Rapport préliminaire ALIA

|  |  |  |  |
| --- | --- | --- | --- |
|  | Nom | Signature | Date |
| Rédigé par | B. DELERUE & A. MERRY DE MONTIGNY |  | 2012/09/27 |
| Validé par |  |  |  |

Contenu

[1. Entrée en matière 3](#_Toc336895729)

[2. Prise en main 3](#_Toc336895730)

[1. membre(a,L) : 3](#_Toc336895731)

[2. Règle append : 3](#_Toc336895732)

[3. Single(X) : 3](#_Toc336895733)

[3. Thème : Généalogie 3](#_Toc336895734)

[1. Introduction 3](#_Toc336895735)

[2. Règle 1 : grandParent 4](#_Toc336895736)

[3. Règle 2 : oncle et tante 4](#_Toc336895737)

[4. Règle 3 : ancetres 4](#_Toc336895738)

[5. Règle 4 : dernierAncetres 4](#_Toc336895739)

[6. Règle 5 : arbregenea 5](#_Toc336895740)

[7. Règle 6 : nbDesc 5](#_Toc336895741)

[4. Sujet de projet 6](#_Toc336895742)

# Entrée en matière

Comme Il a été dit pendant le cours, Prolog n’a rien à voir avec les langages conventionnels. Il paraît donc nécessaire de bien appréhender son fonctionnement avant de se lancer dans quelque projet un peu plus conséquent.

Pour ce faire, nous avons d’abord analysé et testé un certain nombre de règles qui nous ont été fournies au début du TP pour finir par créer nos propres règles sur un système de généalogie.

# Prise en main

Les tests des règles préalablement écrites dans le fichier intro\_prolog.pl nous a permis de démystifier une partie du fonctionnement de Prolog. Nous ne parlerons dans cette partie que des règles les plus intéressantes à noter.

## membre(a,L) :

La particularité de cette requête réside dans le fait que L est une variable ainsi le programme cherche la totalité des listes possibles contenant un a. Les solutions sont infinies.

## Règle append :

Cette règle permet la concaténation des listes et son fonctionnement vaut le coup de s’y arrêter un peu. Ainsi dans cet algorithme, nous avons trois listes L1, L2 et L3. L1 et L2 sont les deux listes à concaténer et L3 est le réceptacle pour concaténer les deux listes. Le principe de la règle est de copier membre à membre le contenu de L1 dans L3 puis une fois que L1 est vide, on copie tout ce qu’il y a de L2 dans L3 pour satisfaire la contrainte *append([], L1, L1).*

## Single(X) :

Cet exemple est édifiant. En effet, il met en évidence l’importance de l’enchainement des contraintes. Dans ce cas, nous pouvons voir que le compilateur cherche premièrement les personnes mariées. En l’occurrence, il n’y a que tom qui est marié donc le programme le trouve et dit qu’il n’est pas bon car il est marié. Les autres n’étant pas mariés, le programme les ignore. En revanche, si nous intervertissons les deux parties de la règle, *single(X)* retourne Lisa et Bob car il cherche les hommes et les femmes puis regarde s’ils sont mariés.

# Thème : Généalogie

## Introduction

Nous avons choisi de prendre ce thème de par la représentation mentale assez simple des résultats. Cela nous permettra entre autres de bien comprendre le fonctionnement des algorithmes utilisés pour effectuer une requête dans la base de données que nous avons constituée. Ainsi, nous allons essayer d’appliquer un maximum de façon de construire une requête comme nous l’avions vu précédemment. Il est intéressant de noter que toutes les relations que nous avons établies entre les données peuvent se rapporter à un arbre avec de multiples racines.

## Règle 1 : grandParent

Dans une mesure de précaution, nous avons voulu coder une règle simple afin de tester les fondements de base. C’est pourquoi nous avons écrits la règle GrandParent sans support :

*grandParent(X,Y):-parent(X,Z),parent(Z,Y).*

Cela nous a permis de tester rapidement et simplement la jointure. Ainsi une requête *grandParent(X,Y)* donne toutes les combinaisons de grands-parents et petits-enfants indifféremment du sexe des deux parties. *grandParent(x,Y)* donne tous les petits-enfants de x et *grandParent(X,y)* donne tous les grands-parents de y.

## Règle 2 : oncle et tante

Sur cette requête notre but fût de compliquer un peu plus les choses en introduisant les structures logique ET et OU dans la même requête. Nous pouvons ainsi gérer plusieurs conditions nécessaires à la validation de la requête posée.

*oncle(X,Y):-homme(X),(frereousoeur(X,Z);frereousoeur(Z,X)),parent(Z,Y).*

*tante(X,Y):-femme(X),(frereousoeur(X,Z);frereousoeur(Z,X)),parent(Z,Y).*

Nous pouvons ainsi rechercher tous les oncles et tantes d’un enfant via la recherche de ses parents dans un prédicat frereousœur. La seule difficulté que nous avons rencontrée dans l’écriture de cette règle fut de savoir comment gérer les Et et OU ensemble.

## Règle 3 : ancetres

Cette requête nous a permis de tester les calculs de fermetures transitives vus en cours. Ainsi, nous avons codé une règle permettant de trouver tous les couples d’ancêtre et descendant quel que soit le nombre de générations.

*ancetres(X,Y):-parent(X,Y).*

*ancetres(X,Y):-parent(Z,Y),ancetres(X,Z).*

Cet exemple très similaire à l’un des exemples dans intro\_prolog.pl ne fut pas dur à écrire. Il est intéressant de noter que si l’on change l’ordre des deux lignes, on peut noter avec la trace que les résultats n’ayant qu’une génération d’écart sont affichés en dernier.

Lorsque nous effectuons une requête sur cette règle le programme recherche soit la totalité des couples ancetre/descendants lorsqu’on ne fixe aucune variable soit tous les descendants d’une personne ou tous ses ancêtres en fonction de quelle variable s’est transformée en constante.

## Règle 4 : dernierAncetres

Avec cette règle nous voulions voir si l’on était capable de retrouver n’importe quel ancêtre de fin de généalogie pour une personne donnée. Nous avons donc commencé à introduire les notions de not et du caractère marquant l’indifférence afin de satisfaire à ce problème.

*dernierAncetres(X,Y):-parent(X,Y),not(parent(\_,X)).*

*dernierAncetres(X,Y):-parent(H,Y),dernierAncetres(X,H).*

Au départ, nous avions commencés à chercher une règle dont la condition de fin ne soit pas gérée via l’utilisation d’un NOT. Ce fut un peu laborieux si bien que l’ajout du not nous apparut naturellement (pour aller au bout d’un chemin, il est plus facile de chercher s’il n’y a plus rien au bout que de regarder si l’on peut continuer).

Ainsi, *dernierAncetres(X,y)* retourne les racines généalogiques d’un individu et *dernierAncetres(x,Y)* retourne les individus pour lesquels x est leur ancêtre terminal. *dernierAncetres(X,Y)* cherche quant à lui la totalité les couples dernier ancêtre/descendants. Le résultat est par conséquent un peu plus restreint que ancetres.

## Règle 5 : arbregenea

Cette règle a été conçue pour pouvoir manipuler les listes. Notre but était de pouvoir lister la descendance d’un individu. Pour cela nous faisons appel à deux règles différentes : arbregenea2 qui permet de faire la recherche et arbregenea qui permet de modifier l’affichage du résultat.

*arbregenea2(X,\_,[X]):-not(parent(X,\_)).*

*arbregenea2(X,Y,[X|L]):-parent(X,Y),arbregenea2(Y,\_,L).*

*arbregenea(X,Z):-arbregenea2(X,Y,Z).*

Au départ, nous avions effectués des tests par analogie du poly mais rien ne fonctionnait. Il s’est avéré plus tard que nous nous étions trompés sur la constante utilisée, nous avions mis une majuscule au début (c’était un nom donc nous l’avions mise machinalement). Une fois que nous avons décelé ce problème, le temps était venu de se familiariser avec les listes. Nous nous sommes rendu compte que notre algorithme s’arrêtait après une génération. Le problème venait d’une mauvaise gestion de la liste visant à réceptionner nos résultats. Une chose qui nous posait problème fût la présence d’une barre verticale à la place de la dernière virgule. Nous ne comprenions pas cette erreur et nous pensions que ce n’était qu’un bug du programme. Il s’est révélé que cette barre ne représentait pas une virgule mais bel et bien autre chose. Elle signifiait que l’élément en fin de tableau n’était pas une liste mais un élément propre. Nous avons donc changé ce qui était *arbregenea2(X,\_,X):-not(parent(X,\_))* en *arbregenea2(X,\_,[X]):-not(parent(X,\_))* pour résoudre ce problème.

Cette règle permet de retourner la liste des descendants d’un individu. Si X est libre, le programme renvoi la totalité des combinaisons entre ancêtre/descendants sinon il renvoi la descendance du même individu. D’un autre côté, si l’on met en second membre une liste de personnes, la requête renverra faux si ce n’est pas une descendance du plus ancien au plus jeune sinon elle affichera l’ancêtre de la liste.

## Règle 6 : nbDesc

Par curiosité, nous avons voulu appliquer la règle longueur de l’exemple sur notre chaîne retournée par arbregenea. Cela avait pour but de compter le nombre de personnes que compte chaque descendance pour l’individu X.

*longueur([],0).*

*longueur([\_|Q],N) :- longueur(Q,N1), N is N1+1.*

*nbDesc(X,N) :- arbregenea(X,Z), longueur(Z,N).*

Cet exemple ne nous a pas posé trop de difficulté mis à part que c’est cet exemple qui nous a permis de déceler l’erreur de formatage de liste de arbregenea. Nous n’avions avec cet règle qu’une intention de manipuler une liste variable dont les éléments seraient comptés dans un second temps. C’est pourquoi nous avons réutilisé du code fourni au début du TP.

Ainsi cette règle renvoie le nombre de générations qui composent une relation ancêtre = X/descendants. Une fois avoir appliqué cela et vu comment cela fonctionnait en détail, nous avons réécrits cette règle d’une manière différente histoire d’assimiler correctement le comptage.

*nbDesc2(X,1):-not(parent(X,\_)).*

*nbDesc2(X,N):-parent(X,Y),nbDesc2(Y,N1), N is N1+1.*

Cette règle fait exactement la même chose que la précédente sans manipuler de listes et en ayant simplifié son fonctionnement.

Ainsi, une requête sur deux variables renvoie le nom du dernier ancêtre de la liste et le nombre de générations qu’il compose avec chaque ramification dans ses descendants.

# Sujet de projet

Nous aimerions travailler sur un projet de Blackjack. Le but serait de créer à chaque round une liste composée de chaque carte d’un jeu de 54 cartes classique puis de tirer aléatoirement celles-ci dans la liste sans remise après (on retire la carte piochée de la liste).

Nous commencerions par un seul joueur qui jouera contre le dealer. Le joueur aura donc le choix de continuer à tirer des cartes (dont la valeur sera calculée à chaque pioche) afin de se rapprocher du nombre 21 sans le dépasser. Une fois qu’il sera resté sur son chiffre ou qu’il aura dépassé 21, le dealer piochera les cartes finales pour déterminer le gagnant. Ceci constituera le noyau dur de notre projet. S’il nous avançons rapidement, nous introduirons un second joueur. Le second joueur aura la même fonction que le premier à ceci près que la haute valeur en dessous de 21 sera recherchée parmi les trois entités jouant. Ainsi, si le joueur 1 a 20 points, le joueur 2 a 19 points et que la banque a 23, ce sera le joueur 1 qui aura gagné. Il est donc nécessaire que les joueurs 1 et 2 ne connaissent pas les jeux de l’un et l’autre.

Dans un troisième temps si le temps et le grand dieu de Prolog nous sont favorables (à traduire par si nous ne sommes pas mauvais), nous introduirons la notion de mise et de capitaux propres à chaque personne. Le joueur qui gagne remporte deux fois sa mise, un joueur qui perd voit sa mise disparaitre. Le grand perdant sera donc celui qui tombera en banqueroute en premier.

Ceci résume succinctement notre projet en prolog. Si ce projet semble réalisable, nous le mettrons en pratique sinon nous nous rabattrons sur un Puissance 4.