Prolog - TP3

Exercice 1 la coupure (sur feuille)

Soit le programme suivant :

```
s(a). (1)
s(b). (2)
t(a,b). (3)
t(b,a). (4)
p1(X,Y):- s(X),t(Y,X). (5)
p1(a,a). (6)

p2(X,Y):- s(X),t(Y,X),!. (7)
p2(a,a). (8)

p3(X,Y):- s(X),!,t(Y,X). (9)
p3(a,a). (10)

p4(X,Y):- !,s(X),t(Y,X). (11)
p4(a,a). (12)
```

Quel est l'arbre de recherche prolog à la question `p1(X,Y)`, `p2(X,Y)`, `p3(X,Y)` et `p4(x,y)` ?

Exercice 2 calcul du maximum

Commencez par recopiez le programme suivant :

```
max1(X,Y,Z):- X >= Y.
max1(X,Y,Y):- X < Y.

max2(X,Y,X):- X >= Y, !.
max2(X,Y,Y):- X < Y.

max3(X,Y,X):- X >= Y,!.
max3(X,Y,Y).
```

Comprenez comment fonctionne ces prédicats et vérifiez leur comportement. Voici quelques exemples que vous pouvez utiliser :

```
max*(1,2,X).
max*(4,2,X).
max*(4,2,2).
```

Quel est le problème de max3 ? Ecrivez un nouveau prédicat max4 qui corrige ce problème (indice : utilisez `=/2`)

Exercice 3 insérer un élément dans une liste

Ecrivez un prédicat `insérer/3` qui permet d'insérer un entier dans une liste triée par ordre croissant. Vous ferez en sorte que ce prédicat soit déterministe.

```
| ?- inserer(3,[],X).
X = [3]
yes

| ?- inserer(3,[2,4],X).
X = [2,3,4]
yes

| ?- inserer(3,[4,5],X).
X = [3,4,5]
yes
```

Exercice 4 inversion d'une liste

Au cours du TP 2, vous avez écrit le prédicat `inverser/2` qui permet d'inverser une liste. Ecrivez un nouveau prédicat permettant d'inverser une liste en utilisant un accumulateur et un wrapper.

```
? inverse([a,b,c,d], [d,c,b,a]).
yes
```

Exercice 5 nombre d'occurrences d'un terme

Ecrivez un prédicat `occurrence/3`. `occurrence(X,Liste,N)` compte le nombre `N` d'éléments de la liste déjà identique à `X`. Cela signifie que ce prédicat est d'un niveau métalogique car Prolog ne doit pas essayer d'unifier les termes au risque de rajouter de termes identiques.

Appuyez vous sur les exemples suivants pour bien comprendre :
Juste :

```
| ?- occurrence(a,[a,b,c,X,a,a],N).
N = 3
yes
```

Faux :

```
| ?- occurrence2(a,[a,b,c,X,a,a],N).
N = 4
X = a
yes
```

Dans ce cas Prolog a unifié `X` avec a alors qu'en fait nous on veut compter les termes déjà identiques
Juste :

```
| ?- occurrence(X,[a,b,c,X,a,a],N).
N = 1
yes
```

Faux :

```
| ?- occurrence2(X,[a,b,c,X,a,a],N).
N = 4
X = a
```

Cet exemple montre bien que l'on veut compter le nombre de terme `X` dans la liste et cela sans l'instancier au préalable.

Exercice 6 les dérivées

Le but de cet exercice est de calculer formellement les dérivées d'une fonction. Rappelez vous qu'en Prolog les expressions arithmétiques sont déjà représentées comme des structures.

Vous vous baserez sur les transformations suivantes. On considère x comme paramètre de la fonction, $*$ représente la multiplication et c une constante.
$c' \rightarrow 0$
$x' \rightarrow 1$
$(-U)' \rightarrow -(U')$
$(U+V)' \rightarrow U'+V'$
$(U–V)' \rightarrow U'–V'$
$(c*U)' \rightarrow c*U'$
$(U*V)' \rightarrow U*V'+V*U'$
$(U/V)' \rightarrow (UV-1)'$
$(Uc)' \rightarrow cU-1U'$

L'opérateur exposant est représenté en prolog par `**/2` :

```
| ?- X is 3**3.
X = 27.0
yes
```

Ecrivez le prédicat `derive(E,X,R)` qui réussit quand `R` est la dérivée de l'expression `E` avec comme paramètre `X`. On considérera que les deux premiers paramètres sont instanciées et que le dernier est une variable (le résultat de la dérivée).

Voici quelques exemples :

```
| ?- derive(x**2+2*x+1,x,X).
X = 2*x**(2-1)*1+2*1+0
yes

| ?- derive(-x/2,x,X).
X = -1*2**(-1-1)*0* -x+ - 1*2** -1
yes

| ?- derive(3*x**2+x**2+1+3/(x**2),x,X).
X = 3*(2*x**(2-1)*1)+2*x**(2-1)*1+0+3*(-1*(x**2)**(-1-1)*(2*x**(2-1)*1))
yes
```

Vous pouvez ensuite définir un prédicat `simp(E1,E2)` qui à partir d'une expression `E1` instanciée calcule `E2` qui est une forme simplifiée. Il faut considérer 2 cas :

- on ne peut pas simplifier car on n'a pas de terme composé
- on peut simplifier car on un terme avec un opérateur et deux arguments

Ce deuxième cas peut être traité par un prédicat annexe `change/4`. `change(op,gauche,droit,E)` réussit avec `E` l'expression simplifiée de l'expression 'gauche op droit'. Ce prédicat peut être construit de manière incrémentale. Commencez par exemple par traiter le cas 0 + Y = Y. Complétez ensuite les différents cas de l'addition puis les cas de la multiplication.

Exercice 7 Tic-Tac-Toe

Cet exercice traite du fameux jeu du Tic-Tac-Toe (Morpion). Nous n'allons pas créer un programme entier qui permettrait de jouer contre une IA mais seulement un programme d'aide à la décision très basique.

Le jeu se compose d'une grille de 9 cases numérotées comme ci-dessous :

1	2	3
4	5	6
7	8	9

Le jeu se joue à deux joueurs jouant à tour de rôle:

- le premier peut écrire un 'x' dans une case
- le second peut écrire un 'o' dans une case

L'objectif est d'aligner trois 'x' ou 'o' pour gagner.

Créez un programme permettant d'aider le joueur avec les 'o' dont le but est de lui indiquer quelle case jouer dans le cas où il est certain qu'il perde au prochain coup s'il ne joue pas cette case.

Nous considérerons le plateau comme une structure : `plateau(C1,C2,C3,C4,C5,C6,C7,C8,C9)`

`Ci` représente la case i de la grille. `Ci` est soit une variable non instanciée (case non jouée) ou un 'x' ou un 'o'.

Cet exercice est un cas classique d'un programme qui génère et teste. Il faut donc faire en sorte de générer les différents cas à vérifier (c'est à dire où le joueur risque de perdre) puis de vérifier si le joueur risque vraiment de perdre dans ce cas.

Vous pouvez utiliser les prédicats `arg/3`, `var/1` et `nonvar/1` pour accéder aux cases du plateau et examiner leur contenu.

Créez un prédicat `mouvement_obligatoire/2` tel que `mouvement_obligatoire(Plateau,X)` retourne le numéro de la case à jouer en fonction du plateau de jeu courant.

On veut obtenir :

```
| ?- mouvement_obligatoire(prateau(C1,C2,C3,C4,C5,C6,C7,C8,C9),X).
no

| ?- mouvement_obligatoire(plateau(x,C2,C3,C4,x,o,o,C8,C9),X).
X = 9
yes
```

x		
	x	o
o		!!!