Master d'ingénierie Informatique de Paris Diderot - Paris 7
M1: Prolog et programmation par contraintes

Examen partiel du 31 octobre 2013- Durée: 1h45

Documents autorisés; le barème est donné à titre indicatif.

Exercice 1 (**5 points**) Pour chacune des 10 requêtes suivantes, donner le résultat renvoyé par l'interpréteur Prolog (sans justifier) :

1. `(!-> X=3; X=0),X<2.`
2. `(X=1,!); X=2.`
3. `[X,Y|Z]=[Z,1,2].`
4. `X=3,X==3,X=:=3.`
5. `X==3,X=:=3,X=3.`
6. `X=:=3,X=3,X==3.`

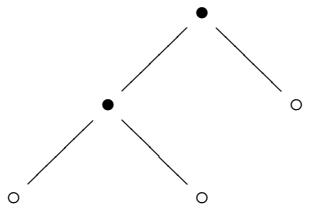
Dans les questions suivantes, on suppose que le fichier ci-dessous ait été consulté:

```
/* fichier fact.pl */
fact1(0,1).
fact1(N,R) :- N>0, P is N-1, fact1(P,V), R is N*V.
fact2(X,1) :- X=:=0.
fact2(N,R) :- N>0, P is N-1, fact2(P,V), R is N*V.
/* fin de fact.pl */
```

7. `fact1(3-3,R).`
8. `fact2(3-3,R).`
9. `fact1(7-3,R).`
10. `fact2(7-3,R).`

Exercice 2 (**7 points**)

On représente les arbres binaires non étiquetés en Prolog à l'aide d'une constante `f` (*feuille*) et d'un symbole de fonction binaire `n` (*nœud*). Par exemple, le terme `n(n(f,f),f))` désigne l'arbre binaire dessiné ci-dessous:



Les prédicats suivants sont à implémenter par récurrence, le cas de base étant la feuille.

1. Écrire un prédicat `feuilles(+arbre,-resultat)` qui prend un terme désignant un arbre comme premier argument, et renvoie comme résultat le nombre de feuilles de cet arbre. Par exemple

```
| ?- feuilles(n(n(n(f,f),f),n(f,f)),R).
R = 5
yes
```

2. Écrire un prédicat `miroir(+arbre,-resultat)` qui prend un terme désignant un arbre comme premier argument, et renvoie comme résultat le terme désignant *l'arbre miroir* de l'argument, c.à.d. l'arbre obtenu en échangeant partout fils gauches et droits. Par exemple

```
| ?- miroir(n(n(n(f,f),f),n(f,f)),R).
R = n(n(f,f),n(f,n(f,f)))
yes
```

3. Écrire un prédicat `chemins(+arbres,-résultat)` qui prend un terme désignant un arbre comme premier argument, et renvoie comme résultat la liste de tous le chemins de cet arbre. Un chemin est une liste de `g` (pour "aller à gauche") et de `d` (pour "aller à droite"), qui mène de la racine à une feuille (par exemple, la liste des chemins de l'arbre dessiné ci-dessus est `[[g,g],[g,d],[d]]` Un autre exemple:

```
| ?- chemins(n(n(n(f,f),f),n(f,f)),R).
R = [[g,g,g],[d,g],[g,d],[d,d],[g,g,d]] ?
yes
```

   Les chemins peuvent être listés dans un ordre quelconque.

Les trois prédicats ci-dessus, et bien d'autres, peuvent être implémentes comme instances d'un prédicat d'itération sur les arbres binaires (inspiré par le *fold_right* sur les listes), qu'on appellera `reduce(+p,+a,+e,-résultat)` où `p` est un symbole de prédicat à trois arguments, implémentant une opération binaire, `a` est un arbre, `e` est l'élément neutre de l'opération implémentée par `p`.

Une spécification de `reduce` est la suivante: si l'arbre `a` est une feuille on renvoie l'élément neutre `e`, sinon on applique récursivement `reduce` aux fils de `a`, puis on applique l'opération implémentée par `p` aux résultats partiels ainsi obtenus, et on renvoie le résultat.

Voici un exemple d'utilisation de `reduce`: si le prédicat ternaire `max_plus_un` est défini par

```
max_plus_un(X,Y,R):-X>Y,!,R is X+1.
max_plus_un(_,Y,R):-R is Y+1.
```

on pourra calculer la hauteur d'un arbre, par exemple de `n(n(n(f,f),f),f)`, comme suit:

```
| ?- reduce(max_plus_un,n(n(n(f,f),f),f),0,R).

R = 3 ?
yes
```

4. Écrire le prédicat `reduce(+p,+a,+n,-résultat)` (on pourra utiliser le prédicat prédéfini `=..`).

5. Re-implémenter les prédicats des points 1,2 et 3 de cet exercice en utilisant `reduce`. Il faudra implémenter les trois prédicats ternaires qui conviennent et donner ensuite les requêtes équivalentes à `feuilles(A,R)`, `miroir(A,R)` et `chemins(A,R)` respectivement, comme vu dans l'exemple du calcul de la hauteur d'un arbre.

Exercice 3 (**8 points**) Les questions de cet exercice ne sont pas indépendantes les unes des autres. Comme d'habitude, il est autorisé de sauter une question et d'utiliser dans le traitement des questions suivantes le prédicat qu'on était censé implémenter dans la question non traitée.

1. Écrire un prédicat `dec2bin_inv(+n,-res)` dont le premier argument est un entier n (qu'on suppose non négatif) et qui calcule la représentation de n en binaire **inversée** (c.à.d. chiffre moins significatif en tête), sous forme de liste de bits. Par exemple:

```
?- dec2bin_inv(32,L).
L = [0,0,0,0,0,1]
yes
```

Rappel: si `N` est un entier, `N mod 2` est le reste et `N//2` le quotient de la division entière de `N` par 2.

2. Écrire un prédicat `somme_xor(+l1,+l2,-res)` qui prends deux listes de bits (non nécessairement de même longueur) comme premier et deuxième arguments, et calcule la liste obtenue en appliquant la fonction *xor* ("ou exclusif") chiffre par chiffre aux deux arguments. Rappel: a *xor* $b = 1$ si et seulement si $a \neq b$. Par exemple:

```
| ?- dec2bin_inv(16,X),dec2bin_inv(32,Y),somme_xor(X,Y,Z).
X = [0,0,0,0,1]
Y = [0,0,0,0,0,1]
Z = [0,0,0,0,1,1] ?
yes
```

3. Écrire un prédicat `somme_xor_iteree(+l,-res)` qui prends une liste de listes de bits comme premier argument, et calcule la `somme_xor` de ces listes. Par exemple:

```
| ?- dec2bin_inv(16,W),dec2bin_inv(48,X),dec2bin_inv(19,Y),somme_xor_iteree([W,X,Y],Z).
W = [0,0,0,0,1]
X = [0,0,0,0,1,1]
Y = [1,1,0,0,1]
Z = [1,1,0,0,1,1] ?
yes
```

4. Écrire un prédicat `bin_inv2dec(+l,-res)` qui calcule l'inverse de `dec2bin_inv`.

   Par exemple:

```
| ?- bin_inv2dec([0,0,0,0,1],R).
R = 16
yes
```

5. La *somme nim* d'une liste d'entiers est l'entier correspondant à la somme xor itérée des représentations binaires des entiers de la liste. Par exemple, la somme nim de $1, 2, 4, 8$ est 15, celle de $2, 3, 8$ est 9 et celle de $5, 3, 6$ est 0. Écrire un prédicat `somme_nim(+l,-res)` qui prend une liste `l` d'entiers non négatifs et calcule leur somme nim, de la manière suivante:

   - on construit la liste des représentations binaires inversées des éléments de `l`, en mappant sur `l` le prédicat `dec2bin_inv`.
   - on applique le prédicat `somme_xor_iteree` à la liste ainsi obtenue.
   - on converti le résultat en utilisant `bin_inv2dec`

6. Le "jeu de Marienbad" vu en cours est l'instance de jeu de Nim dont la position initiale est $1, 3, 5, 7$. Un théorème dû au mathématicien americain Charles Bouton dit qu'une position $n_1, ..., n_k$ du jeu de Nim est gagnante si et seulement si la somme nim de $n_1, ..., n_k$ est strictement positive.[1].

   Vérifier en utilisant le théorème de Bouton que $1, 3, 5, 7$ est perdante.

   Écrire un prédicat `gagne(+l)` qui prends une liste d'entiers non négatifs représentant une position du jeu de Nim et réussit si cette position est gagnante, en utilisant le théorème de Bouton. Comparer ce prédicat au prédicat `gagne` vu en cours pour le jeu de Marienbad.

[1] Ceci vaut en toute généralité pour la "version normale" du jeu, dans laquelle le joueur qui enlève la dernière allumette gagne, mais aussi pour la version vue en cours, sous certaines conditions qu'on ne va pas spécifier ici.