TP numéro 3

Intelligence artificielle, ensIIE

Semestre 4, 2025–26

Exercice 1 : Allons au restaurant

Considérons la base de données suivante qui représente la carte d'un restaurant

```
horsdoeuvre(artichauts).
horsdoeuvre(crevettes).
horsdoeuvre(oeufs).
viande(grillade-de-boeuf).
viande(poulet).
poisson(loup).
poisson(sole).
dessert(glace).
dessert(tarte).
dessert(fraises).
```

1. Créer le fichier `menu.pl` avec les faits ci-dessus et le charger dans Prolog en entrant

   ```
   ?- [menu].
   ```

2. Demander la liste des hors d'œuvre disponibles. Utiliser `;` après la première solution pour obtenir la solution suivante.
3. Définissez la relation plat qui exprime qu'un plat est à base de viande ou de poisson.
   Remarque : après avoir modifié `menu.pl`, il suffit de taper

   ```
   ?- make.
   ```

   pour le recharger.
4. Définissez la relation repas/3 qui précise qu'un repas est constitué d'un hors d'oeuvre, d'un plat et d'un dessert
5. Demander la liste des repas possibles.
6. Considérons maintenant les valeurs caloriques des différents aliments. Par exemple (valeurs fantaisistes) :

   ```
   calories(artichauts, 150).
   calories(crevettes, 250).
   calories(oeufs, 200).
   calories(grillade-de-boeuf, 500).
   calories(poulet, 430).
   calories(loup, 250).
   calories(sole, 200).
   calories(glace, 300).
   calories(tarte, 400).
   calories(fraises, 250).
   ```

Demandez l'affichage de la valeur calorique des hors d'œuvre.

7. Définissez la valeur calorique d'un repas. Comment demander cette valeur ?
8. Définissez la relation repas-equilibre (valeur calorique inférieure à 900) et demandez la liste des repas équilibrés à base de viande.
9. Complétez le programme de façon qu'un repas comporte une boisson à choisir parmi le vin, l'eau minérale et la bière.

Exercice 2 : Listes

1. Définir le prédicat `app(X,Y,Z)` vrai si `Z` est la concatenation de `X` et `Y` (défini en cours).
2. Définir le prédicat `premier(X,L)` qui réussit si `X` est le premier élément de `L`.
3. Définir le prédicat `dernier(X,L)` qui réussit si X est le dernier élément de L. Donner deux versions de ce prédicat, avec et sans utilisation de `app`.
4. Définir le prédicat `mem(X,L)` qui définit l'appartenance d'un élément `X` à une liste `L`.
5. Définir le prédicat `double(L, S)` est vrai si chaque élément de `L` apparaît deux fois consécutivement dans `S`.
   Exemple :

```
?- double([a,b,a], A).
A=[a, a, b, b, a, a]

?- double([a,b,a], [a, a, b, a, a])
no
```

6. Définir deux prédicats (mutuellement récursifs) `longueurpaire/1` et `longueurimpaire/1` qui réussissent si leur argument est une liste avec un nombre pair (impair) d'éléments (n'utilisez pas d'opérations arithmétiques. Ne calculez pas la longueur de la liste).
7. Définir le prédicat `rev/2` pour inverser une liste. Par exemple `rev([a,b,c],L)` donne `L = [c,b,a]`. Est-ce que votre programme marche aussi avec `rev(L,[a,b,c])` ?
8. Définir le prédicat `prefixe(L1,L2)` vrai si la liste `L1` est un préfixe de la liste `L2`. Vous en écrirez deux versions, l'une utilisant `app` et l'autre non.
9. Définir le prédicat `palindrome(L)` vrai si la liste `L` est sa propre image renversée. (exemple : `[r,a,d,a,r]`.)
10. Définir le prédicat `perm(L1,L2)` vrai si la liste `L2` est une permutation de la liste `L1`.

    On peut obtenir une permutation de `L` en prenant une permutation `P` de son reste et en insérant sa tête à une certaine position dans la permutation `P`.

    Rechercher toutes les permutations de `[1,2,3,4]`.

Exercice 3 : Quelques prédicats arithmétiques

1. Écrire le prédicat `fact/2` qui calcule la factorielle d'un entier. Calculer factorielle 4. Peut-on calculer N tel que sa factorielle soit égale à 2 ?
2. Écrire le prédicat `division/4` défini par : `division(A, B, Q, R)` est satisfait si `Q` (resp. `R`) est le quotient (resp. reste) dans la division euclidienne de `A` par `B`.

3. Définir un prédicat `longueur(L, N)` satisfait si `N` est la longueur de la liste `L`.
4. Définir un prédicat `sumlist(L, N)` satisfait si `N` est la somme des éléments de la liste d'entiers `L`.
5. Écrire un prédicat `sorted` qui réussit si son argument est une liste d'entiers triée dans l'ordre croissant.
6. Écrire un prédicat ternaire `merge(L, M, N)` qui réussit si `N` est la liste triée contenant les éléments des listes triées `L` et `M`.

   Testez et vérifiez en utilisant le prédicat précédent que si `L` et `M` sont triées alors `N` est triée.
7. Écrire un prédicat `split(L, L1, L2)` tel que le contenu de `L` est le même que celui des listes `L1` plus `L2`, et la taille de `L1` et `L2` diffère d'au plus un.
8. Implémenter le tri fusion.

Exercice 4 : Problème des N reines

Le problème de N consiste à placer N reines sur un plateau d'échecs de taille $N \times N$, telle qu'aucune reine n'en menace une autre. (Autrement dit, toutes les reines sont sur des lignes et des colonnes différentes, et aucune reine n'est sur la même diagonale qu'une autre.)

On va trouver des solutions à ce problème en générant d'abord une tentative de solutions, c'est-à-dire en plaçant les reines sur N lignes, puis en vérifiant si cette tentative est effectivement une solution.

Une solution sera donnée par une liste contenant la position des reines sur les lignes consécutives, par exemple `[5,7,1,4,2,8,6,3]` pour

				X			
						X	
X							
			X				
	X						
							X
					X		
		X					

1. Définir un prédicat `n_to_one(N, L)` tel que L est la liste des nombres de `N` à 1.
2. Définir un prédicat `safe(Q1, Q2, K)` qui teste que deux reines placées en colonne `Q1` et `Q2` ne se menacent pas, en supposant que la deuxième reine est `K` lignes en dessous de la première.
3. Définir un prédicat `all_safe(Q, L, K)` qui teste qu'une reine placée en colonne `Q` ne menace aucune des reines de la liste de positions dans des lignes `L` situées `K` lignes en dessous.
4. Définir un prédicat `check(Q)` qui vérifie que pour la liste de positions de reines `Q` aucune reine n'en menace une autre.
5. Définir un prédicat `queens(N, Q)` qui est vrai quand `Q` est une solution du problème de `N` reines. Pour cela, on crée la liste de `N` à 1, on en prend une permutation, et on vérifie si cette permutation est une solution.