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

Examen partiel du 23 octobre 2014 - Durée: 2h00
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. `f(X,3,4)=..L, X=15.`
2. `f(X,3,4)=..L, X==15.`
3. `f(X,3,4)=..L, X>15.`
4. `[X|[Y|[Z]]]=[1,2,3].`
5. `[X|[Y|Z]]=[1,2,3,4,5,6,7].`
6. `[X|[Y|[Z]]]=[1,2,3,4,5,6,7].`

Dans les questions suivantes, on suppose que le fichier `ex.pl` ci-dessous ait été compilé:

```
/*****ex.pl****/
ex1(X):-X=0.
ex2(X):-X=:=0.
ex3(X):-X==0.
/*************/
```

7. `ex1(1-1).`
8. `ex2(1-1).`
9. `ex3(1-1).`
10. `ex1(Z).`
11. `ex2(Z).`
12. `ex3(Z).`

Exercice 2 (**5 points**) On implémente les piles en Prolog par des liste; les opérations d'empilement et de dépilement se font sur la tête de la liste.

1. Écrire les prédicats:
   - `is_empty(?Pile)`[1].
   - `push(+Element,+Pile_avant_empilement,-Pile_apres_empilement)`,
   - `pop(+Pile_avant_depilement,-Pile_apres_depilement,-Element)`,

   qui impléméntent les opération de base sur les piles. Dans le reste de cet exercice, les piles seront gérées *uniquement* par le biais de ces trois prédicats, qu'on utilisera dans les solutions des points suivants. Il convient d'oublier que les piles sont implementée par des listes. En particulier, *on n'a pas le droit d'unifier une pile et une liste.*
2. On considère le programme:

[1] Si on appelle `is_empty` sur une liste, on teste si la liste est vide; si on l'appelle sur une variable, on crée une liste vide.

```
parse([a|L],P):-push(_,P,P1),parse(L,P1).
parse([b|L],P):-pop(P,P1,_),parse(L,P1).
parse([],P):-is_empty(P).
accept(L):-is_empty(P), parse(L,P).
```

Le premier argument de `parse` est une liste représentant un mot sur l'alphabet `a,b`, le deuxième est une pile

- Quel est l'ensembles des listes `L` closes, c'est à dire sans variables, telles que `accept(L)` réussit? Justifier brièvement.
- Modifier le prédicat `parse` de telle manière que l'ensembles des listes `L` telles que `accept(L)` réussit soit {`[]`,`[a,b]`,`[a,a,b,b]`,`[a,a,a,b,b,b]` ...} (un argument supplementaire, représentant l'état de l'automate à pile, sera nécessaire).

3. Écrire un prédicat `miroir(+Pile_Argument,-Pile_Resultat)` qui permet de renverser une pile (vous pouvez vous servir de prédicats auxiliaires).

Exercice 3 (**5 points**)

Soit `p/1` le prédicat défini par:

```
p(1).
p(2).
p(3).
p(4).
```

1. Dessiner l'arbre de dérivation de la requête `p(X)`.
2. Dessiner l'arbre de dérivation de la requête `p(X),!`.

On se propose de définir un prédicat `p2/1` tel que la requête `p2(X)` renvoie les deux premiers résultats renvoyés par `p(X)`, et termine. Voici deux définitions de `p2`, dont une seule est correcte.

(a)

```
p_aux(X,Y) :- p(X), p(Y), X\==Y, !.
p2(X) :- p_aux(X,_).
p2(X) :- p_aux(_,X).
```

(b)

```
p_aux(X) :- p(X) , !.
p2(X):- p_aux(X) ; p_aux(X).
```

3 Quel prédicat satisfait la spécification donnée, celui défini en (a) ou en (b) ci-dessus? Que fait l'autre prédicat? Justifiez vos réponses.

4 Proposez un prédicat `p3/1`, inspiré par `p2/1`[2], tel que la requête `p3(X)` renvoie les trois premiers résultats renvoyés par `p(X)`, et termine.

Exercice 4 (**5 points**)

Considérons le jeu combinatoire dont les positions sont des couples d'entiers non négatifs. A partir de la position (n, m), on peut jouer:

- $(n-1, m)$, à condition que $n > 0$.
- $(n-2, m)$, à condition que $n > 1$.
- $(n, m-1)$, à condition que $m > 0$.

[2]Cela veut dire que la réponse: `p3(1).p3(2).p3(3).` n'est pas celle attendue.

Le jeu est *normale*, c'est à dire que le joueur qui a la main quand la position est $(0,0)$ a perdu.

1. Dessiner l'arbre de jeu de racine $(2,2)$ et vérifier que le joueur qui a la main à la position $(2,2)$ possède une stratégie gagnante.

2. Considerons les deux implémentations suivantes de la règle du jeu:

```
move_a((X,Y),(X1,Y1)) :- (X>0, X1 is X-1, Y1 is Y);
                         (Y>0, X1 is X, Y1 is Y-1);
                         (X>1, X1 is X-2, Y1 is Y).

move_b((X,Y),(X1,Y1)) :- (X>1, X1 is X-2, Y1 is Y);
                         (X>0, X1 is X-1, Y1 is Y);
                         (Y>0, X1 is X, Y1 is Y-1).
```

   et les stratégies "force brute" correspondantes, données par

   `gagne_a((X,Y)):-move_a((X,Y),(Z,T)),not(gagne_a((Z,T))).`

   et

   `gagne_b((X,Y)):-move_b((X,Y),(Z,T)),not(gagne_b((Z,T))).`

   Expliquer le phénomène suivant:

```
[eclipse 1]: gagne_a((17,17)).
Yes (25.52s cpu, solution 1, maybe more) ?

[eclipse 2]:  gagne_b((17,17)).
Yes (0.02s cpu, solution 1, maybe more) ?
```

3. Considérons la stratégie "force brute avec mémoire" donnée par

   `gagne(X):- move_b(X,Y), not(gagne(Y)),asserta(gagne(X)).`

   Expliquer le phénomène suivant:

```
[eclipse 3]: gagne((200,200)).
Yes (22.51s cpu, solution 1, maybe more) ?

[eclipse 4]: gagne((201,201)).
Yes (0.80s cpu, solution 1, maybe more) ?
```