TD1 — Mise à niveau PROLOG

Jacques Malenfant, Olena Rogovchenko

Exercice 1 : Arbre généalogique

`family.ecl` :

```
man(eric).                      woman(amelie).
man(ludovic).                   woman(caroline).
man(paul).                      woman(angela).
man(tristan).                   woman(laura).
man(alban).                     woman(christiane).
man(adam).                      woman(stephanie).
man(frederic).                  woman(phoebe).
                                woman(felicia).
married(alban, christiane).     woman(jacqueline).
married(tristan, jacqueline).
married(eric, stephanie).       mother(jacqueline, stephanie).
married(adam, laura).           mother(christiane, eric).
                                mother(stephanie, caroline).
father(alban, eric).            mother(stephanie, angela).
father(tristan, stephanie).     mother(amelie, adam).
father(eric, adam).             mother(amelie, paul).
father(eric, paul).             mother(laura, ludovic).
father(eric, angela).           mother(laura, phoebe).
father(frederic, caroline).
father(adam, ludovic).          sibling(X, Y) :- father(F, X), father(F, Y), X \= Y.
father(adam, phoebe).           sibling(X, Y) :- mother(M, X), mother(M, Y), X \= Y.
```

Q1. Formuler les buts suivants en PROLOG :

1. Est-ce-que Ludovic et Phoebe sont frère et soeur ?
2. Trouver tous les frères et soeurs d'Adam.
3. Est-ce-que Jacqueline et Tristan sont mariés ?

Q2. Définir les relations de famille suivantes : `sister`, `brother`, `son_of`, `mother_in_law`, `aunt`.

Q3. Définir le prédicat `parent`. Comment peut-on reécrire le prédicat `sibling` maintenant ?

Q4. Définir le prédicat `has_children` et expliquer comment peut-on l'utiliser pour obtenir la liste de toutes les personnes qui ont un enfant.

Q5. Définir la relation `predecessor(X,Y)`.

Q6. On veut rendre maintenant le prédicat `married` valide pour deux personnes étant mariées, et ce indépendamment de l'ordre des arguments. Apporter les modifications nécessaires.

Exercice 2 : Les listes

Dans un premier temps, les exercices portent sur une liste d'entiers.

Q1. Donner le résultat de l'unification des listes suivantes :

1. `[X|Y]` et `[1,2,3]`
2. `[X|Y]` et `[1|[2|3]]`
3. `[X|Y]` et `[[1|2]|3]`
4. `[X,Y]` et `[1|[2|3]]`
5. `[X,Y]` et `[1,[2,3]]`
6. `[X,Y]` et `[1|[2|[]]]`

Q2. Définir le prédicat `max_list(List, Max)`, qui unifie `Max` avec le maximum de la liste `List`. Rappel : en PROLOG l'affectation n'existe pas, on utilise uniquement le concept d'unification.

```
[eclipse 5]: max_list([5,2,8,12,0], Max).

Max = 12
```

Q3. Définir les prédicats `delete_elem` et `insert_elem`.

```
[eclipse 6]: delete_elem(5, [6,5,4,3], X).

X = [6, 4, 3]

[eclipse 9]: insert_elem(5, [6,4,3], X).

X = [5, 6, 4, 3]
X = [6, 5, 4, 3]
X = [6, 4, 5, 3]
X = [6, 4, 3, 5]
```

Q4. Définir le prédicat `permutation(List, X)`, qui génère la liste de toutes les permutations possibles.

```
[eclipse 10]: permutation([1,2,3], X).

X = [1, 2, 3]
X = [2, 1, 3]
X = [2, 3, 1]
X = [1, 3, 2]
X = [3, 1, 2]
X = [3, 2, 1]
```

Q5. Définir le prédicat `no_doubles(List, NDList)`, qui prend une liste `List` et génère une nouvelle liste en enlevant tous les doublons.

```
[eclipse 11]: no_doubles([3,5,2,3,5,4,2,5,5], X).

X = [4, 2, 5, 3]
```

On va maintenant travailler avec des listes de couples $(nombre, lettre)$:

Q6.* Écrire le prédicat qui est valide uniquement si les deux couples passés en argument sont égaux.

```
eq_pairs(pair(1,a), pair(1,a)).

Yes (0.00s cpu)
```

Q7. Écrire le prédicat `zip(L1, L2, Res)` qui unifie une liste d'entiers et une liste de caractères avec une liste de couples.

```
[eclipse 18]: zip([1,2,3],[a,b,c],Y).

Y = [pair(1, a), pair(2, b), pair(3, c)]
```

Q8. Écrire le prédicat `unzip(L, R1, R2)` qui unifie une liste de couples avec une liste d'entiers et une liste de caractères.

```
[eclipse 20]: unzip([pair(1, a), pair(2, b), pair(3, c)], Int, Alpha).

Int = [1, 2, 3]
Alpha = [a, b, c]
```

Q9.* Adapter les prédicats `max_list` et `no_doubles` aux listes de couples.

Exercice 3 : Quicksort

Le tri rapide (en anglais *quicksort*) est un tri fondé sur le paradigme "diviser pour régner". Pour rappel, cet algorithme se divise en trois etapes :

1. choisir un élément arbitraire du vecteur, que nous appelons élément pivot
2. réorganiser les éléments du vecteur de sorte que tous les éléments inférieurs ou égaux au pivot soient à gauche du pivot et les éléments supérieurs au pivot à droite du pivot dans la liste
3. trier récursivement la partie gauche et la partie droite du vecteur

Q1. Écrire la fonction qui prend une liste d'entiers et un pivot P et qui produit deux listes : la première contenant tous les éléments inférieurs ou égaux à P et la deuxième tous les élément supérieurs.

```
[eclipse 3]: split(8, [5, 10, 3, 12], Smaller, Greater).

Smaller = [5, 3]
Greater = [10, 12]
```

Q2. Écrire l'algorithme de tri. Pour concaténer les listes, on pourra utiliser le prédicat `append(?List1, ?List2, ?List3)` qui unifie `Liste3` avec la concaténation des listes `Liste1` et `Liste2`.

```
[eclipse 4]: quicksort([8, 5, 10, 3, 12], X).

X = [3, 5, 8, 10, 12]
```

Q3. Est-ce que cette implémentation est efficace, par rapport à une implémentation classique en C ? Pourquoi ?

Exercice 4 : Problèmes de logique

Loucherbem

Écrire un programme qui lit des mots sur l'entrée standard et les affiche après les avoir convertis en "loucherbem" (langage des bouchers).

Cette conversion consiste en :

1. reporter la 1ère lettre du mot en fin de mot
2. rajouter aléatoirement un des suffixes “em”, “ji”, “oc” ou “muche”
3. remplacer la 1ère lettre du mot par la lettre 'l'.

Exemples :

```
[eclipse 9]: loucherbem([b,o,u,c,h,e,r], X).

X = [l, o, u, c, h, e, r, b, e, m]

[eclipse 6]: loucherbem([v,a,c,h,e], X).

X = [l, a, c, h, e, v, o, c]
```

SEND MORE MONEY*

Trouver des valeurs pour les variables S, E, N, D, M, O, R, Y entre 0 et 9 telles que :

```
    S E N D
+   M O R E
= M O N E Y
```

Tous les nombres sont bien formés (aucun ne commence par 0).

On va utiliser l'approche “generate and test” : décrire la relation entre les variables et ensuite générer toutes les combinaisons possibles d'entiers, jusqu'à trouver celle qui vérifie la relation. On pourra utiliser le prédicat `member(X, List)`, qui unifie `X` avec une valeur dans la liste `List`.

SEND MORE MONEY**

Un autre façon de résoudre ce problème est de l'exprimer sous forme de contraintes ECL^iPS^e et ensuite de rechercher une instantiation valide. Laquelle des deux solutions est la plus efficace ? Pourquoi ?

Le fermier, le loup, la chèvre et le chou*

Un homme doit faire traverser un loup, une chèvre et un chou dans un bateau. Le bateau est tellement petit, qu'il ne peut embarquer qu'un des trois et lui-même pour chaque traversée. Comment peut-il faire pour les faire traverser tous les trois sans laisser l'occasion au loup de manger la chèvre ou à la chèvre de manger le chou ?

Pour résoudre ce problème on va encoder les 4 objets `(fermier, loup, chevre, chou)` sous forme d'une liste d'états. Les deux états possibles sont r (pour rive droite) et l (pour rive gauche). Ainsi, on peut représenter les états suivants :

- $[r, r, r, r]$: tout le monde est sur la rive droite
- $[l, l, r, r]$: le fermier mène le loup sur la rive gauche (la chèvre mange le chou)
- $[l, l, l, l]$: l'état gagnant, tout le monde est sur la rive gauche

Pour encoder les transitions possibles on va définir le prédicat `move(State, Transition, NewState)`. Où `State` et `NewState` sont les états représentés par les quadruplets définis précédamment et `Transition` appartient à la liste des transitions possibles :

nothing : le fermier se déplace seul

$wolf$: le fermier se déplace avec le loup

goat : le fermier se déplace avec la chèvre

cabbage : le fermier se déplace avec le chou

Ensuite, il faut définir le prédicat `safe(State)` qui vérifie si un état est valide (le loup ne peut pas manger la chèvre et la chèvre ne peut pas manger le chou).

La dernière étape consiste à définir le prédicat `solution(Config, MoveList)`. Qui à partir d'une configuration initiale cherche une liste de transitions menant à la configuration gagnante.

Remarque : on pourra fixer la taille de la solution recherchée en appliquant à `MoveList` le prédicat `length(MoveList, X)`, où X est la taille souhaitée.

Le problème du Zèbre / problème d'Einstein**

Cinq maisons consécutives, de couleurs différentes, sont habitées par des hommes de différentes nationalités. Chacun possède un animal différent, a une boisson préférée différente et fume des cigarettes de marques différentes. De plus, on sait que :

Le norvégien habite la première maison,
La maison à coté de celle du norvégien est bleue,
L'habitant de la troisième maison boit du lait,
L'anglais habite la maison rouge,
L'habitant de la maison verte boit du café,
L'habitant de la maison jaune fume des kools,
La maison blanche se trouve juste après la verte,
L'espagnol a un chien,
L'ukrainien boit du thé,
Le japonais fume des cravens,
Le fumeur de old golds a un escargot,
Le fumeur de gitanes boit du vin,
Le voisin du fumeur de Chesterfields a un renard,
Le voisin du fumeur de kools a un cheval.

Qui boit de l'eau ? À qui appartient le zèbre ?

On peut résoudre ce problème avec l'approche "generate and test", mais il existe aussi une solution plus astucieuse.

En TP

On utilise ECL^iPS^e, un langage de programmation par contraintes basé sur PROLOG.

Un première liste de commandes :

Lancement/execution :

`eclipse` : pour lancer ECL^iPS^e en ligne de commande

`[nom_module].` : charger/recharger le module *nom_module* en ECL^iPS^e

`halt.` : arrêt d' ECL^iPS^e

Tracer l'exécution :

`trace`/`notrace` : active/desactive l'exécution pas à pas

`spy(predicate_name/arity)` : trace pas à pas l'exécution du prédicat `predicate_name`

Modules :

`:- module(name).` : au début d'un fichier, il définit le module `name`.

`:- use_module(name).` : permet d'importer le module `name` à l'intérieur d'un autre module.

`:- export(name/arity).` : rend la fonction `name` visible à l'extérieur du module.

Pour plus d'informations, il y a le tutoriel d'ECL^iPS^e.