INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Année universitaire 2022–2023

Programmation logique

Cahier de TP

Bertrand Coüasnon : bertrand.couasnon@insa-rennes.fr
Pascal Garcia : pascal.garcia@insa-rennes.fr
Yann Ricquebourg : yann.ricquebourg@insa-rennes.fr
Laurence Rozé : laurence.roze@insa-rennes.fr
Pascale Sébillot : pascale.sebillot@insa-rennes.fr

3e année
Département informatique
INSA de Rennes

Remerciements

Ce cahier de TP est le fruit du travail collectif de divers contributeurs au fil du temps : Bertrand Coüasnon, Mireille Ducassé, Pascal Garcia, Édouard Monnier, Laurence Rozé, Yann Ricquebourg, Pascale Sébillot...

Sommaire

Introduction

1 Introduction à ECL^{i}PS^e Prolog

Tous les TP seront réalisés sous Linux en ECL^{i}PS^e [1]. Vous trouverez des informations complémentaires sur Moodle.

1.1 Démarrer ECL^{i}PS^e

Voici un exemple de début de session :

```
$ eclipsep
ECLiPSe Constraint Logic Programming System [kernel]
...
Version 6.0 #162 (i386_linux), Mon Sep 20 06:09 2010
[eclipse 1]:
```

1.2 Compiler un fichier

Les fichiers ECL^{i}PS^e portent l'extension `ecl` ou `pl`. Un fichier `tp1.ecl` se charge dans l'environnement par « `["tp1"].` » (commande standard des environnements Prolog).

Les guillemets ne sont pas obligatoires ; cependant, si vous ne les utilisez pas, faites attention à ne pas débuter le nom du fichier par une lettre majuscule car Prolog l'interprétera alors comme le nom d'une variable.

1.3 Exécuter un but

Pour exécuter un but, il suffit d'entrer une requête suivie d'un point. Une première réponse est produite. L'utilisateur peut alors soit appuyer sur la touche `;` pour obtenir la réponse suivante, soit sur la touche `Entrée` pour arrêter la recherche de réponse(s).

Exemple :

```
[eclipse 2]: dessert(D).

D = sorbet_aux_poires
Yes (0.00s cpu, solution 1, maybe more) ? ;

D = fraises_chantilly
Yes (0.00s cpu, solution 2, maybe more) ?
[eclipse 3]:
```

1. ECL^{i}PS^e est téléchargeable gratuitement depuis le site du projet : http://eclipseclp.org.

1.4 Historique

La requête `h.` permet d'afficher l'historique de la session ECL^{i}PS^e courante.
Exemple :

```
[Eclipse 4]: h.
1 ["tp1"].
2 dessert(D).
3 dessert(melon_en_surprise).
```

Pour réexécuter un but de l'historique, il suffit de taper le numéro de ce but dans l'historique suivi d'un point :

```
 [Eclipse 4]: 3.
dessert(melon_en_surprise).

Yes (0.00s cpu)
```

Une autre façon encore plus pratique de gérer l'historique est de lancer ECL^{i}PS^e dès le départ sous le contrôle de rlwrap *(read-line wrapper)*, puis d'utiliser les flèches du clavier :

```
$ rlwrap eclipsep
ECLiPSe Constraint Logic Programming System [kernel]
...
[eclipse 1]:
```

1.5 Quitter ECL^{i}PS^e

Pour quitter ECL^{i}PS^e, vous pouvez utiliser le prédicat `halt` ou le raccourci `Ctrl`+`D`.

2 Déroulement des travaux pratiques

2.1 Structure de vos programmes

Le code de vos TP devra respecter l'ordre suivant :
— prédicat principal chargé de résoudre le problème posé (quand cela est pertinent),
— prédicats auxiliaires.

2.2 Qualité du code et conventions de codage

Vous devrez porter attention à la qualité du code et, plus particulièrement :
— utiliser des noms de prédicats et de variables significatifs (pas de p(X,Y)...) ;
— **indenter le code** ;
— ne lister **qu'un seul prédicat par ligne,** même, et surtout, si le prédicat est court. Même si cela peut paraître une perte de place, cela vous fera gagner beaucoup de temps lors de la mise au point. Cela permettra également aux enseignants de vous aider plus facilement ;
— ne pas « entrelarder » son code de commentaires : **regroupez tous les commentaires en en-tête d'un prédicat.** Un commentaire à l'intérieur d'un prédicat est souvent le signe qu'il faudrait découper ce prédicat en prédicat(s) auxiliaire(s) dont le nom pourrait véhiculer l'essentiel de ce qu'on voulait mettre dans le commentaire ;
— ne pas mettre de constantes en dur dans un programme ;
— faire du code modulaire (pas de prédicats « fourre-tout ») ;
— faire du code réutilisable lorsque cela est pertinent.

2.3 Documentation

La documentation d'ECL^{i}PS^e est disponible en ligne à l'url : http://eclipseclp.org/doc/bips.

Ajoutez des signets lors du premier TP sur ces pages et ayez systématiquement le *manuel de référence* ouvert lors des TP.

2.4 Mise au point

Pour finir, voici quelques éléments pour vous aider à détecter des erreurs dans vos programmes :

— **faites systématiquement disparaître les *warnings* du compilateur**. En effet, la plupart du temps les avertissements remontés par le compilateur sont des symptômes d'erreurs. De plus, les nombreux avertissements (ex : variables « singleton ») vous empêchent de remarquer les cas où le compilateur remonte un problème plus grave ;

— **utilisez le mode trace**. C'est le seul moyen de voir exactement ce qu'il se passe à l'exécution. Les prédicats à invoquer sont `trace` et `notrace` (voir chapitre 14, *Debugging* du manuel ECL^{i}PS^e ainsi qu'un résumé sur Moodle). En mode trace, l'état courant est résumé en une ligne (listant le nombre d'invocations, la profondeur, le port et le but) terminée par une invite de commande.
Exemple :
```
(1) 1 CALL  dessert(X)   %>
```
Les ports les plus couramment affichés sont CALL (un but est appelé), EXIT (un but a réussi), FAIL (un but a échoué), REDO (un but ayant déjà fourni une solution est appelé à nouveau). Les commandes de base sont `c` (*creep*, pour avancer d'un pas) et `h` (*help*, pour afficher l'aide).

— **Faites vos tests**. Vous devez absolument faire vos propres tests. Une fois que ceux ci sont faits, vous pouvez utiliser les tests qui vous sont fournis. Généralement pour chaque tp vous avez deux fichiers fournis un premier contenant généralement une base de faits à compléter, par exemple *tp1.pl* et un deuxième contenant des tests par exemple *tp1_test.pl*. Les tests ont été uniformisés sur les tps, vous pouvez les exécuter tous en exécutant : tests. Vous pouvez les exécuter un par un. Si par exemple vous devez écrire le prédicat *mon_predicat*, vous pouvez le tester en exécutant *test_mon_predicat*.

TP 1

Interrogation style base de données

L'objectif de ce premier TP est de se familiariser avec l'environnement et le fonctionnement de base de ECL^{i}PS^{e} Prolog : charger un fichier de clauses, explorer simplement les connaissances qu'il représente, écrire des prédicats pour extraire des réponses plus élaborées. Le travail sera réalisé successivement sur deux bases fournies : la base Menu, déjà utilisée en cours, qui vous permettra une prise en main, puis la base Valois.

Si ECL^{i}PS^{e} Prolog refuse de s'exécuter, indiquant qu'il y a un problème dans le fichier *.eclipse_history*, il suffit d'aller effacer ce fichier pour pouvoir de nouveau exécuter prolog.

1 Base Menu

1. Copier dans votre répertoire de travail la base Menu du fichier `basemenu.pl` présente sous Moodle.
2. Ouvrir votre version de ce fichier dans un éditeur de texte.
3. Exécuter vos prédicats dans une fenêtre de terminal.

1.1 Travail à réaliser

Question 1.1. Interroger la base afin de visualiser le contenu des relations. Par exemple vous pouvez demander à visualiser tous les hors d'oeuvres, tous les poissons.

Question 1.2. Écrire les règles correspondant aux définitions suivantes et poser des questions pour tester :

1) *Un plat de résistance est un plat à base de viande ou de poisson.*
2) *Un repas se compose d'un hors d'œuvre, d'un plat et d'un dessert.*
3) *Plat dont le nombre de calories est compris entre 200 et 400.*
4) *Plat plus calorique que le « Bar aux algues ».*
5) *Valeur calorique d'un repas.*
6) *Un repas équilibré est un repas dont le nombre total de calories est inférieur à 800.*

Question 1.3. Se mettre en mode trace et effectuer de nouveau des requêtes des questions précédantes. Vous pouvez maintenant suivre le mécanisme prolog de résolution des requêtes.

1.2 Contenu de la base Menu

```
hors_d_oeuvre(artichauts_Melanie).
hors_d_oeuvre(truffes_sous_le_sel).
hors_d_oeuvre(cresson_oeuf_poche).

viande(grillade_de_boeuf).
viande(poulet_au_tilleul).

poisson(bar_aux_algues).
poisson(saumon_oseille).

dessert(sorbet_aux_poires).
dessert(fraises_chantilly).
dessert(melon_en_surprise).
```

```
calories(artichauts_Melanie, 150).
calories(truffes_sous_le_sel, 202).
calories(cresson_oeuf_poche, 212).
calories(grillade_de_boeuf, 532).
calories(poulet_au_tilleul, 400).
calories(bar_aux_algues, 292).
calories(saumon_oseille, 254).
calories(sorbet_aux_poires, 223).
calories(fraises_chantilly, 289).
calories(melon_en_surprise, 122).
```

1.3 Tests finaux

1. Copier dans votre répertoire de travail la base Menu du fichier `basemenu_test.pl` présente sous Moodle.
2. Charger les prédicats correspondants aux tests dans votre environnement : `[basemenu_test]`.
3. Poser la question `tests.`.

2 Base Famille de France (De Valois)

Copier, dans votre répertoire de travail, le fichier `basevalois.pl` qui contient des informations sur la famille De Valois de rois et reines de France.

Divers prédicats sont prédéfinis dans `basevalois.pl` :

— homme(H) : *H est un homme* ;
— femme(F) : *F est une femme* ;
— pere(P, E) : *P est le père de E* ;
— mere(M, E) : *M est la mère de E* ;
— epoux(X, Y) : *X et Y sont mariés* ;
— roi(R, S, D1, D2) : *R de surnom S a régné de l'année D1 à l'année D2.*

2.1 Travail à réaliser

Question 2.1. Écrire les règles qui définissent les liens de parenté suivants et poser des questions pour tester chacun de ces prédicats :

— enfant(E,P) : *E est un enfant de P* ;
— parent(P,E) : *P est un parent direct de E, P est donc le père ou la mère de E* ;
— grand_pere(G,N) : *G est un grand-père de N* ;
— frere(F,N) : *F est un frère de N* ;
— oncle(O,N) : *O est un oncle de N* ;
— cousin(C,N) : *C est un cousin de N* ;
— le_roi_est_mort_vive_le_roi(R1,D,R2) : *en l'an D, le règne du roi R1 se termine et celui du roi R2 débute.*

Interroger la base et utiliser le mode trace pour explorer l'arbre de recherche.

Question 2.2. Définir le prédicat *ancetre(X,Y)* exprimant que X est un ancêtre de Y. S'intéresser à son algorithme en variant l'ordre des clauses et l'ordre des buts dans la récursivité.

Constater les effets en posant des questions appropriées, en faisant varier le mode d'utilisation du prédicat.

2.2 Contenu de la base Valois

```
homme(charles_V).
homme(charles_VI).
homme(charles_VII).
homme(louis_XI).
homme(charles_VIII).
homme(louis_XII).
homme(francois_I).
homme(henri_II).
homme(francois_II).
homme(charles_IX).
homme(henri_III).
homme(jean_II).
homme(philippe_VI).
homme(charles_d_Orleans).
homme(charles_de_Valois).
homme(louis_d_Orleans).
homme(jean_d_angouleme).
homme(charles_d_angouleme).

femme(anne_de_cleves).
femme(louise_de_Savoie).
femme(claude_de_france).
femme(anne_de_Bretagne).
femme(catherine_de_medicis).
femme(charlotte_de_Savoie).
femme(marie_d_anjou).
femme(isabeau_de_Baviere).
femme(valentine_de_milan).
femme(jeanne_de_Bourbon).
femme(bonne_de_luxembourg).
femme(jeanne_de_Bourgogne).
femme(marie_Stuart).
femme(elisabeth_d_autriche).
femme(louise_de_lorraine).
femme(marguerite_de_Rohan).

mere(marguerite_de_Rohan, charles_d_angouleme).
mere(jeanne_de_Bourgogne, jean_II).
mere(bonne_de_luxembourg, charles_V).
mere(jeanne_de_Bourbon, charles_VI).
mere(jeanne_de_Bourbon, louis_d_Orleans).
mere(valentine_de_milan, charles_d_Orleans).
mere(valentine_de_milan, jean_d_angouleme).
mere(isabeau_de_Baviere, charles_VII).
mere(marie_d_anjou, louis_XI).
mere(charlotte_de_Savoie, charles_VIII).
mere(anne_de_Bretagne, claude_de_france).
mere(claude_de_france, henri_II).
mere(anne_de_cleves, louis_XII).
mere(louise_de_Savoie, francois_I).
mere(catherine_de_medicis, francois_II).
mere(catherine_de_medicis, charles_IX).
mere(catherine_de_medicis, henri_III).

epoux(marguerite_de_Rohan, jean_d_angouleme).
epoux(louise_de_lorraine, henri_III).
epoux(elisabeth_d_autriche, charles_IX).
epoux(marie_Stuart, francois_II).
epoux(jeanne_de_Bourgogne, philippe_VI).
epoux(bonne_de_luxembourg, jean_II).
epoux(jeanne_de_Bourbon, charles_V).
epoux(valentine_de_milan, louis_d_Orleans).
epoux(isabeau_de_Baviere, charles_VI).
```

```
epoux(marie_d_anjou, charles_VII).
epoux(charlotte_de_Savoie, louis_XI).
epoux(catherine_de_medicis, henri_II).
epoux(anne_de_cleves, charles_d_Orleans).
epoux(louise_de_Savoie, charles_d_angouleme).
epoux(claude_de_france, francois_I).
epoux(anne_de_Bretagne, charles_VIII).
epoux(anne_de_Bretagne, louis_XII).
epoux(H, F) :- homme(H), femme(F), epoux(F, H).

pere(louis_XII, claude_de_france).
pere(charles_de_Valois, philippe_VI).
pere(philippe_VI, jean_II).
pere(jean_II, charles_V).
pere(charles_V, charles_VI).
pere(charles_VI, charles_VII).
pere(charles_VII, louis_XI).
pere(charles_d_Orleans, louis_XII).
pere(charles_d_angouleme, francois_I).
pere(francois_I, henri_II).
pere(henri_II, francois_II).
pere(henri_II, charles_IX).
pere(henri_II, henri_III).
pere(louis_d_Orleans, charles_d_Orleans).
pere(charles_V, louis_d_Orleans).
pere(jean_d_angouleme, charles_d_angouleme).
pere(louis_d_Orleans, jean_d_angouleme).

roi(charles_V, le_sage, 1364, 1380).
roi(charles_VI, le_bien_aime, 1380, 1422).
roi(charles_VII, xx, 1422, 1461).
roi(louis_XI, xx, 1461, 1483).
roi(charles_VIII, xx, 1483, 1498).
roi(louis_XII, le_pere_du_peuple, 1498, 1515).
roi(francois_I, xx, 1515, 1547).
roi(henri_II, xx, 1547, 1559).
roi(francois_II, xx, 1559, 1560).
roi(charles_IX, xx, 1560, 1574).
roi(henri_III, xx, 1574, 1589).
roi(jean_II, le_bon, 1350, 1364).
roi(philippe_VI, de_valois, 1328, 1350).
```

TP 2

Manipulation de termes construits

L'objectif de ce TP est de manipuler des termes construits afin de maîtriser cette notion fondamentale en Prolog. L'application sur laquelle vous allez travailler est un jeu de cartes qui vous permettra de construire des termes tels que *carte* ou *main*. Attention à ne pas confondre un terme construit et un prédicat.

1 Sujet : le monde du poker

Chaque carte d'un jeu de 52 cartes a
— une hauteur (*deux, trois, quatre, cinq, six, sept, huit, neuf, dix, valet, dame, roi, as*) et
— une couleur (*trefle, carreau, cœur, pique*).
Les hauteurs comme les couleurs sont ici données en ***ordre croissant***.

Au poker, une main est constituée de cinq cartes.

2 Questions

Question 2.1. Écrire le prédicat *est_carte*, à **un seul argument**, définissant une carte du jeu. La requête : $est\_carte(C)$ doit donc réussir 52 fois.

Question 2.2. Écrire le prédicat *est_main*, à **un seul argument**, définissant une main. La requête : $est\_main(M)$ doit énumérer toutes les mains possibles d'un jeu de 52 cartes. Il faut bien sûr imposer que toutes les cartes d'une main soient différentes !

Pour évaluer plus facilement une main, il est intéressant d'avoir les cinq cartes en ***ordre croissant***. Pour cela, il faut définir la relation *inferieure* entre toutes cartes $C1$ et $C2$.

$$\begin{array}{rll} C1 < C2 & \text{si} & \text{hauteur}(C1) < \text{hauteur}(C2) \\ & & \text{ou} \\ & \text{si} & \text{hauteur}(C1) = \text{hauteur}(C2) \text{ et} \\ & & \text{couleur}(C1) < \text{couleur}(C2) \end{array}$$

On s'intéresse au prédicat $inf\_carte(C1, C2)$ qui réussit quand la carte $C1$ est *inferieure* à la carte $C2$.

Question 2.3. Définir le prédicat *inf_carte* et l'utiliser pour connaître toutes les cartes inférieures au cinq de cœur.

Pour toutes les questions suivantes, le but est de vérifier la propriété sans regarder si elle est « maximale ». Par exemple le prédicat *une_paire* aboutira à un succès même si la main contient un brelan.

Question 2.4. Écrire le prédicat *est_main_triée* à **un seul paramètre**, indiquant si l'élément passé en paramètre est une main triée.

Dans la suite, nous allons écrire des prédicats évaluant une main triée, vous commencerez par vérifier que la main est effectivement triée.

Question 2.5. Écrire le prédicat *une_paire*(M). M étant instancié, ce prédicat réussit si M contient 2 cartes de même hauteur.

Question 2.6. Écrire le prédicat *deux_paires*(M). M étant instancié, ce prédicat réussit si M contient 2 fois deux cartes de même hauteur.

Question 2.7. Écrire le prédicat *brelan*(M). M étant instancié, ce prédicat réussit si M contient 3 cartes de même hauteur.

Question 2.8. Écrire le prédicat *suite*(M). M étant instancié, ce prédicat réussit si M contient 5 cartes dont les hauteurs se suivent.

Question 2.9. Écrire le prédicat $full(M)$. M étant instancié, ce prédicat réussit si M contient une paire et un brelan (les cartes de la paire et du brelan étant bien sûr disjointes).

TP 3

Listes

L'objectif de ce TP est de se familiariser avec les listes Prolog, et de maîtriser l'utilisation des modes des prédicats. Outre des prédicats nouveaux, le TP porte aussi sur certains prédicats vus en cours-TD : il est en effet intéressant d'analyser leur comportement et les modes qu'ils supportent.

Nous avons indiqué ici les modes des prédicats lorsque cela paraissait être une aide pertinente. Dans les autres cas, vous analyserez et indiquerez les modes supportés par vos solutions.

1 Quelques classiques sur les listes

Question 1.1. Programmez et testez chacun des prédicats suivants, en utilisant le traceur en cas de problème. Il sera intéressant de tester et vérifier les modes effectivement supportés. Après vos propres tests, vous pourrez lancer les tests automatiques fournis pour corroborer vos tests.

- *membre(?A,+X)* : A est élément de la liste X.
- *compte(+A,+X, ?N)* : N est le nombre d'occurrences de A dans la liste X.
- *renverser(+X, ?Y)* : Y est la liste X à l'envers.
- *palind(+X)* : X est une liste palindrome.
- *enieme(+N,+X,–A)* : A est l'élément de rang N dans la liste X.
- *hors_de(+A,+X)* : A n'est pas élément de la liste X.
- *tous_diff(+X)* : les éléments de la liste X sont tous différents.
- *conc3(+X,+Y,+Z, ?T)* : T est la concaténation[1] des listes X, Y et Z.
- *debute_par(+X, ?Y)* : la liste X débute par la liste Y.
- *sous_liste(+X, ?Y)* : la liste Y est sous-liste de la liste X.

1. On pourra réutiliser le prédicat concat/3 programmé en cours, sans que cela soit obligatoire.

— *elim(+X,–Y)* : la liste X étant donnée, on construit la liste Y qui contient tous les éléments de X une seule fois.

— *tri(+X,–Y)* : la liste Y est le résultat du tri par ordre croissant de la liste d'entiers X. Vous pourrez, par exemple, créer un prédicat auxiliaire *inserer(+E,+L1,-L2)* qui insère l'entier E à sa place dans la liste $L1$ déjà triée par ordre croissant, pour produire la liste $L2$.

2 Retour sur les modes et la génération de solutions

Comme évoqué dans la question précédente, vous avez certainement constaté que certains prédicats ne fonctionnaient pas dans d'autres modes que ceux demandés. Nous allons essayer d'en augmenter certaines capacités, afin par exemple de pouvoir obtenir une production de solutions dans plus de modes.

— *enieme(+N,+X,–A)* :
Proposer une seconde version *enieme2* permettant le mode (–,+,+).
Disposant des deux versions, peut-on écrire une solution *eniemefinal* combinant les possibilités des deux ?
Idée : réussir à tester le mode d'appel pour orienter sur la bonne version.

— *conc3(+X,+Y,+Z, ?T)* :
conc3 sait-il découper la liste T de toutes les façons possibles ? En d'autres termes, peut-on obtenir un seul algorithme *conc3final* qui permet aussi le mode (–,–,–,+) ? Si non, écrivez la variante.
Idée : s'inspirer de l'algorithme réalisant concat/3.

— *compte(+A,+X, ?N)* :
Votre solution fonctionne-t-elle en mode (–,+,+) ou (–,+,–) ? Si non voyez-vous une solution pour écrire *comptefinal* et permettre ces modes où A doit être trouvé ?
Idée : réfléchir au moment du test d'égalité, et à la différence entre = et == en Prolog.

TP 4

Arbres binaires

En cours et au TP précédent, nous avons manipulé une première sorte de structure récursive binaire : les listes, construites à l'aide d'une constante symbolique [] permettant d'arrêter la récursion, et d'un symbole fonctionnel d'arité deux (le point). Nous nous intéressons à nouveau ici à Prolog en tant que langage offrant la possibilité de calculer dynamiquement des structures potentiellement infinies dont le type est défini récursivement (on parle de l'aspect programmation récursive de Prolog) en nous focalisant sur une seconde structure récursive binaire : les arbres binaires.

D'autre part, sur un plan plus « technique », ce TP va également être pour vous l'occasion d'utiliser la coupure (le *cut*). Toutefois, veillez à en faire un usage aussi limité que possible, c'est-à-dire à ne l'introduire dans vos prédicats que lorsque vous avez une vraie raison de le faire (voir remarque en début de section 2 à ce sujet).

1 Représentation des arbres binaires

On choisit de représenter un arbre binaire par un terme construit à l'aide des 2 symboles fonctionnels suivants :

- une constante symbolique *vide* permettant de représenter un arbre vide ;
- un symbole fonctionnel d'arité 3 *arb_bin* tel que *arb_bin(R, G, D)* représente l'arbre non vide dont la racine est étiquetée R, et dont le sous-arbre gauche (respectivement droit) est G (respectivement D).

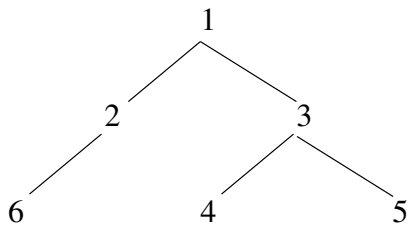

FIGURE 4.1 – Arbre binaire d'entiers

Ainsi, l'arbre de la figure 4.1 correspond à la représentation :
arb_bin(1, arb_bin(2, arb_bin(6, vide, vide), vide), arb_bin(3, arb_bin(4, vide, vide), arb_bin(5, vide, vide))).

Bien que les prédicats demandés ci-dessous soient, en règle générale, non dédiés à un type d'arbre binaire particulier, on se focalisera, par souci de simplification, sur les seuls arbres

binaires d'entiers. Sauf mention explicite du contraire (voir questions vers la fin du TP), ces arbres binaires d'entiers seront quelconques, c'est-à-dire ni triés, ni équilibrés.

2 Questions

Remarque générale : si, au cours de la rédaction de vos prédicats, vous utilisez des coupures, vous devez expliquer, dans votre compte rendu, la raison qui vous les ont fait introduire (et ceci pour chaque prédicat où vous en insérez une).

Question 2.1. Écrire le prédicat *arbre_binaire(+B)* qui réussit si B est un arbre binaire d'entiers. On utilisera le prédicat *integer/1* pour tester si l'étiquette d'un nœud donné est un entier.

Question 2.2. Écrire le prédicat *dans_arbre_binaire(+E, +B)* qui réussit si E est l'une des étiquettes de l'arbre binaire B.

Question 2.3. Écrire le prédicat *sous_arbre_binaire(+S, +B)* qui réussit si S est un sous-arbre de B.

Question 2.4. Écrire le prédicat *remplacer(+SA1, +SA2, +B, -B1)* où *B1* est l'arbre B dans lequel toute occurrence du sous-arbre *SA1* est remplacée par le sous-arbre *SA2*.

Question 2.5. Deux arbres binaires B1 et B2 sont isomorphes si B2 peut être obtenu par réordonnancement des branches des sous-arbres de B1. L'isomorphisme définit donc la notion d'identité de deux arbres binaires. Ainsi, dans la figure 4.2, les 2 premiers arbres sont isomorphes, mais pas le troisième.

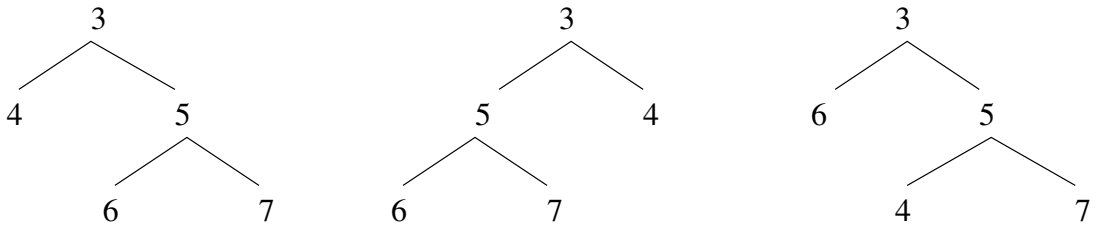

FIGURE 4.2 – Deux premiers arbres isomorphes, et un 3^e non isomorphe aux 2 autres

Écrire le prédicat *isomorphes(+B1, +B2)* qui réussit si les arbres *B1* et *B2* sont isomorphes.

Question 2.6. Pour récupérer les informations présentes dans les étiquettes des différents nœuds d'un arbre binaire, il y a trois possibilités de parcours :

- le parcours *préfixe* qui récupère d'abord l'information présente au nœud, puis les informations du sous-arbre gauche, et enfin celles du sous-arbre droit ;
- le parcours *postfixe* qui récupère d'abord les informations du sous-arbre gauche, puis celles du sous-arbre droit et enfin l'information au nœud ;
- le parcours *infixe* qui récupère d'abord les informations du sous-arbre gauche, puis l'information au nœud, et enfin celles du sous-arbre droit.

Appliqués à l'arbre de la figure 4.1, ces parcours produisent donc :

- pour le parcours préfixe : 1, 2, 6, 3, 4, 5
- pour le parcours postfixe : 6, 2, 4, 5, 3, 1
- pour le parcours infixe : 6, 2, 1, 4, 3, 5

Écrire le prédicat *infixe(+B, -L)* qui construit la liste L des informations contenues dans l'arbre binaire B par parcours infixe.

Un arbre binaire B est dit **ordonné** si pour tout nœud N de cet arbre B, les étiquettes des nœuds du sous-arbre gauche de N sont plus petites que l'étiquette du nœud N, et toutes les étiquettes des nœuds du sous-arbre droit de N sont plus grandes que l'étiquette N.

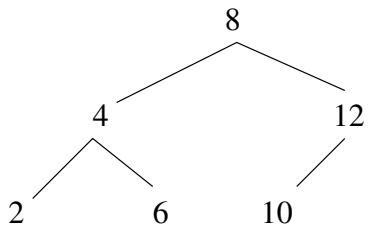

FIGURE 4.3 – Arbre binaire d'entiers ordonné

L'arbre de la figure 4.3 est, par exemple, un arbre binaire ordonné, alors que celui de la figure 4.1 ne l'est pas.

Pour les 2 dernières questions, on considérera des arbres binaires d'entiers **ordonnés** et, pour simplifier les comparaisons, on admettra, sans le vérifier, que toutes les étiquettes d'un arbre portent une valeur différente.

Question 2.7. Écrire le prédicat *insertion_arbre_ordonne(+X, +B1, -B2)* qui réussit si *B2* est l'arbre ordonné d'entiers obtenu par l'insertion de la valeur X dans l'arbre ordonné d'entiers *B1*.

TP 5

Arithmétique

Dans ce TP, nous allons évaluer des expressions arithmétiques. Nous allons voir comment représenter des nombres et les évaluer à partir des principes premiers de Prolog. L'objectif est donc d'essayer de se passer des entiers et du prédicat spécial `is` pour évaluer les expressions arithmétiques. En effet, le prédicat `is` ne s'évalue pas de la même façon qu'un prédicat « normal ». Par exemple :

```
?- 5 is X + 2.
instantiation fault in +(X, 2, _260)
Abort
```

On voit ici que l'évaluation de `5 is X + 2` produit une erreur alors qu'on aurait pu s'attendre à obtenir `X = 3` ou, à la limite, un échec.

1 Arithmétique de *Peano*

En cours de Prolog a été vue l'arithmétique de *Peano*. C'est en se basant sur cette arithmétique que nous allons créer notre première représentation des nombres et certains opérateurs associés. Nous aurons donc :

- le terme `zero` qui représentera le nombre 0 ;
- `s(X)` qui représentera le successeur du nombre `X`. Par exemple `s(s(zero))` représentera le nombre 2.

Question 1.1. Écrire le prédicat permettant de calculer la somme de deux entiers de *Peano* :

```
%add(?, ?, ?): peano number * peano number * peano number
%add(Op1, Op2, Add) avec Add = Op1 + Op2
```

Par exemple :

```
?- add(s(zero), s(s(zero)), Sum).
Sum = s(s(s(zero)))
```

Les formules suivantes :

$$\forall x(zero + x = x)$$
$$\forall x \forall y(s(x) + y = s(x + y))$$

sont deux axiomes de l'arithmétique de *Peano* et devraient vous aider à élaborer ce prédicat. Contrairement au prédicat `is`, nous allons pouvoir utiliser le prédicat `add` de différentes manières, par exemple en ayant qu'une seule variable instanciée :

```
?- add(X, Y, s(s(zero))).
X = zero
Y = s(s(zero))
Yes

X = s(zero)
Y = s(zero)
Yes

X = s(s(zero))
Y = zero
Yes
```

Question 1.2. Écrire le prédicat permettant de calculer la différence entre deux entiers de *Peano* :

```
%sub(?, ?, ?): peano number * peano number * peano number
%sub(Op1, Op2, Sub) avec Sub = Op1 - Op2
```

Question 1.3. Écrire le prédicat permettant de calculer le produit de deux entiers de *Peano* :

```
%prod(+, +, -): peano number * peano number * peano number
%prod(Op1, Op2, Prod) avec Prod = Op1 * Op2
```

Par exemple :

```
?- prod(s(s(zero)), s(s(s(zero))), Prod).
Prod = s(s(s(s(s(zero)))))
Yes
```

Les axiomes suivants vous seront utiles :

$$\forall x(zero \times x = zero)$$
$$\forall x \forall y(s(x) \times y = x \times y + y)$$

Notons que le mode du prédicat cette fois n'est pas `prod(?, ?, ?)`. Même si le prédicat que vous allez écrire pourra fonctionner dans différents modes, sans un peu plus de travail, il pourra se mettre à boucler dans la recherche d'une solution pour certains modes. Il n'est pas trop difficile de l'adapter pour tous les modes, mais ce n'est pas notre objectif [1].

Question 1.4. Écrire le prédicat permettant de calculer la factorielle d'un entier de *Peano* :

```
%factorial(+, -): peano number * peano number
%factorial(N, Fact) avec Fact = N!
```

Par exemple :

```
?- factorial(s(s(s(zero))), F).
F = s(s(s(s(s(s(zero))))))
Yes
```

Notons que cette représentation prend beaucoup de place pour représenter un nombre ($3 \times n + 1$ pour un entier n). C'est en plus illisible et peu efficace (vous pouvez essayer de calculer la factorielle de 12 par exemple).

1. On peut y arriver en utilisant un prédicat inférieur par exemple.

2 Représentation binaire

Nous allons toujours représenter des nombres à partir des principes premiers, mais, cette fois-ci, de manière plus efficace tant en terme de place pour la représentation qu'en efficacité de calcul. Nous allons pour ce faire utiliser la représentation binaire d'un nombre (on aurait pu utiliser la représentation décimale, mais cela nécessite l'écriture de plus de prédicats). Pour représenter un nombre n, il faudra ici uniquement $\lfloor \log n \rfloor + 1$ bits (pour $n > 0$).

Un nombre en binaire est représenté par une liste. Les bits de poids faibles sont en tête de liste, contrairement à la représentation traditionnelle, car ceci va faciliter l'écriture des prédicats. Par exemple 4 est représenté par la liste `[0, 0, 1]` et 12 par `[0, 0, 1, 1]`. Le nombre 0 est représenté de plusieurs manières : `[]` (liste vide) ou `[0]` (ou comme tout nombre en mettant des 0 inutiles en poids forts (`[0, 0]`, `[0, 0, 0]`, ...)).

On vous donne (voir le squelette) le prédicat suivant :

```
%add_bit(?, ?, ?, ?, ?): bit * bit * bit * bit * bit
%add_bit(Bit1, Bit2, CarryIn, Res, CarryOut)
add_bit(0, 0, 0, 0, 0).
add_bit(0, 0, 1, 1, 0).
add_bit(0, 1, 0, 1, 0).
add_bit(0, 1, 1, 0, 1).
add_bit(1, 0, 0, 1, 0).
add_bit(1, 0, 1, 0, 1).
add_bit(1, 1, 0, 0, 1).
add_bit(1, 1, 1, 1, 1).
```

Ce prédicat permet d'additionner deux bits et un bit de retenue et de récuperer le bit résultat et la retenue en sortie.

Question 2.1. Écrire le prédicat permettant de calculer la somme de deux entiers en représentation binaire :

```
%add(?, ?, ?): bit list * bit list * bit list
%add(Op1, Op2, Sum) avec Sum = Op1 + Op2
```

Par exemple :

```
add([1], [0, 0, 1, 1], Sum).
Sum = [1, 0, 1, 1]
Yes
```

Question 2.2. Écrire le prédicat permettant de calculer la différence entre deux entiers en représentation binaire :

```
%sub(?, ?, ?): bit list * bit list * bit list
%sub(Op1, Op2, Sub) avec Sub = Op1 - Op2
```

Question 2.3. Écrire le prédicat permettant de calculer le produit de deux entiers en représentation binaire :

```
%prod(+, +, -): bit list * bit list * bit list
%prod(Op1, Op2, Prod) avec Prod = Op1 * Op2
```

Là encore, on pourrait faire en sorte que ce prédicat fonctionne dans tous les modes mais il faudrait un peu plus de travail.

Question 2.4. Écrire le prédicat permettant de calculer la factorielle d'un entier en représentation binaire :

```
%factorial(+, -): bit list * bit list
%factorial(N, Fact) avec Fact = N!
```

Ceci permet de calculer les factorielles de grands nombres efficacement.

3 Utilisation du prédicat spécial `is`

Question 3.1. Écrire le prédicat permettant de calculer la factorielle d'un entier en utilisant le prédicat `is` :

```
%factorial(+, -): int * int
%factorial(N, Fact) avec Fact = N!
```

Vous pourrez comparer cette implémentation avec celle utilisant des nombres binaires sous forme de liste et remarquer qu'on a encore gagné en efficacité (le `is` est implémenté en utilisant les opérations arithmétiques de la machine).

TP 6

Bases de données déductives

Une base de données déductive (BDD) est une base de données capable d'effectuer des déductions construites sur des règles et des faits. Les données de la base sont stockées sous forme de faits. Une BDD peut être vue comme une combinaison de la programmation logique et des bases de données relationnelles. Pour schématiser, il s'agit de remplacer un langage de manipulation de données tel que SQL par Prolog. En réalité, des langages spécialisés, comme DATALOG, dérivés de Prolog, ont été développés pour une mise en œuvre permettant de gérer de très grandes bases. Pour approfondir le sujet, vous pouvez consulter par exemple le chapitre XV de [1] [1] sur les BDD et DATALOG.

Un des intérêts des bases de données déductives est de pouvoir profiter de toute la puissance de Prolog pour effectuer des requêtes complexes impossibles à exprimer sous forme d'opérations relationnelles [2] en SQL, comme par exemple des requêtes récursives.

Nous allons, dans ce TP, étudier et simuler une base de données déductive en Prolog, en commençant par définir des opérations relationnelles classiques, puis en définissant des requêtes hors du cadre de l'algèbre relationnelle.

1 Représentation de la base de données

Comme dans le TP1, nous allons représenter une base de données en Prolog par une base de faits. Chaque table est représentée par un prédicat et chaque enregistrement est représenté par un fait. Pour vous faire gagner du temps, vous pourrez copier dans votre répertoire de travail le fichier `baseauto.pl` que vous trouverez sous Moodle. Ce fichier contient la base de données suivante pour gérer des pièces et fournisseurs pour un constructeur automobile :

1. [1] Georges Gardarin : "Bases de données : les systèmes et leurs langages", Eyrolles 1999. Disponible sur http ://georges.gardarin.free.fr
2. On appelle ici *opérations relationnelles* les requêtes exprimables à l'aide d'opérateurs ensemblistes ou de l'algèbre relationnelle, voire, par extension, de fonctions implémentées dans les langages fondés sur l'algèbre relationnelle tels SQL.

Table d'assemblage : définit l'assemblage de pièces et composants pour former un composant.

Assemblage

Composant	Composé de	Quantité
voiture	porte	4
voiture	roue	4
voiture	moteur	1
roue	jante	1
porte	tôle	1
porte	vitre	1
roue	pneu	1
moteur	piston	4
moteur	soupape	16

Table des pièces : représente les pièces de base et leur lieu de fabrication.

Pièce

NumPiece	Nom	Lieu fabrication
p1	tôle	lyon
p2	jante	lyon
p3	jante	marseille
p4	pneu	clermont-Ferrand
p5	piston	toulouse
p6	soupape	lille
p7	vitre	nancy
p8	tôle	marseille
p9	vitre	marseille

Table des demandes fournisseurs : représente les fournisseurs ayant demandé à être référencés par le constructeur automobile, et la ville de leur siège social.

Demande Fournisseur

Nom	Ville
dupont	lyon
michel	clermont-Ferrand
durand	lille
dupond	lille
martin	rennes
smith	paris
brown	marseille

Table des fournisseurs référencés : représente les fournisseurs référencés par le constructeur automobile, et la ville de leur siège social.

Fournisseur Référencé

NumFournisseur	Nom	Ville
f1	dupont	lyon
f2	durand	lille
f3	martin	rennes
f4	michel	clermont-Ferrand
f5	smith	paris
f6	brown	marseille

Table des livraisons : représente les quantités de pièces livrées par les fournisseurs référencés.

Livraison

NumFournisseur	Pièce	Quantité
f1	p1	300
f2	p2	200
f3	p3	200
f4	p4	400
f6	p5	500
f6	p6	1000
f6	p7	300
f1	p2	300
f4	p2	300
f4	p1	300

2 Opérations relationnelles

Sur cette base de faits, construisez des exemples d'opérations relationnelles :

Question 2.1. Sélection : quelles sont les pièces fabriquées à Lyon ?

Question 2.2. Projection : quels sont les noms des pièces et leur lieu de fabrication ?

Question 2.3. Union, **intersection** et **différence ensembliste** : construisez chacune de ces opérations entre la table des `Demande Fournisseur` et la projection sur `Nom` et `Ville` de la table `Fournisseur Référencé`. Dans cette question, n'utilisez pas le prédicat prédéfini *not/1* mais programmez un prédicat spécifique à chaque opération concernée.

Question 2.4. Produit cartésien entre fournisseurs référencés et livraisons.

Question 2.5. Jointure :
- construisez la jointure entre les fournisseurs référencés et les livraisons ;
- construisez la jointure entre les fournisseurs référencés et les livraisons de pièces à plus de 350 exemplaires.

Question 2.6. Division : construisez la requête permettant de connaître les fournisseurs qui (chacun) fournissent *toutes les* pièces fabriquées à Lyon.

Question 2.7. Construisez une requête pour calculer le total de pièces livrées par fournisseur. Pour résoudre cette question, vous pourrez consulter la documentation du prédicat *findall/3*, prédicat à utiliser toutefois avec parcimonie.

3 Au-delà de l'algèbre relationnelle

Toutes les requêtes précédentes peuvent également s'exprimer en SQL. Un des intérêts des bases de données déductives est de pouvoir exprimer des requêtes récursives, ce qui est impossible en SQL.

Question 3.1. Construisez une requête permettant d'obtenir l'ensemble des composants et pièces nécessaires pour réaliser un composant, une voiture par exemple.

Question 3.2. Construisez une requête pour calculer le nombre de pièces total nécessaire à la construction d'un composant (voiture, moteur...).

TP 7

Mondes possibles

L'objectif de ce TP est de montrer l'intérêt de Prolog, et, en particulier de son non-déterminisme, pour mettre en œuvre l'approche « générer et tester » très fréquemment utilisée en algorithmie. Cette approche, comme son nom l'indique, consiste à générer des candidats solutions potentielles pour le problème visé, puis à tester si ces candidats résolvent en fait bien le problème. Cette méthode est cependant souvent peu efficace et ne peut être appliquée de façon « brute » qu'à des problèmes dont l'espace de recherche n'est pas trop grand. Il est possible de l'améliorer en cherchant à « pousser » la partie testeur à l'intérieur de la partie générateur afin de produire des candidats qui ont de fortes chances d'être des solutions acceptables (voire uniquement des solutions acceptables si le générateur et le testeur sont complètement entrelacés). Ceci peut se faire de plusieurs façons (*cf.* TP suivant par exemple).

Nous allons appliquer ici les bases de l'approche « générer et tester », pour résoudre ce que l'on appelle un puzzle (on parle aussi d'énigme logique), générant chaque état du monde correspondant au problème traité puis testant si l'état généré est une solution. Une énigme logique est un ensemble de faits qui concerne un petit nombre d'objets ayant différents attributs. Un nombre suffisant de faits est fourni pour trouver (au moins) une solution à l'énigme posée.

1 Relations d'amitié dans une confrérie

Une petite confrérie possède 4 membres : Abby, Bess, Cody et Dana. Certaines de ces personnes s'aiment et d'autres non. On voudrait savoir qui aime qui pour ne pas commettre d'impair et, après une petite enquête, on réussit à récolter les informations suivantes :

1. Dana aime Cody.
2. Bess n'aime pas Dana.
3. Cody n'aime pas Abby.
4. Personne n'aime quelqu'un qui ne l'aime pas.
5. Abby aime tous ceux qui aiment Bess.
6. Dana aime tous ceux que Bess aime.
7. Tout le monde aime quelqu'un.

Plusieurs relations d'amitié peuvent rendre ces informations vraies. Un *monde candidat* est un ensemble de relations d'amitié entre les membres de la confrérie (amitié effective ou pas) alors qu'un *monde possible* est un ensemble de relations d'amitié compatible avec les propositions précédentes.

Connaissant Prolog et sa puissance, on voudrait utiliser ce langage pour trouver qui aime qui dans cette confrérie. Pour ce faire, on va énumérer chaque monde candidat et tester si les

7 relations listées ci-dessus y sont vérifiées, c'est-à-dire si le monde est possible. Notons qu'il existe peut-être plusieurs mondes possibles.

Question 1.1. On va représenter la relation d'amitié par le terme `likes(_, _)`. Par exemple, le fait que Dana aime Cody sera représenté par `likes(dana, cody)`. Vous allez écrire un prédicat :

```
make_all_pairs(+, -): 'a list * likes('a, 'a) list
```

qui va donner, pour une liste d'éléments, la liste de toutes les relations d'amitié imaginables pour ces éléments. Par exemple :

```
?- make_all_pairs([1, 2], Res).
Res = [likes(1, 2), likes(2, 1), likes(1, 1), likes(2, 2)]
Yes
```

La liste n'est pas forcément dans cet ordre.

Question 1.2. Pour représenter tous les mondes candidats, écrivez le prédicat :

```
sub_list(+, -): 'a list * 'a list
```

qui va réussir autant de fois qu'il y a de sous-listes générables à partir du premier paramètre (ne pas utiliser le prédicat `subset`/2 de la bibliothèque *lists*). Par exemple :

```
?- sub_list([1, 2], Res).
Res = [1, 2]
Yes
Res = [1]
Yes
Res = [2]
Yes
Res = []
Yes
```

Les réponses ne sont pas forcément dans cet ordre.

Question 1.3. Écrire les 7 prédicats représentant les 7 propositions. Ces prédicats auront la forme suivante :

```
proposition1(+): likes('a, 'a) list
```

qui réussit si le monde donné en argument vérifie la proposition 1. Par exemple :

```
?- proposition1([likes(cody, dana), likes(dana, dana)]).
No
```

Remarquons que si `likes(X, Y)` n'apparaît pas dans le monde, ça veut dire que `X` n'aime pas `Y`.

Question 1.4. Écrire le prédicat :

```
possible_worlds(-): likes('a, 'a) list
```

qui réussit autant de fois qu'il y a de mondes possibles.

Question 1.5. Il se peut que vous obteniez plusieurs fois le même monde avec votre programme. Si c'est le cas, essayez d'ajouter une ou plusieurs coupures pour ne pas obtenir de doublons.

On voudrait analyser ce qui est exécuté et combien de fois lors de l'exécution du prédicat `possible_worlds`. Pour ce faire, on va utiliser le prédicat suivant (voir squelette) :

```
test_possible_worlds :-
  possible_worlds(World),
  writeln(World),
  fail.
```

Question 1.6. Utiliser la bibliothèque *coverage* (voir le manuel) pour voir quels prédicats sont utilisés par l'exécution de `test_possible_worlds` et combien de fois. Tester ensuite avec la liste des membres : `[abby, bess, cody, dana, peter]`. Proposer une borne inférieure (la plus haute possible) pour la complexité du prédicat `test_possible_worlds`, en fonction de la taille de la liste.

Question 1.7. Changer l'ordre des littéraux dans `possible_worlds` et tester de nouveau successivement avec la liste de 4 et la liste de 5 membres. Est-ce que ça change :

— l'ensemble des solutions ?
— les résultats de *coverage* ?
— la complexité (voir question précédente) ?

TP 8

Dominos

L'objectif de ce TP est à nouveau d'utiliser Prolog pour résoudre un puzzle ; cependant, contrairement au TP précédent, nous n'allons pas utiliser l'approche « générer et tester » brute, mais, grâce à une bonne structure de données, nous allons élaguer l'espace de recherche au fur et à mesure de la construction de la solution.

1 Énoncé

Le jeu de *dominos solitaire* se joue avec des pièces (les dominos) qui comportent deux nombres. Ces nombres sont représentés par des configurations de points comme illustré en figure 8.1.

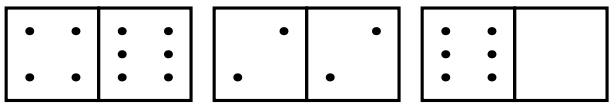

FIGURE 8.1 – Exemples de dominos

Dans une partie de *dominos solitaire*, le joueur dispose d'un ensemble de pièces qu'il doit placer en totalité, les unes au contact des autres, en respectant les contraintes suivantes :

- deux pièces se touchant doivent avoir le même nombre sur leurs côtés en contact ;
- on peut relier une pièce comportant deux nombres différents à deux autres au maximum (via chacune des deux extrémités de la pièce) ;
- on peut relier une pièce comportant deux nombres identiques à trois autres au maximum (via les deux extrémités de la pièce et la partie centrale).

Un domino sera représenté par le terme : `stone(X, Y)` où `X` et `Y` appartiennent à $\{0, \ldots, 6\}$. L'ensemble des pièces sera représenté par une liste, par exemple :

```
[stone(2, 2), stone(4, 6), stone(1, 2), stone(2, 4), stone(6, 2)]
```

Une solution possible pour cet exemple est donnée en figure 8.2.

Question 1.1. Nous allons tout d'abord écrire le prédicat suivant :

```
choose(+, -, -): 'a list * 'a * 'a list
```

Ce prédicat permet de choisir un élément dans une liste et de retourner la liste privée de celui-ci. Il doit réussir exactement autant de fois qu'il y a d'éléments dans la liste. Par exemple :

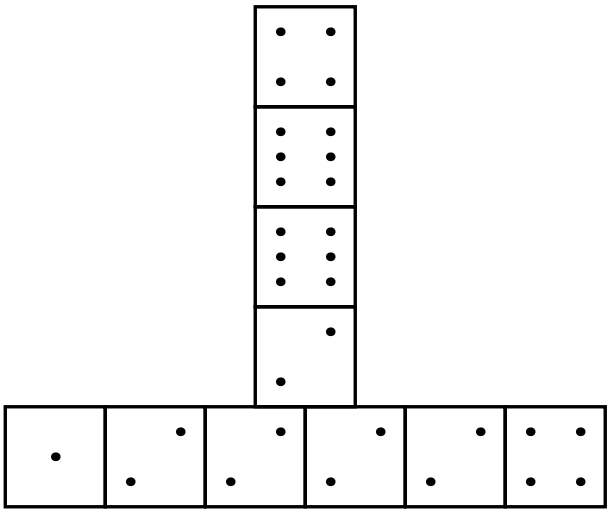

FIGURE 8.2 – Une solution possible pour les 5 pièces de l'exemple

```
?- choose([1, 2, 3], Elt, Rest).
Elt = 1
Rest = [2, 3]
Yes
Elt = 2
Rest = [1, 3]
Yes
Elt = 3
Rest = [1, 2]
Yes
```

Question 1.2. On va définir une structure de données représentant le problème assez efficacement. Cette structure de données va permettre d'élaguer au fur et à mesure l'espace de recherche.

On remarque dans la figure 8.2 qu'il peut exister plusieurs chaînes dans une solution : ici il y en a deux, celle horizontale et celle verticale. On va représenter une chaîne par un terme :

`chain(L1, L2)`

où `L1` et `L2` sont des listes, avec en tête de chacune d'elle la face libre d'un domino. Par exemple, si on a uniquement le domino `stone(2,4)`, on va créer le terme :

`chain([2], [4])`

On a ainsi dans chaque tête de liste la face libre.

Si on ajoute le domino `stone(2, 1)` à la première extrémité, on obtient le terme :

`chain([1, 2], [4])`

Remarquons qu'on ajoute un seul nombre à la liste pour un nouveau domino.
On aurait pu ajouter les 2 nombres du nouveau domino pour représenter cette chaîne de façon plus naturelle par : `chain([1, 2, 2], [4])` mais cette représentation contient des informations redondantes inutiles car des dominos qui se touchent ont forcément les mêmes nombres

sur leurs faces en contact.

Et si on ajoute le domino `stone(4, 5)` à la deuxième extrémité, on obtient la chaîne :

```
chain([1, 2], [5, 4])
```

Si on ajoute ensuite un double, il faut aussi amorcer une chaîne supplémentaire (car un double ajoute une nouvelle direction de progression depuis sa partie centrale). Par exemple, si on ajoute le domino `stone(5, 5)` à la deuxième extrémité, on obtiendra deux chaînes :

```
chain([1, 2], [5, 5, 4])
```

et

```
chain([5], [double])
```

`double` est utilisé pour limiter à un seul sens la nouvelle chaîne, et simplifie l'implémentation au lieu de la liste vide. Notons que la seconde liste de cette dernière chaîne n'évoluera pas.

L'ensemble des chaînes sera représenté par une liste de termes `chain`. Ainsi les deux chaînes précédentes seront représentées par la liste :

```
[chain([1, 2], [5, 5, 4]), chain([5], [double])]
```

Vous avez à votre disposition (voir le squelette sur Moodle) le prédicat :

```
%print_chains(+): chain list
```

qui affiche une liste de chaînes de manière plus conviviale que la représentation compacte.

Faire quelques essais manuels d'appels de ce prédicat pour voir l'affichage produit.

Question 1.3. Définir un prédicat `chains` qui permet de produire les chaînes finales d'une solution à partir de la liste des dominos encore à jouer et de la situation actuelle du jeu :

```
%chains(+, +, -): stone list * chain list * chain list
chains(Stones, PartialAcc, Chains)
```

L'argument `Stones` est la liste des dominos à jouer.
L'argument `PartialAcc` est un accumulateur qui contient le résultat partiel (les chaînes actuelles du jeu).

Ce prédicat choisira un domino, puis jouera ce domino dans un prédicat auxiliaire qui tentera de le placer à toute extrémité compatible puis retournera la valeur actualisée de l'accumulateur afin que `chains` poursuive récursivement.

Question 1.4. Écrire finalement le prédicat qui résout le problème :

```
%domino(+, -): stone list * chain list
%domino(Stones, Chains)
```

Ce prédicat n'aura qu'à lancer la recherche, en plaçant le 1erdomino de la liste fournie, puis en passant le relais au prédicat récursif `chains` et afficher à l'écran chaque solution fournie par ce dernier.

fin de fichier.