Année universitaire 2014–2015

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

4e année
Département informatique
INSA de Rennes

v14.0901

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 en ECL^{i}PS^e[1].

1.1 Démarrer ECL^{i}PS^e

Sur les machines du département informatique, ECL^{i}PS^e se lance à l'aide de la commande `eclipsep`. 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 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 Comptes rendus

Les travaux pratiques se feront par binômes et chaque TP devra faire l'objet d'un compte rendu dont trois seront notés par binôme. **Ces comptes rendus devront être fournis au début du TP suivant sous la (double) forme suivante : une copie papier sera remise à l'enseignant et un fichier Prolog compilable déposé sur Moodle.**

Les comptes rendus incluront au minimum :

1. le code de votre solution. Remarque : la notation tiendra compte de la qualité du code et des commentaires ;
2. **les tests** vous ayant permis de valider votre code. Chaque test sera décrit par :
   - les données utilisées (si ce ne sont pas celles du texte du TP),
   - le but exécuté,
   - les premières réponses d'ECL^{i}PS^e,
   - le temps d'exécution de la requête,
   - une indication du nombre de réponses.

   Ces tests devront être inclus dans un grand commentaire en fin de code source, le fichier déposé sur Moodle devant pouvoir être exécuté directement par le correcteur.

2.2 Structure de vos programmes

Le code de vos TP devra respecter la structure suivante :
- prédicat principal chargé de résoudre le problème posé (quand cela est pertinent),
- prédicats auxiliaires.

2.3 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.

2.4 Documentation

La documentation d'ECL^{i}PS^e est disponible en local à l'adresse : `/usr/local/stow/eclipse6.0_136/doc/bips`. Utilisez en priorité cette documentation. En cas d'absence d'une information, vous pouvez aussi consulter la documentation 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.

2.5 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 empêche de remarquer les cas où le compilateur remonte un problème 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).

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 de connaissances : la base Menu, déjà utilisée en cours, qui vous permettra une prise en main, puis la base Valois dont les questions feront l'objet de votre compte rendu.

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 lister le contenu des relations.

Question 1.2. Écrire les règles correspondant aux définitions suivantes :
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 poser des questions qui permettent d'illustrer l'ordre d'évaluation des prédicats et d'instanciation des variables.

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).
```

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 :

- enfant(E,P) : *E est un enfant de P* ;
- parent(P,E) : *P est un parent direct de E* ;
- grand_pere(G,E) : *G est un grand-père de E* ;
- frere(F,E) : *F est un frère de E* ;
- oncle(O,N) : *O est un oncle de N* ;
- cousin(C,E) : *C est un cousin de E* ;
- 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).
```

3 Compte rendu

Nous vous demandons de rendre uniquement votre code concernant la base Valois ainsi que les tests de chacun des prédicats (*cf.* modalités page 8).

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*.

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 $C2$.

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

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 argument**, indiquant si l'élément passé en paramètre est une main triée.

Dans la suite, nous allons écrire des prédicats évaluant une main triée. On fera donc l'hypothèse que la main fournie en paramètre de ces prédicats sera triée ; inutile de le contrôler.

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).

3 Compte rendu

Nous vous demandons de rendre votre code ainsi que les tests de chacun des prédicats (tests à la fin de votre fichier).

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.

1 Quelques classiques sur les listes

Question 1.1. Programmez et testez chacun des prédicats suivants, en utilisant le traceur. Il sera intéressant de tester et vérifier les modes effectivement supportés.

- *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 ».
- *nieme(+N,+X,-A)* : A est l'élément de rang N dans la liste X.
  Si nécessaire, on proposera une seconde version permettant le mode (-,+,+).
  Dans ce cas, pouvait-on écrire un seul algorithme combinant les possibilités des deux versions ?
- *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 des listes X, Y et Z.
  conc3 sait-il découper la liste T de toutes les façons possibles ? En d'autres termes, peut-on obtenir un seul algorithme qui permet aussi le mode (-,-,-,+) ? Si non, écrivez la variante.
- *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.
- *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 Modélisation des ensembles

Question 2.1. Dans les prédicats suivants, les ensembles seront représentés par des listes Prolog. Nous admettrons donc qu'il n'y a alors jamais deux éléments identiques dans une liste. Remarque : **n'utiliser que *membre* et *hors_de* dans les 3 définitions suivantes à écrire** (et bien sûr la récursivité !)

– *inclus(+X,+Y)* : tous les éléments de l'ensemble X sont présents dans l'ensemble Y.

– *non_inclus(+X,+Y)* : au moins un élément de l'ensemble X est hors de l'ensemble Y.

– *union_ens(+X,+Y, ?Z)* : Z est l'union ensembliste des ensembles X et Y.

Question 2.2. Votre prédicat *inclus/2* de la question précédente fonctionne-t-il ou pas en mode $(?,?)$? Si tel n'est pas le cas, proposez une nouvelle version acceptant ce mode.

3 Compte rendu

Nous vous demandons de rendre votre code, ainsi que les tests de chacun des prédicats en fin de fichier.

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).

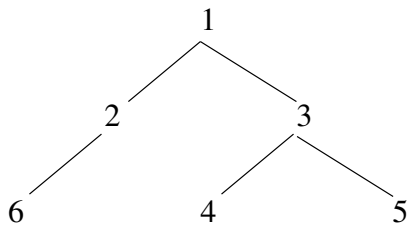


Fig. 4.1 – Arbre binaire d'entiers

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.

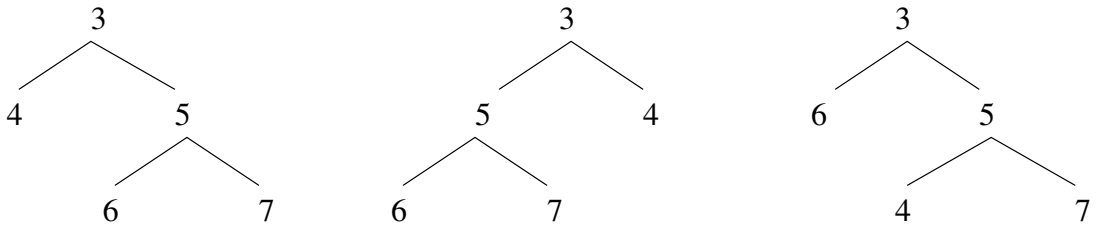


FIG. 4.2 – Deux premiers arbres isomorphes, et un 3^e non isomorphe aux 2 autres

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.

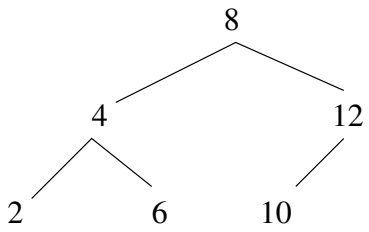


FIG. 4.3 – Arbre binaire d'entiers ordonné

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*.

Question 2.8. La question précédente nécessite la manipulation d'un arbre en entrée mais également d'un arbre en sortie ; en effet, pour faire l'insertion, il est nécessaire de faire une copie de l'arbre initial. Il est cependant possible en Prolog de faire une mise à jour « en place » de certaines structures de données. Pour ce faire, on modifie la représentation actuelle des arbres binaires en remplaçant les arbres vides par des variables logiques. Ainsi, la constante symbolique *vide* sera remplacée par une variable non instanciée (on rappelle qu'on peut tester qu'une variable est non instanciée grâce au prédicat *free/1*).

Écrire le prédicat *insertion_arbre_ordonne1(+X, +B)* qui, utilisant cette nouvelle représentation, insère la valeur X dans l'arbre ordonné d'entiers B.

3 Compte rendu

Nous vous demandons de rendre votre code, ainsi que les tests de chacun des prédicats en fin de fichier. N'oubliez pas la remarque générale au début de la section 2.

TP 5

Arithmétique

Dans ce TP, nous allons évaluer des expressions arithmétiques. Nous allons tout d'abord 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.

Pour ceux qui ont envie d'aller plus loin, dans la seconde partie de ce TP (qui est optionnelle (et plus difficile)), nous manipulerons des termes représentant des expressions arithmétiques. Ces termes pourront notamment représenter des fonctions arithmétiques récursives. L'évaluation des termes représentant les fonctions arithmétiques nécessitera un renommage de variables et ceci nous permettra d'étudier la différence entre les opérateurs `=` (unification) et `==` (égalité stricte).

1 Construction des nombres à partir de principes premiers

1.1 Arithmétique de *Peano*

En cours de logique et 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(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* :

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

```
%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).

1.2 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 1.5. 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 1.6. 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 1.7. 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 1.8. 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.

1.3 Utilisation du prédicat spécial `is`

Question 1.9. 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).

2 Évaluation d'expressions arithmétiques (partie optionnelle)

Nous allons maintenant représenter des expressions arithmétiques par des termes que l'on évaluera grâce à un prédicat. Le terme représentant l'expression arithmétique pourrait être l'arbre syntaxique abstrait obtenu après analyse lexicale et syntaxique d'un fichier, comme vu en compilation. Notre but est d'évaluer cette expression obtenue. Les expressions que nous allons manipuler contiendront les expressions classiques :

- des entiers ;
- des additions : `add(Expr1, Expr2)` où `Expr1` et `Expr2` sont des expressions ;
- des soustractions : `sub(Expr1, Expr2)` où `Expr1` et `Expr2` sont des expressions ;

– des multiplications : `prod(Expr1, Expr2)` où `Expr1` et `Expr2` sont des expressions ;

mais aussi des fonctions, des tests et des conditionnelles (qui sont eux aussi des expressions) :

– un test d'égalité : `eq(Expr1, Expr2)` où `Expr1` et `Expr2` sont des expressions représentant des entiers. Le test d'égalité rend le terme `t` (pour vrai) si la comparaison est vraie et rend le terme `f` (pour faux) sinon ;
– des conditionnelles : `if(Cond, Then, Else)` qui rend l'expression `Then` si la condition `Cond` est vraie et rend l'expression `Else` si elle est fausse ;
– des fonctions : `fun(X, Expr)` où `X` est une variable représentant le paramètre de la fonction et `Expr` est une expression représentant le corps de la fonction ;
– l'application d'une fonction à une expression : `apply(Expr1, Expr2)` où `Expr1` est une expression représentant une fonction et `Expr2` est une expression qui sera appliquée à la fonction.

Notre objectif est de réaliser le prédicat suivant :

```
%evaluate(+, -): expr * {int, fun, t, f}
```

qui permet d'évaluer une expression quelconque et rend un entier ou une fonction ou `t` ou `f`. Voici des exemples :

```
?- evaluate(prod(2, add(3, 1)), Res).
Res = 8
Yes

?- evaluate(eq(prod(2, add(3, 1)), sub(9, 1)), Res).
Res = t
Yes

?- evaluate(fun(X, add(X, 1)), Res).
X = X
Res = fun(X, add(X, 1))
Yes

?- evaluate(apply(fun(X, add(X, 1)), 41), Res).
Res = 42
Yes
```

On vous fournit une très grande partie du prédicat `evaluate` :

```
evaluate(N, N) :- number(N).

evaluate(add(N1, M1), N) :-
        evaluate_numbers(N1, M1, N2, M2),
        N is N2 + M2.

evaluate(sub(N1, M1), N) :-
        evaluate_numbers(N1, M1, N2, M2),
        N is N2 - M2.
```

```
evaluate(prod(N1, M1), N) :-
        evaluate_numbers(N1, M1, N2, M2),
        N is N2 * M2.

evaluate(eq(N1, M1), Res) :-
        evaluate_numbers(N1, M1, N2, M2),
        (
            N2 = M2, Res = t
         ;
            N2 \= M2, Res = f
        ).

evaluate(fun(X, Body), fun(X, Body)).

%evaluate_numbers(+, +, -, -): expr * expr * expr * expr
evaluate_numbers(N1, M1, N2, M2) :-
        evaluate(N1, N2),
        evaluate(M1, M2),
        number(N2),
        number(M2).
```

On voit, par exemple, que pour évaluer l'expression `add(Expr1, Expr2)`, on évalue tout d'abord les sous-expressions `Expr1` et `Expr2` (grâce au prédicat `evaluate_numbers`) puis on effectue l'addition.

Question 2.1. Écrire la clause du prédicat `evaluate` permettant d'évaluer une conditionnelle.

On voudrait maintenant écrire la clause traitant de l'application d'une fonction à une expression. Une première tentative consiste à écrire ceci :

```
evaluate(apply(Expr, Param), Res) :-
        evaluate(Expr, fun(X, Body)),
        X = Param,
        evaluate(Body, Res).
```

Cependant, si on exécute la requête suivante :

```
?- F = fun(X, prod(X, X)), evaluate(apply(F, 1), Res1), evaluate(apply(F, 2), Res2).
```

elle échoue ! On aurait voulu obtenir `Res1 = 1` et `Res2 = 4`.
Le problème vient du fait que le paramètre `X` de la fonction `fun(X, prod(X, X))`, lors de l'application `evaluate(apply(F, 1), Res1)`, ne devrait être lié à `1` que pendant l'évaluation de cette fonction. Mais ici `X` est égal à `1` pour tout le reste de la requête et donc l'évaluation `evaluate(apply(F, 2), Res2)` échoue.
On va essayer de résoudre ce problème en créant le prédicat suivant permettant de transformer une expression pour lui associer de nouvelles variables :

```
%fresh_variables(+, -): expr * expr
%fresh_variables(Expr, FreshExpr) avec FreshExpr l'expression Expr où les
%                                 variables ont été renommées de manière cohérente.
```

Par exemple :

```
fresh_variables(fun(X, fun(Y, add(Y, prod(X, X)))), Fresh).
X = X
Y = Y
Fresh = fun(_203, fun(_211, add(_211, prod(_203, _203))))
Yes

fresh_variables(apply(fun(X, X), fun(X, X)), Fresh).
X = X
Fresh = apply(fun(_194, _194), fun(_203, _203))
Yes
```

On vous donne les prédicats suivants :

```
fresh_variables(Expr, Res) :-
       fresh_variables(Expr, [], Res).

%fresh_variables(+, +, -): expr * (var * var) list * expr
fresh_variables(X, Assoc, Y) :-
        var(X),
        !,
        assoc(X, Assoc, Y).

fresh_variables(add(X1, Y1), Assoc, add(X2, Y2)) :-
        fresh_variables(X1, Assoc, X2),
        fresh_variables(Y1, Assoc, Y2).

fresh_variables(prod(X1, Y1), Assoc, prod(X2, Y2)) :-
        fresh_variables(X1, Assoc, X2),
        fresh_variables(Y1, Assoc, Y2).

fresh_variables(sub(X1, Y1), Assoc, sub(X2, Y2)) :-
        fresh_variables(X1, Assoc, X2),
        fresh_variables(Y1, Assoc, Y2).

fresh_variables(eq(X1, Y1), Assoc, eq(X2, Y2)) :-
        fresh_variables(X1, Assoc, X2),
        fresh_variables(Y1, Assoc, Y2).

fresh_variables(if(Cond1, X1, Y1), Assoc, if(Cond2, X2, Y2)) :-
        fresh_variables(Cond1, Assoc, Cond2),
        fresh_variables(X1, Assoc, X2),
        fresh_variables(Y1, Assoc, Y2).

fresh_variables(Number, _, Number) :- number(Number).

fresh_variables(fun(X, Body1), Assoc, fun(Y, Body2)) :-
        fresh_variables(Body1, [(X, Y) | Assoc], Body2).

fresh_variables(apply(Fun1, Param1), Assoc, apply(Fun2, Param2)) :-
        fresh_variables(Fun1, Assoc, Fun2),
        fresh_variables(Param1, Assoc, Param2).
```

On voit que l'essentiel du travail se fait dans le prédicat `fresh_variables(..., Assoc, ...)` qui possède une liste d'association entre variables. En effet, c'est cette liste qui va permettre de faire un renommage cohérent, comme on peut le voir dans la clause s'occupant du renommage d'une fonction. Il reste le prédicat `assoc` à compléter pour que le prédicat `fresh_variables` soit complet.

Question 2.2. Écrire le prédicat suivant :

```
%assoc(+, +, -): var * (var * var) list * var
%assoc(X, Assoc, Res) avec Res la variable correspondant à X dans Assoc
%                     ou une nouvelle variable libre si X n'est pas présente.
```

Vous ne pourrez pas utiliser l'unification pour comparer la variable `X` à une variable dans `Assoc` car cela unifierait les variables. Par exemple, si le premier élément de la liste `Assoc` est `(Z, T)` et que l'on souhaite tester si `X` est égale à `Z`, cela réussira toujours et liera les variables `X` et `Z`. Vous allez donc devoir utiliser le prédicat `==` qui permet d'indiquer si deux termes sont égaux même si les termes comparés sont des variables. Le prédicat `==` teste si les termes sont identiques (et pas simplement unifiables). Par exemple :

```
?- X == Y.
No

?- 1 == X.
No

?- X == X.
Yes
```

Vous disposez aussi du prédicat `\==` qui teste si deux termes sont différents même si les termes comparés sont des variables.

Question 2.3. Vous pouvez maintenant finaliser le prédicat `evaluate` en complétant la clause :

```
evaluate(apply(Expr, Param), Res) :-
   ...
```

Vous pourrez tester `evaluate` grâce à la requête suivante qui calcule la factorielle de 42 :

```
?- Fun = fun(N, fun(F, if(eq(N, 0), 1, prod(N, apply(apply(F, sub(N, 1)), F))))),
   Factorial = fun(N, apply(apply(Fun, N), Fun)),
   evaluate(apply(Factorial, 42), Res).
Fun = fun(N, fun(F, if(eq(N, 0), 1, prod(N, apply(apply(F, sub(N, 1)), F)))))
N = N
F = F
Factorial = fun(N, apply(apply(fun(N, fun(F, if(eq(N, 0), 1,
prod(N, apply(apply(F, sub(N, 1)), F))))), N),
                        fun(N, fun(F, if(eq(N, 0), 1, prod(N,
                        apply(apply(F, sub(N, 1)), F)))))))
Res = 1405006117752879898543142606244511569936384000000000
Yes
```

3 Compte rendu

Nous vous demandons de rendre votre code de la partie obligatoire, suivi des tests de chacun de ses prédicats. Ceux qui ont fait la partie optionnelle peuvent également la rendre : le code et les tests de cette partie seront mis après l'intégralité des éléments concernant la partie obligatoire.

TP 6

Machine de Turing

On pourrait se demander si Prolog permet de faire autant de choses qu'un langage tel que Java™. Par faire « autant de choses », on entend calculer tout ce qui peut se calculer sur un ordinateur en écrivant les programmes en Java™. En fait, il s'avère que tout ce qu'on peut calculer sur un ordinateur peut l'être par une machine de *Turing* (et vice-versa). Par conséquent, si on arrive à simuler une machine de *Turing* en Prolog, on aura montré que Prolog permet lui aussi de calculer tout ce qu'il est possible de calculer sur un ordinateur.

Dans la première partie de ce TP, nous allons construire un simulateur de machine de *Turing* en Prolog, ce qui montrera que Prolog est *Turing*-complet. On créera aussi une trace d'exécution de la machine de *Turing* pour pouvoir visualiser celle-ci.

Écrire un programme dans le formalisme des machines de *Turing* n'est pas chose aisée. Dans la seconde partie, optionnelle, de ce TP, nous automatiserons la création de programmes (simples) pour les machines de *Turing*. On s'intéressera à calculer deux valeurs d'une fonction non calculable : la fonction du « castor affairé ».

1 Simulation d'une machine de *Turing* en Prolog

Une machine de *Turing* comporte :
- une bande contenant des cases (théoriquement la bande contient une infinité dénombrable de cases) ;
- une tête de lecture positionnée sur une des cases ;
- un programme.

Nous allons représenter la bande par le terme :

```
tape(Left, Right)
```

où `Left` et `Right` sont deux listes. Le symbole sous la tête de lecture est le premier élément de `Right`, ce qui est à gauche de la tête de lecture est dans `Left` et ce qui est à droite est dans le reste de `Right`. On s'assurera que les listes `Left` et `Right` ne soient jamais vides et contiennent au moins un symbole *blanc* (représenté par le terme `' '`). En fait, on suppose que `Left` se prolonge indéfiniment vers la gauche et que les cases non représentées contiennent le symbole blanc. De même, on suppose que `Right` se prolonge indéfiniment vers la droite et que les cases non représentées contiennent le symbole blanc.

Par exemple, la bande suivante, où la tête de lecture est sur la case mise en évidence :

...		1	1	1			1	1		...

sera représentée par le terme :

```
tape([1, 1, 1], [' ', 1, 1])
```

Un programme est représenté par un terme similaire au terme de la figure 6.1 où :
- le premier élément du terme représente l'état de départ de la machine ;
- le deuxième élément représente la liste des états terminaux de la machine ;
- le troisième élément est la fonction de transition.

```
program(
           start,
           [stop],
           [delta(start, ' ', ' ', right, stop),
            delta(start, 1, ' ', right, s2),
            delta(s2, 1, 1, right, s2),
            delta(s2, ' ', ' ', right, s3),
            delta(s3, 1, 1, right, s3),
            delta(s3, ' ', 1, left, s4),
            delta(s4, 1, 1, left, s4),
            delta(s4, ' ', ' ', left, s5),
            delta(s5, 1, 1, left, s5),
            delta(s5, ' ', 1, right, start)]
      )
```

FIG. 6.1 – Représentation d'un programme pour la machine de *Turing*

Nous allons expliquer le fonctionnement de la machine de *Turing* sur un exemple. Au départ, la machine se trouve dans l'état `start` et la bande est : `tape([' '], [1, ' '])`. L'exécution du programme de la figure 6.1 va se faire de la manière suivante, où l'on représente l'état de la machine et la configuration de la bande par :

```
Configuration initiale :
+---------------------------+
| state: start              |
| tape([' '], [1, ' '])     |
+---------------------------+
```

On va donc effectuer une transition, sachant que le symbole sous la tête de lecture est `1` et que l'état de la machine est `start`. La seule transition possible est : `delta(start, 1, ' ', right, s2)`. En effet, la signification des éléments du quintuplet `delta` sont, dans l'ordre :

1. l'état courant : `start` dans notre configuration ;
2. le symbole sous la tête de lecture : `1` ici ;
3. le nouveau symbole par lequel on remplace le symbole actuellement sous la tête de lecture : `' '` ici, qui remplacera donc `1` ;

4. le déplacement de la tête de lecture (gauche (`left`) ou droite (`right`)), effectué après le remplacement : ici on déplace la tête de lecture à droite ;
5. le nouvel état de la machine : ici `s2`.

On obtient donc, après avoir effectué la transition `delta(start, 1, ' ', right, s2)`, la nouvelle configuration :

```
+---------------------------+
| state: s2                 |
| tape([' ', ' '], [' '])   |
+---------------------------+
```

La suite des transitions est la suivante :

```
+-----------------------------+
| state: s3                   |
| tape([' ', ' ', ' '], [' ']) |
+-----------------------------+

+-----------------------------+
| state: s4                   |
| tape([' ', ' '], [' ', 1])  |
+-----------------------------+

+-----------------------------+
| state: s5                   |
| tape([' '], [' ', ' ', 1])  |
+-----------------------------+

+-----------------------------+
| state: start                |
| tape([' ', 1], [' ', 1])    |
+-----------------------------+

+-----------------------------+
| state: stop                 |
| tape([' ', 1, ' '], [1])    |
+-----------------------------+
```

La machine s'arrête dans cette dernière configuration car l'état `stop` appartient à la liste des états finaux. Ce programme effectue simplement une copie des `1` consécutifs sur la bande de départ.

Nous allons maintenant réaliser un simulateur de machine de *Turing* en Prolog.

Les prédicats suivants (qui sont fournis dans le squelette sur Moodle) permettent d'accéder aux différents éléments du programme :

```
%initial_state(+, -): program * state
initial_state(program(InitialState, _, _), InitialState).
```

```
%final_states(+, -): program * state list
final_states(program(_, FinalStates, _), FinalStates).

%transitions(+, -): program * delta list
transitions(program(_, _, Deltas), Deltas).
```

Question 1.1. Écrire le prédicat suivant qui permet d'effectuer une transition :

```
%next(+, +, +, -, -, -):
%program * state * symbol * symbol * direction (left or right) * state
%next(Program, State0, Symbol0, Symbol1, Dir, State1)
```

En entrée de ce prédicat, on a le programme, l'état courant de la machine et le symbole sous la tête de lecture. On récupère le nouveau symbole, la direction de déplacement de la tête de lecture et le nouvel état.

Par exemple, avec `Program` instancié au programme de la figure 6.1, on a :

```
?- Program = ..., next(Program, start, 1, NewSymbol, Dir, NextState).
Program = ...
NewSymbol = ' '
Dir = right
NextState = s2
Yes
```

Question 1.2. Écrire le prédicat :

```
%update_tape(+, +, +, -): tape * symbol * direction * tape
%update_tape(Tape, Symbol, Direction, UpdatedTape)
```

qui met à jour la bande de la machine. `Tape` est la bande de lecture, `Symbol` est le nouveau symbole à placer sous la tête de lecture, `Direction` est la direction de déplacement de la tête et `UpdatedTape` est la nouvelle bande. Par exemple :

```
?- update_tape(tape([' '], [1, ' ']), ' ', right, UpdatedTape)
UpdatedTape = tape([' ', ' '], [' '])
Yes
```

Question 1.3. Écrire le prédicat :

```
%run_turing_machine(+, +, -, -): program * symbol list * symbol list * state
%run_turing_machine(Program, Input, Output, FinalState)
```

qui exécute le programme `Program` sur l'entrée `Input` et rend la sortie `Output` (qui correspondra au contenu de la bande en fin d'exécution) et l'état final `FinalState`. La tête de lecture sera placée sur le début de la liste `Input`. Par exemple, avec `Program` instancié au programme de la figure 6.1, on obtient :

```
?- Program = ..., run_turing_machine(Program, [1], Output, FinalState)
Program = ...
Output = [' ', 1, ' ', 1]
FinalState = stop
Yes
```

On voudrait pouvoir visualiser les différentes étapes de calcul effectuées par la machine de *Turing*. On veut récupérer une liste de couples `(State, Tape)` où `State` représente un état de la machine et `Tape` est le contenu de la bande. Vous pourrez ensuite créer un fichier de type `MetaPost` (un format permettant de décrire des documents) grâce au prédicat suivant qui vous est fourni dans le squelette :

```
%dump_to_mpost(+, +): string * (state * symbol list) list
%dump_to_mpost(Filename, Dump)
```

Ce prédicat permet d'écrire dans le fichier `Filename` une représentation des différentes étapes de l'exécution de la machine contenues dans la liste `Dump`. Une fois le fichier `Filename` généré, il vous faudra taper dans un shell :

```
> mpost Filename
> epstopdf Filename.1
```

Vous obtiendrez alors un fichier *pdf* de nom `Filename` qui vous permettra de visualiser les différentes étapes de l'exécution de votre programme.

Question 1.4. Réécrire le prédicat `run_turing_machine` pour qu'il produise une liste représentant les différentes étapes de l'exécution de la machine :

```
%run_turing_machine(+, +, -, -, -):
%program * symbol list * symbol list * state * (state * symbol list) list
%run_turing_machine(Program, Input, Output, FinalState, Dump)
```

2 Création automatique de programmes (simples) dans le formalisme des machines de *Turing* (partie optionnelle)

Il est assez difficile d'écrire un programme dans le formalisme des machines de *Turing*. On va donc essayer, pour de petits programmes, de les faire écrire par Prolog. On vous donne les prédicats suivants (fournis dans le squelette) :

```
%make_pairs(+, +, -): 'a list * 'a list * ('a * 'a) list
%make_pairs(L1, L2, PairList)

%complete_list(+, +, +, +, -):
%(state * symbol) list * symbol list * direction list * state list * delta list
%complete_list(StateSymbolList, SymbolList, DirectionList, StateList, DeltaList)
```

Le prédicat `make_pairs` permet de créer une liste contenant le produit cartésien des listes `L1` et `L2`. Par exemple :

```
?- make_pairs([s1, s2, s3], [' ', 1], Res).
Res = [(s1, ' '), (s1, 1), (s2, ' '), (s2, 1), (s3, ' '), (s3, 1)]
Yes
```

Le prédicat `complete_list` va réussir autant de fois qu'il est possible de compléter les paires contenues dans la liste `StateSymbolList` en utilisant les symboles de `SymbolList`, les directions de `DirectionList` et les états de `StateList` pour obtenir une liste de transitions qui permettra d'écrire un programme. Par exemple :

```
?- complete_list([(s1,' '),(s1,1)], [' ',1], [left,right], [init,s1,stop], Res).
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, init)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, s1)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', left, stop)]
Yes
Res = [delta(s1, ' ', ' ', left, init), delta(s1, 1, ' ', right, init)]
Yes
...
```

Vous allez écrire un prédicat qui réussit autant de fois qu'il y a de programmes possibles. S'il y a deux états non terminaux `init` et `s1` et deux symboles `' '` et `1` par exemple, le programme comportera 4 transitions (celles correspondant à `(init, ' ')`, `(init, 1)`, `(s1, ' ')` et `(s1, 1)`). Donc, s'il y a n états non terminaux et m symboles, le programme comportera $n \times m$ transitions.

Question 2.1. Écrire le prédicat suivant :

```
%all_programs(+, +, +, +, -):
%state * state list * symbol list * state list * program
%all_programs(Init, StateList, SymbolList, FinalStateList, Program)
```

où `Init` est l'état initial de la machine, `StateList` correspond aux états non terminaux sans l'état `Init`, `SymbolList` est la liste de symboles, et `FinalStateList` est la liste des états terminaux. Par exemple :

```
?- all_programs(init, [s2, s3], [' ', 1], [stop], Program).
Program = program(init, [stop],
[delta(init, ' ', ' ', left, init),
 delta(init, 1, ' ', left, init),
 delta(s2, ' ', ' ', left, init),
 delta(s2, 1, ' ', left, init),
 delta(s3, ' ', ' ', left, init),
 delta(s3, 1, ' ', left, init)])
Yes
...
```

On va maintenant essayer de trouver un programme permettant de calculer la valeur d'une fonction pour une entrée donnée. Cette fonction s'appelle la fonction du *castor affairé*. Vous êtes invités à lire une partie de l'article suivant : http://fr.wikipedia.org/wiki/Castor_affairé. Nous allons essayer de faire découvrir à Prolog un programme permettant de produire la valeur de cette fonction pour $n = 2$ et $n = 3$.

Puisque parmi les programmes que l'on va générer certains vont faire boucler indéfiniment la machine de *Turing*, il va nous falloir modifier le prédicat `run_turing_machine` pour qu'il prenne en paramètre un nombre d'étapes maximum lors de l'exécution de la machine de *Turing* pour le programme donné.

Question 2.2. Écrire le prédicat :

```
%run_turing_machine(+, +, -, -, -, +, +):
%program * symbol list * symbol list * state * (state * symbol list) list * int
%run_turing_machine(Program, Input, Output, FinalState, Dump, NbMaxIter)
```

Question 2.3. Vous allez maintenant pouvoir écrire le prédicat suivant :

```
%find_busy_beaver(+, +, +, +, +, +, -):
%state * state list * state list * symbol list * int * int * program
%find_busy_beaver(Init, States, FinalStates, Symbols, NbMaxIter,
        BusyBeaverNumber, Program)
```

où `Init` est l'état initial de la machine, `States` est l'ensemble des états non terminaux sans l'état `init`, `FinalStates` est l'ensemble des états terminaux, `NbMaxIter` est le nombre maximum de pas d'exécution d'une machine, `BusyBeaverNumber` est la valeur de la fonction du castor affairé et `Program` est un programme correspondant au castor affairé. En faisant l'appel suivant :

```
?- find_busy_beaver(init, [s1], [stop], [' ', 1], 6, 4, Program).
```

vous obtiendrez les programmes correspondant au castor affairé pour $n = 2$. Notez que, normalement, on ne devrait pas connaître le nombre d'itérations maximal et la valeur n à l'avance ; l'adaptation nécessaire pour pouvoir se passer de ces 2 valeurs n'est pas très difficile à réaliser.

3 Compte rendu

Nous vous demandons de rendre votre code de la partie obligatoire, suivi des tests de chacun de ses prédicats. Ceux qui ont fait la partie optionnelle peuvent également la rendre : le code et les tests de cette partie seront mis après l'intégralité des éléments concernant la partie obligatoire.

TP 7

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`.

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 de toutes les pièces fabriquées à Lyon.

Question 2.7. Construisez une requête pour calculer le total de pièces livrées par fournisseur.

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...).

Question 3.3. Construisez une requête pour calculer le nombre de voitures qu'il est possible de construire avec les quantités de pièces livrées ?

4 Compte rendu

Nous vous demandons de rendre uniquement votre code concernant les requêtes (sans la base de données) ainsi que les tests pour chacune d'entre elles (*cf.* modalités page 8).

TP 8

Traitement automatique des langues

Prolog a été conçu initialement pour traiter automatiquement des langues naturelles, c'est-à-dire du Français, de l'Anglais... par opposition à un langage formel. L'objectif de ce TP est de vous donner une très courte introduction à un des aspects du traitement automatique des langues (TAL). Nous nous limiterons en effet à réaliser un analyseur syntaxique de quelques structures grammaticales du Français, en effectuant un nombre limité de contrôles de leur correction. Nous n'aborderons donc par exemple pas ici tout ce qui a trait à la compréhension de textes, qu'elle soit fine ou à plus gros grain, laissant par conséquent de côté des domaines du TAL tels que les systèmes de questions-réponses, la traduction automatique ou le résumé automatique pour ne citer que quelques exemples.

Le but de ce TP est donc de construire en Prolog un analyseur d'un sous-ensemble de la grammaire du Français. Cet analyseur sera mis en œuvre avec la méthode dite **d'analyse de la tête de liste**, seconde méthode vue en cours.

1 Représentation grammaticale

Question 1.1. Écrire une grammaire reconnaissant les quatre phrases suivantes :

le chien aboie.
les enfants jouent.
Paul marche dans la rue.
la femme qui porte un pull noir mange un steak.

Vous devrez utiliser les non-terminaux suivants :
`adjectif`, `article`, `gp_nominal`, `gp_prepositionnel`, `gp_verbal`, `nom_commun`, `nom_propre`, `phrase_simple`, `prep`, `pronom`, `relative`, `verbe`.

Vous écrirez cette grammaire en commentaire dans votre fichier source. Pour vous faire gagner du temps, vous pouvez récupérer sous Moodle le fichier `tal.pl` contenant les non-terminaux ainsi que quelques phrases à tester. Bien entendu, il est nécessaire de tester votre analyseur avec des phrases supplémentaires.

2 Analyseur en Prolog

Question 2.1. Traduire cette grammaire en un analyseur en Prolog grâce à la méthode d'analyse de la tête de liste. Cet analyseur sera appelé par le prédicat `analyse` que vous définirez. Tester les phrases précédentes.

Quelle question faut-il poser pour générer toutes les phrases reconnues par la grammaire ?

Question 2.2. Modifier votre analyseur pour qu'il produise également l'arbre syntaxique de la phrase analysée. Vous utiliserez les constructeurs (*i.e.*, symboles fonctionnels de termes construits) tels qu'ils sont définis dans l'exemple suivant :

```
[eclipse 6]: analyse([paul,qui,porte,un,pull,noir,mange,un,steack],R).

R = phr(gn(nom_prop(paul), rel(pronom(qui), gv(verbe(porte),
        gn(art(un), nom_com(pull), adj(noir))))), gv(verbe(mange),
        gn(art(un), nom_com(steack))))
```

Tous les noms terminaux sont représentés dans l'arbre syntaxique de l'exemple, à l'exception de `gp_prepositionnel` et `prep`, que vous représenterez respectivement par les constructeurs `gp` et `prep`.

Question 2.3. On désire maintenant éviter de reconnaître des phrases contenant des fautes d'accord en genre et en nombre. Modifier la grammaire afin de vérifier les accords en genre et en nombre pertinents.

Question 2.4. On désire en plus éviter de reconnaître des phrases sémantiquement incorrectes, telles que *les enfants jouent dans le chien*, *Paul mange la rue* ou *Paul aboie*. Ajouter les contrôles nécessaires.

Le principe est de définir des attributs hérités ou synthétisés supplémentaires, afin de représenter des traits sémantiques. Par exemple, en simplifiant, le verbe `marcher` ne s'applique qu'à des `humain`, alors que le verbe `jouer` peut concerner un `humain` ou un `animal`.

3 Compte rendu

Vous devez rendre le code ainsi que les tests réalisés (*cf.* modalités page 8). Cependant, contrairement aux TP précédents où vous deviez regrouper les tests à la fin du fichier, vous devez ici mettre les tests à la suite du code correspondant à chaque question. En outre, comme les différentes questions de ce TP correspondent en fait à des améliorations successives du code de la question 2.1, vous devrez recopier le code solution de la question `n` pour commencer la question `n+1`, puis mettrez ce code de la question `n` en commentaire, avec ses tests associés.

TP 9

Binômes

Le but de ce TP est de répondre à une question qui avait été posée par un étudiant d'une promotion précédente à un des enseignants de Prolog. Voici la question :

« Bonjour !
Avec cinq de mes amis, nous nous sommes trouvés confrontés à un problème qui, bien qu'au premier abord semblait simplissime, s'est révélé assez délicat. L'idée est simple : au cours du semestre qui arrive, nous devrons participer à cinq TP différents et nous aurions tous souhaité être en binôme avec tous les membres du groupe.
Après quelques essais non concluants au papier, il nous a semblé que Prolog apparaissait comme un outil de choix pour résoudre ce problème. Pensez-vous qu'il soit possible de le résoudre ? Ce problème pourrait faire l'objet d'un sujet de TP pour l'année prochaine ou pour la programmation par contraintes ?
Bonne soirée. »

Dans la première partie de ce TP, nous commencerons par proposer une solution simple mais coûteuse (suffisante cependant pour la taille du problème considéré) pour cette question. Dans la seconde partie, optionnelle, de ce TP, nous présenterons une approche optimale.

1 Résolution par force brute

Soit N étudiants (avec N un nombre pair). Nous proposons la démarche globale suivante :

1. créer la liste `Binomes` des $\frac{N\times(N-1)}{2}$ binômes possibles ;
2. extraire récursivement de cette liste $(N-1)$ sous-listes de $\frac{N}{2}$ binômes. Chacune de ces sous-listes est telle que chaque personne n'y figure qu'une fois ;
3. la liste de ces sous-listes est la liste des binômes pour chacun des TP.

Question 1.1. Écrire le prédicat :

```
%combiner(+, -): 'a list * ('a * 'a) list
%combiner(Copains, Binomes)
```

qui permet de produire la liste `Binomes` de tous les binômes que l'on peut créer à partir de la liste `Copains`. Par exemple :

```
?- combiner([pluto, riri, fifi, loulou], Binomes).
Binomes = [(pluto, riri), (pluto, fifi), (pluto, loulou),
           (riri, fifi), (riri, loulou), (fifi, loulou)]
Yes
```

Question 1.2. Écrire le prédicat suivant :

```
%extraire(+, +, -, -): ('a * 'a) list * int * ('a * 'a) list * ('a * 'a) list
%extraire(AllPossibleBinomes, NbBinomes, Tp, RemainingBinomes)
```

qui permet d'extraire `NbBinomes` de la liste de tous les binômes possibles `AllPossibleBinomes`. Ces `NbBinomes` sont mis dans la liste `Tp` et le reste des binômes non utilisés de la liste `AllPossibleBinomes` se retrouve dans `RemainingBinomes`. Dans la liste `Tp`, chaque étudiant ne doit apparaître qu'une fois. Par exemple :

```
?- combiner([pluto, riri, fifi, loulou], Binomes),
   extraire(Binomes, 2, Tp, R).
Binomes = [(pluto, riri), (pluto, fifi), (pluto, loulou),
           (riri, fifi), (riri, loulou), (fifi, loulou)]
Tp = [(riri, fifi), (pluto, loulou)]
R = [(pluto, riri), (pluto, fifi), (riri, loulou), (fifi, loulou)]
Yes (0.00s cpu, solution 1, maybe more) ? ;

Binomes = [(pluto, riri), (pluto, fifi), (pluto, loulou),
           (riri, fifi), (riri, loulou), (fifi, loulou)]
Tp = [(riri, loulou), (pluto, fifi)]
R = [(pluto, riri), (pluto, loulou), (riri, fifi), (fifi, loulou)]
Yes (0.00s cpu, solution 2, maybe more) ? ;

Binomes = [(pluto, riri), (pluto, fifi), (pluto, loulou),
           (riri, fifi), (riri, loulou), (fifi, loulou)]
Tp = [(fifi, loulou), (pluto, riri)]
R = [(pluto, fifi), (pluto, loulou), (riri, fifi), (riri, loulou)]
Yes (0.00s cpu, solution 3, maybe more) ? ;

No (0.00s cpu)
```

On pourra, par exemple, se servir d'un prédicat auxiliaire `extraire_aux` qui contient un paramètre de plus que `extraire`. Ce paramètre supplémentaire sera un accumulateur permettant de construire progressivement la liste `Tp`.

Question 1.3. On peut enfin répondre au mail :-). Écrire le prédicat :

```
%les_tps(+, -): 'a list * ('a * 'a) list list
%les_tps(Copains, Tps)
```

qui fournit dans `Tps` tous les binômages possibles pour les étudiants `Copains` pour tous les TP. Par exemple :

```
?- les_tps([pluto, riri, fifi, loulou], Tps).
Tps = [[(riri, fifi), (pluto, loulou)],
       [(riri, loulou), (pluto, fifi)],
       [(fifi, loulou), (pluto, riri)]]
Yes
```

2 Résolution optimale (partie optionnelle)

Le problème de la solution précédente est sa complexité. Pour vous en rendre compte, vous pouvez, par exemple, essayer d'utiliser le prédicat `les_tps/2` avec une liste de 20 étudiants.

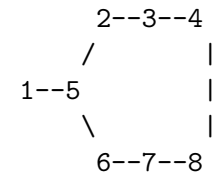
Nous allons décrire et implémenter une solution dont l'ordre de grandeur de complexité est optimal. Ce problème est un problème très connu (*round robin*). Il décrit aussi, par exemple, les appariements d'équipes dans un championnat.

Voici le principe de l'algorithme.

Soit un groupe de huit amis notés : `1, 2, 3, 4, 5, 6, 7, 8` (nous fondons la discussion sur ce cas précis, mais l'argumentation se généralise sans problème).
On coupe la liste en deux et on obtient :

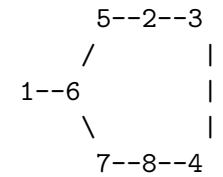
```
1 2 3 4
5 6 7 8
```

On représente cela géométriquement par :

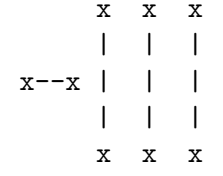


On peut voir que `1` et `5` sont reliés entre eux ainsi que `5` et `2`, `2` et `3`, ... et `6` et `5` (la lecture se fait dans le sens des aiguilles d'une montre). On suppose qu'il y a une distance de une unité entre chaque couple (`(1, 5)`, `(5, 2)`, ..., `(6, 5)`).

Dans cette structure, le `1` va rester fixe et les autres chiffres vont effectuer une rotation d'un cran à chaque fois dans le sens des aiguilles d'une montre. Donc, après une rotation, on obtient :



On mémorise à chaque tour les paires suivantes :



Donc on obtiendrait initialement la liste de paires : `[(1,5), (2,6), (3,7), (4,8)]` et après la première rotation la liste de paires : `[(1,6), (5,7), (2,8), (3,4)]`. On arrête de faire des rotations quand on revient dans la configuration initiale. L'ensemble des listes de paires obtenues (les paires initiales et celles après chaque rotation) représente la solution au problème.

Mais pourquoi cela marche-t-il ?

On ne peut pas obtenir la même paire deux fois si on crée 7 listes de paires par ce principe. En effet, soit :

```
      C  D  E
      |  |  |
A--B  |  |  |
      |  |  |
      H  G  F
```

Il existe une distance de 5 entre les emplacements `C` et `H` (en comptant dans le sens des aiguilles d'une montre), une distance de 3 entre `D` et `G`, une distance de 1 entre `E` et `F`, une distance de 2 entre `H` et `C`, une distance de 4 entre `G` et `D` et une distance de 6 entre `F` et `E`. Par conséquent, une paire qui a été appariée entre `C` et `H` ne pourra pas se faire apparier, après une ou plusieurs rotations, entre `D` et `G`, `E` et `F`, `H` et `C`, `G` et `D` ou `F` et `E` car la distance ne correspond pas (comme on effectue une rotation, la distance entre deux éléments le long de la chaîne ne change pas). On peut appliquer le même raisonnement aux paires `(D, G)` et `(E, F)`. Il faut donc faire 7 rotations pour revenir en configuration initiale et avoir un doublon possible.

Le lien `A--B` ne permet pas d'avoir de doublon avant la septième rotation car l'élément en `A` ne change pas (et l'élément en `B` change à chaque rotation).

Tout le monde a été apparié à tout le reste. En effet, on obtient 7 listes sans doublons, et la seule manière de ne pas avoir de doublons pour un i donné est d'être apparié à tous les $j \neq i$ (il y en a 7).

Quelle est la complexité de cet algorithme ? Si N est le nombre d'amis, il y a $(N-1)$ rotations à effectuer. Pour chaque rotation, on doit afficher $N/2$ paires. La complexité est donc en $\Theta(N^2)$. La complexité est optimale car nous venons de voir ci-dessus que pour N personnes, il y a $(N-1)$ appariements et, pour chaque appariement, il faut lister les $N/2$ binômes.

Question 2.1. En vous fondant sur la description de l'algorithme ci-dessus, écrire une nouvelle version du prédicat `les_tps`/2 précédent (il vous faudra écrire des prédicats auxiliaires) :

```
%les_tps(+, -): 'a list * ('a * 'a) list list
%les_tps(Copains, Tps)
```

Vous pourrez maintenant facilement exécuter ce prédicat avec une liste de 20 étudiants.

3 Compte rendu

Nous vous demandons de rendre votre code de la partie obligatoire, suivi des tests de chacun de ses prédicats. Ceux qui ont fait la partie optionnelle peuvent également la rendre : le code et les tests de cette partie seront mis après l'intégralité des éléments concernant la partie obligatoire.

TP 10

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 l'approche « générer et tester », de façon basique, 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 faire 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 vraies, 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 librairie *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.

Ce problème des mondes possibles est tiré des exercices 1.2 et 1.3 du cours d'introduction à la logique sur `www.coursera.org`. En vous inscrivant à ce cours, vous pourrez tester la validité de vos réponses.

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 librairie *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) ?

2 Compte rendu

Nous vous demandons de rendre les réponses aux questions, votre code, ainsi que les tests de chacun des prédicats en fin de fichier.

TP 11

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 *domino solitaire* se joue avec des pièces (les dominos) qui comportent deux nombres. Ces nombres sont représentés par des ensembles de points comme illustré en figure 11.1.

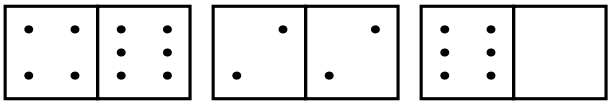


Fig. 11.1 – Exemple de dominos

Dans une partie de *domino solitaire*, le joueur dispose d'un ensemble de pièces qu'il doit toutes placer, les unes à la suite 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 les 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 $\{0, ..., 6\}$. L'ensemble des pièces sera représenté par une liste, par exemple :

```
stones([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 11.2.

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 autant de fois qu'il y a d'éléments dans la liste. Par exemple :

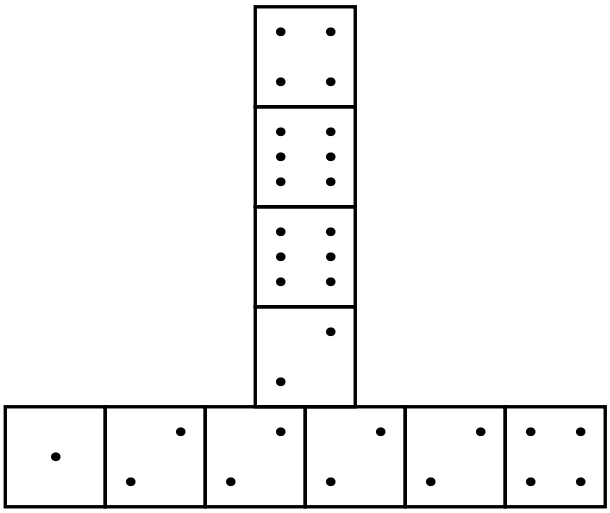


FIG. 11.2 – Une solution possible pour la configuration de l'exemple

```
?- 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 pour représenter le problème assez efficacement. En effet, cette structure de données va nous permettre d'élaguer au fur et à mesure l'espace de recherche.

On remarque dans la figure 11.2 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.

L'idée est d'avoir en tête de chaque liste 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)`, on obtient le terme :

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

Remarquons qu'on aurait pu représenter cette chaîne de façon plus naturelle par : `chain([1, 2, 2], [4])` mais cette représentation contient des informations redondantes car des dominos qui se touchent ont forcément les mêmes nombres sur leurs faces communes.

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

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

Si on ajoute ensuite un double, il faut créer une chaîne supplémentaire. Par exemple, si on ajoute le domino `stone(5, 5)`, on obtiendra deux chaînes :

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

et

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

`double` est utilisé au lieu de la liste vide car cela simplifie l'implémentation. 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 permet d'afficher de manière plus conviviale une liste de chaînes.

Écrire le prédicat :

```
%domino(-): chain list
%domino(Chains)
```

qui résoud le problème.

Pour écrire ce prédicat, vous pourrez faire appel à (et donc écrire) un prédicat `chains` qui permet de calculer le résultat :

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

Le deuxième argument de ce prédicat est un accumulateur qui contient le résultat partiel. Vous pouvez avoir à définir un (ou plusieurs) prédicats supplémentaires pour faciliter l'écriture de ce prédicat `chains`.

2 Compte rendu

Nous vous demandons de rendre les réponses aux questions, votre code, ainsi que les tests de chacun des prédicats en fin de fichier.

TP 12

Méta-interpréteurs en Prolog

Un interpréteur est un outil informatique ayant pour tâche d'analyser, de traduire et d'exécuter les programmes écrits dans un langage informatique. Dans ce TP, nous nous intéressons au langage Prolog et au programme capable d'exécuter un programme Prolog. Comme le programme que nous allons écrire sera lui-même écrit en Prolog, nous parlerons de méta-interpréteur (interpréteur écrit dans le langage du programme interprété). On peut se questionner sur l'utilité d'écrire un méta-interpréteur, sachant que, pour l'écrire, il faut déjà posséder un interpréteur du langage. Cela peut en fait permettre d'enrichir l'exécution. Par exemple, dans ce TP, nous allons garder une trace de l'exécution d'une question (quels nœuds de l'arbre de recherche ont permis d'aboutir à la solution). Cela permettra de bien comprendre le cheminement suivi par l'interpréteur pour arriver à son résultat.

« A meta-interpreter for a language is an interpreter for the language written in the language itself. Being able to write a meta-interpreter easily is a very powerful feature of a programming language. It gives access to the computation process of the language and enables the building of an integrated programming environment. »[1]

1 Objectif

Avant de commencer le TP, récupérer sur Moodle le squelette de nom :
`TP12_meta_interpreteur_squelette.pl`.

Dans ce TP, vous allez programmer un méta-interpréteur qui garde une trace de l'exécution d'une requête Prolog. Au passage, nous allons introduire la notion de code dynamique. Lorsqu'un prédicat est déclaré dynamique, il est possible d'ajouter des clauses pour ce prédicat pendant l'exécution.

2 Un méta-interpréteur traceur

Comme dit précédemment, l'objectif est de programmer un méta-interpréteur qui garde une trace d'exécution sous la forme d'une branche de l'arbre de recherche pour chaque solution trouvée. L'appel au méta-interpréteur se fera par l'appel au prédicat *solve* d'arité 2 : *solve(+X, -Z)*. X est le but à démontrer et Z contiendra les nœuds parcourus de l'arbre de recherche sous la forme d'une liste (*cf.* exemple ci-après). S'il existe plusieurs solutions, la requête aboutira à plusieurs succès avec des instanciations de Z différentes.

Considérons l'exemple suivant :

[1] The Art of Prolog, Leon Sterling and Ehud Shapiro, MIT Press 1994

```
enfant(e1).
pere(p1,e1).
pere(gp1,p1).
parent(X,Y):-pere(X,Y).
```

À la question *solve*(*parent*$(X, Y), Z$)., nous voulons, à la fin du TP, avoir des réponses de la forme :

```
X = p1
Y = e1
Z = [parent(p1, e1) , pere(p1, e1) , true]
Yes (0.00s cpu, solution 1, maybe more) ? ;
```

Z indique le chemin dans l'arbre de recherche : *parent(p1,e1)* est la racine ; on est ensuite passé par le nœud fils *pere(p1,e1)* puis par un troisième nœud succès (*true*). On voit ici que l'on a enrichi l'exécution de la requête puisque l'on a conservé une trace des nœuds de l'arbre d'exécution par lesquels on est passé.

Nous allons partir d'un méta-interpréteur très simple, puis nous l'enrichirons progressivement pour arriver finalement au prédicat *solve* présenté ci-dessus.

2.1 solve0 : interpréteur basique

Le but de ce prédicat *solve0* est de répondre aux requêtes Prolog simples. La version la plus simple d'un méta-interpréteur en Prolog est :

```
solve0(A) :-
            A.
```

Cette version est très utile quand on ne sait pas, au moment où on écrit le code, quel sera le but à exécuter, par exemple parce qu'on le construit dans le programme. En revanche, il ne permet pas de simuler le raisonnement Prolog : le raisonnement est fait par l'interpréteur que l'on utilise.

Question 2.1. Tester le but *solve0(enfant(X))*.

Pour pouvoir utiliser la conjonction dans le méta-interpréteur, il faut entourer les buts de parenthèses et les séparer par des virgules. Ainsi *solve0((B1,B2,B3))* vérifiera si B1 et B2 et B3 sont vrais.

```
solve0((B1,B2,B3))
      |
      |
      |
  B1,B2,B3
```

Pour trouver tous les enfants de p1 qui ont 10 ans, on peut par conséquent poser la question suivante : *solve0((pere(p1,X),age(X,10)))*.

Question 2.2. Quel interpréteur est appelé pour résoudre le but *pere*$(p1, X)$? Modifier le prédicat *solve0* pour appeler votre interpréteur *solve0* dans le cas d'une conjonction.

Si on veut maîtriser l'exécution (par exemple, connaître les nœuds de l'arbre par lesquels on est passé), il faut programmer un méta-interpréteur plus détaillé.

2.2 solve1 : interpréteur simulant la réduction des buts

Réduction des buts

Nous allons maintenant aller plus loin en simulant la façon de fonctionner de Prolog. Pour résoudre A, Prolog recherche s'il existe dans sa base une clause avec la tête de laquelle il peut unifier A. Si c'est le cas et si, par exemple, la clause est de la forme A :- B, il cherche alors à résoudre B. On peut noter au passage qu'un fait est modélisé par la règle : $true : -fait$ et que $true$ est toujours vrai.

Prédicat clause

Pour écrire le prédicat *solve1*, vous allez devoir utiliser le prédicat Prolog clause/2 ; $clause(+head, -body)$ aboutit à un succès si $head : -body$ est une clause existante au moment de l'exécution de clause/2 et si head est un prédicat dynamique.

Question 2.3. Tester $clause(parent(X, Y), Z)$. Quel message obtenez-vous ?

Ceci est tout à fait normal puisque le prédicat *clause* ne fonctionne que sur des prédicats dynamiques.

Question 2.4. Rendre dynamiques les prédicats *enfant*, *age*, *pere* et *parent*.

Attention : si le prédicat a déjà été chargé en NON dynamique, on ne peut pas le déclarer en dynamique sans sortir et re-rentrer sous ECL^{i}PS^{e}. Il vous faut donc quitter ECL^{i}PS^{e}, le relancer et recharger votre programme.

Question 2.5. Tester de nouveau le prédicat *clause(parent(X,Y),Z).*

On considère le prédicat *my_append* dont le code est donné ci-dessous :

```
my_append([],Ys,Ys).
my_append([X|Xs],Ys,[X|Zs]) :-
    writeln(X),
    my_append(Xs,Ys,Zs).
```

Question 2.6. Tester $my\_append(X, Y[a, b, c])$.

Question 2.7. Tester

```
:-clause(my_append(A,B,C),Body).
:-clause(my_append([],B,C),Body).
```

Bien regarder le travail d'unification.

Question 2.8. Tester *clause(true).* Quel message obtenez-vous ?

Il va donc falloir faire attention dans la suite à appeler le prédicat *clause* uniquement sur des prédicats accessibles.

Question 2.9. Écrire une première version du prédicat *solve*1, que l'on appellera *solve*1_1, de façon à être capable d'exécuter un but en Prolog pur (Prolog sans coupure, négation...) en décomposant l'exécution des sous-buts des clauses qui s'unifient avec le but initial.

Prise en compte des prédicats prédéfinis

Question 2.10. Tester $solve1\_1(my\_append(X, Y, [1, 2, 3]))$. Quel message obtenez-vous ?

La seconde version du prédicat *solve*1, que l'on appelera *solve*1_2, va permettre de résoudre ce problème. Pour savoir si un prédicat est un prédicat prédéfini de Prolog, nous allons définir un prédicat *built_in* d'arité 1 dans lequel tous les prédicats prédéfinis que l'on s'autorise à utiliser vont être listés.

Pour accepter *writeln* avec un paramètre, on écrira : $built\_in(writeln(\_))$.

Question 2.11. Écrire le prédicat *solve*1_2 prenant en compte les prédicats prédéfinis *writeln* et *read*.

2.3 solve2 : interpréteur gardant une trace de l'exécution

Question 2.12. Écrire le prédicat $solve2\_1(X, Z)$ qui garde une trace de l'exécution pour chaque solution sous la forme d'une branche de l'arbre de recherche tel que vu en cours.

Exemple :

```
solve2_1(my_append(A, B, [a, b, c]), Branch).

A = []
B = [a, b, c]
Branch = [my_append([], [a, b, c], [a, b, c]), true]
Yes (0.00s cpu, solution 1, maybe more) ? ;

A = [a]
B = [b, c]
Branch = [my_append([a], [b, c], [a, b, c]), writeln(a), builtin,
my_append([], [b, c], [b, c]), true]
Yes (0.00s cpu, solution 2, maybe more) ?
```

Question 2.13. Tester $solve2\_1(my\_append([a, c], B, [a, b, c]), Branch)$.

Question 2.14. Écrire le prédicat $solve2\_2(X, Z)$ tel que la trace d'exécution soit également rendue en cas d'échec, tout en conservant cet échec.

Exemple :

```
[eclipse] : solve2_2(my_append([a,c], B, [a, b, c]), Branch).

B = B
Branch = [my_append([a, c], B, [a, b, c]), writeln(a), builtin,
my_append([c], B, [b, c]), echec]
Yes (0.00s cpu, solution 1, maybe more) ? ;

B = B
Branch = [my_append([a, c], B, [a, b, c]), echec]
Yes (0.00s cpu, solution 2, maybe more) ? ;

No (0.00s cpu)
```

Comme lors de tous les TP, veillez à bien tester vos prédicats.

2.4 La boucle d'interprétation (partie optionnelle)

On souhaite écrire un prédicat *my_top* qui affiche un prompt, permet de saisir des prédicats, retourne les résultats en utilisant le prédicat *solve1_2* écrit en fin de partie 2.2 et recommence. *my_top* est donc une boucle infinie. La sortie de cette boucle se fait par `Ctrl C puis abort`. Vous avez toute latitude pourvu que les éléments indispensables soient présents. Attention, si le prédicat entré par l'utilisateur a plusieurs solutions, `my_top/0` doit les proposer.

Voici un exemple d'exécution :

[eclipse 7] : my_top.
[my_toplevel] : enfant(X).

Result : enfant(e2)
(more solutions ?) yes.

Result : enfant(e1)
(more solutions ?) no.

[my_toplevel] :

interruption : type a, b, c, e, or h for help : ? abort
Aborting execution ...
Abort

Il est utile d'avoir lu la documentation des prédicats `read/1`, et éventuellement `get_char/1` ou `get_char/1`.

3 Compte-rendu

Vous devez rendre

1. le code final du méta-interpréteur de la question 2.11. Dans un commentaire en en-tête du prédicat principal, identifiez bien quelles sont les parties qui répondent à quelles étapes ;

2. les résultats des tests de la partie obligatoire.

Ceux qui ont fait la partie optionnelle peuvent également la rendre : le code et les tests de cette partie seront mis après l'intégralité des éléments concernant la partie obligatoire.