

M1 Informatique – Université Paris Cité
Programmation Logique et par Contraintes

Examen partiel du 23 octobre 2025 - Durée: 2 heures

Documents autorisés; le barème est donné à titre indicatif.

La lisibilité et les commentaires du code seront pris en compte dans l'évaluation.

Pour les exercices 2 et 3, vous pouvez :

- définir librement des **prédicats auxiliaires**, à condition de les expliquer;
- réutiliser les prédicats définis dans les questions précédentes, **même s'ils n'ont pas été réalisés**;
- pour l'exercice 2 : utiliser les prédicats standards du module `lists` de Prolog (`member/2`, `append/3`, `length/2`, etc.), ainsi que `between/3`, illustré dans l'exercice 1, pour la génération d'entiers.

Exercice 1 (6 points) Pour chacune des requêtes suivantes, donner le résultat renvoyé par l'interpréteur Prolog (sans justifier). Dans les cas où plusieurs résultats sont renvoyés par des retours en arrière successifs, les indiquer tous, séparés par de “;”. Le comportement du prédicat `between/3` est illustré ci-dessous :

```
?- between(1,3,N).  
N = 1 ;  
N = 2 ;  
N = 3
```

1. `member(X, [a,b]), !, member(Y, [a,b,c]).`
2. `Q=..[member, X, [1,2,3]],!,Q.`
3. `Q=..[member, X, [1,2,3]],Q,!.`
4. `setof(X, (X mod 2 =:= 0, between(2, 5, X)), L).`
5. `setof(X, (between(2, 5, X), X mod 2 =:= 0), L).`
6. `! -> (fail -> fail ; !) ; (! -> ! ; fail).`

Exercice 2 (7 points)

On s'intéresse à la décomposition d'un entier naturel n en produit de nombres entiers supérieurs ou égaux à 2. Une telle décomposition sera représentée par une liste de ces facteurs dont le produit vaut n .

1. Décomposition exhaustive — decomp/2

Définir un prédicat `decomp(+N, -L)` qui, pour un entier positif N , génère **toutes** les listes L de nombres ≥ 2 dont le produit est N . La décomposition de 1 est la liste vide.

Exemples :

```
?- decomp(1, L).  
L = [].  
  
?- decomp(12, L).  
L = [2, 6] ;  
L = [3, 4] ;  
L = [4, 3] ;  
L = [6, 2] ;  
L = [2, 2, 3] ;  
L = [2, 3, 2] ;  
L = [3, 2, 2] ;  
L = [12].
```

Indication : on pourra utiliser le prédicat `between/3` pour générer les facteurs possibles.

2. Test de primalité — `est_premier/1`

Définir un prédicat `est_premier(+N)` qui est vrai si et seulement si N est premier. On utilisera le prédicat précédent : un entier positif N est premier si et seulement si il admet **exactement une décomposition**, à savoir la liste `[N]`.

3. Facteurs premiers — `factor/2`

Définir un prédicat `factor(+N, -F)` qui renvoie la **liste croissante des facteurs premiers** de N .

Pour cela :

- on génère les décompositions L via `decomp(N, L)` ;
- on ne garde que celle qui est **croissante au sens large**, et dont tous les éléments sont **premiers**.

Exemples :

```
?- factor(12, F).  
F = [2, 2, 3].  
  
?- factor(60, F).  
F = [2, 2, 3, 5].
```

4. Plus grand commun diviseur — `pgcd/3`

Définir `pgcd(+N, +M, -K)` qui calcule le plus grand commun diviseur de deux entiers naturels. Procéder comme suit :

1. calculer les listes des facteurs premiers de N et M ;
2. déterminer leur intersection multienssembliste (facteurs communs avec multiplicités :
par exemple $[2,2,3,5], [2,2,3;3] \mapsto [2,2,3]$) ;
3. calculer le produit des facteurs communs.

Exemples :

```
?- pgcd(60, 36, K).  
K = 12.  
  
?- pgcd(21, 14, K).  
K = 7.
```

5. Plus petit commun multiple — `ppcm/3`

Définir `ppcm(+N, +M, -K)` à partir des factorisations de N et M :

- on effectue l'union multienssembliste des facteurs premiers :
par exemple $[2,2,3,5], [2,2,3;3] \mapsto [2,2,3,3,5]$;
- on calcule le produit des facteurs obtenus.

Exemples :

```
?- ppcm(6, 15, K).  
K = 30.  
  
?- ppcm(12, 18, K).  
K = 36.
```

Exercice 3 (7 points) On considère le jeu combinatoire suivant : une position est un **arbre binaire non étiqueté** défini par :

- `nil` pour un arbre vide (ou feuille) ;
- `t(G, D)` pour un nœud avec sous-arbre gauche `G` et droit `D`.

La position `nil` est finale, et les positions atteignables à partir de `t(G, D)` sont `G` et `D`. Le joueur qui atteint la position `nil` **gagne** (car son adversaire ne peut plus jouer).

Question 1. Définir le prédicat `move(+T, -S)` qui énumère tous les coups possibles depuis la position `T`.

```
?- move(t(nil, nil), S).  
S = nil ;  
S = nil.
```

Question 2. Définir le prédicat `wins(+T)` en utilisant la stratégie “brute force”.

Question 3. Dessiner les arbres de dérivation des requêtes `wins(nil)` et `wins(t(nil, nil))`.

Un arbre est *complet* si tous les chemins de la racine à une feuille ont la même longueur.

Question 4. Définir le prédicat `complete(+T)` qui réussit si `T` est un arbre complet.

Un arbre complet de hauteur impaire est une position gagnante au sens *fort* suivant : si t est un arbre complet de hauteur impaire et t' est atteignable à partir de t , alors t' est une position perdante.

Question 5. Définir le prédicat `complete_odd(+T)` qui réussit si `T` est un arbre complet de hauteur impaire.

Question 6. Donner une version optimisée du prédicat du point précédent qui visite chaque nœud de l'arbre `T` une seule fois (si le prédicat défini au point précédent ne le fait pas).

Question 7. Définir le prédicat `wins_strongly(+T)` qui implémente la définition de position gagnante au sens fort donnée ci-dessus. Autrement dit, `wins_strongly(T)` réussit si et seulement si pour tout arbre `T1` tel que `move(T, T1)` réussit, on a que `not(gagne(T1))` réussit.