M1 Informatique – Paris Diderot - Paris 7
Programmation Logique et par Contraintes

Examen partiel du 24 octobre 2017 - Durée: 1h50

Documents autorisés; le barème est donné à titre indicatif.

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

1. `[ [ Z | X ] ] = [ a , a ].`
2. `[ Z | X ] = [ [ a , a ] ].`
3. `[ [ Z | X ] ] = [ [ a , a ] ].`
4. `not ( ! ) ; true.`
5. `( ! , fail ) ; true.`
6. `bagof( X , ( X is 1 ; X = 1 ; X is 2 ) , L ).`
7. `setof( X , member( X , [ 1 , 1 , 2 , 3 ] ) , L ).`
8. `Q =.. [ member , X , [ 1 , 2 ] ] , Q.`
9. `Q =.. [ not , true ] , Q.`
10. `Q =.. [ not , true ] , not ( Q ) .`

Exercice 2 (**4 points**) On considère les définitions:

```
p1(P):-!,P,fail.
p1(_).
p2(P):-P,!,fail.
p2(_).
p3(P):-P,fail,!.
p3(_).
```

1. Donner les six arbres de dérivation des buts: `p1(true).` `p1(fail).` `p2(true).` `p2(fail).` `p3(true).` `p3(fail).`
2. Définir de manière aussi concise que possible des prédicats unaires `q1, q2, q3` tels que pour tout `P` le but `qi(P)` réussit si et seulement si e le but `pi(P)` réussit, pour `i=1,2,3`. Justifier.

Exercice 3 (**6 points**) De nombreuses opérations sur les listes peuvent être définies *par itération d'une operation binaire à partir de la droite* de la liste.

Voici le principe de l'itération de l'opération binaire `f` à partir de la droite de la liste `l`:

- si `l` est vide (cas de base): on renvoie le résultat approprié.
- si `l=[x|h]` (cas récursif): soit `y` le résultat de l'itération de `f` à partir de la droite de `h`. On renvoie `f(x,y)`.

En Prolog, l'opération binaire `f` sera implémentée par un prédicat ternaire `pf(+Arg1,+Arg2,-Res)`.

1. Définir, en suivant le principe exposé ci-dessus, les prédicats[1]:
   (a) `somme(+Liste_argument,-Resultat)` qui calcule la somme des éléménts d'une liste d'entiers par itération de l'operation d'addition à partir de la droite. Exemple:

   ```
   [eclipse 1]: somme([1,2,3],R).
   R = 6
   Yes (0.00s cpu)
   ```

   (b) `aplatir(+Liste_argument,-Resultat)` qui aplatit une liste de listes par itération de `append` à partir de la droite. Exemple:

[1] Vous supposerez que les listes passées en argument à ces prédicats sont bien formées: il s'agira de listes d'entiers pour `somme`, de listes de listes pour `aplatir` et ainsi de suite. Il n'est pas nécessaire de le vérifier.

```
[eclipse 2]: aplatir([[1,2],[3],[4,5]],R).
R = [1, 2, 3, 4, 5]
Yes (0.00s cpu)
```

(c) `pair(+Liste_argument,-Resultat)` qui compte le nombre d'entiers pairs dans une liste d'entiers par itération de l'opération qui sur `(x,y)` renvoie `y+1` si `x` est pair, `y` sinon, à partir de la droite. Exemple:

```
[eclipse 3]: pair([1,2,3,4,7,8,0],R).
R= 4
Yes (0.00s cpu)
```

2. Définir un prédicat d'ordre supérieur `fold_right(+Fonction,+Neutre,+Liste,-Result)` qui prend en argument (1) le nom du prédicat ternaire qui implémente l'opération binaire à itérer, (2) le résultat escompté sur la liste vide, (3) la liste sur laquelle on veut itérer, et qui renvoie (4) le résultat de l'itération de l'opération à partir de la droite de la liste. On aura par exemple:

```
[eclipse 4]: fold_right(append,[],[[1,2],[3],[4,5]],R).
R = [1, 2, 3, 4, 5]
Yes (0.00s cpu)
```

3. Si on voulait implémenter l'inversion des listes `reverse(L,R)` par itèration d'une opération binaire à partir de la droite de `L`, quelle serait l'opération à itérer? Serait-ce une implémentation efficace de `reverse` (justifier brièvement)?

4. Le principe de l'itération d'une opération binaire `f` *à partir de la gauche* de la liste `l` avec accumulateur `e` est le suivant:

   - si `l` est vide (cas de base): on renvoie `e`.
   - si `l=[x|h]` (cas récursif): on itère `f` à partir de la gauche de `h`, avec accumulateur `f(e,x)`.

   (a) Définir un prédicat `fold_left(+Fonction,+Accumulateur,+Liste,-Result)` qui implémente le principe d'itération à partir de la gauche.

   (b) Proposez une implementation de `reverse(L,R)` utilisant `fold_left`. Quelle est l'opération qu'on itère dans ce cas? Est-ce efficace (justifier brièvement)?

Exercice 4 (**5 points**) Une position du jeu "Fibonacci Nim" est un couple (n, m) d'entiers tels que $n \geq 0$ et $m > 0$: n représente le nombre de "pièces" qui restent en jeu, et m est le nombre maximum de piéces que le jouer qui à la main peut retirer.

Le joueur qui a la main choisit un entier $1 \leq k \leq min(n, m)$ et il retire k pièces de la position courante. La nouvelle position est alors $(n - k, 2k)$ (au coup suivant, son adversaire pourra retirer jusqu'au double des pièces qu'il vient de retirer).

Par exemple, à partir de $(7, 3)$ on peut jouer $(6, 2)$, $(5, 4)$, $(4, 6)$. Lorsque $n = 0$, c.à.d lorsqu'il ne reste plus de pièces pouvant être retirées, le jeu termine et le joueur qui à la main perd.

- Dessiner l'arbre de jeu à partir de la position $(4, 2)$. Le joueur *max* joue à la racine. Évaluer cet arbre à l'aide de l'algorithme MiniMax (une feuille vaut -1 si *max* doit jouer, 1 sinon).
- Écrire le prédicat `move(+Pos_courante,-Pos_suivante)` qui implémente la règle du jeu (utilisez des termes `(N,M)` pour les positions. Par exemple, la position $(4, 2)$ sera représentée par le terme `(4,2)`, tout simplement).
- Écrire le prédicat `gagne(+Pos_courante)` qui implémente la stratégie "force brute avec mémoire" pour le Fibonacci Nim.