Master d'Ingéniérie Informatique de Paris 7
M1: Prolog et programmation par contraintes

Examen partiel du 6 novembre 2007 - Durée: 1h45

Documents autorisés; le barème est donné à titre indicatif.

Exercice 1 (**3 points**) Pour chaque requête de la liste suivante, indiquer la réponse de l'interpreteur Prolog.

1. `X==3.`
2. `X=:=3.`
3. `X is 3.`
4. `X=3.`
5. `[[3]]=[X|L2].`
6. `f(X,g(Y))=f(Y,X).`

Exercice 2 (**5 points**)

Tous les prédicats auxiliaires utilisés dans la solution de cet exercice doivent être définis.

Nous désignons un mot binaire en Prolog par une liste de `0` et de `1`. Par exemple, la liste `[0,0,1,0]` désigne le mot `0010`.

Définir un prédicat `palyndrome(+liste_binaire)` tel que `palyndrome(L)` réussit si et seulement si `L` désigne un mot palyndrome, c.à.d. un mot dont les lectures de gauche à droite et de droite à gauche donnent le même résultat.

Par exemple:

```
  ?- palyndrome([0,1,1,0]).
yes
   ?-  palyndrome([1,0,1]).
yes
   ?- palyndrome([1,1,0]).
no
```

Exercice 3 (**6 points**)

Nous considérons les prédicats `membre_a/2,membre_b/2,membre_c/2,` définis par:

1. ```
   membre_a(X,[Y|_]):-X=Y,!.
   membre_a(X,[_|L]):-membre_a(X,L).
   ```
2. ```
   membre_b(X,[Y|_]):-!,X=Y.
   membre_b(X,[_|L]):-membre_b(X,L).
   ```
3. ```
   membre_c(X,[Y|_]):-X=Y.
   membre_c(X,[_|L]):-membre_c(X,L).
   ```

et les couples de termes suivantes:

(+,+): `(3,[4,3,7])`
   ```

(-,+): `(X,[4,3,7])`

(+,-): `(3,[X1,X2,X3]).`

Indiquer les reponses de l'interpreteur pour chacune des 9 requêtes:

1. `membre_a(3,[4,3,7]).`

⋮

9. `membre_c(3,[X1,X2,X3]).`

Dessiner les arbres de dérivation des requêtes

`membre_a(3,[4,3,7]).`

et

`membre_b(3,[4,3,7]).`

Exercice 4 (**6 points**) Deux jouers `J` et `O` jouent sur une liste d'entiers non négatifs. Chaque joueur, alternativement, doit:

- Soit retirer un entier de la liste.
- Soit décrémenter *d'une unité* un des entiers positifs de la liste.

Par exemple, les configurations atteignables par un coup à partir de `[0,2,3]` sont:
`[2,3]`, `[0,3]`, `[0,2]`, `[0,1,3]`, `[0,2,2]`
Le joueur qui vide la liste a perdu.

1. Définir un prédicat `move(+liste,-nouvelle_liste)` qui implémente la règle du jeu.

Un *arbre de jeu* est un arbre dont les noeuds sont etiquetés par des couples:

- état du jeu (une liste d'entiers non négatifs),
- joueur qui a la main (`J` ou `O`).

Un noeud d'un arbre de jeu d'étiquette `(etat_courant,_)` a autant de fils que d'états atteignables par un coup à partir de `etat_courant`.
L'état correspondant à une feuille est `[]`.
Un *arbre de jeu évalué* est un arbre de jeu dont les noeuds sont etiquetés par des triplets: état du jeu, joueur qui a la main, entier qui évalue la valeur du noeud (du point de vue de `J`).
Les valeurs des noeuds se calculent à partir des feuilles:

- La valeur d'une feuille `([],J)` est 1 (`J` a gagné). La valeur d'une feuille `([],O)` est -1 (`J` a perdu).
- Si `l` est non vide:
  - `(l,J)` vaut le maximum des valeurs de ses fils.
  - `(l,O)` vaut le minimum des valeurs de ses fils.

2. Dessiner l'arbre de jeu évalué de racine `([0,0,1],J)`.
3. Définir un prédicat `mimimax(+etat,+joueur,-valeur)` qui évalue, en utilisant la définition ci-dessus, la position de jeux `etat` quand `joueur` a la main.