TP - Logique

L2 info – math

29 novembre 2021

1 Consignes

Le but du TP est de coder en `Prolog`. Il est divisé en trois grandes parties. La première se penche sur l'aspect base de connaissance (base de données) de `Prolog`. La seconde parle de récursivité et graphes. Enfin, on s'intéressera aux listes et aux langages construits par induction. Les *règles du jeu* sont les suivantes :

— En plus du code, qui doit être *commenté*, il faut rendre un rapport en `Markdown` qui décrit le fonctionnement d'un prédicat.

— Il faudra soumettre code et documentation dans une archive sur l'ENT *2 semaines* après votre dernier TP.

— Vous pouvez vous mettre par groupe de 1 à 3 *maximum*.

— Le barème de chaque exercice est donné à titre indicatif, il dépasse 20 points au total.

2 Requête et base de connaissances (8 points)

2.1 Introduction (1 point)

Sur l'ENT, vous trouverez dans le cours « LOGIQUE2021 » un fichier nommé `boit.pl`. Le fichier décrit des *faits* à propos d'une relation *boit*. Par exemple, la ligne `boit(john, the).` signifie que John boit du thé. Notez qu'une ligne se *termine toujours par un point*.

Rendez vous sur à cette addresse `https://swish.swi-prolog.org/` (`SWISH`) et cliquez sur le bouton `Program`. Dans l'éditeur (à gauche) copiez-coller le code contenu dans le fichier `boit.pl`. Ensuite, répondez aux questions suivantes

1. D'après ce qui est écrit dans le fichier, est-il vrai que Vincent boit de la vodka ?
2. Dans l'interpréteur (à droite), tapez `boit(vincent, vodka).` (attention au point !). Quelle est la réponse ?
3. D'après ce qui est écrit dans le fichier, est-il vrai qu'Otto boit de la vodka ?
4. Tapez `boit(otto, vodka).` et notez la réponse de l'interpréteur.
5. Ajoutez la ligne `boit(otto, vodka).` au fichier et retapez la commande `boit(otto, vodka).`. La réponse de l'interpréteur a-t-elle changé ?
6. Tapez `boit(X, eau).`, à quoi correspond la réponse de l'interpréteur ? Tapez `;` (plusieurs fois) pour avoir d'autres solutions. Dans cette ligne, `X` représente une *variable* et `eau` une *constante*. En `Prolog` toutes les variables commencent par une *majuscule* ou par `_`. Une constante en revanche est un nombre ou une chaîne de caractère commençant par une *minuscule*. Du coup, `boit` est un *prédicat* d'arité 2. Enfin, un *fait* est un prédicat appliqué à des constantes ou des variables.
7. Tapez `boit(lhouari, X).`, que donne l'interpréteur ?

À partir de tous ces faits, on peut ajouter des *règles* qui permettent de déduire de nouvelles connaissances. Par exemple, on peut dire que si une constante apparaît en deuxième position du prédicat `boit`, c'est que c'est une boisson. On peut modéliser cette règle par une implication logique :

$$\forall X, Y \quad boit(Y, X) \rightarrow boisson(X)$$

En `Prolog`, cette règle s'écrit de la manière suivante :

```
boisson(X) :- boit(_, X).
```

Cette ligne signifie que `X` est une boisson s'il existe quelqu'un qui boit `X`. Autrement dit, `boisson(X)` s'il existe un fait de la forme `boit(_, X)`, où `_` est une variable dite *anonyme*, c'est-à-dire que sa valeur ne nous intéresse pas.

8. En prenant exemple sur `boisson`, donner l'implication logique et le prédicat `personne` qui détermine si `X` est une personne ou non.

Attention : Si vous copiez-collez directement le code écrit dans le PDF, certains symboles peuvent être mal convertis, par exemple - sera transformé en –.

2.2 La famille (3 points)

Dans cet exercice, on s'intéresse à la famille. Dans le fichier `famille.pl` sont donnés des faits sur des personnes, par exemple

```
homme(alexandre). % alexandre est un homme
homme(jules).

femme(aure). % aure est une femme
femme(emmanuelle).

parent(remy, claude).  % remy est un parent de claude
parent(catherine, claude).
```

À partir de ces faits là, on peut déduire qu'un homme est le père d'une autre personne s'il est son parent. Logiquement on peut l'écrire :

$$\forall X, Y \quad (parent(X, Y) \wedge homme(X)) \rightarrow pere(X, Y)$$

Ce qu'on peut traduire en `Prolog` par la règle :

```
pere(X, Y) :- parent(X, Y), homme(X).
```

On remarquera que l'on *« inverse »* la lecture de la règle, et que la quantification universelle ($\forall$) est *automatique* et *implicite*. À partir de ces données, proposez pour chacun des cas suivants une modélisation logique (comme pour *pere*) et le code `Prolog` correspondant :

1. Le prédicat `mere(X, Y)` signifiant « `X` est la mère de `Y` ».
2. Les prédicats `fils(X, Y)` et `fille(X, Y)`.
3. Les prédicats `grandpere(X, Y)` et `grandmere(X, Y)`.
4. Les prédicats `soeur(X, Y)` et `frere(X, Y)`.
5. Les prédicats `cousin(X, Y)` et `cousine(X, Y)`.
6. Déterminer toutes les relations de parenté et dessiner avec l'application de votre choix (Google draw par exemple) le(s) arbre(s) généalogique(s) que vous trouvez dans les données.

Remarque : on peut tester l'égalité de deux variables `X`, `Y` avec `X == Y`. De manière similaire, on utilise `X \== Y` ou `not(X == Y)` pour tester que `X` et `Y` sont différents.

2.3 La fin de la solitude (2 points)

Vous travaillez pour le site de rencontres `www.mitoc.gov` qui propose des rencontres entre les gens inscrits sur la base des prédicats suivants :

- `personne(I, N, T, C, A)` décrit un(e) inscrit(e). `N` est son nom, `T` sa taille, `C` la couleur de ses cheveux et `A` son âge. `I` est un identifiant (un nombre entier) ;
- `gout(I, M, L, S)` : la personne dont l'identifiant est `I` aime le genre de musique `M`, le genre de littérature `L`, et pratique le sport `S` ;
- `recherche(I, T, C, A)` : la personne `I` recherche un partenaire de taille `T`, ayant des cheveux de couleur `C` et dont l'âge est `A`.

On considère que deux personnes X et Y sont assorties si X convient à Y et si Y convient à X. On dira que X convient à Y si d'une part X convient physiquement à Y (la taille, l'âge et la couleur des cheveux de X sont ceux que Y recherche) et si d'autre part les goûts de X et Y en matière de musique, littérature et sport sont identiques.

1. Donner un ensemble de faits représentant le fichier des candidats.
2. Écrire les règles définissant `convient-physiquement(X, Y)` et `ont-memes-gouts(X, Y)`.
3. En déduire le programme qui détermine les couples assortis.

Remarque : donner des noms suffisamment rigolos aux personnes peut amener des points bonus.

2.4 Attention à ne pas dépasser ! (2 points)

On se propose de définir un prédicat permettant de colorier la carte de la Figure 1. Les règles sont les suivantes : on dispose des trois couleurs vert, jaune et rouge ; deux zones contiguës doivent avoir des couleurs différentes.

1. Écrivez un prédicat `coloriage(C1, C2, C3, C4)` qui comportera deux parties. La première partie génère toutes les valeurs possibles de `C1`, `C2`, `C3` et `C4`. La seconde vérifie si les colorations obtenues sont conformes à la carte par l'utilisation du prédicat `X \== Y` sur les couleurs des zones contiguës.
2. Reprenez ce prédicat, et modifiez le programme en déplaçant les tests de différence de couleurs le plus tôt possible dans l'écriture du prédicat, c'est-à-dire en vérifiant les différences de couleurs dès que celles-ci sont instanciées. Quelle en est la conséquence ?

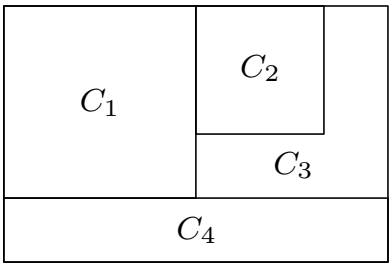


FIGURE 1

3 Graphes et récursivité (7 points)

3.1 Introduction (4 points)

Le système de règles et de faits de `Prolog` en font un langage adapté à la récursivité. Dans cette introduction nous allons nous atteler à programmer quelques fonctions arithmétiques récursives. Par exemple, tester qu'un nombre entier est pair peut se faire de manière récursive : on sait que 0 est pair, et que 1 ne l'est pas. De plus, si n est un entier strictement positif et que $n-2$ est pair, alors nécessairement n est pair. On peut modéliser ça par la logique de la manière suivante :

$$pair(0)$$
$$(n > 0 \wedge pair(n-2)) \rightarrow pair(n)$$

Ces règles ont une traduction directe en `Prolog` :

```
pair(0).
pair(N) :- N > 0, M is N - 2, pair(M).
```

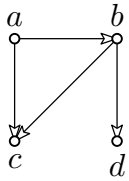
On notera que `M is N - 2` permet de faire en sorte que `M` « prenne » la valeur `N - 2` (pour plus de détail, voir l'idée d'*unification* dans l'aide/ le cours). Pour chacune des fonctions suivantes, donnez les règles de construction et le prédicat correspondant en `Prolog` :

1. La fonction factorielle. **Indice :** on peut créer un prédicat `fact(N, R)` où la variable `R` contiendra le résultat de la fonction, un cas de base est `fact(0, 1)`.
2. La somme des entiers de 1 à n (en utilisant la récursivité !).
3. La suite de fibonacci $F(n) = F(n-1) + F(n-2)$ avec $F(1) = F(2) = 1$.
4. La fonction d'Ackermann $A(m, n)$, avec m, n entiers positifs, définie par :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1, A(m, n-1)) & \text{si } m > 0 \text{ et } n > 0 \end{cases}$$

3.2 Graphes dirigés acycliques (3 points)

Dans cet exercice on s'intéresse aux graphes dirigés acycliques (abrégé DAG). Un DAG $G = (V, E)$ peut-être décrit en `Prolog` par le prédicat `arete` d'arité 2 qui traduit les arêtes de G (donc E). On donne un exemple dans la Figure 2.



(a) Un graphe dirigé acyclique G

```
sommet(a).
sommet(b).
sommet(c).
sommet(d).

arete(a, b).
arete(b, d).
arete(b, c).
arete(a, c).
```

(b) Le code `Prolog` correspondant

FIGURE 2

1. Écrivez un prédicat `chemin-oriente(X, Y)` qui teste s'il existe un chemin orienté de `X` à `Y` dans le graphe. Un chemin orienté est un ensemble de sommets $u_1, \ldots, u_n$ tels que pour tout $1 \leqslant i < n$, il y a une arête allant de u_i à u_{i+1}. Dans notre exemple, il existe un chemin orienté entre a et d passant par les arêtes (a, b) et (b, d).
2. Modifiez le prédicat `chemin-oriente` pour y ajouter un paramètre `N`. Ainsi, le nouveau prédicat `chemin-oriente(X, Y, N)` testera s'il existe un chemin orienté de `X` à `Y` de taille `N`. Dans l'exemple, Le chemin orienté $(a, b), (b, d)$ est de taille 2.
3. Écrivez un prédicat `chemin(X, Y)` qui teste s'il existe un chemin (non-orienté) de `X` à `Y` dans le graphe. Un chemin est un ensemble de sommets $u_1, \ldots, u_n$ tels que pour tout $1 \leqslant i < n$, il y a une arête allant de u_i à u_{i+1} ou de u_{i+1} à u_i. Dans notre exemple, d, b, c, a est un chemin entre d et a. En déduire un prédicat `connecte` qui teste si le graphe est connecté. Un graphe est connecté si toutes les paires de sommets sont reliées par un chemin.

4 Listes et langages (17 points)

4.1 Introduction (1 point)

Non seulement on peut faire de la récursivité en `Prolog`, mais on peut aussi utiliser des structures récursives : les *listes*. Les listes se déclarent entre crochets `[ ]`. Par exemple `[a, b, c, d]` est une liste. En `Prolog` les listes sont pensées de manière récursive : une liste est composée d'une *tête*, et d'une autre liste, le *corps*. Dans notre exemple, `a` est la tête et `[b, c, d]` le corps. Ainsi, une autre écriture (équivalente) de la liste `[a, b, c, d]` est `[a | [b, c, d]]`.

1. Dans l'interpréteur, tapez la ligne `[X | Y] = [a, b, c, d]`. Quelle est la réponse ? Que représente `X` ? `Y` ?
2. Ensuite, tapez `[X] = [a, b, c, d]`. Quelle est la réponse ? À votre avis, pourquoi ? **Indice :** `X` est *une* variable, mais la liste contient *quatre* éléments.

À noter que notre exemple ne contient que des constantes, mais une liste peut très bien contenir éléments plus complexes, comme des listes par exemple ! En effet, la liste `[[a, b], [c, d]]` est une liste tout à fait valide !

4.2 Opérations sur les listes (5 points)

Dans cet exercice, on dispose d'une liste `L` sur laquelle on aimerait effectuer des opérations élémentaires comme par exemple ajouter un élément au début ou à la fin.

1. Donner le code du prédicat `head(X, L)` qui renvoie le premier élément de la liste `L`.
2. Coder `addhead(X, L, L1)`, le prédicat qui ajoute un élément `X` au début de la liste `L`.
3. Coder le prédicat `last(X, L)`, qui renvoie le dernier élément de la liste `L`.
4. On aimerait maintenant coder un prédicat `addlast(X, L, L1)` qui ajoute un élément à la fin de la liste `L`. Après des jours et des jours de réflexion, on est arrivé au code suivant :

```
addlast(X, [], [X]).
addlast(X, L, [L | X]).
```

   Exécutez la ligne suivante : `addlast(e, [a,b,c,d], Resultat)`. Que se passe-t-il ? Quel est le problème à votre avis ? Corrigez le prédicat en fonction de vos observations.
5. Enfin, on souhaiterait retourner la liste, par exemple transformer la liste `[a, b, c, d]` en `[d, c, b, a]`. Écrire le prédicat `reverse(L, L1)` correspondant.

4.3 Construction inductive et langages (11pt)

Dans cet exercice on va s'intéresser aux *langages* construits de manière inductive. Soit $\{a, b\}$ notre alphabet et $\mathcal{L}$ le langage contenant tous les *mots* sur $\{a, b\}$ qui peuvent s'écrire $(ab)^n$, avec $n \in \mathbb{N}$. Par exemple, *abab* est un mot de $\mathcal{L}$, mais *aba* ne l'est pas. Le mot vide ϵ fait aussi partie de $\mathcal{L}$. On peut consruire ce langage à partir de quelques règles inductives (S est un mot sur $\{a, b\}$) :

$$\epsilon \in \mathcal{L}$$
$$S \in \mathcal{L} \rightarrow abS \in \mathcal{L}$$

On va traduire ces règles et mots en `Prolog`. On va représenter un mot par une *liste*. Par exemple le mot *abab* est représenté par la liste `[a, b, a ,b]` et le mot vide ϵ par `[ ]`, la liste vide. En utilisant des règles, on peut également coder un prédicat qui reconnaît le langage $\mathcal{L}$:

```
% le mot vide est dans le langage
langage([]).

% si S est un mot valide, alors abS est un mot valide.
langage([a, b | S]) :- langage(S).
```

À présent, on peut tester notre prédicat sur nos exemples :

```
?- langage([]).
true.

?- langage([a, b, a, b]).
true.

?- langage([a, b, a]).
false.

?- langage(X).
X = [] ;
X = [a, b] ;
X = [a, b, a, b].
```

Le but de cet exercice est de reconnaître des langages en utilisant `Prolog`. Pour chacun des langages suivants :

— donner ses règles de construction inductive ;

— donner un prédicat `Prolog` prenant un mot en entrée et qui, sur la base de vos règles inductives, renvoie `true` si le mot appartient au langage, et `false` sinon ;

— chaque question vaut 1 point.

1. $a^n b$, $n \in \mathbb{N}$.
2. ab^n, $n \in \mathbb{N}$.
3. $a^n b^m$, $n, m \in \mathbb{N}$.
4. a^{2n}, $n \in \mathbb{N}$.
5. $a^n b^n$, $n \in \mathbb{N}$.
6. L'ensemble des mots *palindromes* sur $\{a, b\}$. Un mot sur cet alphabet est palindrome si on peut le lire de gauche à droite ET de droite à gauche. Par exemple, *abba* et *bab* sont des palindromes, mais *abab* et *bba* n'en sont pas.
7. $a^l b^m c^n$, $l, m, n \in \mathbb{N}$.

8. $a^m b^n c^m$, $m, n \in \mathbb{N}$.
9. Le langage $\mathcal{L}$ qui contient tout les mots sur $\{a, b\}$ contenant au plus un b. Par exemple *aaabaaaaaa* et *aa* sont dans $\mathcal{L}$ mais pas *baaaaaaaab*.
10. Le langage $\mathcal{L}$ qui contient tous les mots ayant le même nombre de a et de b. Par exemple, *aaababbb* et *abba* sont dans $\mathcal{L}$ mais pas *aabbb*.
11. $a^n b^n c^n$, $n \in \mathbb{N}$. **Indice :** `l(S) :- l_count(S, N, N, N).`