UNIVERSITÉ DE
SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

LE LANGAGE PROLOG

Notes de cours
GEI457–Intelligence artificielle et langage associés

Charles-Antoine Brunet
François Michaud

Sherbrooke (Québec) Canada
Octobre 2001

Table des matières

1 Objectifs

- Introduire les concepts de base de la programmation logique avec Prolog.
- Présenter les différentes fonctionnalités de Prolog utiles pour le cours, avec des exemples.
- Présenter les exercices à faire en laboratoire pour comprendre le langage et programmer.
- Donner des indications sur le fonctionnement du logiciel utilisé pour les laboratoires, LPA Prolog. Vous pouvez télécharger une version de Prolog et sa documentation du site `http://www.lpa.co.uk`.
- Présenter les notions rattachées aux chapitres 6 à 10 du livre de déférence du cours [2].

2 Introduction

La logique sert de fondement pour déduire des conséquences à partir de prémisses, d'étudier la véracité et la fausseté d'énoncés en fonction de la véracité ou la fausseté d'autres énoncés, ainsi que d'établir la consistance et de vérifier la validité d'énoncés. Le but premier de la programmation logique est de permettre de programmer à un haut niveau.

Prolog est un langage de programmation qui a comme base deux principes associés à la logique :

- un programme est un ensemble de règles (communément appelées axiomes) ;
- le calcul consiste à construire une preuve à partir du programme et en partant de l'énoncé de but.

Pour le cours, l'objectif est de vous faire connaître les mécanismes de base d'un tel langage pour que vous puissiez évaluer son utilité dans les applications propres à l'intelligence artificielle. Bien entendu, il nous est impossible de couvrir en détails toutes les fonctionnalités d'un tel langage dans le laps de temps alloué pour le cours. C'est pourquoi ce document vient synthétiser les informations requises pour rencontrer les objectifs du cours. Les objectifs sont d'être en mesure de concevoir des programmes simples ainsi que d'exploiter Prolog pour la résolution de problèmes logiques.

La section 3 présente les principes de base de Prolog, et la section 4 décrit des exemples de structures de programmes communément utilisées, avec des exercices. Les solutions sont données à l'annexe B. Ces exercices et les différents programmes présentés dans le document devront être expérimentés en laboratoire car c'est le seul moyen de bien comprendre ce type de programmation. Des indications sur le logiciel utilisé sont données à l'annexe A. Enfin, la matière rattachée aux chapitres 6 à 10 du volume de cours de Russell et Norvig [2] est illustrée à la section 5 par des programmes en Prolog pour résoudre des problèmes logiques. Encore une fois, le choix de la représentation du problème est un point important que vous devrez bien comprendre.

Pour obtenir plus de détails sur l'ensemble des fonctionnalités de Prolog, veuillez consulter les références.

Enfin, il est possible que certaines erreurs ou manquements se soient glissés de façon non intentionnelle dans ce document, ou que certaines parties demandent plus d'éclaircissement. Tout commentaire pour améliorer ces notes sera très apprécié par l'auteur.

3 Programmation logique avec Prolog

Un programme logique est un ensemble de règles définissant des relations entre des objets. Le traitement d'un programme logique est la déduction de conséquences (soit le résultat d'une règle) du programme. Un programme définit un ensemble de conséquences, ce qui lui donne sa signification. L'art de la programmation logique est de construire des programmes concis et élégants qui possèdent la signification désirée.

L'interprétation procédurale de ce langage correspond à la clause logique de Horn :

$$B_1, B_2, \ldots, B_n \Rightarrow A$$

Ce qui signifie A est vrai si B_1 est vrai et B_2 est vrai et ... et B_n est vrai ; ou encore, pour résoudre A, résoudre B_1 et B_2 et ... et B_n.

La procédure de preuve de clause de Horn est l'interpréteur de Prolog. L'algorithme d'unification, qui est au coeur de la procédure de résolution de preuve, effectue les opérations de manipulations de données pour l'assignation de variable, le passage de paramètres, la sélection de données et la construction de données. En fait, Prolog offre un système de déduction capable de répondre à toute question dont la réponse est logiquement déductible des connaissances fournies préalablement. Cette réponse est obtenue par une exploration systématique de l'ensemble des cheminements logiques permettant de remonter de la question aux faits par l'entremise de règles.

En Prolog, il y a trois énoncés de base : les faits, les règles et les requêtes (ou questions), et une seule structure de données : le terme logique.

3.1 Faits

- Établissent une relation (ou un prédicat) entre objets (ou atomes)
- Donnent une description de la situation. Ils sont similaires à une base de données. Un fait indique que la relation établie entre les objets est vraie. Les faits sont toujours vrais. Plusieurs faits du même type correspondent à une disjonction de la relation entre les objets (OU).
- Tous termes doivent utiliser une minuscule comme première lettre, à l'inverse de la convention utilisée dans le volume de Russell et Norvig. Ils se terminent par un point.
- Un exemple de base de faits couramment rencontré dans la littérature est les liens de parenté :

```
father(terach,abraham).  male(terach).
father(terach,nachor).   male(abraham).
father(terach,haran).    male(nachor).
father(abraham,isaac).   male(haran).
father(haran,lot).       male(isaac).
father(haran,milcah).    male(lot).
father(haran,yiscah).
                         female(sarah).
mother(sarah,isaac).     female(milcah).
                         female(yiscah).
```

3.2 Requêtes

- Permet d'obtenir de l'information d'un programme logique. En demandant `P?`, on demande si un but `P` est vrai.
- Une requête sera préfixée par `?-`.
- Répondre à une requête en fonction d'un programme consiste à déterminer si elle est une conséquence logique du programme à partir de règles de déduction. Trois principales règles sont utilisées :

  1. **L'identité :** vérifie si un but correspond à un des faits présents dans la base de faits :

     ```
     ?-father(abraham,isaac).    yes
     ?-father(abraham,lot).      no
     ```

  2. **La généralisation :** Pour exploiter cette règle de déduction, on doit introduire le concept de variable. Les variables sont le moyen de résumer plusieurs requêtes. Une variable est une entité unique mais non définie (pas de type). On identifie une variable en employant la majuscule comme première lettre, à l'inverse de la convention utilisée dans le volume de Russell et Norvig. La généralisation consiste à effectuer une requête vérifiant l'existence (quantificateur existentiel) d'une substitution valide pour la variable, permettant de rendre vraie la requête.

     ```
     ?-father(abraham,X).  X = isaac
     ?-father(haran,X).    X = lot, X = milcah, X = yiscah
     ```

     Il est à noter que Prolog peut instancier temporairement des variables dans l'espoir qu'elles pourront l'être plus tard avec des éléments de la base de connaissances. Ces variables sont présentées sous le format `_#` (i.e., `_` suivi d'un numéro de variable, par exemple : `X=_143`). S'il ne peut instancier ces variables, c'est un signe qu'une erreur est présente dans la base de connaissances.

  3. **L'instanciation ou l'unification :** L'utilisation de variables dans les faits permet de résumer ou de synthétiser plusieurs faits. L'instanciation consiste à trouver les substitutions à cette variable permettant de rendre le fait vrai. Cette façon de procéder est similaire à rendre le fait universel ($\forall$).

     L'unification consiste à considérer deux termes et de tenter de les rendre identiques par l'instanciation de leurs variables. Prolog unifie des termes pour gérer les appels

nécessaires à l'exécution d'un programme ainsi que la transmission des paramètres. Les règles de l'unification des termes `S` et `T` sont :

- si `S` et `T` sont des constantes, ils s'unifient s'ils correspondent au même objet.
- si `S` est une variable et `T` est un objet, l'unification consiste à instancier `S` à `T`. Ceci est aussi valable si `S` est un objet et `T` est une variable.
- si `S` et `T` sont des termes composés, alors il s'unifient s'ils ont la même racine et que leurs composantes s'unifient deux à deux.

Voici quelques exemples :

(a) `titi(X,2,tutu(Z))` et `titi(tutu(1),Y,tutu(4))`
Unification `X=tutu(1), Y=2, Z=4`

(b) Si `titi(Z,Y,tutu(2))` avec `titi(X,2,tutu(Z))`
Non unifiable car `Z=X=???`

(c) Sachant que `plus(0,3,3)` existe, `plus(0,X,X)` peut être instancié avec `X=3`

- Les requêtes peuvent comporter des conjonctions de requêtes. On représente la conjonction entre les buts d'une requête par une virgule. Il ne faut pas confondre ceci avec l'utilisation de la virgule à l'intérieur d'un fait qui sert à séparer les objets d'une relation.
- Les conjonctions de requêtes sont intéressantes lorsqu'elles partagent des variables entre les buts. La portée de la variable est la conjonction en entier.
  `?-father(haran,X),male(X).  X = lot`

- La réponse à une question peut être positive, dans ce cas les variables sont instanciées aux réponses obtenues étant vraies selon le programme logique. La réponse peut aussi être négative lorsque aucune substitution permet de rendre les buts de la requête vrais.
- Si plusieurs réponses peuvent être apportées à une question, alors Prolog donnera la première, puis les suivantes à la demande de l'utilisateur.

3.3 Règles

- Elles déclarent des choses vraies sous certaines conditions.
- Les règles ont la forme $A \Leftarrow B_1, B_2, \ldots, B_n$, soit $Tête \Leftarrow Corps$. Le corps est une conjonction de buts séparés par des virgules. Les caractères `:-` correspondent au symbole $\Leftarrow$.
  ```
  son(X,Y) :- father(Y,X),male(X).
  grandfather(X,Z) :- father(X,Y),father(Y,Z).
  ```

- Interprétation procédurale (comment obtenir les solutions) : pour connaître si `X` est le grand-père de `Z`, il faut déterminer si `X` est le père de `Y` et `Y` est le père de `Z`. Pour exécuter A il faut exécuter B_1 et puis B_2, etc.
- Interprétation déclarative (quelles sont les solutions) : pour tout `X`, `Y` et `Z`, `X` est le grand-père de `Z` si[1] `X` est le père de `Y` et `Y` est le père de `Z`.

[1]Formulé autrement : s'il existe un Y tel que, ce qui est équivalent à un quantificateur existentiel à l'intérieur de la règle.

– La loi du Modus ponens établit que de B' (un ensemble de faits) et `A:-B` (une règle $A \Leftarrow B_1, B_2, \ldots, B_n$), on peut déduire A'.
– Un fait est un cas spécial de règle avec aucun antécédents, i.e. une règle qui a son corps vide. Une règle se termine donc aussi par un point. Une requête ne possède qu'un corps et pas de tête.
– Une règle permet d'exprimer une requête sous forme de plusieurs requêtes (plus simples on l'espère !).

3.4 Terme

– Un terme est soit une constante, une variable ou un terme composé. Il sert à définir des objets.
– Une constante peut être un nombre (par exemple, 1, -5, 0.5), ou un atome (une chaîne de lettres qui commence par une minuscule, par exemple abraham ou une chaîne entre ' ', par exemple 'composante X').
– Un terme composé est formé d'un foncteur, soit son nom f, et de ses arguments t_i, donc de la forme $f(t_1, \ldots, t_n)$ et d'arité n. `tree(tree(nil,3,nil),5,R)`, `père(Luc,Éric)` et `foo(X)` sont des exemples de termes composés. On fait habituellement référence aux prédicats en employant la structure foncteur/arité.
– Certains prédicats ont certaines restrictions quand aux types d'arguments admissibles. Convention : `f(+Arg1, ?Arg2, -Arg3)`, le `+` signifiant un argument d'entrée qui doit être instancié lors de l'appel, le `-` référant à un argument de sortie qui doit être non-instancié lors de l'appel et qui sera instancié si le prédicat réussit, et le `?` réfère à un argument d'entrée ou de sortie, instancié ou non-instancié.
– Un terme est dit *ground* s'il ne contient pas de variables.
– Un but est soit un atome ou un terme composé. On utilise le terme but pour illustrer que Prolog envisage les questions comme autant de buts à satisfaire.
– Le type d'un objet est défini entièrement par son apparence syntaxique.
– La portée lexicale d'une variable est d'une seule clause. Le même nom dans deux clauses correspond donc à deux variables différentes.

3.5 Opérateurs artihmétiques

Des opérateurs arithmétiques peuvent être programmés en utilisant des clauses logiques, mais ceci n'est pas très efficace. Des opérateurs ont donc été prédéfinis pour effectuer ces opérations. Ils occasionnent toutefois des contraintes sur les types possibles des arguments.

Opérateur	Description	Exemple
=:=/2	Égalité entre deux expressions arithmétiques	?-4+(2*1)=:= 6. yes
=\=/2	Inégalité entre deux expressions arithmétiques	?-4=\=5+2. yes
=/2	Unification entre deux termes (i.e. $X = X$)	?-X=[1,2,3]. X=[1,2,3] ?-a(1)=a(X). X=1
\=/2	Vérifie la non-unification entre deux termes	?-5\=6. yes
==/2	Vérifie si deux termes sont identiques	?-a(1)==a(X). no
\==/2	Vérifie si deux termes sont non identiques	?-a(2)\==a(1). yes ?-X\==Y. yes
=</2	Plus petit ou égal à	?-2=<3. yes
>=/2	Plus grand ou égal à	?-3>=3. yes
>/2	Plus grand	?-3>2. yes
</2	Plus petit	?-2<3. yes
is/2	Évaluation d'expression	?-T is 0+1. T=1

Avec le prédicat `is/2`, une série de fonctions arithmétiques peuvent être utilisées. Les arguments assignés à ces opérateurs mathématiques doivent être instanciés. Voici quelques-unes de ces fonctions :

Fonction	Description	Exemple
X + Y	Somme de X et Y	?-T is 1 + 3. T=4
X - Y	Différence de X et Y	?-T is 1 - 3. T=-2
X * Y	Produit de X et Y	?-T is 1 * 3. T=3
X / Y	Quotient de X et Y	?-T is 6 / 3. T=2
X ^ Y	X élevé à la puissance Y	?-T is 2 ^ 3. T=8
sqrt(X)	Racine carrée de X	?-T is sqrt(4). T=2
-X	Négatif de X	?-T is -3. T=-3
abs(X)	Valeur absolue de X	?-T is abs(-3). T=3
max(X,Y)	Maximum entre X et Y	?-T is max(-3,1). T=1
min(X,Y)	Minimum entre X et Y	?-T is min(-3,1). T=-3

3.6 Prédicats de contrôle

Il existe aussi des prédicats qui permettent de contrôler l'exécution du programme. Normalement, le programme cherche de façon non-déterministe pour une solution, effectuant des retours en arrière et terminant quand la première solution est trouvée. Les prédicats de contrôle permettent de modifier ce comportement. La virgule pour la conjonction entre les buts dans le corps d'une clause est un de ces prédicats. Voici quelques-uns de ces prédicats :

Prédicat	Description	Exemple
`!/0`	Le Cut, il élimine les points de choix pour les clauses qui le précède. Dans l'exemple fourni, il n'y aura pas de retour en arrière pour le but b et la deuxième clause a.	`a:-b,!,c. a:-d.`
`not/1`	Négation logique. Retourne vrai si le but auquel il est appliqué échoue (attention aux variables non instanciées)	`?-not father(abraham,lot). yes`
`true/0`	Succès	`?-true. yes`
`false/0`	Force une faute	`?-false. no`
`fail/0`	Force une faute	`?-fail. no`

L'usage du cut est contreversé, car son utilisation doit souvent être interprétée de façon procédurale, ce qui va à l'encontre du style de programmation logique. Toutefois, son usage peut améliorer l'efficacité de programmes sans compromettre leur interprétation. Pour le cours, nous allons tenter de ne pas l'utiliser, car des consignes doivent être respectées lors de son utilisation et ces aspects s'éloignent des objectifs du cours.

3.7 Listes

La liste est aussi une structure de données importante en Prolog. Elle s'exprime par la forme `[X|Xs]`, où `X` est la tête de la liste et `Xs` la queue de la liste. La liste vide se représente par `[]`. Voici quelques exemples :

```
[a,b,c,d,e]
[a,[1,2],[b],q,r,s,[[3],2]]
```

Des prédicats pré-définis existent aussi pour la manipulation de listes :

Prédicat	**Description**	**Exemple**

`append/3`
`append(?L1,?L2,?L3)`
	Joindre (`L1` avec `L2` pour donner `L3`), divise (connaissant `L1` et `L3`, trouver `L2` ; connaissant `L2` et `L3`, trouver `L1` ; connaissant `L3`, trouver toutes les `L1` et `L2`) ou vérifie la jonction des listes.	`?-append([a,b,c],[1],L).`
		`L=[a,b,c,1]`
		`append([a,b],L,[a,b,c]).`
		`L = [c]`
		`append(L,[c],[a,b,c])`
		`L = [a, b]`
		`append(L,L2,[a,b,c])`
		`L = [], L2 = [a, b, c]`
		`L = [a], L2 = [b, c]`
		`L = [a, b], L2 = [c]`
		`L = [a, b, c], L2 = []`
		`No more solutions`

`length/2` `length(?X,?L)`	Retourne ou vérifie la longueur `L` d'une liste `X`. Si `X` n'est pas instanciée, la fonction retourne des variables non unifiées.	`?-length([1,2],X).` `X=2`

`member/2`
`member(?E,?L)`
	Vérifie ou obtient un membre `E` d'une liste `L`. Si `E` est une variable, alors le prédicat retourne toutes les possibilités de membres de la liste `L`.	`?-member(1,[1,2]).`
		`yes`
		`?-member(X,[1,2]).`
		`X=1`
		`X=2`

`member/3`
`member(?E,?L,?P)`
	Vérifie ou obtient un membre `E` d'une liste `L` à la position `P`. Si `E` est une variable, alors le prédicat retourne toutes les possibilités de membres de la liste `L`.	`?-member(1,[1,2],P).`
		`P=1`
		`?-member(X,[1,2,3],P).`
		`X=1, P=1`
		`X=2, P=2`
		`X=3, P=3`

`remove/3`
`remove(?E,?L,?R)`
	Élimine un élément `E` de la liste `L`, et retourne `R`. Le deuxième ou troisième argument doit être initialisé.	`?-remove(1,[1,2,1],R).`
		`R=[2,1]`
		`R=[1,2]`
		`?-remove(E,[1,2],[1]).`
		`E=2`
		`?remove(1,L,[2]).`
		`L=[1,2]`
		`L=[2,1]`

`removeall/3` `removeall(?E,+L,?R)`	Élimine toutes les occurrences d'un item `E` dans une liste `L`, et retourne `R`.	`?-removeall(1,[1,2,1],R).` `R=[2]`

`reverse/2`
`reverse(?L,?R)`
	Vérifie (`L` et `R` sont instanciés) ou obtient la liste inverse.	`?-reverse([1,2,3],R).`
		`R=[3,2,1]`
		`?-reverse(L,[3,2,1]).`
		`L=[1,2,3]`

3.8 Programme logique

– Un programme logique est un ensemble fini de règles.
– Une procédure est un ensemble de règles avec le même prédicat en tête (partie A) de la règle.

Programme pour déterminer un repas léger

```
/* Menu du restaurant */
horsDOeuvre(radis).
horsDOeuvre(pate).

poisson(sole).
poisson(thon).

viande(porc).
viande(boeuf).

dessert(glace).
dessert(fruit).

plat(X):-poisson(X). /* Tout poisson est un plat */
plat(Y):-viande(Y). /* Toute viande est un plat */

/* Points en fonction des calories */
points(radis,1).
points(pate,6).
points(sole,2).
points(thon,4).
points(porc,7).
points(boeuf,3).
points(glace,5).
points(fruit,1).

/* Règle définissant un repas */
repas(H,P,D):-horsDOeuvre(H),plat(P),dessert(D).

/* Règle definissant un repas léger */
repasLeger(H,P,D):-repas(H,P,D),points(H,X),points(P,Y),points(D,Z),
SousTotal is X+Y,Total is SousTotal+Z, Total<10.
```

Trouver tout les repasLeger

```
?-repasLeger(X, Y, Z).
No.1 : X = radis, Y = sole, Z = glace
No.2 : X = radis, Y = sole, Z = fruit
```

```
No.3 : X = radis, Y = thon, Z = fruit
No.4 : X = radis, Y = porc, Z = fruit
No.5 : X = radis, Y = boeuf, Z = glace
No.6 : X = radis, Y = boeuf, Z = fruit
No.7 : X = pate, Y = sole, Z = fruit
No more solutions
```

Liens de parenté

```
father(terach,abraham). male(terach).
father(terach,nachor).  male(abraham).
father(terach,haran).   male(nachor).
father(abraham,isaac).  male(haran).
father(haran,lot).      male(isaac).
father(haran,milcah).   male(lot).
father(haran,yiscah).
                        female(sarah).
mother(sarah,isaac).    female(milcah).
                        female(yiscah).

child(Child,Parent):- father(Parent,Child).
child(Child,Parent):- mother(Parent,Child).

parent(Dad,Child):- father(Dad,Child).
parent(Mom,Child):- mother(Mom,Child).

grandparent(X,Y):- parent(X,Z),parent(Z,Y).

son(Son,Parent):- parent(Parent,Son),male(Son).
```

Exemples de requête

```
?-child(X, Y).
No.1 : X = abraham, Y = terach
No.2 : X = nachor, Y = terach
No.3 : X = haran, Y = terach
No.4 : X = isaac, Y = abraham
No.5 : X = lot, Y = haran
No.6 : X = milcah, Y = haran
No.7 : X = yiscah, Y = haran
No.8 : X = isaac, Y = sarah
No more solutions

?-son(isaac, Y).  ; Trouver les Y pour qui isaac est le fils de Y

No.1 : Y = abraham
No.2 : Y = sarah
```

```
No more solutions

?-grandparent(terach, isaac). ; terach est-il le grand-père de isaac?

No.1 : yes
No more solutions
```

- La signification d'un programme logique est l'ensemble des buts pouvant être déduits du programme. On se doit de vérifier si cette signification est correcte et complète.
- En Prolog, établir si un objet répond à une question est un processus mettant en jeu des inférences logiques, l'exploration de plusieurs possibilités, ainsi que de possibles retours en arrière (*backtracking* - offrant le non-déterminisme).
- On pourrait comparer le processus d'exploration à instancier des variables pour satisfaire le prédicat B_1, et répéter le processus avec ces instanciations pour instancier d'autres variables dans les prédicats $B_2, \ldots, B_n$. On retourne un résultat si on peut instancier toutes les variables et vérifier si tous les prédicats du corps de la requête sont vrais. Si on ne peut plus instancier de variables permettant de rendre vrai un B_n, alors on remonte en arrière et on tente une nouvelle solution. On applique le même processus lorsqu'on cherche à trouver toutes les solutions possibles. Le processus revient automatiquement en arrière pour examiner les différentes possibilités. Ce principe de retour en arrière et l'absence de déclaration de types pour les variables font que Prolog diffère des langages de programmation procédurale. Une variable logique réfère à un individu au lieu d'un espace mémoire. Une fois qu'elle réfère à un individu, elle ne peut plus référer à un autre. En d'autres mots, la programmation logique ne supporte pas l'assignation destructive où le contenu d'une variable initialisée peut changer. Il en résulte une structure arborescente de données qui permet la récursivité. On récupère l'espace mémoire des variables inutilisées avec un garbage collector.
- Un arbre de démonstration est utilisé pour représenter la séquence d'appels et d'instanciation. Un arbre comporte des noeuds et des arcs représentant les buts réduits pendant le calcul. La racine de l'arbre de démonstration pour une question simple est la question elle-même. Les noeuds de l'arbre sont les buts qui ont été explorés pendant le calcul. Il y a un arc orienté d'un noeud vers tout noeud qui correspond à un but dérivé du but réduit. L'arbre de démonstration pour une question conjonctive est simplement la collection des arbres de démonstration pour les buts individuels de la conjonction. Exemple (voir aussi fig. 9.3 et 9.4, volume de Russell et Norvig) :

3.9 Exercices série A

1. Définir des règles qui convertissent les degrés Farenheit en Celsius (et vice-versa), sachant que $Celcius = (F + 40) * 5/9 - 40$ et $Farenheit = (C + 40) * 9/5 - 40$.
2. Définir une procédure qui retourne M, le maximum entre X, Y et Z :
   (a) En utilisant le prédicat >=.
   (b) En utilisant seulement >.
   (c) Est-ce que les résultats sont similaires ?

3. En employant seulement `append`, construire les prédicats suivants. Faites des tests en effectuant des requêtes avec une liste de votre choix.

```
memberA(?X,+L)      % X est un élément membre de la liste L
dernier(?X,+L)      % X est le dernier élément de la liste L
enleveUn(?X,+L,?M)  % M est la liste L moins l'élément X enlevé
adjacent(?X,?Y,+Zs) % X et Y sont adjacents dans la liste L
avant(?X,?Y,+L)     % X est avant Y dans la liste L
```

4. Un palindrome est une liste qui a la même séquence d'éléments lorsqu'elle est lue de droite vers la gauche ou de gauche vers la droite. Définir `palindrome(?L, ?P)`, une procédure qui retourne le palindrome `P` d'une liste `L` qui a deux fois sa longueur (truc : utiliser append et reverse).
5. Définir une procédure `rotationGauche(+L,?R)` qui effectue une rotation vers la gauche `R` d'une liste `L`, et une procédure `rotationDroite(+L,?R)` qui effectue une rotation vers la droite `R` d'une liste `L` (truc : utiliser `append` ou `reverse`).
6. Cet exercice cherche à vous faire réfléchir sur le choix des prédicats pour représenter un problème. Pour un programme mettant en évidence les horaires d'ensignement d'un département, il serait possible d'utiliser la représentation suivante où on place dans une seule relation l'état de la situation (en utilisant des termes imbriqués) :

```
prof(Prof,Cours) :- cours(Cours,Heure,Prof,Local).

duree(Cours,Longueur) :-
  cours(Cours,temps(Jour,Debut,Fin),Prof,Local),
  plus(Debut,Longueur,Fin).

enseigne(Prof,Jour) :- cours(Cours,temps(Jour,Debut,Fin),Prof,Local).

occupe(Local,Jour,Heure) :-
  cours(Cours,temps(Jour,Debut,Fin),Prof,Local),
  Debut =< Heure, Heure =< Fin.

cours(ia,temps(lundi,13,16),prof(ca,brunet),lieux(facGenie,304)).
```

   On pourrait toutefois représenter le même problème en utilisant des relations binaires, i.e., entre deux arguments.

```
jourCours(ia,lundi).
heureDebut(ia,13).
heureFin(ia,16).
prof(ia,charles_antoine_brunet).
edifice(ia,facGenie).
local(ia,304).
```

   En utilisant cette représentation, définir les règles pour les prédicats suivants :

```
enseigne(?Prof,?Jour)
occupe(?Prof,?Heure)
nePeutRencontrer(?Prof1,?Prof2)
conflitCedule(?Heure,?Place,?Cours1,?Cours2).
```

7. À partir du programme suivant :

```
d(X,Y):-X>1,Y>1.
a(0,1).
a(0,2).
a(2,1).
a(M,N):-b(P,Q),b(Q,P),M is P+1,N is Q+1.
c(0).
b(3,1).
b(2,1).
b(1,2).
```

Déterminer **manuellement**, en reproduisant la trace des requêtes Prolog, toutes les réponses à la question suivante, et **ensuite** vérifier avec Prolog.

```
?-a(X,Y),not(c(X)),d(X,Y).
```

4 Structures de programmes couramment utilisées

4.1 Points importants

– L'ordre des règles détermine l'ordre dans laquelle les solutions sont trouvées. Il ne change pas l'arbre de recherche. La convention est de placer les règles récursives avant les clauses de base.
– L'ordre des buts dans une règle détermine l'arbre de recherche en spécifiant le flot séquentiel de contrôle dans les programmes Prolog. Il faut éviter les règles récursives avec le but récursif comme premier but dans le corps de la règle. Pour limiter les calculs, il est plus efficace de placer les clauses les plus restrictives ou celles qui doivent initialiser les variables au début dans le corps d'une règle. En Prolog, l'objectif est d'échouer le plus rapidement possible pour limiter l'arbre de recherche et trouver la bonne solution plus tôt.
– L'arbre de démonstration est traversé profondeur d'abord (*depth-first*), et si une branche est infinie, le processus d'évaluation ne se terminera jamais. La non-terminaison survient lors d'appels récursifs. Les règles récursives qui ont le but récursif comme premier but dans le corps de la règle sont dites récursives par la gauche. Par exemple :

```
married(X,Y):-married(Y,X).
married(abraham, sarah).
?-married(X,Y).
  married(abraham,sarah)
    married(sarah,abraham)
      married(abraham,sarah)
```

...

La meilleure solution au problème de récursion par la gauche est d'éviter d'utiliser de telles clauses lorsque c'est possible. Des relations commutatives (comme `married`) peuvent être traitées en définissant un nouveau prédicat qui a une clause pour chaque permutation des arguments de la relation. Par exemple :

```
areMarried(X,Y):-married(X,Y).
areMarried(X,Y):-married(Y,X).
```

Enfin, il faut éviter les définitions circulaires :

```
parent(X,Y):-child(Y,X).
child(X,Y):-parent(Y,X).
```

– Faire attention à la redondance des solutions :
  – en s'assurant que chaque cas est traité tout au plus par une seule règle. Par exemple :

    ```
    minimum(X,Y,X) :- X =< Y.
    ```

    est changé par

    ```
    minimum(X,Y,Y) :- Y < X.
    minimum(X,Y,Y) :- Y <= X.
    ```

  – en évitant d'avoir trop de cas spéciaux.
  – Pour mieux comprendre les principes de la récursion avec Prolog, le meilleur moyen est de faire des exercices. Ces exercices sont divisés en trois catégories, soit les problèmes récursifs avec opérations arithmétiques (série B), les problèmes récursifs pour le traitement de listes (série C), et des problèmes récursifs généraux (série D).

4.2 Exercices série B

1. Définir la procédure `entre(+I,+J,?K)` qui indique que `K` est un entier entre `I` et `J` inclusivement.
2. Définir une procédure pour évaluer de façon récursive sans accumulateur la fonction $f(n) = n!$. Proposer aussi une procédure récursive avec accumulateur.
3. Définir une procédure `triangle(+N,?F)` qui retourne `F`, la somme des entiers de `0` jusqu'à `N`. Proposer une version récursive sans accumulateur et une version récursive avec accumulateur.
4. Définir une procédure `puissance(+X,+N,?V)` qui retourne `V`, soit `X` élevé à la `N`. Proposer une version récursive sans accumulateur et une version récursive avec accumulateur.
5. Mettez en oeuvre avec Prolog la fonction de Ackerman définie comme étant (avec x et y des entiers positifs) :

$$A(x,y) \equiv \begin{cases} y+1, \text{ si } x = 0 \\ A(x-1,1), \text{ si } y = 0 \\ A(x-1, A(x,y-1)), \text{ sinon} \end{cases}$$

6. Écrire un programme qui effectue la conversion d'unités métriques, sachant que :

1 mille = 1.609 kilomètre	1 kilomètre = 100 décamètres
1 décamètre = 10 mètres	1 mètre = 10 décimètres
1 décimètre = 10 centimètres	1 centimètre = 10 millimètres
1 millimètre = 0.03936 pouce	1 pouce = 1/12 pied
1 pied = 1/3 verge	

   Débutez par écrire les faits représentant ces relations. Ensuite, définir un prédicat qui permettra de trouver le facteur correspondant à la conversion en se promenant dans les faits. **Faites attention aux boucles !** Enfin, définir `converti(N,U1,U2,Conv)`, qui convertit le nombre `N` exprimé dans l'unité `U1` en unité `U2` pour retourner `Conv`.

4.3 Exercices série C

1. Redéfinissez les prédicats prédéfinis suivants pour les listes en écrivant vos propres fonctions récursives :

   `append/3`
   `reverse/2`
   `length/2`

2. (a) En utilisant la procédure suivante, qu'arrive-t-il si le deuxième argument est une variable ? Comment devrait-on représenter le prédicat ?

   ```
   memberA(E,[E|Ys]).
   memberA(E,[Y|Ys]) :- memberA(E,Ys).
   ```

   (b) Comparativement à a), qu'arrive-t-il si on utilise la procédure suivante ?

   ```
   memberB(X,[X|Xs]).
   memberB(X,[Y|Ys]) :- X \= Y, memberB(X,Ys).
   ```

   (c) Quel est le rôle de la procédure suivante :

   ```
   membersA([X|Xs],Ys) :- member(X,Ys), membersA(Xs,Ys).
   membersA([],Ys).
   ```

   (d) Quelle différence y a-t-il entre la procédure précédente et la suivante ?

   ```
   membersB([X|Xs],Ys):-remove(X,Ys,Ys1),membersB(Xs,Ys1).
   membersB([],Ys).
   ```

3. Soit la procédure suivante :

   ```
   removeA(X,[X|Xs],Xs).
   removeA(X,[Y|Ys],[Y|Zs]) :- removeA(X,Ys,Zs).
   ```

   Qu'arrive-t-il si la deuxième ligne est modifée pour obtenir la procédure suivante :

   ```
   removeB(X,[X|Xs],Xs).
   removeB(X,[Y|Ys],[Y|Zs]) :- X\=Y,removeB(X,Ys,Zs).
   ```

4. Soit la procédure suivante :

```
% removeall(+Liste,+X,-NoX):-la liste NoX est le
% résultat de l'élimination de toutes les occurrences
% de X de la liste Liste.
removeallA([X|Xs],X,Z) :- removeallA(Xs,X,Z).
removeallA([X|Xs],Y,[X|Zs]) :- X \= Y, removeallA(Xs,Y,Zs).
removeallA([],Y,[]).
```

   (a) Qu'arrive-t-il si on omet la condition `X \= Y` ?

   (b) Qu'arrive-t-il si la deuxième ligne est modifiée pour obtenir la procédure suivante :

```
removeallC([X|Xs],X,Z) :- removeallC(Xs,X,Z).
removeallC([X|Xs],X,Xs).
removeallC([],Y,[]).
```

5. Définir les procédures suivantes en utilisant la récursion.

```
dernier(?X,+L) :- donne X, le dernier élément de la liste L
adjacent(?X,?Y,?L) :- X est l'élément adjacent
                      et en avant de Y dans la liste L
gardePremiers(+N,+Xs,?Ys) :- Ys est la liste des N premiers
                             éléments de Xs
skipPremiers(+N,+Ys,?Zs) :- Zs est la liste des éléments
                            de Ys moins les N premiers
nth(+N,+Xs,?Y) :- Y est le Nième élément de la liste Xs
```

6. Soit la procédure suivante :

```
substitutionA(X,Y,[X|Zs],[Y|Ys]):-substitutionA(X,Y,Zs,Ys).
substitutionA(X,Y,[Z|Zs],[Z|Ys]):-substitutionA(X,Y,Zs,Ys).
substitutionA(X,Y,[],[]).
```

   (a) À quoi sert-elle ?

   (b) Qu'arrive-t-il si la deuxième ligne est modifiée pour obtenir la procédure suivante :

```
substitutionB(X,Y,[X|Zs],[Y|Ys]):-substitutionB(X,Y,Zs,Ys).
substitutionB(X,Y,[Z|Zs],[Z|Ys]):-X\=Z,substitutionB(X,Y,Zs,Ys).
substitutionB(X,Y,[],[]).
```

7. Soit la procédure `noDouble(+Xs,?Ys)` qui élimine tous les éléments en double dans `Xs` pour retourner `Ys`.

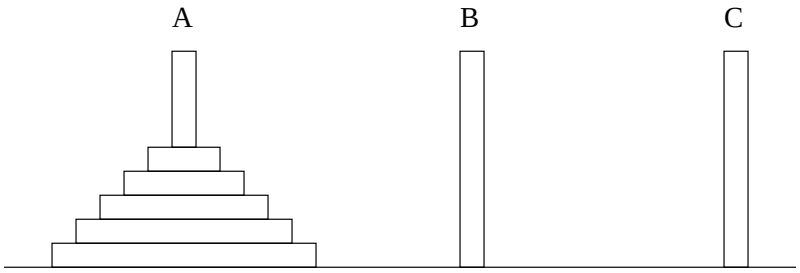
   (a) Est-ce que la procédure suivante est valide ?

```
noDoubles1([X|Xs],Ys):-member(X,Xs),noDoubles1(Xs,Ys).
noDoubles1([X|Xs],[X|Ys]):-noDoubles1(Xs,Ys).
noDoubles1([],[]).
```

   (b) Concevoir la procédure `nonmember(X,L)`, retournant vrai si `X` n'est pas un élément de `L` (ne pas utiliser `not`).

(c) Corriger le problème trouvé en a) en utilisant `nonmember/2`.

8. Définir la procédure `ndReverse(+Xs,?Ys)` où `Ys` est l'inverse de `Xs` avec les éléments dupliqués éliminés.
9. Définir les procédures pour l'union, l'intersection et la différence ensemblistes, les ensembles étant représentés par des listes (n'ayant pas d'éléments dupliqués).
10. Le problème de la tour de Hanoi consiste à déplacer les disques de la tige A vers la tige B sans placer un disque plus large sur un disque moins large. Les règles du jeu sont :
    – seulement un disque peut être déplacé à la fois ;
    – aucun disque plus gros ne peut être placé sur un disque plus petit ;
    – initialement tous les disques se trouvent sur le bâton A selon la contrainte précédente.



Le truc pour arriver à la bonne séquence de déplacement consiste à tout d'abord déplacer $n-1$ disques de A vers C en utilisant B comme tige transitoire ; ensuite déplacer le plus gros disque de A (soit le disque n) vers B, pour finalement déplacer les $n-1$ disques de C vers B en utilisant A comme tige transitoire. Essayez avec $n=2$ pour commencer, et ensuite comprendre la récursion.

Définir `hanoi(+N,+A,+B,+C,-M)`, où `M` est la séquence de déplacements pour résoudre le problème des tours de Hanoi avec `N` disques et trois bâtons, `A`, `B` et `C`.

L'important ici n'est pas de passer trop de temps pour trouver le truc pour la mise en oeuvre de la solution, mais bien voir comment la récursion permet de résoudre des problèmes complexes.

4.4 Exercices série D

1. Définir les procédures suivantes :

```
sommeEntiers(+M,+N,?R) :- R est la somme des entiers de M à N.
absTout(+Xs,?Ys):- Ys est la liste des valeurs absolues de la liste Xs.
sommeListe(+L,?Somme) :- Somme est la somme des entiers de la liste L.
maximum(+L,?N) :- N est le maximum de la liste L.
minimum(+L,?N) :- N est le minimum de la liste L.
```

2. Définir une procédure pour évaluer le produit scalaire de deux vecteurs représentés par des listes. Proposer une version récursive sans accumulateur et une version récursive avec accumulateur.
3. Définir une procédure pour générer une liste L des entiers entre `M` et `N`. Proposer une version récursive sans accumulateur et une version récursive avec accumulateur.

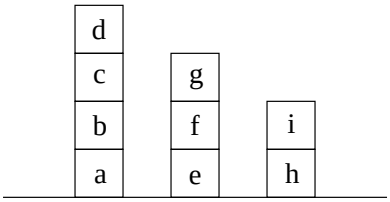
4. Définir une procédure pour trier les éléments d'une liste en :
   (a) Définissant une procédure permutant tous les éléments d'une liste ;
   (b) Définissant une procédure vérifiant si les éléments d'une liste sont en ordre ;
   (c) Triant une liste par permutation.
5. Définir une procédure pour trier les éléments d'une liste `[X|Xs]` à partir du principe d'insertion suivant : on trie la queue `Xs` et ensuite on insère `X` dans la liste triée de `Xs`, de façon à retourner la liste `[X|Xs]` triée.

5 Résolution de problèmes par déduction logique

Cette section fait davantage référence aux chapitres 6 à 10 du volume de cours. L'objectif ici est que vous soyez en mesure d'exploiter un environnement comme Prolog pour résoudre des problèmes variés, exprimés dans des termes courants et non sous forme d'une description mathématique ou procédurale. Cette habilité est nécessaire pour l'ingénierie des connaissances. Le choix de la représentation est aussi un point très important. Encore une fois, le meilleur moyen d'y arriver est de faire des exercices.

5.1 Exercices série E

1. Expliquer en utilisant les concepts de Prolog les différentes significations du mot ET dans les phrases suivantes :
   (a) Une récolte de nourriture est tout ce qui est de la famille des végétaux et qui constitue une nourriture pour les hommes.
   (b) Le président du département est un professeur et il est responsable de la direction des réunions.
   (c) Pour éteindre un feu, vos options sont d'appliquer des ralentisseurs de feu et de le laisser mourir.
   (d) Tom et Sue sont des managers.
   (e) Tom et Sue sont copains.
2. Représenter en Prolog les faits décrivant les relations entre les blocs. Utiliser deux prédicats, un disant que le bloc est immédiatement sur un autre, et un établissant qu'un bloc est immédiatement à la gauche d'un autre et sur la table. Pour le deuxième prédicat, donner seulement les faits pour les blocs au bas de la pile.



(a) Sur quel bloc le bloc a se trouve-t-il ?

(b) uels blocs sont sur d'autres blocs ?

(c) Quels blocs se trouvent sur des blocs qui sont immédiatement à la gauche d'autres blocs ?

(d) Définir un nouveau prédicat auDessus qui est vrai quand un bloc est n'importe où dans la pile au-dessus d'un autre bloc. Définir un nouveau prédicat pileDeGauche qui indique qu'un bloc est dans une pile immédiatement à la gauche de la pile d'un autre bloc. Quel bloc est sur un autre bloc et n'est pas à gauche d'une autre pile ?

3. À partir des énoncés de l'exercice 9.4, p. 294 du volume de cours, écrire le programme Prolog qui permet de résoudre la requête formulée à l'exercice 9.5, partie a).

4. À partir de l'arbre généalogique de la figure 7.4, p. 215 du volume de cours, écrire les prédicats demandés à l'exercice 7.6 (sauf la définition concernant le mième cousin), et trouver les réponses aux requêtes demandées. Les prédicats de faits permis sont `pere/2`, `mere/2`, `homme/1`, `femme/1`, `parent/2` et `maries/2`. Vérifier vos relations et vos réponses avec l'arbre généalogique. En ne définissant pas aucune autre relation, demander aussi à votre programme :
   - Quels sont les parents de Anne ?
   - William est le grand-père de qui ?
   - Quels sont les neveux de Andrew ?
   - Quels sont les belle-soeurs de qui ?

5. En utilisant le programme développé à la question 4, formuler une requête qui répond à la question 9.6, p. 295 du volume de cours.

6. Représenter les faits suivants en Prolog (les prédicats de relations binaires, i.e., deux arguments, sont recommandés). Représenter ce que les phrases signifient, non pas ce qu'ils disent ; chacune des phrases ne doivent pas utiliser des noms de prédicat différents ou des arguments différents.
   - Une plaque chauffante Acme a un cordon et un corps.
   - Une partie du corps de la plaque chauffante Acme est l'élément chauffant.
   - L'élément chauffant est en métal.
   - Une autre partie du corps de la plaque chauffante Acme est son couvert.
   - Le couvert possède une poignée.
   - Du plastique est toujours utilisé pour les poignées.
   - Une des choses qui constitue un cordon est le fil électrique.
   - Le métal est le matériel constituant le fil électrique.
   - Une partie du cordon est l'isolant.
   - L'isolant est en fibre.

   (a) Vérifier vos relations en posant les questions suivantes : quelles sont les pièces qui contiennent du métal ? Qu'est-ce qui fait partie du corps de la plaque chauffante ?

   (b) Définir une règle permettant de dire qu'une pièce d'une autre pièce de la plaque chauffante est aussi une pièce de la plaque chauffante.

(c) Définir une règle permettant de dire par exemple que si l'élément chauffant est fait de métal, alors la plaque chauffante est aussi constituée de métal. Effectuer les requêtes suivantes :
   - Quels éléments contiennent du plastique ?
   - Quels éléments contiennent du métal et des fibres ?
   - Quels éléments ne contiennent pas de fibre ?

7. Représenter les prochaines phrases en des faits et des règles Prolog (représenter leur signification, et non ce qu'elles signifient littérairement).
   - Une VW Rabbit est une VW.
   - La voiture de Tom est une VW Rabbit.
   - La voiture de Dick est une VW Rabbit.
   - Une VW a un système électrique.
   - Une partie du système électrique est l'alternateur.
   - L'alternateur est défectueux sur toutes les VW.

   Écrire les règles d'inférence en Prolog qui vont permettre d'arriver à la conclusion que la voiture de Tom ou de Dick est défectueuse.

5.2 Puzzles ou énigmes logiques

Trois amis sont arrivés respectivement premier, deuxième et troisième lors d'une compétition. Chacun a un prénom different, aime un sport différent, et est de nationalité différente. Michel aime le basketball, et fut meilleur que l'américain. Simon, l'israëlien, fut meilleur que le joueur de tennis. Le joueur de cricket arriva premier.

Qui est l'autralien ? Quel sport pratique Richard ?

Un puzzle logique consiste en des faits concernant un nombre restreint d'objets ayant différents attributs. Un minimum de faits est donné concernant les objets et les attributs afin de fournir une façon unique d'assigner les attributs aux objets. À partir d'un programme logique, il est possible de résoudre ces puzzles en instanciant les valeurs d'une structure de données appropriée, et en extractant ces valeurs.

Pour y arriver, le principe de méta-variable est utilisé. Une méta-variable est une variable que le programme tente d'instancier en la rendant vraie. Ceci permet à une variable d'apparaître comme but dans le corps d'une clause ou d'une conjonction de buts. La variable doit être instanciée avant d'être utilisée comme but, mais elle peut contenir d'autres variables qui seront instanciées lors de l'évaluation comme but de la méta-variable.

Prenons l'exemple suivant :

```
testMeta :- initMeta(Meta), Meta.

initMeta(member(X,[1,2,3])).
```

Le prédicat `testMeta` commence par initialiser la méta-variable `Meta` par la clause `initMeta`, la variable `Meta` est alors instanciée au but `member(X, [1,2,3])`. Pour que le prédicat `testMeta` soit vrai, la règle demande ensuite que le but de `Meta` soit aussi vrai. Esuite est évalué `member(X, [1,2,3])` et trois réponses sont fournies :

```
testMeta
No.1 : yes
No.2 : yes
No.3 : yes
```

Les instanciations de la variable `X` dans le but de `Meta` ne sont pas retournées ici, car cette variable est propre à la méta-variable. Nous allons voir un peu plus loin comment faire pour lier des variables propres à des méta-variables pour obtenir des solutions plus détaillées.

Revenons au puzzle logique. Pour résoudre un puzzle logique, nous utilisons la procédure suivante : initialiser la structure (la représentation) du problème, les indices et les requêtes, pour ensuite appliquer la procédure pour résoudre le puzzle. La requête initiale pour trouver la solution au puzzle est `puzzleLogique(competition,S)`. Le premier argument correspond au nom du puzzle car nous utilisons le même programme pour les différentes énigmes demandées dans les exercices.

```
puzzleLogique(Nom,Solution) :-
  structure(Nom,Structure),                  % Défini le problème
  indices(Nom,Structure,Indices),            % Défini les indices
  requetes(Nom,Structure,Requetes,Solution), % Défini les requêtes
  resoudPuzzle(Indices,Requetes,Solution).   % Résoud le puzzle
```

Des méta-variables `Indices` et `Requetes` sont utilisées respectivement pour décrire les indices fournis et les informations demandées sous forme de conjonction de buts à satisfaire par le mécanisme d'inférence de Prolog. Ces méta-variables sont évaluées par le prédicat `resoudPuzzle/3`.

```
resoudPuzzle(Indices,Requetes,Solution):-resoud(Indices),resoud(Requetes).

resoud([Indice|Indices]) :- Indice, resoud(Indices).
resoud([]).
```

Pour qu'une telle procédure fonctionne, nous devons trouver un moyen de faire des liens entre les variables propres aux méta-variables `Indices` et `Requetes`. Pour y arriver, nous définissons une structure de données qui sert de représentation pour le problème et qui vient définir un contexte pour la résolution des buts dans les méta-variables. Pour notre problème, la structure choisie est une liste d'amis, avec leurs attributs.

```
structure(competition,[ami(N1,C1,S1),ami(N2,C2,S2),ami(N3,C3,S3)]).
% Chaque personne a trois attributs: nom (N), nationalité (C) et sport
% pratiqué (S).
% L'ordre définit le résultat de la compétition.
```

On peut ensuite coder les indices sous la forme d'une conjonction de buts. Chaque indice est codé comme un fait concernant la structure de données, liée à la structure de données du problème (Amis). Noter que des termes entre parenthèses sont évalués comme une expression. Voici les indices formulés (remarquez l'importance des variables ayant le même nom, décrivant les contraintes formulées dans les indices) :

```
( futMeilleur(Ami1Indice1,Ami2Indice1,Amis),                  % Indice 1
  nom(Ami1Indice1,michel),
  sport(Ami1Indice1,basketball),
  nationalite(Ami2Indice1,americain))

  ( futMeilleur(Ami1Indice2,Ami2Indice2,Amis),                % Indice 2
    nom(Ami1Indice2,simon),nationalite(Ami1Indice2,israelien),
    sport(Ami2Indice2,tennis))

  ( premier(Amis,AmiIndice3),sport(AmiIndice3,cricket))       % Indice 3
```

Ces indices exprimés selon le format de la clause `indices/3` ont la forme suivante :

```
indices(competition,Amis, [
  ( futMeilleur(Ami1Indice1,Ami2Indice1,Amis),                % Indice 1
    nom(Ami1Indice1,michel),
    sport(Ami1Indice1,basketball),
    nationalite(Ami2Indice1,americain)),
  ( futMeilleur(Ami1Indice2,Ami2Indice2,Amis),                % Indice 2
    nom(Ami1Indice2,simon),nationalite(Ami1Indice2,israelien),
    sport(Ami2Indice2,tennis)),
  ( premier(Amis,AmiIndice3),sport(AmiIndice3,cricket)) ]).   % Indice 3
```

Il est alors possible de formuler les prédicats pour valider les indices.

```
futMeilleur(A,B,[A,B,C]).
futMeilleur(A,C,[A,B,C]).
futMeilleur(B,C,[A,B,C]).

nom(ami(A,B,C),A).
nationalite(ami(A,B,C),B).
```

```
sport(ami(A,B,C),C).

premier([X|Xs],X).
```

Ensuite, il suffit de faire la même chose pour les requêtes. Noter que la variable `Solution` sera instanciée par la clause `requetes/4` à partir des variables internes propres à la méta-variable `Requetes` une fois cette dernière évaluée.

```
requetes(competition, Amis,
 [ member(Q1,Amis),
   nom(Q1,Nom),
   nationalite(Q1,australien),            % Requête 1
   member(Q2,Amis),
   nom(Q2,richard),
   sport(Q2,Sport)                        % Requête 2
 ],
   [['Autralien est', Nom], ['Richard joue au ', Sport]]
 ).
```

5.3 Exercices série E (suite)

8. Reproduire le programme décrit précédemment et donner les réponses au puzzle décrit précédemment.
9. Résoudre l'énigme donnée à l'exercice 6.12, page 182 du volume de cours.
10. Cinq maisons de couleurs différentes sont habitées par des hommes de nationalités différentes, ayant chacun son animal favori, sa boisson préférée et sa marque de cigarettes. Sachant que :
    - l'anglais vit dans la maison rouge ;
    - le chien appartient à l'espagnol ;
    - on boit du café dans la maison verte ;
    - l'ukranien boit du thé ;
    - la maison verte est à droite de la maison ivoire ;
    - le fumeur de la marque ExportA élève des escargots ;
    - on fume des Matinée dans la maison jaune ;
    - on boit du lait dans la maison du milieu ;
    - le norvégien vit dans la première maison à gauche ;
    - l'homme qui fume des Players vit dans la maison à côté de l'homme avec un renard ;
    - on fume des Matinée dans la maison à côté du cheval ;
    - le fumeur de duMaurier boit du jus d'orange ;
    - le japonais fume des Menthol ;
    - le norvégien habite à côté de la maison bleu.

    À qui appartient le zèbre ? Qui boit du vin ?

A Notions importantes sur LPA Prolog

Avant de commencer à utiliser LPA Prolog, il est recommandé de lire certaines parties du manuel LPA Win-Prolog 4.1 User Guide. Les parties importantes sont les chapitres 2 et 3 (pages 25 à 36), les chapitres 5 et 6 (pages 40 à 47) et le chapitre 9. Notez à la page 30 l'indication d'utiliser la touche `<space bar>` pour obtenir l'ensemble des solutions à une requête. Une autre caractéristique importante est le *Box Model Debugger* (page 72) , qui vous permet de garder dans une fenêtre la trace des appels avec l'instanciation des variables. Les commentaires dans le texte sont indiqués entre les symboles `/*` et `*/`. De plus, `%` place le restant de la ligne en commentaire.

B Solutions aux exercices

B.1 Série A

1.

```
% Conversion de Farenheit en Celsius: f_en_c(+F,?C)
f_en_c(F,C) :- C is ((5 / 9) * (F + 40)) - 40.

% Conversion de Celsius en Farenheit: c_en_f(+C,?F)
c_en_f(C,F) :- F is ((9 / 5) * (C + 40)) - 40.
```

2.

```
% max(+X,+Y,+Z,-M) retourne M, le maximum entre X,Y,Z
max(X,Y,Z,X) :- X>=Y,X>=Z.
max(X,Y,Z,Y) :- Y>X,Y>=Z.
max(X,Y,Z,Z) :- Z>X,Z>Y.

% max2(X,Y,Z,M) retourne M,
% le maximum entre X,Y,Z en employant seulement >
max2(X,Y,Z,X) :- X>Y,X>Z.
max2(X,Y,Z,Y) :- Y>X,Y>Z.
max2(X,Y,Z,Z) :- Z>X,Z>Y.
max2(X,X,Z,X) :- X>Z.
max2(X,Y,Y,Y) :- Y>X.
max2(Z,Y,Z,Z) :- Z>Y.
max2(X,X,X,X).
```

```
% Avec la solution suivante, qu'arrive-t-il
% comme situation si les X=Y=Z? Trois solutions
/*
  max2(X,Y,Z,X) :- X>Y,X>Z.
  max2(X,Y,Z,Y) :- Y>X,Y>Z.
  max2(X,Y,Z,Z) :- Z>X,Z>Y.
  max2(X,X,Z,X).
  max2(X,Y,Y,Y).
  max2(Z,Y,Z,Z).
*/
```

3.

```
/* En employant seulement append, construire les predicats suivants: */
memberA(X,L) :-append(L1,[X|L2],L).
dernier(X,L) :-append(L1,[X],L).
enleveUn(X,L,M) :-append(L1,[X|L2],L),append(L1,L2,M).
adjacent(X,Y,Zs) :-append(L1,[X,Y|Ys],Zs).
avant(X,Y,L) :-append(L1,[Y|L2],L),append(L3,[X|L4],L1).
```

4.

```
/* P est le palindrome de L qui a deux fois sa longueur */
palindrome(L,P) :- append(L,R,P), reverse(L,R).

/* Inverser append et reverse occasionne des problèmes
   si on demande de trouver L selon un P donné: la première
   solution sera donnée, mais une erreur suiviendra si toutes
   les solutions sont demandées. */
```

5.

```
/* Rotation gauche d'une liste */
rotationGauche([X|Xs],R) :- append(Xs,[X],R).

/* Rotation droite d'une liste */
rotationDroite(L,R): -
   reverse(L,Z),rotationGauche(Z,G),reverse(G,R).

/* Autre solution: */
rotationDroite(L,R) :-
   reverse(L,[X|Xs]),reverse(Xs,Ys),append([X],Ys,R).

/* Attention boucle sans fin:
   rotationDroite3(L,R):-rotationGauche(R,L).  */
```

6.

```
enseigne(Prof,Jour) :- prof(Cours,Prof),jourCours(Cours,Jour).

% occupe(?Prof,+Heure)
occupe(Prof,Heure) :-
   prof(Cours,Prof), heureDebut(Cours,Debut),
   heureFin(Cours,Fin), Debut =< Heure, Heure =< Fin.

% Suppose qu'ils ont besoin de la journee
nePeutRencontrer(Prof1,Prof2) :-
   prof(Cours1,Prof1), prof(Cours2,Prof2), Prof1 \= Prof2,
   jourCours(Cours1,Jour),jourCours(Cours2,Jour).

% conflitCedule(+Heure,?Place,?Cours1,?Cours2)
conflitCedule(Heure,Place,Cours1,Cours2) :-
   jourCours(Cours1,Jour), jourCours(Cours2,Jour),
   Cours1 \= Cours2, edifice(Cours1,Place),
   edifice(Cours2,Place), local(Cours1,Local),
   local(Cours2,Local), heureDebut(Cours1,Debut1),
   heureFin(Cours1,Fin1), Debut1=<Heure,Heure=<Fin1,
   heureDebut(Cours2,Debut2), heureFin(Cours2,Fin2),
   Debut2=<Heure,Heure=<Fin2.

% Variante
conflitCedule(Heure,Place,Cours1,Cours2) :-
   jourCours(Cours1,Jour), jourCours(Cours2,Jour),
   Cours1 \= Cours2, edifice(Cours1,Place),
   edifice(Cours2,Place), local(Cours1,Local),
   local(Cours2,Local), prof(Cours1,Prof1),
   prof(Cours2,Prof2), occupe(Prof1,Heure),
   occupe(Prof2,Heure).
```

7.

```
Trapped Spypoint: a(_39134,_39136)
0024 H UNIFY a(0,1)
0024 H EXIT  a(0,1)
0024 H UNIFY a(0,2)
0024 H EXIT  a(0,2)
0024 H UNIFY a(2,1)
0024 H EXIT  a(2,1)
0024 H UNIFY a(M,N)
0025 B CALL  b(P,Q)
0025 H UNIFY b(3,1)
0025 H EXIT  b(3,1)
```

```
0025 B EXIT  b(P,Q)
             P = 3, Q = 1
0026 B CALL  b(Q,P)
             Q = 1, P = 3
0026 H FAIL  b(1,2)
             2 \= 3
0026 B FAIL  b(Q,P)
             Q = 1, P = 3
0025 B REDO  b(P,Q)
             P = 3, Q = 1
0025 H UNIFY b(2,1)
```

```
0025 H EXIT  b(2,1)
0025 B EXIT  b(P,Q)
             P = 2, Q = 1
0027 B CALL  b(Q,P)
             Q = 1, P = 2
0027 H UNIFY b(1,2)
0027 H DONE  b(1,2)
0027 B DONE  b(Q,P)
             Q = 1, P = 2
0028 B CALL  M is P + 1
             P = 2
0028 B DONE  M is P + 1
             M = 3, P = 2
0029 B CALL  N is Q + 1
             Q = 1
0029 B DONE  N is Q + 1
             N = 2, Q = 1
0024 H EXIT  a(M,N)
             M = 3, N = 2
0024 H REDO  a(M,N)
             M = 3, N = 2
0025 B REDO  b(P,Q)
             P = 2, Q = 1
0025 H UNIFY b(1,2)
0025 H DONE  b(1,2)
0025 B DONE  b(P,Q)
             P = 1, Q = 2
0030 B CALL  b(Q,P)
             Q = 2, P = 1
0030 H UNIFY b(2,1)
0030 H DONE  b(2,1)
0030 B DONE  b(Q,P)
             Q = 2, P = 1
0031 B CALL  M is P + 1
             P = 1
0031 B DONE  M is P + 1
             M = 2, P = 1
0032 B CALL  N is Q + 1
             Q = 2
0032 B DONE  N is Q + 1
             N = 3, Q = 2
0024 H DONE  a(M,N)
             M = 2, N = 3

%(Note: cette trace fut obtenue
% avec MacProlog)
a(X,Y),not(c(X)),d(X,Y)
*a(X,Y)=>X=0,Y=1
not(c(0))
c(0)=>yes
*a(X,Y)=>X=0,Y=2
not(c(0))
c(0)=>yes
*a(X,Y)=>X=2,Y=1
not(c(2))
c(2)=>no
d(2,1):-2>1,1>1=>no
a(M,N):-b(P,Q),b(Q,P),
    M is P+1,N is Q+1.
*b(P,Q)=>P=3,Q=1
b(1,3)=>no
*b(P,Q)=>P=2,Q=1
b(1,2)=>yes
M is 3
N is 2
not(c(3)) =>yes
d(3,2):-3>1,2>1
=>X=3,Y=2
%Reprise au point de retour
% en arrière (backtrack)
b(P,Q)=>P=1,Q=2
b(2,1)=>yes
M is 2
N is 3
not(c(2)) =>yes
d(2,3):-2>1,3>1.
=>X=2,Y=3
```

B.2 Série B

1.

```
entre(I,J,I) :- I =< J.
entre(I,J,K) :- I < J, I1 is I + 1, entre(I1,J,K).
```

2.

```
/* Version récursive */
fact(N,F) :- N > 0, N1 is N-1, fact(N1,F1), F is N*F1.
fact(0,1).

/* Version récursive avec accumulateur, de type croissante */
factAcc(N,F) :- factAcc(0,N,1,F).

factAcc(I,N,T,F) :- I < N, I1 is I+1, T1 is T*I1, factAcc(I1,N,T1,F).
factAcc(N,N,F,F).

/* Un autre version récursive avec accumulateur,
   de type décroissante */
factAcc2(N,F) :- factAcc2(N,1,F).

factAcc2(N,T,F) :-  N > 0, T1 is T*N, N1 is N-1, factAcc2(N1,T1,F).
factAcc2(0,F,F).
```

3.

```
/* Récursion sans accumulateur */
triangle(N,F) :- N > 0, N1 is N-1,
                 triangle(N1,F1), F is N+F1.
triangle(0,0).

/* Récursion avec accumulateur */
triangleAcc(N,F) :- triangleAcc(N,0,F).
triangleAcc(N,A,F) :- N > 0, A1 is N + A,
   N1 is N-1, triangleAcc(N1,A1,F).
triangleAcc(0,A,A).
```

4.

```
/* Récursion sans accumulateur */
puissance(X,N,V) :- N > 0, N1 is N-1,
   puissance(X,N1,V1), V is X * V1.
puissance(X,0,1).

/* Récursion avec accumulateur */
puissanceAcc(X,N,V) :- puissanceAcc(X,N,1,V).
```

```
puissanceAcc(X,N,A,V) :- N>0, A1 is X*A,
    N1 is N-1, puissanceAcc(X,N1,A1,V).
puissanceAcc(X,0,A,A).
```

5.

```
ackermann(0,N,A) :- A is N+1.
ackermann(M,0,A) :- M1 is M-1, M1>=0, ackermann(M1,1,A).
ackermann(M,N,A) :- M1 is M-1, M1>=0, N1 is N-1, N1>=0,
   ackermann(M,N1,A1),ackermann(M1,A1,A).
```

6.

```
%facteur(U1,U2,F):-1*U1=F*U2.
facteur(mille,kilometre,1.609).
facteur(kilometre,decametre,100).
facteur(decametre,metre,10).
facteur(metre,decimetre,10).
facteur(décimetre,centimetre,10).
facteur(centimetre,millimetre,10).
facteur(millimetre,pouce,0.03936).
facteur(pouce,pied,1/12).
facteur(pied,verge,1/3).

%conversion(U1,U2,F):-1*U1=F*U2.
conversion(X,X,1).
conversion(X,Y,F) :- checkUp(X,Y,F).
conversion(X,Y,F) :- checkDown(X,Y,F).

%checkDown(U1,U2,F) :- recherche le facteur vers le bas de la chaine
checkDown(X,Y,F) :- facteur(X,Y,F).
checkDown(X,Y,F) :- facteur(X,Z,F1), checkDown(Z,Y,F2), F is F1*F2.

%checkUp(U1,U2,F) :- recherche le facteur vers le haut de la chaine
checkUp(X,Y,F) :- facteur(Y,X,F1), F is 1/F1.
checkUp(X,Y,F) :- facteur(Z,X,F1), checkUp(Z,Y,F2), F is F2/F1.

%converti(+N,+U1,+U2,?Conv) :- N en U1 correspond a Conv en unites U2.
converti(N,U1,U2,Conv) :- conversion(U1,U2,F), Conv is N*F.
```

B.3 Série C

1.

```
% appendA(Xs,Ys,XsYs) :-
%    XsYs est le resultat de la concaténation de Xs et Ys.
appendA([],Ys,Ys).
appendA([X|Xs],Ys,[X|Zs]) :- appendA(Xs,Ys,Zs).


% reverse(+Liste,-Etsil):-Etsil est la liste inverse de Liste
% a) Recursif
reverseR([],[]).
reverseR([X|Xs],Zs) :- reverseR(Xs,Ys), append(Ys,[X],Zs).

% b) Iteratif ou plutot de la base vers le haut
reverseAcc(Xs,Ys) :- reverseAcc(Xs,[],Ys).
reverseAcc([X|Xs],Acc,Ys) :- reverseAcc(Xs,[X|Acc],Ys).
reverseAcc([],Ys,Ys).

% length(Xs,N) :- Xs est une liste de longueur N
% a) Recursive croissante
lengthA([X|Xs],N) :- lengthA(Xs,N1), N is N1+1.
lengthA([],0).

% b) Iterative
lengthAcc(Xs,N) :- lengthAcc(Xs,0,N).
lengthAcc([X|Xs],I,N) :- I1 is I+1, lengthAcc(Xs,I1,N).
lengthAcc([],I,I).
```

2.

   (a) Génère plusieurs solutions avec des variables non instanciées ; `memberA(?E,+L)`

   (b) Retourne seulement le premier élément trouvé à une requête `memberB(X,[1,2,3])`.

   (c) Vérifie si les éléments du premier argument sont des éléments du second argument. Les deux arguments doivent être initialisés. Si le second argument n'est pas initialisé, le prédicat donne une suite de liste avec des variables non-initialisées. Si le premier élément n'est pas initialisé, le prédicat member redonne toujours le même résultat après chaque appel.

   (d) Le premier élément peut ne pas être instancié : on vient prendre les éléments de Ys un par un et on les élimine après chaque appel.

3. Seulement le premier élément trouvé est remplacé.

4.

   (a) On obtient alors une variante permettant d'enlever des nombres variables d'occurrence de X. Si toutes les solutions possibles sont demandées, il va se produire des

problèmes. L'interpréteur utilise alors la troisième clause dans des cas déjà traités par la deuxième, de sorte qu'on obtient des solutions où toutes les occurrences ne sont pas enlevées.

(b) Il n'y a alors plus de récursion pour éliminer l'élément dans le restant de la liste une fois qu'un élément à enlever a été retrouvé dans la liste.

5.

```
dernier(X,[X]).
dernier(X,[Y|Ys]) :- dernier(X,Ys).

adjacent(X,Y,[X,Y|Zs]).
adjacent(X,Y,[Z|Zs]) :- adjacent(X,Y,Zs).

gardePremiers(1,[Y|Ys],[Y]).
gardePremiers(N,[Y|Ys],[Y|Zs]) :-
  N \= 1, N1 is N - 1, gardePremiers(N1,Ys,Zs).

skipPremiers(1,[Y|Ys],Ys).
skipPremiers(N,[Y|Ys],Zs) :- N\=1, N1 is N-1, skipPremiers(N1,Ys,Zs).

nth(1,[X|Xs],X).
nth(N,[X|Xs],Y) :- N \= 1, N1 is N - 1, nth(N1,Xs,Y).
```

6.

(a) `substitution(?X,?Y,+L1,?L2):-` substitue X par Y dans la liste L1 pour donner L2.

`substitutionA/4` retourne toutes les combinaisons possibles substituant X à Y (de 0 au nombre d'éléments X dans la liste L1).

(b) Tous les éléments X sont remplacés par Y.

7.

(a) Elle retourne plusieurs solutions car on ne peut examiner ce qui fut placé dans la liste dans les appels précédents (lorsqu'on se déplace en profondeur dans l'arbre).

(b)

```
nonmember(X,[Y|Ys]) :- X \= Y, nonmember(X,Ys).
nonmember(X,[]).
```

(c)

```
noDoubles([X|Xs],Ys) :- member(X,Xs),noDoubles(Xs,Ys).
noDoubles([X|Xs],[X|Ys]):- nonmember(X,Xs), noDoubles(Xs,Ys).
noDoubles([],[]).
/* Vient déterminer si un élément va apparaître,
   au lieu de déterminer s'il a déjà apparu /*
```

Par contre, lorsque member(X,Xs) est utilisé avec des éléments similaires, on obtient plusieurs solutions :

```
member(a,[a,a,a]) => 3 yes
member(a,[a,a]) => 2 yes
member(a,[a]) => 1 yes
```

Pour corriger, il faut modifier `member` pour retourner vrai qu'une seule fois même si l'élément se retrouve plusieurs fois dans la liste. Une solution serait :

```
membre(X,[Y|Ys]) :- X \= Y, membre(X,Ys).
membre(X,[Y|Ys]) :- X = Y.

noDoublesf([X|Xs],Ys):-membre(X,Xs),noDoublesf(Xs,Ys).
noDoublesf([X|Xs],[X|Ys]):-nonmember(X,Xs), noDoublesf(Xs,Ys).
noDoublesf([],[]).

?-noDoublesf ([a, a, b, a], D)
No.1 : D = [b, a]
No more solutions
```

8.

```
/* bottom-up reverse: avantage de construire de bottom-up */

ndReverse(Xs,Ys) :- ndReverse(Xs,[],Ys).

ndReverse([X|Xs],Revs,Ys) :- member(X,Revs), ndReverse(Xs,Revs,Ys).
ndReverse([X|Xs],Revs,Ys) :- nonmember(X,Revs),
   ndReverse(Xs,[X|Revs],Ys).
ndReverse([],Ys,Ys).

/* Autre solution avec résultats multiples */
ndReverse2(Xs,Ys) :- reverse(Xs,R),noDoubles(R,Ys).

ndReverse3([],[]).
ndReverse3([X|Xs],R) :- nonmember(X,Xs), ndReverse3(Xs,R1),
   append(R1,[X],R).
ndReverse3([X|Xs],R) :- member(X,Xs), ndReverse3(Xs,R).
```

9.

```
/* union(+S1,+S2,?S):- S est l'union des listes
   S1 et S2, sans dupliqués. */
union([X|Xs],Ys,S) :- member(X,Ys), union(Xs,Ys,S).
union([X|Xs],Ys,[X|Zs]) :- not member(X,Ys), union(Xs,Ys,Zs).
union([],S2,S2).
```

```
/* intersection(+S1,+S2,?S):- S est l'intersection
   des listes S1 et S2, sans dupliqués. */
intersection([X|Xs],Ys,[X|Zs]) :- member(X,Ys), intersection(Xs,Ys,Zs).
intersection([X|Xs],Ys,S) :- not member(X,Ys), intersection(Xs,Ys,S).
intersection([],S2,[]).

/* difference(+S1,+S2,?S):- S est la différence des
   listes S1 et S2, sans dupliqués. */
difference([X|Xs],Ys,Zs) :- member(X,Ys), difference(Xs,Ys,Zs).
difference([X|Xs],Ys,[X|Zs]) :- not member(X,Ys), difference(Xs,Ys,Zs).
difference([],S2,[]).
```

10.

```
/* hanoi(+N,+A,+B,+C,-M) :- M est la séquence de
  déplacements pour résoudre le problème des tours
  de Hanoi avec N disques et trois bâtons, A, B et
  C. :-hanoi(3,a,b,c,M).
*/

hanoi(1,A,B,C,[vers(A,B)]).
hanoi(N,A,B,C,M) :- N > 1, N1 is N-1,
  hanoi(N1,A,C,B,Ms1),
  hanoi(N1,C,B,A,Ms2),
  append(Ms1,[vers(A,B)|Ms2],M).

/* #10 autre solution (semblable) qui représente
   les déplacements par des sous-listes */
% hanoi(+Nb,+Debut,+Fin,+Inter,-Mouvements).
hanoi2(1,D,F,I,[D, F]).
hanoi2(N,D,F,I,[M1, M2, M3]) :- N > 1, N1 is N - 1,
  hanoi2(N1,D,I,F,M1),
  hanoi2(1, D,F,I,M2),
  hanoi2(N1,I,F,D,M3).
```

B.4 Série D

1.

```
sommeEntiers(M,N,R) :- M < N, N1 is N-1,
   sommeEntiers(M,N1,R1), R is R1 + N.
sommeEntiers(M,M,M).

absTout([X|Xs],[Y|Ys]) :- Y is abs(X), absTout(Xs,Ys).
absTout([],[]).

/* Récursive */
sommeListe([X|Xs],Somme) :- sommeListe(Xs,Somme1),
   Somme is X + Somme1.
sommeListe([],0).

/*Récursion avec accumulateur */
sommeListeAcc(L,Somme) :- sommeListeAcc(L,0,Somme).

sommeListeAcc([X|Xs],Temp,Somme) :- Temp1 is Temp+X,
   sommeListeAcc(Xs,Temp1,Somme).
sommeListeAcc([],Somme,Somme).

maximum([X|Xs],M) :- maximum(Xs,X,M).

maximum([X|Xs],Y,M) :- X =< Y, maximum(Xs,Y,M).
maximum([X|Xs],Y,M) :- X > Y, maximum(Xs,X,M).
maximum([],M,M).

minimum([X|Xs],M) :- minimum(Xs,X,M).

minimum([X|Xs],Y,M) :- X < Y, minimum(Xs,X,M).
minimum([X|Xs],Y,M) :- X >= Y, minimum(Xs,Y,M).
minimum([],M,M).
```

2.

```
/* produitScalaire(+Xs,+Ys,?R) :- R est le produit
   vectoriel des vecteurs Xs et Ys représentés par
   des listes. */
produitScalaire([X|Xs],[Y|Ys],R) :- produitScalaire(Xs,Ys,XYsR),
  R is X * Y + XYsR.
produitScalaire([],[],0).
```

```
/* Version récursive avec accumulateur */
produitScalaireAcc(Xs,Ys,R) :- produitScalaireAcc(Xs,Ys,0,R).

produitScalaireAcc([X|Xs],[Y|Ys],Temp,R) :- Temp1 is X*Y+Temp,
   produitScalaireAcc(Xs,Ys,Temp1,R).
produitScalaireAcc([],[],R,R).
```

3.

```
/* interval(+M,+N,?L) :- L est la liste des entiers
   entre M et N */
/* Récursion sans accumulateur */
interval(M,N,[M|Ns]) :- M < N, M1 is M+1, interval(M1,N,Ns).
interval(N,N,[N]).

/* Récursion avec accumulateur. Noter qu'on doit diminuer
   au lieu d'augmenter, sinon la liste est inversée */
intervalAcc(M,N,Ns):-intervalAcc(M,N,[],Ns).

intervalAcc(M,N,L,Ns) :- M < N, N1 is N-1, intervalAcc(M,N1,[N|L],Ns).
intervalAcc(M,M,L,[M|L]).
```

4.

```
/* triPermutation(Xs,Ys) :- Ys est la liste triée
   par permutation de la liste Xs.*/

triPermutation(Xs,Ys) :- permutation(Xs,Ys), ordre(Ys).

permutation(Xs,[Z|Zs]) :- remove(Z,Xs,Ys), permutation(Ys,Zs).
permutation([],[]).

ordre([X]).
ordre([X,Y|Ys]) :- X =< Y, ordre([Y|Ys]).
```

5.

```
/* Pourquoi donne une seule solution si element
   en double, contrairement a triPermutation? */

triInsertion([X|Xs],Ys) :- triInsertion(Xs,Zs), insertion(X,Zs,Ys).
triInsertion([],[]).

insertion(X,[],[X]).
insertion(X,[Y|Ys],[Y|Zs]) :- X > Y, insertion(X,Ys,Zs).
insertion(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.
```

B.5 Série E

1.

   (a) Ici, le ET associe deux caractéristiques d'une chose pour en donner une définition, comme la relation `récolte(végétaux, nourriture)`.

   (b) Le ET associe encore deux caractéristiques mais ne permet pas de donner une définition de la chose. Celles-ci ne sont que des spécifications de cette chose (le président du département). On pourrait établir une relation donnant les caractéristiques du président de département par les relations suivantes :
   `carac_président(professeur).` et `carac_président(dirige_réunions).`

   (c) Ici, le ET s'interprète davantage comme une union. Pour éteindre une feu, on peut ou bien appliquer des ralentisseurs de feu, ou bien le laisser mourir, ou bien les deux. C'est un peu le sens de la phrase qui vient énoncer les options pour éteindre un feu. On pourrait représenter cette affirmations par les relations suivantes :
   `éteindre_feu(ralentisseurs).` et `éteindre_feu(laisser_mourir).`

   (d) Le ET vient associer une caractéristique à deux identités. Nous pourrions utiliser deux faits `manager(sue).` et `manager(tom).`

   (e) Le ET vient établir la relation entre deux identités, et non, comme l'exemple précédent, associer une caractéristique à celles-ci. Les deux identités doivent être présentes dans ce cas. On pourrait avoir des relations en Prolog comme :
   `copains(tom,sue).` et comme :
   `sontCopains(X,Y) :- copains(Y,X). sontCopains(X,Y):- copains(X,Y).`
   pour les définir en évitant les définitions circulaires.

2.

```
% sur(X,Y) :- X est au-dessus de Y
% blocs immédiatement sur
sur(b,a).
sur(c,b).
sur(d,c).
sur(f,e).
sur(g,f).
sur(i,h).
% aGauche(X,Y) :- X est immédiatement à la gauche de Y (bas de la pile)
aGauche(a,e).
aGauche(e,h).
```

   (a) `?-sur(a,X).`

   (b) `?-sur(X,Y).`

   (c) `?-sur(X,Y), aGauche(Y,Z).`

   (d)
```
% auDessus(X,Y) :- X est au-dessus de Y
auDessus(X,Y) :- sur(X,Y).
auDessus(X,Y) :- sur(X,Z),auDessus(Z,Y).
```

```
% pileGauche(X,Y) :- X est a gauche Y
pileDeGauche(X,Y) :- aGauche(X,Y).
pileDeGauche(X,Y) :- auDessus(X,Z), aGauche(Z,Y).
pileDeGauche(X,Y) :- aGauche(X,Z), auDessus(Y,Z).
pileDeGauche(X,Y) :- auDessus(X,Z), aGauche(Z,W), auDessus(Y,W).

% Note: si on avait utilisé les clauses suivantes, on aurait
% obtenu toutes les combinaisons possibles, et donc des
% répétions de solutions
pileDeGauche2(X,Y) :- aGauche(X,Y).
pileDeGauche2(X,Y) :- aGauche(X,Z), auDessus(Y,Z).
pileDeGauche2(X,Y) :- auDessus(X,Z), pileDeGauche2(Z,Y).

pileDeGauche(X) :- pileDeGauche(X,Y).
% Pour éviter des problèmes d'instanciation avec le not (car Y
% ne serait pas instancié)

?-auDessus(X, Y), not pileDeGauche(X).
```

3.

```
mammal(horse).
mammal(cow).
mammal(pig).

horse(bluebeard).
horse(X) :- father(Y,X), horse(Y).

father(bluebeard,charlie).

?-horse(X).
```

4.

```
% Relation pere(X,Y) signifie X est le père de Y
pere(george,elizabeth). pere(george,margaret). pere(spencer,diana).
pere(philip,charles). pere(philip,anne). pere(philip,andrew).
pere(philip,edward). pere(charles,william). pere(charles,harry).
pere(mark,peter). pere(mark,zara).
pere(andrew,beatrice). pere(andrew,eugenie).

% Relation mere(X,Y) signifie X est la mère de Y
mere(mum,elizabeth). mere(mum,margaret). mere(kydd,diana).
mere(elizabeth,charles). mere(elizabeth,anne). mere(elizabeth,andrew).
mere(elizabeth,edward). mere(diana,william). mere(diana,harry).
mere(anne,peter). mere(anne,zara). mere(sarah,beatrice).
mere(sarah,eugenie).
```

```
% Relation homme(X) signifie X est un homme
homme(george). homme(spencer). homme(philip).
homme(charles). homme(mark). homme(andrew).
homme(edward). homme(william). homme(harry).
homme(peter).

% Relation homme(X) signifie X est une femme
femme(mum). femme(kydd). femme(elizabeth).
femme(margaret). femme(diana). femme(anne).
femme(sarah). femme(zara). femme(beatrice).
femme(eugenie).

% Relation marie(X,Y) signifiant que X est marié à Y
maries(george,mum). maries(spencer,kydd). maries(philip,elizabeth).
maries(charles,diana). maries(mark,anne). maries(andrew,sarah).

% marie(X,Y):-marie(Y,X). % Génère une boucle sans fin
% Pour éviter la redondance, utiliser:
sontMaries(X,Y) :- maries(X,Y).
sontMaries(X,Y) :- maries(Y,X).


% Relations de base
% Relation parent(X,Y) signifie que X est un parent de Y
parent(X,Y) :- pere(X,Y).
parent(X,Y) :- mere(X,Y).

% Relation fils(X,Y) signifie X est le fils de Y
fils(X,Y) :- parent(Y,X), homme(X).

% Relation fille(X,Y) signifie X est la fille de Y
fille(X,Y) :- parent(Y,X), femme(X).

% Formulation des relations demandées
% Relation petitEnfant(X,Y) signifie X est le petit-enfant de Y
petitEnfant(X,Y) :- parent(Z,X), parent(Y,Z).

% Relation arriereGrandParent(X,Y) signifie
% X est l'arrière-grand-parent de Y
arriereGrandParent(X,Y) :- parent(X,Z), parent(Z,W), parent(W,Y).

% Relation frere(X,Y) signifie que X est le frère de Y.
frere(X,Y) :- fils(X,Z), homme(Z), parent(Z,Y), Y\=X.
% frere2(X,Y) :- parent(Parent,X), parent(Parent,Y),
```

```
%    homme(X), X\=Y. %répétition

% Relation soeur(X,Y) signifie que X est la soeur de Y.
soeur(X,Y) :- fille(X,Z), femme(Z), parent(Z,Y),Y\=X.

% Notez que si on omet homme(Z) et femme(Z), frere et soeur
% donnent deux fois les mêmes réponses car on est le fils ou
% la fille de notre mère et de notre père, etc.

% Relation oncle(Oncle,X) signifie que Oncle est l'oncle de X
oncle(Oncle,X) :- frere(Oncle,Parent), parent(Parent,X).

% Relation tante(Tante,X) signifie que Tante est la tante de X
tante(Tante,X) :- soeur(Tante,Parent), parent(Parent,X).

% beauFrere(X,Y) signifie X est le beau-frère de Y
% beauFrere par mariage
beauFrere(X,Y) :- frere(X,Z), sontMaries(Z,Y).
% beauFrere par affiliation
beauFrere(X,Y) :- homme(X), sontMaries(X,Z), frere(Z,Y).
beauFrere(X,Y) :- homme(X), sontMaries(X,Z), soeur(Z,Y).
beauFrere(X,Y)  :-homme(X),sontMaries(X,Z),frere(Z,W),sontMaries(W,Y).
beauFrere(X,Y) :- homme(X),sontMaries(X,Z),soeur(Z,W),sontMaries(W,Y).

% belle-soeur(X,Y) signifie X est la belle-soeur de Y
belleSoeur(X,Y) :- soeur(X,Z), sontMaries(Z,Y).
belleSoeur(X,Y) :- femme(X), sontMaries(X,Z), frere(Y,Z).
belleSoeur(X,Y) :- femme(X), sontMaries(X,Z), soeur(Y,Z).
belleSoeur(X,Y) :- femme(X),sontMaries(X,Z),frere(Z,W),sontMaries(W,Y).
belleSoeur(X,Y) :- femme(X),sontMaries(X,Z),soeur(Z,W),sontMaries(W,Y).

% cousin(X,Y) signifie que X est le cousin de Y
cousin(X,Y) :- parent(Z,X), parent(W,Y), frere(Z,W).
cousin(X,Y) :- parent(Z,X), parent(W,Y), soeur(Z,W).
```

```
?-petitEnfant(X,elizabeth).
?-beauFrere(X,diana).
?-arriereGrandParent(X,zara).

?-parent(Z,anne).
?-petitEnfant(william,Z).
?-oncle(andrew,Z),homme(Z).
?-belleSoeur(X,Y).
```

5.

```
?-pere(Pere,Homme), pere(Pere,MonFils), MonFils\=Homme,
    fils(MonFils,I), I\=Pere, not frere(I), not soeur(I).

frere(I):-frere(X,I).
soeur(I):-soeur(X,I).
% Faire not frere(X,I) occasionne des problèmes lorsque I
% n'est pas instancié.
```

6.

```
% pieceDe(X,Y):- X est une pièce directe de Y.
pieceDe(cordon, plaquechauff).
pieceDe(corps, plaquechauff).
pieceDe(elementchauff, corps).
pieceDe(couvert, corps).
pieceDe(poignée, couvert).
pieceDe(fil, cordon).
pieceDe(isolant, cordon).

% On peut aussi établir une relation concernant le
% constituant des pièces (en quoi elles sont faites):

% constitueDe(X,Y):- X est constitué de Y.
constitueDe(elementchauff, metal).
constitueDe(poignee, plastique).
constitueDe(fil, metal).
constitueDe(isolant, fibre).
```

(a) Nous pouvons voir une première grande relation : `pieceDe`. En effet, on énumère une série de faits établissant ce qui fait partie de quoi. Nous pourrions donc avoir les faits suivants énumérant les pièces de la plaque chauffant de type Acme, en y référant directement comme des pièces de la plaque chauffante :
Les requêtes à formuler sont :

```
?-constitueDe(X, metal)
No.1 : X = elementchauff
No.2 : X = fil
No more solutions

?-pieceDe(X, corps)
No.1 : X = elementchauff
No.2 : X = couvert
No more solutions
```

(b)

```
piece(X,Y):-pieceDe(X,Y).
piece(X,Y):-pieceDe(X,W),piece(W,Y).

?-piece(X, corps)
No.1 : X = elementchauff
No.2 : X = couvert
No.3 : X = poignée
No more solutions
```

(c)

```
constitue(X,Y):-constitueDe(X,Y).
constitue(X,Y):-pieceDe(Z,X),constitue(Z,Y).

?-constitue(X, plastique)
No.1 : X = poignee
No.2 : X = plaquechauff
No.3 : X = corps
No.4 : X = couvert
No more solutions

?-constitue(X, metal), constitue(X, fibre)
No.1 : X = plaquechauff
No.2 : X = plaquechauff
No.3 : X = cordon
No more solutions
```

Pour la troisième requête, la réponse devrait être le corps, l'élément chauffant, le couvert et la poignée. Ici, il faut porter une attention particulière à la manière de formuler la question. En effet, si on formule tout simplement la question suivante :

```
?- not constitue(X, fibre)
```

Une erreur de contrôle survient. On demande en fait s'il n'existe pas une variable X qui est constituée de fibre : autrement dit, on demande de trouver une substitution pour X qui est invalide. Prolog n'est alors pas en mesure d'y arriver, et une erreur de contrôle est signalée. Si on pose plutôt la requête suivante :

```
?- constitue(X, Y),not (Y=fibre)

No.1 : X = elementchauff, Y = metal
No.2 : X = poignee, Y = plastique
No.3 : X = fil, Y = metal
No.4 : X = plaquechauff, Y = metal
No.5 : X = plaquechauff, Y = metal
No.6 : X = plaquechauff, Y = plastique
No.7 : X = corps, Y = metal
No.8 : X = corps, Y = plastique
```

```
No.9 : X = couvert, Y = plastique
No.10 : X = cordon, Y = metal
No more solutions
```

Les réponses se répètent. Le programme trouve les pièces directement constituées de matériel différent des fibres, et ensuite donne les autres pièces constituées de ces premières pièces. Puisque par exemple le corps est constitué de l'élément chauffant et du couvert, et que chacun d'eux sont fait de matériel différent des fibres, alors la pièce corps sera citée deux fois. Il faut donc ajouter une autre règle permettant de retrouver seulement les nouvelles réponses. Pour ce faire, on peut s'organiser pour que le programme analyse les pièces qu'une fois seulement. Ceci peut se réaliser par :

```
% choixPiece(X):-sélection de la pièce X.
choixPiece(X):-pieceDe(X,Y).
```

La relation permet de prendre chaque pièce une par une, et examiner si elle ou une de ses pièces constituantes possède du matériel fibreux. Le résultat est :

```
?-choixPiece(X), not constitue(X, fibre)
No.1 : X = corps
No.2 : X = elementchauff
No.3 : X = couvert
No.4 : X = poignee
No.5 : X = fil
No more solutions
```

7.

```
%voitureType(Sorte,Marque) :- Sorte est une voiture de type Marque.
voitureType(vw_rabbit,vw).

%voitureDe(Sorte,Proprio) :- Sorte est la voiture de Proprio
voitureDe(vw_rabbit,tom).
voitureDe(vw_rabbit,dick).

%partieDe(X,Y) :- X est une partie de Y.
partieDe(systeme_electrique,vw).
partieDe(alternateur,systeme_electrique).

partie(X,Y) :- partieDe(X,Y).
partie(X,Y) :- partieDe(X,Z),partie(Z,Y).

%defectueux(Piece,Marque) :- Piece est défectueuse sur les Marques.
defectueux(alternateur,vw).

voitureDefectueuse(Proprio):-
  defectueux(Piece,Marque),
```

```
  partie(Piece,Marque),  % Inutile pour cette requête
  voitureType(Sorte,Marque),
  voitureDe(Sorte,Proprio).

?-voitureDefectueuse(X).
No.1 : X = tom
No.2 : X = dick
No more solutions
```

8. Vous devrez tester le programme pour le savoir. Un conseil : il est préférable de taper vous même le code pour bien comprendre les mécanismes utilisés.

9. Du petit texte, on peut déduire que
   - John n'est pas un programmeur (il ne peut pas s'emprunter de l'argent à lui-même) ;
   - John n'est pas un manager (car un manager ne peut emprunter de l'argent, à cause de son épouse) ;
   - Smith n'est pas un manager (car il n'est pas marié).

   On a choisi d'utiliser le prédicat emploi au lieu de marie, owe, borrow, etc. parce qu'il est pertinent à ce que l'on cherche à établir (il faut faire abstraction des détails et s'occuper du but seulement).

```
structure(jobs,[job(X,programmeur),job(Y,manager),job(Z,ingenieur)]).

indices(jobs,Structure,[
( member(M0,Structure), not emploi(M0,programmeur),
not emploi(M0,manager), nom(M0,john)),
( member(M1,Structure), not emploi(M1,manager), nom(M1,smith)),
( member(M2,Structure), nom(M2,clark))]).
    % Il faut savoir que Clark existe

requetes(jobs, Structure,
[ member(Q1,Structure), nom(Q1,Name1), emploi(Q1,programmeur),
    member(Q2,Structure), nom(Q2,Name2), emploi(Q2,manager),
member(Q3,Structure), nom(Q3,Name3), emploi(Q3,ingenieur)
],
[ [Name1,' est programmeur'],
[Name2,' est manager'],
[Name3,' est ingenieur']]).

nom(job(P,J),P).
emploi(job(P,J),J).
```

10.

```
structure(zebre,[maison(C1,N1,P1,D1,S1),
                 maison(C2,N2,P2,D2,S2),
                 maison(C3,N3,P3,D3,S3),
                 maison(C4,N4,P4,D4,S4),
                 maison(C5,N5,P5,D5,S5)]).
% Maisons de gauche à droite, avec une couleur C, la nationalité de
% l'occupant N, l'animal domestique P, la boisson B et la marque de
% cigarette S.

% Devrait placer M9, M8 et M16 en premier pour contraindre la
% recherche member est important: vient prendre un élément dans
% Structure

indices(zebre,Structure,[
(member(M0,Structure),nation(M0,anglais),couleur(M0,rouge)),
(member(M1,Structure),animal(M1,chien),nation(M1,espagnol)),
(member(M2,Structure),boire(M2,cafe),couleur(M2,verte)),
(member(M3,Structure),boire(M3,the),nation(M3,ukranien)),
(member(M4,Structure),member(M5,Structure),
  couleur(M4,verte),couleur(M5,ivoire),aDroite(M4,M5,Structure)),
(member(M6,Structure),cigarette(M6,exportA),animal(M6,escargots)),
(member(M7,Structure),cigarette(M7,matinee),couleur(M7,jaune)),
(member(M8,Structure),boire(M8,lait),milieu(M8,Structure)),
(member(M9,Structure),nation(M9,norvegien),
  premiereGauche(M9,Structure)),
(member(M10,Structure),member(M11,Structure),
 cigarette(M10,players),animal(M11,renard),aCote(M10,M11,Structure)),
(member(M12,Structure),member(M13,Structure),
  cigarette(M12,matinee),animal(M13,cheval),aCote(M12,M13,Structure)),
(member(M14,Structure),cigarette(M14,duMaurier),boire(M14,orange)),
(member(M15,Structure),cigarette(M15,menthol),nation(M15,japonais)),
(member(M16,Structure),member(M17,Structure),
  nation(M16,norvegien),couleur(M17,bleu),aCote(M16,M17,Structure))]).

requetes(zebre,Structure,[
  member(Q1,Structure),
  animal(Q1,zebre),
  nation(Q1,Nom1),
  member(Q2,Structure),
  boire(Q2,vin),
  nation(Q2,Nom2)],
[['Le proprietaire du zebre est ',Nom1],['  ',Nom2,' boit du vin']]).

nation(maison(C,N,P,D,S),N).
```

```
couleur(maison(C,N,P,D,S),C).
animal(maison(C,N,P,D,S),P).
boire(maison(C,N,P,D,S),D).
cigarette(maison(C,N,P,D,S),S).

aDroite(M1,M2,[M2|M]) :- member(M1,M).
aDroite(M1,M2,[M|Ms]) :- M\==M2, aDroite(M1,M2,Ms).

milieu(M,[M1,M2,M,M3,M4]).

premiereGauche(M,[M|Ms]).

aCote(M1,M2,[M1,M2|Ms]).
aCote(M1,M2,[M2,M1|Ms]).
```

Références

[1] Clocksin, W.F. et Mellish, C.S. *Programming in Prolog*. Springer-Verlag, 4e édition, 281 pages, 1994.

[2] Russell, S. et Norvig, P. *Artificial Intelligence : A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall, 912 pages, 1995.

[3] Sterling, L. et Shapiro, E. *The art of Prolog : Advanced Programming Techniques*. MIT Press, deuxième édition, 560 pages, 1994.