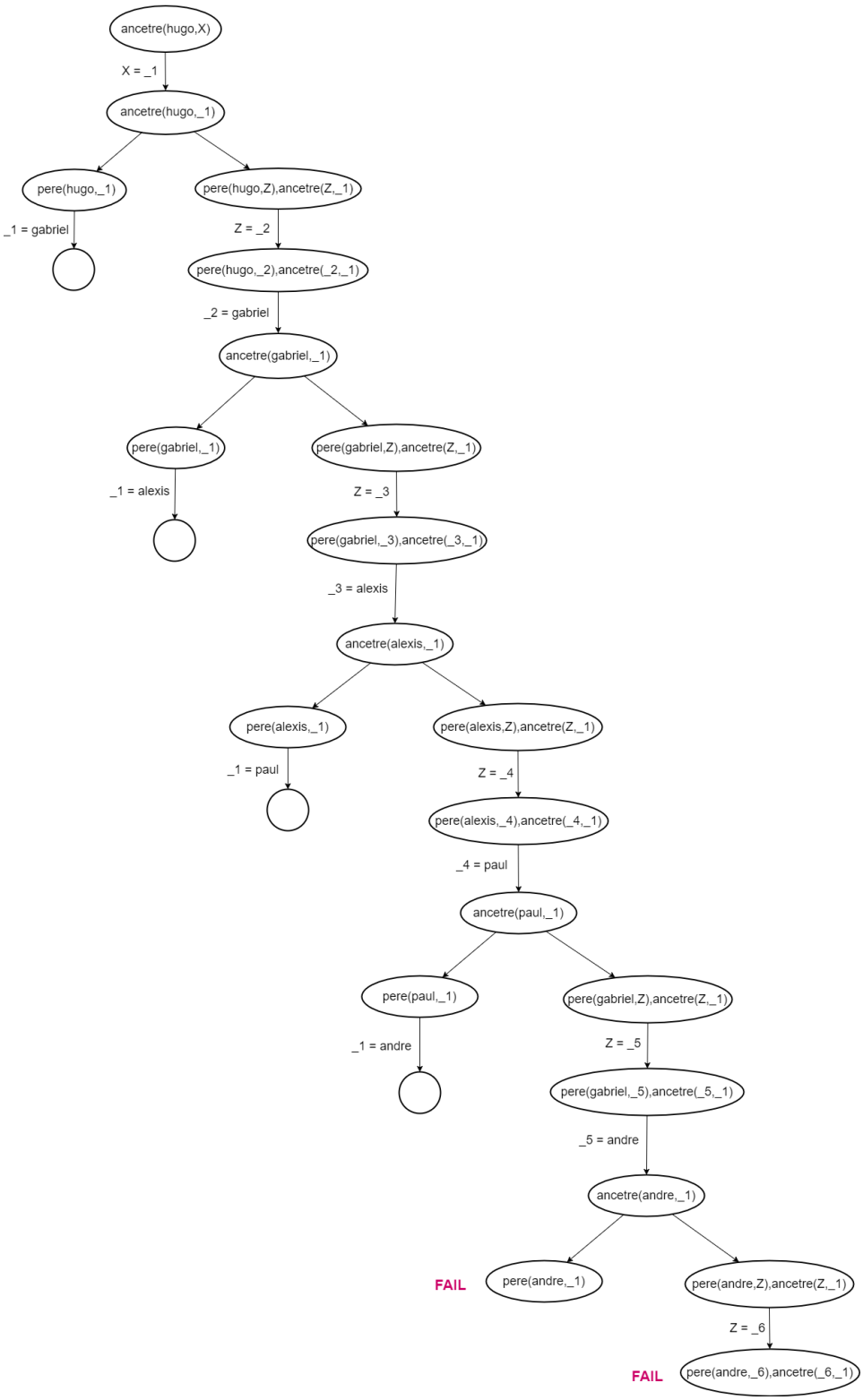
4 Exercices sur le chaînage arrière (Partie 4.5)

4.1 Arbre de recherche (Partie 4.5.1)

On utilise la base de connaissances suivante :

```
pere(hugo,gabriel).
pere(gabriel,alexis).
pere(alexis,paul).
pere(paul,andre).
ancetre(X,Y):-pere(X,Y).
ancetre(X,Y):-pere(X,Z),ancetre(Z,Y).
```

Ci-dessous, l'arbre de recherche de Prolog associé à la requête `ancetre(hugo,X).` :



Cette requête renvoie `X = gabriel; X = alexis; X = paul; X = andre` en Prolog, ce qui correspond bien aux noeuds vides de notre arbre.

4.2 Mots Croisés (Partie 4.5.2)

Dans cet exercice on commence par définir 6 prédicats relatifs aux mots à insérer avec comme premier argument le mot en entier, suivit de ses lettres en arguments subséquents.

L'idée est de rassembler une solution qui fait croiser ces mots. De ce fait, nous auront 9 points d'intersections. Chacun de ces points doit correspondre à une même lettre (dont la position est connue) dans exactement deux mot différents.

On construit alors un prédicat avec les 6 prédicats (ET) des 6 mots. On s'arrange pour que les intersections correspondent. La solution est donnée par le premier argument de chacun de ces 6 prédicats.

On peut insérer un *cut* `!` à la fin du prédicat de résolution `crossword/6` si on ne veut qu'une seule solution. Il va toujours y avoir au moins deux solutions car une deuxième solution peut toujours être obtenue par rotation.

```
?- crossword(V1,V2,V3,H1,H2,H3).
V1 = abalone,
V2 = anagram,
V3 = connect,
H1 = abandon,
H2 = elegant,
H3 = enhance ;
V1 = abandon,
V2 = elegant,
V3 = enhance,
H1 = abalone,
H2 = anagram,
H3 = connect ;
false.
```

5 Utilisation des listes et de la récursivité (Partie 4.8)

Dans cette partie on exploite les propriétés des listes afin de réaliser certaines tâches :

5.1 Maximum d'une liste

On fait usage ici de `max2/3` définie plus tôt. On définit alors deux prédicats de `max/2` :

1. Un premier prédicat de fin de récursion qui prend en argument une liste de deux éléments et qui retourne dans son deuxième argument l'élement le plus grand (à l'aide de `max2/3`). Celui-ci finit avec un *cut* `!` qui marque l'arrêt de récursion (plus besoin d'évaluer `max/2` d'avantage).
2. Un deuxième prédicat qui appelle `max2/3` sur les deux premiers éléments de la liste et qui appelle `max/2` de façon récursive sur ce résultat attaché au restant de la liste à tester.

De cette manière, à chaque appel récursif de `max/2`, la liste se débarrasse de ses éléments les plus petits jusqu'à ce qu'un seul élément reste : le maximum.

5.2 Somme des éléments d'une liste

De façon tout à fait analogue à `max/2`, on effectue les mêmes opérations sauf qu'au lieu de faire appel à `max2/3`, on fait appel à `sum/3` définie plus tôt.

Ces appels récursifs font la somme de la liste en traitant toujours les deux premiers éléments à la fois, jusqu'à ce que tous les éléments soient additionnés.

5.3 N-ème élément d'une liste

On se base sur une indexation à 0. De même que pour les deux premières fonctions, on a besoin de deux prédicats :

1. Un premier prédicat de fin de récursion qui prend en premier argument un 0 et qui retourne dans le troisième argument le premier élément dans la liste au deuxième argument. Celui-ci finit avec un *cut* `!` qui marque l'arrêt de récursion (plus besoin d'évaluer `nth/3` d'avantage).
2. Un deuxième prédicat qui vérifie que le premier argument est un nombre positif, qui décrémente celui-ci de 1 et qui appelle `nth/3` de façon récursive sur la liste démunie de son premier élément à l'aide de l'indice décrémenté.

Cet appel récursif ramène l'élément recherché à l'indice 0, ce qui permet au premier prédicat de `nth/3` de le retourner. La fonction est telle que si l'indice indiqué n'est pas entier ou est en dehors des limites de taille permises, elle retourne `false`. Cela est dû au fait que le prédicat de fin de récursion ne sera jamais atteint dans ces cas avec des paramètres valides (ie la liste deviendra vide à un moment et ne correspondra à aucune définition de `nth/3`).

5.4 Zip

Dans cette opération, on construit une liste de listes qui rassemble des éléments deux à deux provenant de deux listes différentes. Là encore, deux prédicats sont nécessaires :

1. Un premier prédicat de fin de récursion qui effectue l'opération sur deux listes distinctes d'un élément chacune.
2. Un deuxième prédicat qui construit la liste des deux éléments réunis issus des deux listes indiquées, qui appelle récursivement `zip/3` sur le restant des listes et qui construit la liste finale en y insérant les éléments réunis ainsi que le résultat de l'appel récursif de `zip/3`.

De cette façon, on construit la réponse en commençant par le dernier couple jusqu'au premier. Cela est nécessaire si on veut utiliser la construction `[U|V]`. Par ailleurs, la fonction est telle qu'elle retourne `false` si les tailles des listes donnés ne correspondent pas (ie ne sont pas égales).

5.5 Énumération

Dans cette opération, on fait l'énumération des entiers entre 0 et l'entier indiqué (exclu). Pour ce faire, on définit :

1. Un premier prédicat de fin de récursion qui prend en premier argument un 0 et qui retourne une liste vide (en effet, on n'énumère rien à 0).

2. Un deuxième prédicat qui vérifie que le premier argument est positif, qui le décrémente, qui appelle `enumerate/2` de façon récursive avec cet argument décrémenté et qui construit la liste d'énumération à l'aide de `append/3`.

L'usage de `append/3` est important pour pouvoir insérer les nombres qui viennent par ordre croissant de la droite plutôt que de la gauche avec la construction `[U|V]`. De plus, on ne met pas de *cut* `!` à la fin du prédicat de fin de récursion pour pouvoir utiliser `w` afin de `write` les longues listes. Enfin, le prédicat est construit de telle sorte que si le premier argument n'est pas un entier, `false` sera retourné (à cause de la vérification du signe du premier argument qui échoue après un certain nombre de récursions).

5.6 Monnaie

Cette opération repose sur un appel récursif dont le nombre de récursions est connu à la base. En effet, on fera autant de récursions qu'il y a de dénominations dans la devise Canadienne (5 au total pour \$2, \$1, \$0.25, \$0.10 et \$0.05).

On construit les prédicats `rend_monnaie/2` qui admettent deux paramètres : l'argent donné M et ensuite le prix P. Ces prédicats se ressemblent tous et effectuent dans l'ordre les opérations suivantes :

1. Calculer la différence entre M et P et voir si celle-ci est plus grande que la dénomination en cours de traitement.
2. Si oui, calculer la partie entière de la division entre cette différence et la dénomination en cours de traitement.
3. Enlever l'argent remis de M pour le prochain appel récursif.
4. Afficher un message avec le nombre de pièces remis ainsi que la dénomination remise seulement si le nombre de pièces à remettre n'est pas 0.
5. Appeler récursivement `rend_monnaie/2` avec le nouveau M.

De plus, on insère des *cut* `!` à la fin de chacun de ces prédicats pour empêcher une prochaine évaluation. On veut que tous ces prédicats ne soient évalués qu'une seule fois uniquement (on ne sait pas non plus à l'avance quels prédicats seront vrais ou faux).

En outre, on insère un prédicat initial de fin de récursion qui prévoit le cas où l'argent à remettre est de valeur plus petite que la plus petite dénomination possible. Si il n'y a plus rien à remettre, le travail est achevé. Si l'argent donné est plus petit que le prix, le résultat est `false`.

Finalement, il est devenu évident que les erreurs de troncature sont significatives sur Prolog, ce qui fausse les calculs (`rend_monnaie(26.65,20)` par exemple est impossible à calculer sans prendre certaines considérations). Pour contrer cela, on a choisi d'ajouter à l'argent donné en entrée une petite valeur de 0.001. Cette opération n'aura aucun impact sur les calculs mais fera que l'évaluation de la partie entière de la division mise en oeuvre soit toujours correcte et immune aux erreurs de troncatures. En effet, ces dernières tendent à rendre un nombre entier très légèrement plus petit.

5.7 Exemple d'exécution

```
?- max([1,2,3,4],M).
M = 4.

?- max([1,22,3,4],M).
M = 22.

?- somme([4,6,10,9],S).
S = 29.

?- nth(0,[1,2,3,4],R).
R = 1.

?- nth(5,[1,2,3,4],R).
false.

?- nth(3,[1,2,3,4],R).
R = 4.

?- enumerate(0,R).
R = [] .

?- enumerate(10,R).
R = [0, 1, 2, 3, 4, 5, 6, 7, 8|...] [write]
R = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] .
```

```
?- zip([a,b,c],[5,7,11],R).
R = [[a, 5], [b, 7], [c, 11]].

?- rend_monnaie(26.65,20).
3 piece de 2
2 piece de 0.25
1 piece de 0.10
1 piece de 0.05
true.

?- rend_monnaie(26.75,20).
3 piece de 2
3 piece de 0.25
true.

?- rend_monnaie(19.75,20).
false.
```

6 Résolution de problèmes de contraintes (Partie 4.9)

Dans cette partie, on fait appel à la librairie CLPFD, « Constraint Logic Programming over Finite Domains. » Celle-ci permet de résoudre des problèmes de satisfaction de contraintes directement sur Prolog.

6.1 Stratégie de résolution

Il s'agit ici de traduire l'énoncé en un problème de satisfaction de contraintes. Il est alors nécessaire de définir des variables avec leur domaine ainsi que des contraintes qui lient ces variables entre-elles.

Variables On commence par définir une liste de variables qui représente chacune une information. Ainsi, des informations comme *Maison Rouge* ou *A un serpent* se présenteront dans notre implémentation comme des variables (par exemple `MaisonRouge` et `Serpent` respectivement). Il ne faut pas oublier que la question contient elle-même des informations à traduire en variables. On suggère de consulter le code pour la liste complète des variables pour les deux problèmes.

Domaines Avant de définir les domaines en tant que tels, il est nécessaire définir une interprétation valide des valeurs quant aux variables. Dans notre implémentation, on entend par les *valeurs* des variables *quel individu n lui est attribuée telle ou telle variable*. Ceci revient à dire quel individu n lui est attribuée telle ou telle caractéristique. La solution est trouvée alors en faisant correspondre toutes les variables qui se retrouvent avec la même valeur à la fin de la résolution : cette valeur lie les

variables au même individu. C'est-à-dire, cette valeur assigne à l'individu n un ensemble d'attributs. Le domaine au final, et ce pour chacune des variables X, est l'ensemble d'entiers allant de 1 à m, m étant le nombre d'individus.

$$D_X = \{1, 2, ..., m\}$$

Par défaut aussi, on numérote les individus dans l'ordre de leur maison, celle la plus à gauche étant numéro 1. Ainsi, l'individu qui y demeure est l'individu 1.

Contraintes Certaines caractéristiques sont mutuellement exclusives. Ainsi, dans notre problème, un même individu ne peut avoir deux animaux différents ou encore un même individu ne peut être citoyen de deux pays différents. On appelle deux groupes de variables mutuellement exclusifs *des catégories*. Un exemple de catégorie est *Animal de compagnie*. Cette contrainte d'exclusivité se traduit par une contrainte *tous différents* : un individu ne peux avoir deux caractéristiques issues de la même catégorie. Pour les deux problèmes, et pour chacune des catégories C_Y, on force une telle contrainte :

$$\text{AllDiff}(C_Y) \qquad \text{avec} \qquad C_Y = \{\texttt{Var\_1}, \texttt{Var\_2}, ..., \texttt{Var\_m}\}$$

Le reste des contraintes provient des informations fournies dans le texte de l'énoncé :

1. Si un même individu d'attribut `X` dans une catégorie C_M doit aussi avoir un autre attribut `Y` dans une autre catégorie C_N, alors nécessairement $\texttt{X} = \texttt{Y}$.
2. Si une certaine maison, ou toute autre caractéristique `X`, est située dans l'espace de façon absolue à la position u, alors nécessairement $\texttt{X} = u$. Cette contrainte intervient aussi dans les énoncés « telle maison est située au milieu » (cf convention établie dans *Domaines*).
3. Si une certaine maison, ou toute autre caractéristique `X`, est située dans l'espace de façon relative à une autre `Y` de u unités à droite, alors nécessairement $\texttt{X} = \texttt{Y} + u$. Si c'est à gauche, on retranche u (cf convention établie dans *Domaines*).

6.2 Implémentation

Les deux problèmes sont identiques en matière d'implémentation. On commence par inclure la librairie CLPFD puis on définit deux grands prédicats principaux `probleme1/4` et `probleme2/7` dont les prototypes sont les suivants :

— `probleme1(Anglais,Espagnol,Japonais,Serpent)`
— `probleme2(Anglais,Espagnol,Ukrainien,Norvegien,Japonais,Eau,Zebre)`

L'inclusion des variables de la catégorie *Nationalité* dans la liste des arguments permet de faciliter l'identification de la solution. Ainsi, le(s) variable(s) *Nationalité* qui partage(nt) la même valeur avec `Serpent` dans le problème 1 ou avec `Eau` et `Zebre` dans le problème 2 sont les individus recherchés dans le cadre de la solution.

Chacun des deux prédicats est composé des éléments suivants comme il suit, liés avec des ET :

1. On commence par déclarer une liste de toutes les variables (voir plus haut).
2. On définit le domaine de toutes ces variables en utilisant `ins/2` sur une variable qu'on unifie à la liste précédente.
3. On traduit ensuite chacune des déclarations contenues dans l'énoncé en prédicat légal sur Prolog avec CLPFD incluse (égalité avec `#=/2`, OU avec `#\//2`)

4. On insère autant de prédicats `all_different/1` qu'il y a de catégories de variables. Chacune de ces catégories en paramètre se présente comme une liste de ses variables-membre.
5. Finalement, on insère un dernier prédicat `label/1` qui prend en argument la liste de toutes les variables du problème. C'est ce prédicat qui mène à la résolution du problème en tentant systématiquement des valeurs dans le domaine des variables jusqu'à ce que toutes les valeurs soient satisfaisantes. `label/1` et un cas particulier de `labeling/2` (sans options).
6. On termine les prédicats relatifs aux problèmes avec un *cut* `!` car une seule solution est nécessaire pour conclure.

6.3 Problème 1 : Qui possède le serpent ?

```
?- probleme1(Anglais,Espagnol,Japonais,Serpent).
Anglais = 1,
Espagnol = 3,
Japonais = Serpent, Serpent = 2.
```

C'est le japonais qui possède le serpent.

6.4 Problème 2 : Qui boit de l'eau ? Qui possède le zèbre ?

```
?- probleme2(Anglais,Espagnol,Ukrainien,Norvegien,Japonais,Eau,Zebre).
Anglais = 3,
Espagnol = 4,
Ukrainien = 2,
Norvegien = Eau, Eau = 1,
Japonais = Zebre, Zebre = 5.
```

Le norvégien bois de l'eau tandis que le Japonais a un zèbre.

Conclusion

On aura vu lors de ce TP comment utiliser le paradigme ainsi que les fonctions de Prolog afin de réaliser des tâches et raisonnements logiques plus ou moins poussés. On aura vu au final plusieurs applications assez variées sur différents problèmes. On a pu ainsi utiliser le programme pour effectuer diverses opérations mathématiques comme la somme, le max et la dérivée. On a aussi utilisé des méthodes récursives pour réaliser des problèmes plus complexes. Enfin, on a mis en oeuvre une résolution de problèmes de programmation par contraintes en utilisant la bibliothèque CLPFD.