Rapport TP2 INF4215

Adrien Logut (1815142), Elliot Sisteron (1807165)

Introduction

Pour ce second travail pratique, nous allons développer une base de connaissances en Prolog pour représenter un programme académique. Nous devrons répondre à plusieurs spécifications qui permettront à un étudiant de gérer son cheminement dans le programme et au personnel de l'établissement de gérer la population étudiante.

Table des matières

1 Réflexion

1.1 Construction

La base de connaissance, codée en prolog a été divisé en plusieurs sections délimitées par des commentaires indiquant la partie qui suit.

1.2 Définition du problème

La première phase du TP a été de raisonner sur les différentes entités à définir pour notre problème. On a donc déterminé des classes ainsi que des sous-classes qui permettent de hiérarchiser notre problème. On a donc différentes classes : *Cours*, *Sujet*, *Étudiant*, *Programme*, *Langage*, etc... Ainsi que différentes sous classes pour la classe sujet. (Tout se trouve au début de notre fichier .pl).

1.3 Cours Valides

La définition d'un cours valide est la partie la plus compliquée et la plus *messy*, du fait qu'il y a ÉNORMÉMENT de conditions à vérifier (entre autres qu'un étudiant à déjà réussi les pré requis, pour les co-requis aussi etc...). On a essayé de garder des noms intelligibles pour une bonne compréhension, mais on s'excuse en avance du bordel monstre que ça génère. Il a fallu bien lister les différentes choses à vérifier, les coder séparément, les tester séparément et ensuite connecter le tout. C'est la partie où il a fallu le plus réfléchir, et bien poser le problème en cherchant à savoir vraiment ce que nous demande le sujet. Comme dans tout projet qui se base sur un texte du client (on peut aussi citer une base de données demandée par un client), il y a toujours des zones d'ombres ou alors sujettes à différentes interprétations selon les personnes. On a essayé de coller au mieux, en espérant avoir répondu à tous les critères demandés.

1.4 Représentation des dépendances

On a eu le choix pour les différents prédicats. On fait le choix d'utiliser des prédicats lorsque la propriété était unaire (*en_ echange(Étudiant)*), et le mot clé *prop* lorsque nos propriétés étaient binaires. (*aka. tout le reste*). Cela nous a permis de vérifier plus simplement certaines conditions.

1.5 Les données

Les données ont été extraites en partie du site de Polytechnique. Mais certaines ont été inventées pour tester plus facilement nos propriétés et réponses aux questions (comme les cours dispensés à l'udem et McGill par exemple). L'ajout de données est assez facile dans le sens ou chaque propriété est regroupée avec ses voisines.

1.6 Conclusion

Au final le projet a été un grand casse tête, mais nous a forcé à poser le problème réfléchir et ne pas foncer tête baissée dans le code. Il a fallut faire preuve d'une grande discipline pour trier

les différentes parties, classes, propriétés pour ne pas faire de redondances. On en est encore loin, à notre humble avis, et ça pourrait être encore plus clair. C'est une bonne piste d'amélioration.

2 Manuel d'utilisation

1. Pour déterminer si un choix de cours est valide, utilisez la requête valide(Etudiant, [Cours]) :

```
?- valideChoixDeCours(elliot, [mth1101, inf4215])
   Yes
?- valide(adrien, [inf8225, inf4420])
   No
```

2. On peut utiliser la requête qualite(Etudiant, [Cours]) pour savoir si ce choix de cours est de qualité :

```
?- qualite(elliot, [inf4705, inf4215, inf1010, log4715, inf8225])
   No
?- qualite(adrien, [inf8225, log4420, inf4215, inf4725]).
   Yes
```

3. Pour savoir quels cours abordent certains sujets, il suffit d'utiliser la requête prop(Cours, aborde, Sujet) :

```
?- prop(inf1010, aborde, objet)
   Yes
?- prop(inf1010, aborde, chimie)
   No
?- prop(X, aborde, ia)
   X = inf4215;
   X = inf8225;
   X = inf2565;
   No
```

4. Il suffit d'utiliser prop(Cours, utilise, Langage) pour connaître les cours dans lesquels un langage spécifique est utilisé :

```
?- prop(X, langage, cpp)
   X = inf1010;
   X = inf4705;
   ...
```

5. Pour savoir quels sont les programmes dans lesquels un cours est offert, il suffit d'utiliser la requête prop(Programme, offre, Cours) :

```
?- prop(X, offre, inf1010)
    X = genieLog;
    X = genieInfo;
    ...
```

6. Les cours se donnent en classe inversée lorsque le prédicat classe_inversee(Cours) est vérifié :

```
?- classe_inversee(X)
    X = inf4215;
    ...
```

7. On peut savoir si un cours obligatoire, optionnel ou bien projet a été suivi par un étudiant avec prop(Etudiant, Type, Cours) :

```
?- prop(elliot, suivi_obligatoire, X)
    X = inf4215;
    X = log4420;
    ...
?- prop(elliot, suivi_projet, X)
    X = inf2990;
    No
```

8. On peut savoir si un élève est inscrit à une orientation, et laquelle c'est.

```
?- prop(adrien, est_inscrit_orientation, X).
     X = multimedia;
?- prop(elliot, est_inscrit_orientation, X).
     false;
```

9. On peut savoir si une équivalence est disponible pour un cours. Elle a été définie s'il existe un autre cours qui parle du même sujet, dispensé dans une autre école et que celui ci a été réussi par l'étudiant.

```
?- prop(elliot, equivalence, X):-
     X = inf4215;
     X = inf8215;
```

10. Un cours accepte des étudiants en échange lorsque le prédicat accepte_echange(Cours) est vérifié :

```
?- accepte_echange(X)
    X = inf4215;
    X = log4420;
    ...
```

3 Questions

3.1 Question 1

On considère le code *Prolog* suivant :

```
1  aime(paulo,X) :-
2      mama_burger(X),
3      !,
4      fail.
5  aime(paulo,X) :-
6      hamburger(X).
7
8  hamburger(X) :-
9      big_mac(X).
10 hamburger(X) :-
11     mama_burger(X).
12 hamburger(X) :-
13     whopper(X).
14
15 big_mac(a).
16 mama_burger(b).
17 big_mac(c).
18 whopper(d).
```

La combinaison du coupe choix *!* avec *fail* qui force l'échec est ici un moyen de s'assurer que Paulo ne peut pas aimer un burger qui soit de type *mama burger*.

En effet, le code suivant garantit que *Paulo* aime tous les hamburgers (y compris, donc, le fameux *mama burger*) :

```
1  aime(paulo,X) :-
2      hamburger(X).
3
4  hamburger(X) :-
5      big_mac(X).
6  hamburger(X) :-
7      mama_burger(X).
8  hamburger(X) :-
9      whopper(X).
```

En ajoutant la relation :

```
1  aime(paulo,X) :-
2      mama_burger(X),
3      !,
4      fail.
```

On force, dans le cas où X est un *mama burger*, à arrêter la recherche.

Mais, cela pose un problème majeur. Supposons que l'on souhaite obtenir toutes les constantes t.q. *aime(paulo, X)*. On lancerait donc la requête :

```
1 ?- aime(paulo,X)
2    No
```

Ce n'est clairement pas le comportement recherché. Cela vient du fait que l'on déclare avant le reste :

```
1 aime(paulo,X) :-
2     mama_burger(X),
3     !,
4     fail.
```

Mais, en le déclarant avant on n'aurait pas le comportement voulu non plus (cela retournerait *Yes* dans le cas où X est un *mama burger*).

Il faudrait donc séparer cette commande :

```
1 pas(X):-
2     X,
3     !,
4     fail.
5 pas(X).
6
7 aime(paulo,X) :-
8     hamburger(X),
9     pas(mama_burger(X)).
```

On peut interpréter cela comme une négation par l'échec, i.e. *aime(paulo, X)* continuera d'être évalué ssi la tentative de montrer *mama_ burger(X)* n'aboutit pas.

```
1 aime(paulo,X) :-
2     hamburger(X),
3     \+ mama_burger(X).
```

Tout simplement.

3.2 Question 2

Supposons qu'on soit dans la situation du TP, c'est à dire qu'on cherche à savoir, dans le cadre où notre étudiant est en train de constituer son choix de cours, la liste de tous les cours valides qu'il peut prendre et qui satisfasse toutes les contraintes de requis et co-requis. Il semble difficile en Prolog de *trier* cette liste avec en premier le *meilleur* cours à prendre dans la situation actuelle. En effet, Prolog ne résonne qu'avec des *vrai* ou *faux*, c'est tout ou rien. Dans ce cas là on aimerait avoir plus de choix, pas de choix binaire, entre autres en instaurant une file de priorité, ou du moins un *score* à chaque cours.