M1 Informatique – Université de Paris

Programmation Logique et par Contraintes

Examen partiel du 22 octobre 2020 - Durée: 1 heure et 50 minutes

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. `X = 1+1 , X =:= 2.`
2. `X =:= 1+1 , X = 2.`
3. `X == 1+1 , X = 1+1.`
4. `Q =.. [member,X,[1,2,trois]] , Q, number(X).`
5. `[1,2] = [X|Y] , (atomic(X) ; atomic(Y)).`
6. `[1,Z] = [X|Y] , ground(X) , ground(Y).`
7. `not(X = 0).`
8. `setof(X,X = 1,R) , setof(Y,Y = Z,S) , S = R.`
9. `X = 1 , !, (Y=2 ; Y=3).`
10. `! , X=1 , (Y=2 ; Y=3).`

Exercice 2 (**10 points**) On considère les constantes `a,c,g,t` représentant en Prolog les quatre bases de l'ADN. Un brin d'ADN est représenté par une liste de bases. Les bases `a` et `t` s'apparient entre elles, comme les bases `g` et `c`, et aucune autre paire de bases n'est possible. Deux brins d'ADN sont appariés si chaque paire de bases, à partir de la paire formée par les deux bases se trouvant en première position dans ces brins, l'est. Ainsi les brins `[a,c,t,t,g]` , `[t,g,a,a,c]` sont appariés, tandis que `[a,c,t,t,g]`,`[t,g,a,a]` et `[a]`,`[a]` ne le sont pas. Dans toutes les questions, on peut supposer que les listes représentant des brins d'ADN sont bien de listes de bases, sans devoir le tester.

0. Ecrire un prédicat `match(base1 , base2)` qui réussit si les deux bases sont appariés (faites simple, il suffit de 4 axiomes).

1. Ecrire un prédicat `paired(brin1 , brin2)` qui réussit si les deux brins sont appariés. Ce prédicat doit pouvoir s'utiliser en modalité `(+,+)` (pour tester l'appariement de deux brins) et `(+,-)` (pour produire l'apparié d'un brin donné).

2. Ecrire un prédicat `mismatches(+brin1 , +brin2 , -defauts)` qui, en étant donnés deux brins d'ADN supposés de même longueur, produit la liste des entiers correspondants aux positions de non-appariement. Par exemple `mismatches( [a,c,t,t,g],[t,c,a,g,c],L)`, donne `L=[2,4])`.

3. Ecrire un prédicat `repair(+brin1 , +defauts , +bases , -brin2)` qui prend un brin d'ADN, une liste d'entiers répresentants les défauts du brin et une liste de bases de même longueur que la liste de défauts et produit un brin ou les défauts sont corrigés.

   Par exemple `repair([t,c,a,g,c],[2,4],[g,a],L)` produit `L=[t,g,a,a,c]`.

4. Ecrire un prédicat `generate(+entier , -brin)` qui engendre tous les brins d'ADN de longueur `entier`, par des retours en arrière succéssifs (génération exhaustive). Par exemple `generate(1,L)` donne

   ```
   L = [a]
   Yes (0.00s cpu, solution 1, maybe more) ? ;
   L = [c]
   Yes (0.00s cpu, solution 2, maybe more) ? ;
   L = [g]
   Yes (0.00s cpu, solution 3, maybe more) ? ;
   L = [t]
   Yes (0.00s cpu, solution 4)
   ```

5. La distance entre deux brins d'ADN `b1` et `b2` (de même longueur) est le nombre de leur positions de non-appariement. En étant donnés une listes de brins `L` et un brin `b` (tous de même longueur) l'adéquation de `b` à `L` est la somme des distances entre `b` et chacun des élément de `L`. Ecrire une prédicat `fitness(+liste-brins , +brin , -adequation)` qui calcule l'adéquation d'un brin à une liste de brins.

   Par exemple `fitness([[c,c,a],[c,t,g],[a,c,c]],[g,g,g],N)` produit `N = 4`, car les distances entre le brin `[g,g,g]` et les brins `[c,c,a]`,`[c,t,g]`,`[a,c,c]` sont respectivement 1,2 et 1.

6. Ecrire un prédicat `fittest(+liste-brins , +entier , -brin , -adequation)` qui produit le brin le plus adéquats par rapport aux éléments de la liste `liste-brins`, qui sont tous de longueur `entier`, et son adéquation (si plusieurs brins conviennent, on en choisit un). Par exemple, `fittest([[a,c],[t,c]],2,B,F)` produit `B = [t, g], F = 1`.

Exercice 3 (**5 points**) On considère le jeu combinatoire suivant (c'est le "jeu soustractif {2,3,4}"): l'ensemble des positions est **N** (les entiers naturels). Le positions atteignables à partir de $n \in \mathbf{N}$ sont $n-2$, à condition que $n > 1$, $n-3$, à condition que $n > 2$ et $n-4$, à condition que $n > 3$.

1. Déssiner l'arbre de jeu à partir de la position 6 (6 est l'étiquette de la racine de l'arbre, il faut jouer tous les coups possibles à partir de chaque sommet de l'arbre).

2. Définir un predicat `move(+position-courante,-position-suivante)` pour ce jeu.

3. On peut décider si une position est gagnante en utilisant le prédicat suivant en Prolog:

   ```
   gagne(N):-move(N,M), not(gagne(M)).
   ```

   Dessiner les arbres de dérivation des requêtes `gagne(0)` et `gagne(2)`, en utilisant la définition de `gagne` ci-dessus et celle de `move` que vous avez donné au point précedent. Est-ce que `gagne(0)` reussit? Est-ce que `gagne(2)` reussit?

4. Quels sont les positions gagnantes de ce jeu?

5. Donner un prédicat qui `gagne2(+entier)` qui vérifie si une position est gagnante sans explorer l'arbre de jeu.