Module Algorithmes et Raisonnement
S8 - AR - 2A - 2016-2017
BE2 : exercices sur les Listes

I Introduction

• Selon votre groupe, l'enseignant fera la transition avec le BE précédent.
• Les exercices suivants vous permettent la manipulation des listes.
• Le cours prochain fera un bilan sur les listes et autres structures composées de Prolog.

Programme de ce BE
- Explications et exercices sur les listes - Exercice sur les recettes. - Travail à rendre = Recettes avec gestion des quantités - Bonus : Livres, circuits et équipes - Délai : 4 semaines.

II Listes

• Vous pouvez en apprendre davantage sur les termes Prolog (Annexes, section IX.2) et sur les listes en Annexes (section IX.5).

• En Prolog, une liste permet de représenter une collection d'informations (même de différents types ; le mélange est possible) dont on ne connait pas d'avance le nombre .

Pour faire simple, les exemples ci-dessus désignent des listes diverses :

Exemple	
1. []	*la liste vide*
2. [jean]	*liste contenant une constante (dite atomique = non décomposable)*
3. [jean, dupont, 12.5]	*3 éléments (2 constantes littérales et une constante numérique)*
4. [12.5, marie, "Ecole Centrale", 126, personne('Pierre Louis', 'Dominique'), [1,2,3,5,7,11]]	

➙ Cette dernière liste contient :
- des constantes : *12.5, marie, "Ecole Centrale", 126,*
- un arbre : *personne('Pierre Louis', 'Dominique')*
- une liste : *[1,2,3,5,7,11].*

Remarquez que la chaîne de caractères *"Ecole Centrale"* est représentée (en interne) par la liste de ses caractères ;
et que *'Pierre Louis'* n'est pas une chaîne de caractères mais une constante littérale non décomposable où les apostrophes permettent d'utiliser un espace entre "Pierre" et "Louis" et ne pas considérer **Pierre** comme une variable.

☞ Pour créer une liste, il suffit d'écrire des termes entre '[' et ']' en les séparant par des virgules.
➙ Par exemple : **Liste = [sur,10, personnes, 1, comprend, les, nombres, binaires, l_autre, non].**

En résumé :

- Une liste Prolog est une collection d'éléments placés entre '[]'.
- Une liste existe dès lors qu'elle est écrite (pas de déclaration).
- Une liste peut contenir n'importe quel terme (y compris une autre liste).
- La liste vide se note par []
- **[]**, **[jean]**, **[X, 12]** etc. sont des exemples de liste.

• Il est ici utile de rappeler ce qu'est un terme Prolog.
Un Terme est en quelque sorte au sommnt de la hiérarchie des types des données en Prolog (comme dans certains langages objets *propres* (cf. ADA, SmallTalk) et le type *Object* au sommet).

A savoir : Terme Prolog

Un **Terme** Prolog peut être :

- Une **constante** :
  - une constante numérique : **12**, **13.14**, etc
  - une constate littérale : **marie**, **'jean jacques'** (*jean* est la même chose que *'jean'*, mais pour 'jean-jacques' ou 'jean jacques', il faut les apostrophes à cause de l'espace ou du '-').
  - une constante chaîne de caractères : **"Ecole Centrale de Lyon"**. Notons que cette chaine sera représentée par la liste de ses caractères.

    ☞ Quand faut-il utiliser "Ecole Centrale" ou 'Ecole Centrale' ?

    ➙ Réponse : si vous avez besoin de manipuler les caractères un par un, utiliser "Ecole Centrale", sinon 'Ecole Centrale'.

- Un **terme composé** :
  - une arbre : **eleve(u,25)**, **date(21, mai, 2012)**,
    personne(jean, adresse(3, 'rue machine', lyon), profession(chef, 'EDF'))
  - une liste (comme les exemple donnés ci-dessus).
    En fait, une liste est un arbre dont les feuilles sont les éléments de la liste et dont tout autre noeud est un opérateur binaire particulier noté '.' (dot).

    ➙ Ce qui veut dire que la liste [a,12,'anne marie'] est en fait l'arbre binaire
    .(a, .(12, .('anne marie', [])))

    ➥ Essayer *write([a,12,'anne marie']).* puis *display([a,12,'anne marie']).*

- Une **variable** : commence par une lettre majuscule (ou par '_') et peut prendre un terme quelconque comme valeur.

  ➙ Par exemple **X1**, **MonAmieLaRose**, **_Une_variable_dite_anonyme**.

  ☞ On a dit plus haut que *jean* et *'jean'* était identiques. Ce n'est pas le cas pour *Jean* et *'Jean'* car *Jean* est avant tout une variable (et *'Jean'* une constante littérale).

• Donc, la règle pour placer les apostrophes serait : s'ils entourent un seul mot (sans espace) commençant par une lettre minuscule, alors on peut les enlever. Dans tous les autres cas, il faut les laisser.

☞ Un fait, une règle, un prédicat ... et même un programme Prolog est un terme !

II.1 Manipulation des listes

• Nous avons vu plus haut : une liste est en fait une structure binaire et récursive :
• Une liste est
- Soit vide (noté par []) ;
- Soit un élément (la tête) suivie d'une liste (qui est le reste).

➙ Cette définition correspond à celle des structures simplement chainées dans les langages impératifs (C/C++/Java/ADA/...). Elle correspond également à la liste dans les langages fonctionnels (CAML, Lisp, etc).

• Pour pouvoir utiliser et parcourir une liste L en Prolog, on distingue donc 2 cas :

1) La liste L est-elle vide ? : pour cela, on teste légalité L=[] .
2) (sinon), La liste L contient-elle au moins un élément ? :
pour cela, on teste légalité L= [Premier_element | Le_reste_qui_est_une_liste].

La barre verticale sépare la tête de la liste de son reste ; le reste est une liste.

• Ajoutons deux remarques :
➙ 1- Le fait de tester l'égalité de **L** avec **'[]'** impose à L d'être une liste. Les crochets sont donc la marque de fabrique des listes (et uniquement des listes).

➙ 2- Nous savons distinguer une liste vide d'une liste avec au moins un élément.

Nous pouvons aussi écrire **L=[Premier, Seconde | Reste]** pour savoir si L contient au moins deux éléments (lesquels sont unifiés avec les variables *Premier* et *Seconde*).
Ce test peut être utile par exemple si vous avez besoin d'isoler les éléments d'une liste deux-à-deux pour tester une relation d'ordre sur les couples (deux éléments successifs) dans une liste (voir exemple *ordonnee* plus loin).

• On dispose également du prédéfini *length* permettant de connaitre le nombre d'éléments d'une liste.

• Il y a bien d'autres prédicats prédéfinis sur les listes (voir Doc. BProlog).

En savoir plus : Liste vs. Arbres.
Nous avons dit plus haut : une liste est en fait un arbre construit avec l'opérateur binaire '.'. ➙ Pourquoi a-t-on besoin d'une liste puisque on a les arbres (arbres binaires, ternaires, quaternaire, ...) ? Soit l'ensemble des élèves de l'Ecole (d'un nombre indéfini par avance). ➙ Une liste permet simplement d'énumérer ces élèves sans se préoccuper de leur nombre (variable d'année en année). Par contre, pour faire la même chose avec un arbre, • soit on doit utiliser un arbre 350-ernaire, sans disposer de la même structure d'une année sur l'autre (car une autre année, il en faut par exemple un 354-ernaire), • soit se rabattre sur une entité / type dynamique et potentiellement infinie (pouvant représenter vide) mais dont la structure est définie (par exemple : binaire). ➙ c'est exactement la justification des listes : on admet la contrainte qu'elle se décomposent en structure binaire (*Tete* suivie-de *Reste*) ; mais en contre partie, on peut représenter et manipuler facilement n'importe quelle séquence et de longueur quelconque (comme les listes chainées simples de C++).

III Exemples

III.1 Est-ce une liste

• Créer un programme composé du prédicat est_list/1 qui prend un argument et renvoie vrai si son argument est une liste Prolog.
Ce prédicat donné ci-dessous est composé de deux clauses.
➙ une **clause** = un fait ou une règle,
➙ un **prédicat** = un paquet de clauses avec le même symbole et même nombre d'arguments.

Exemple

```
%prédicat pour tester si L est une liste
   est_liste(L) :-   L=[].
   est_liste(L) :-
      L=[_Tete | Reste],        % la valeur de Tete ne nous intéresse pas.
      est_liste(Reste).         % Le reste aussi doit être une liste

Poser des questions (au besoin activer le mode trace) :
   ?- est_liste([a, 12, toto]).
   ?- est_liste(a).                              % ⇐ pas de '[]', pas de liste !
   ?- est_liste("Ecole Centrale").               % ⇐ une liste de codes ASCII
   ?- L = [a, b, c], est_liste(L), writeln(L), display(L) .      % comparer ces 2 affichages
   ?- writeln("Ecole Centrale"), display("Ecole Centrale").      % Comparer !
```

☞ **write** et **writeln** écrivent "normalement" la liste des caractères, **display** en donne la forme interne (dite *canonique*).

☞ Si vous voulez écrire le message (la chaine) "Ecole Centrale", utiliser "format" :
➙ X="Ecole Centrale", format("~s \n",[X]). ou
➙ format("Ecole Centrale \n"). ou format("~s \n",["Ecole Centrale"]).

• Le prédicat *est_liste* peut s'écrire (plus efficace, on l'appellera *est_liste1*) :

```
   est_liste1( [] ).
   est_liste1( [_Tete | Reste] ) :- est_liste1(Reste).
```

☞ Le fait de transférer, dans les entêtes des clauses, les unifications placées à l'intérieur des clauses (*L=[]* et L=*[_Tete | Reste]* de la 1e version) accélère l'exécution.

➙ Plus la tête d'une *clause* filtre les paramètres, mieux Prolog choisit la "bonne" clause à appliquer.

☞ Le prédéfini **is_list/1** (c-à-d. *is_list(.)*) équivalent à *est_liste* est proposé par BProlog.

III.2 Faire la somme

• Que fait le prédicat suivant (admettons que la liste contient des nombres) :

Exemple

```
   somme( [], 0 ).
   somme( [Tete | Reste], Total ) :-
            somme(Reste, Total1),
            Total is Total1 + Tete.      % 'is' force l'évaluation des expressions

Poser des questions (et expliquer les réponses) :
   ?- somme([12, 15, 23], S).
   ?- somme(a, S).       % No car 'a' n'est pas une liste : on ne rentre même pas dans ces clauses.
   ?- somme([X], S).     % Erreur : pour exécuter 'is', il faut des valeurs !
   ?- somme(X, S).       % Une réponse puis une erreur. Expliquer (avec trace).
   ?- somme([12, 15, a], S).      % Expliquer l'erreur.
```

N.B. : on peut éviter ces erreurs d'évaluation en utilisant le prédéfini *number/1* qui teste si son paramètre est un nombre.

Trois remarques :

1) Un prédicat NE PEUT PAS RENVOYER une valeur numérique.
➙ On ne peut pas écrire :
somme([]) :- 0 .
car l'appel d'un prédicat est évalué par *vrai* ou *faux* ; rien d'autre.
L'appel d'un prédicat réussit (vrai) ou échoue (faux).

2) Les arguments et paramètres d'appel des prédicats NE SONT PAS ÉVALUÉS.
➙ On ne peut pas écrire :
write(2+3).
et espérer obtenir 5 (testez-le).

Cependant et dans des conditions particulières (dans les calculs numériques utilisant l'évaluateur **is** dans les expressions arithmétiques), BProlog manipule des expressions fonctionnelles :
➙ Par exemple : *X is sqrt(1).* donne X=1.0
ou *X=5, Y is log(X+5) + sin(1).* donne Y = 3.14
→ X+5 est évalué et on utilise log(10).

☞ Le mot clef **is** provoque les évaluations.

3) Donc, si le prédicat *somme* ci-dessus est décliné en (remarquez l'addition dans les paramètres) :

```
sommeTer([], 0 ).
sommeTer([Tete | Reste], Tete + Total) :- sommeTer(Reste, Total).
```

Prolog n'évaluera pas l'addition *Tete + Total*.
Rares sont les Prologs (comme CLP(R)) qui le permettent sous une syntaxe particulière.

☞ Ajouter le prédicat *sommeTer/2* à votre code et essayer :
?- sommeTer([3,7,2],S)
et expliquer le résultat.

III.3 Refaire la somme avec des contraintes

• Observez cette version du prédicat *somme* appelée *somme1/2* (sur une liste de nombres) avec des contraintes :

Exemple

```
somme1( [], 0 ).
somme1( [Tete | Reste], Total ) :-
        Total #= Total_bis + Tete,
        somme1(Reste, Total_bis).
```

Elle souffre de certaines erreurs d'évaluation si la liste contient autre chose que des nombres mais en contre partie, on dispose d'une équation entre des nombres et leur somme !

Poser des questions (et expliquer les réponses) :

```
?- somme1([12, 15, 23], S).
?- somme1(a, S).          % ERREUR : 'a' n'est pas un nombre.
?- somme1([X], S).        % Pas d'erreur cette fois
?- somme1(X, S).          % Idem : S #=X+0 est une équation.
?- somme1(X, S), S=12.    % Ici, on connait la somme, reste à envisager les listes !
?- somme1([A,B], 12).     % Ici, on connait la somme ; la liste aura 2 éléments.
?- somme1([A,7], -12).    % On connait la somme ; la liste aura 2 éléments dont un connu.
```

➙ On ne pourra pas écrire l'expression "*Total is Total_bis + Tete*" à la place de (même endroit) "*Total #= Total_bis + Tete*" dans la 2e clause de somme1. Pourquoi ?

☞ Voir en annexes section IX.1 une version de somme avec accumulateur.

☞ Pour faciliter les écritures, le prédéfini **sum/1** permet de récupérer la somme des éléments d'une liste :

```
?- X is sum([1,5,2]).          % On aura X=8
?- X is sum([a,5,2]).          % Erreur d'évaluation ('a' n'est pas un nombre).
?- 15 is sum([A,5,2]).         % Erreur d'évaluation ('A' ne reçoit pas de valeur).
```

☞ Préférez l'utilisation des contraintes :

```
?- X #= sum([1,5,2]).          % On aura X=8
?- 15 #= sum([A,5,2]).         % On aura A=8 (c'est une équation)
```

Nota Bene : on a l'impression qu'avec *sum/1*, BProlog utilise une notation fonctionnelle. Il n'en est rien : la somme fournie est une EXPRESSION :
➙ *sum([1,5,2])* est une expression au même titre que la valeur de *1+5+2*. C'est #= qui provoque la résolution de l'équation.

☞ On utilise # avec d'autres opérateurs pour imposer une contrainte (vs. faire un test). On utilise ce symbole avec #=, #>, #<, #>=, #=<, ...

III.4 Liste ordonnée

• Le prédicat suivant (sur une liste quelconque) teste si une liste est ordonnée.
➙ Activer la trace pour comprendre :

Exemple

```
(R1)    ordonnee( [] ).
(R2)    ordonnee( [_Seul] ).
(R3)    ordonnee( [Prem, Second | Reste]) :-
            Prem @>= Second,                % comparaison de deux termes quelconques.
            ordonnee( [Second | Reste] ).   % on remet Second dans la liste.

Poser des questions (et expliquer les réponses) :
   ?- ordonnee([23, 18, 15, 3]).
   ?- ordonnee("zyxwvu").
   ?- ordonnee([c, b, a]).      % car c > b > a (codes ASCII)
   ?- ordonnee([b, 3, 0]).      % idem.
   ?- ordonnee([X]).
   ?- ordonnee(X).
   ?- ordonnee([115, 100, a]).
   ?- ordonnee([a, 10]).
```

☞ Voir en Annexes (section IX.3) plus de détails sur les comparateurs (@>, @=<, *&c.*).

Ordre entre les clauses :
Observez l'ordre des 3 clauses (c-à-d. deux faits et une règle) du prédicat *ordonnee/1*. On peut modifier cet ordre sans modification des résultats ; car les clauses sont ici mutuellement exclusives.

➽ En partant de la version *ordonnee/1*, changeons l'ordre par une simple permutation des 3 clauses (sans rien y changer d'autre) pour obtenir *ordonnee1/1* :

```
ordonnee1([Prem, Second | Reste]) :-
        Prem @>= Second,
        ordonnee1([Second | Reste]).
ordonnee1([_Seul]).
ordonnee1([]).
```

➽ Posez la requête *?- ordonnee1([b,c,a]).* vous aurez les mêmes réponses.

III.5 ordonne/1 revisité

• On peut proposer une version différente du prédicat *ordonnee/1* en partant de la définition suivante :

— Une liste vide ou une liste à un élément est ordonnée

— Une liste avec plus d'un élément ([Prem | Reste]) est ordonnée si son Reste est ordonnée et que son premier élément est plus grand (ou petit si ordre ascendant) que le premier de son Reste (prévoir que Reste peut être vide ou avec au moins 1 élément).

Ce qui donne le prédicat ordonnee2/1 suivant :

Exemple

```
ordonne2([]).                % la liste vide est ordonnée
ordonne2([_]).               % la liste réduite à un singleton est ordonnée
ordonne2([Prem | Reste]) :-
        ordonne2(Reste),             % Cas liste avec
        Reste=[Seconde|_],           % plus de 2 éléments
        Prem @>= Seconde.
```

Exemple de requêtes :
?- ordonne2([c,b,a]). succès
?- ordonne2([a,b,c]). échec car l'ordre doit être descendant.

• La version avec des contraintes de *ordonnee/1* permettra d'utiliser la contrainte $>=$ mais limitera le prédicat aux nombres.

```
ordonne3([]).
ordonne3([_]).
ordonne3([Prem | Reste]) :-
        ordonne3(Reste),
        Reste=[Seconde|_],
        Prem #>= Seconde.            % contrainte sur les nombres
```

➙ Cette version nous limite aux entiers mais permet par exemple d'avoir X dans :
ordonne3([10,X,2]). → $X \in 2..10$ pourquoi ?
ordonne3([3,2,X]). → $X \in -2^{28}..2$ pourquoi ?

III.6 Maximum d'une liste d'entiers

• Recopier et étudier le prédicat suivant et poser les questions suggérées :

Exemple

```
maximum( [Seul], Seul ).             % Le maximum d'un singleton est lui même
maximum( [Prem | Reste], Max) :-
        maximum(Reste, Max_bis),     % trouver le maximum du Reste (dans Max_bis)
        Max is max(Prem, Max_bis).   % choisir entre Prem et Max_bis
```

☞ Le prédéfini max(X,Y) est une expression et représente le maximum des deux entiers X et Y.
De ce fait, son utilisation impose de manipuler des entiers (pas les réels).

➥ Vous pouvez le tester par *?- X is max(3,4).*
➥ Testez-le aussi avec :

```
?- X is max(A,3).        % Erreur d'évaluation à cause de A. Mais
?- X #= max(A,3).        % On constate que max est une expression contrainte.
```

➙ Le domaine de X : 2... , le domaine de A : un entier.

Remarque

Pourquoi nous n'avons pas écrit une clause supplémentaire pour traiter le cas où la liste est vide (cas []) dans *maximum/2* ?

➙ Car par définition, le maximum d'une liste vide n'est pas défini.
En d'autres termes, si une requête demande le maximum d'une liste vide, l'appel échouera par l'absence même du cas prévu (rappelez-vous : en logique binaire, ce qui n'est pas défini est faux) : les deux clauses du prédicat maximum ne prévoient pas le cas de la liste vide.

➙ On peut néanmoins décider de traiter ce cas et et échouer en affichant éventuellement un message. Par exemple, on peut ajouter :

maximum([],_X) :- writeln('Erreur, Tant pis !'), fail.

ce qui veut dire que l'appel **?- maximum([], X).** affichera un message avant d'échouer.

☞ fail est un terme prédéfini en Prolog voulant dire échec. Il fait partie du mécanisme du contrôle de la résolution. Utilisez-le si vous devez explicitement provoquer un échec.

☞ On croise souvent un coupe choix (' !') avant un fail : on juge souvent que si la situation de fail explicite s'est présentée, il ne faut pas aller plus loin (d'où le cut).

➙ Voir aussi le mécanisme d'exception de BProlog (la Doc BProlog).

Version avec contraintes de maximum/2 :

On propose la version *maximum2/2* pour les nombres dans laquelle on utilise les contraintes :

Exemple

```
maximum2( [Seul], Seul ).                % Le maximum d'un singleton est lui même
maximum2( [Prem | Reste], Max) :-
        Max #= max(Prem, Max_bis),       % choisir entre Prem et Max_bis qui est
        maximum2(Reste, Max_bis).        % le maximum du Reste (de la liste)
```

Poser des questions et expliquer les réponses (activer la trace pour mieux comprendre les réponses) :

```
?- maximum2([23, 18, 15, 3], X).         % X=23
?- maximum2([X], 10).                    % X=10 (la liste contient un singleton)
?- maximum2(X, 12).                      % Une infinité de réponses (expliquer).
```

D'autres requêtes :

?- maximum2([A,B,C],3). % Quels sont A,B, C tels que leur maximum = 3 ?
➙ $A, B, C : -2^{28}..3$

?- maximum2([A,B,C,D], M), A=2, B #= A+1, C #> B, D #= A + B - C. % Une équation.
➙ $A = 2, B = 3, C : 4..2^{28}, D = -2^{28}..1, M = 4..2^{28}$

➽ La contrainte expression prédéfinie *max(Liste)* représente le maximum d'une liste d'entiers (comme pour *sum(Liste)*).
➙ Exemple : X #= max([2,6,7]). → X=7.

IV Exercices

L'enseignant vous indiquera les exercices (parmi les suivants) à réaliser en séances.

IV.1 Quelques Exercices simples sur les listes

Écrire et tester les prédicats suivants :

(1) **affiche_liste/1** : affiche_liste(Liste) : affichage des éléments, ligne par ligne, avec le rang de chaque élément affiché.

(2) **dernier/2** : *dernier(Liste, Last_Element)* : dernier élément de la liste.

Le prédéfini **last/2** fait la même chose.

Voir aussi en section VII.4 où dernier est revisité.

(3) **taille/2** : taille(Liste, Taille) : nombre d'éléments de la liste.

Le prédéfini **length/2** fait la même chose. De plus, comme **taille/2**, il permet de créer une liste d'une taille donnée.

Voir aussi en section VII.3 où taille est revisité.

(4) Écrire le prédicat du test d'appartenance d'un élément dans une liste (**membre/2**) puis dans une liste ordonnée (**membre_ord/2**).

(5) **concat/3** : concat(Liste1, Liste2, Liste3) : concaténation de deux listes Liste1 et Liste2 avec résultat dans Liste3.

➙ Il permet également de "découper" une liste en deux et ce de toutes les manières possibles.
➙ Le prédéfini **append/3** fait la même chose.

Le prédicat concat/3 est non seulement capable de concaténer deux listes connues, mais d'établir une relation logique entre ces 3 listes sous la forme *L3* ↔ *L1* *suivie de* *L2*.

Exemple : essayer

```
?-concat(L1, [c,d], [a,b,c,d]).      % Quelle était L1 telle que son ajout à L2 a donné [a,b,c,d].
?-concat(L1, L2, [a,b,c,d]).         % Quelles étaient L1 et L2 telle que leur ajout a donné [a,b,c,d]
```

➙ Cette seconde requête montre le caractère "équationnel" sur listes de concat/3.

Le prédicat concat/3 est revisité en rapport avec le coupe choix en section V.2 et V.1

V Coupe choix

V.1 Prédicat concat et le coupe choix

Rappel du prédicat concat/3 (activer la trace pour avoir les points de choix) :

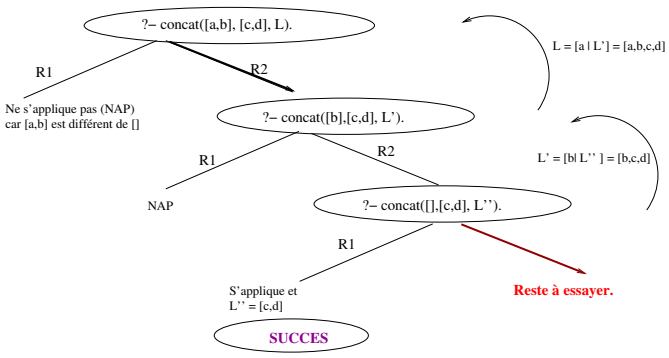
```
(R1)    concat([], L, L).
(R2)    concat( [X | Reste], L, [X | L2] ) :-        %
              concat(Reste, L, L2).

Posons la question simple :
 ?- concat([a,b],[c,d],L).        % L=[a,b,c,d] (activer trace pour voir le point de choix)
```

Remarque :
• Pour la requête *?- concat([a,b],[c,d],L).*
en cas de succès, Prolog maintient des données (internes) pour savoir si d'autres solutions existent (alors qu'il n'y en a pas d'autre).

➙ L'activation de la trace montre **les points de choix** (ici 2 branches dans chaque appel, car 2 clauses).

Dans l'arbre de recherche ci-contre, on note par R1, R2 les 2 clauses du prédicat *concat*.

➙ A droite en bas, on remarque la branche qui reste à explorer : puisqu'un succès est obtenue à l'aide de (R1) sur l'appel
?-concat([],[c,d], L'')
donnant L'' = [c,d].



Prolog envisage de re-tenter également (R2) et l'appel **?-concat([],[c,d],L'')**.

C'est cette branche qui provoque la proposition de Prolog de nous donner d'autres réponses.
• Les noeuds de cet arbre montrent seulement les appels récursifs, les détails des prédicats ne sont pas rapportés.

La branche (marquée Reste à essayer) sera exploitée par Prolog si suite à un succès, on en demande d'autres (par ' ;').
➙ Or, ayant produit un résultat+succès, nous pourrions ne pas être intéressé par une éventuelle seconde réponse et/ou nous voudrions accélérer l'exécution.

☞ Si nous ne voulons une seule réponse à notre requête, nous devons utiliser le mécanisme "Coupe Choix" (dit cut) noté par ' !' (voir ci-dessous).
➙ Ce mécanisme rend les prédicats qui l'utilisent *déterministe*.

Le prédicat *concat(L1,L2,L3)* établit une relation logique entre ces paramètres. Ce prédicat est non seulement capable de concaténer deux listes, mais aussi de résoudre une sorte d'équation sur deux listes et leur concaténation :
?- concat(L1, L2,[a,b,c,d]]).
➙ L1=[], L2=[a,b,c,d] puis L1=[a], L2=[b,c,d], ... ,L1=[a,b,c,d], L2=[]

V.2 Coupe choix sur concat/3

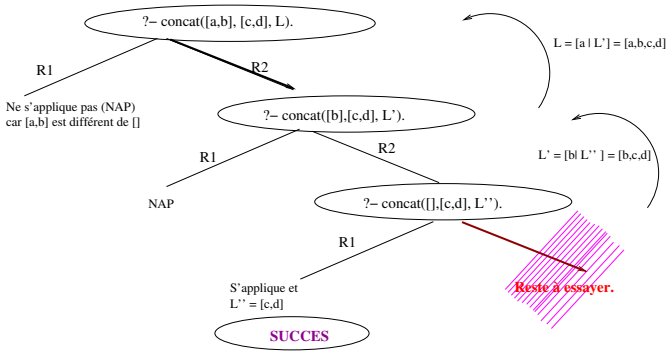
Dans certains cas, on peut souhaiter rendre son code plus efficace (et moins gourmand en mémoire). Pour ce faire, on peut supprimer le point de choix (qui ne nous intéressent pas) en plaçant des coupes choix.
➙ Il faut remarquer que l'efficacité ainsi obtenue est au détriment de la capacité de notre code à donner toutes les réponses possibles.

• *Se contenter d'une seule réponse dans concat/3 :*
➙ La nouvelle version (appelé concat2/3) devient :

```
(R1)    concat2([], L, L) :- ! .                    % coupe choix ' !'
(R2)    concat2( [X | Reste], L, [X | L2] ) :-      % inchangé
            concat2(Reste, L, L2).
```

Poser des questions :
?- concat2([a,b],[c,d],L). → L=[a,b,c,d] et un seul succès
?- concat2(L1,L2,[a,b,c,d]). → L1=[], L2=[a,b,c,d] et un seul succès

• L'arbre de recherche du prédicat *concat2/3* pour la requête
?- concat2([a,b],[c,d],L).



• La partie hachurée (qui ne sera pas exploitée) est le résultat de l'effet du coupe choix dans concat2/3.
• Notons que la présence du coupe choix dans concat2/3 ne permet plus de trouver toutes les combinaisons de L1 et L2 dans la requête
?- concat2(L1,L2,[a,b,c,d]).

• Une manière plus souple d'utiliser le coupe choix, lorsque cela est possible, est de l'utiliser sélectivement et dans la requête. Cela permet non plus de rendre un prédicat (tel que concat/3) déterministe, mais de l'utiliser ponctuellement de manière déterministe.

➙ Nous utilisons donc la première version de concat/3 dans la requête :
?-concat(L1,L2,[a,b,c,d]) , !. % remarquez le ' !' dans la requête

On obtient ainsi le même effet que par concat2([a,b],[c,d],L).

➙ Cette fois, le coupe-choix est placé dans la requête disant : une seule réponse me suffit.

V.2.1 Mots de la fin sur ces coupe-choix

Pour mieux comprendre la signification de ces coups choix, interprétons la version *concat/3* rappelée :

```
(R1)    concat([], L, L).
(R2)    concat( [X | Reste], L, [X | L2] ) :-
            concat(Reste, L, L2).
```

• Ce prédicat implante la définition de "concat" en envisageant 2 cas pour concat(L1,L2,L3) mais pour bien comprendre, gardons en tête que concat est une relation entre ses 3 arguments :
concat(L1,L2,L3) est vrai si L3 = L1 suivi-de L2 :

(R1) L1 est vide : L3 = L2.

OU

(R2) L1 possède une Tête et un Reste : L3 = concaténer Reste avec L2 puis y ajouter Tête

➽ On constate que les 2 cas s'appliquent en concurrence 'selon la définition relationnelle de concat.
Du point de vue logique, il y a une **disjonction** (un **OU**) commutative entre les deux clauses
ET AUCUNE CLAUSE NE DISQUALIFIE L'AUTRE (Disqualifier = empêcher l'appel).
➙ C'est ainsi que concat(L1,L2,[a,b,c,d]) donne 5 réponses.

• L'ajout des coupes choix (donnant la version *concat2/3*) change cette lecture et place des "sinon" entre les clauses :

```
(R1)    concat2([], L, L) :- ! .                    % coupe choix ' !'
(R2)    concat2( [X | Reste], L, [X | L2] ) :-      % inchangé
            concat2(Reste, L, L2).
```

Cette fois, la lecture devient :

(R1) L1 est vide : L3 = L2.

Ou bien = sinon

(R2) L1 possède une Tête et un Reste : L3 = concaténer Reste avec L2 puis y ajouter Tête

⊷ L'ajout des coupe-choix **impose un ordre** entre les clauses et transforme, d'un point de vue opérationnel, le OU en un **OU-exclusif** entre les clauses.

➙ De ce fait, à chaque coupe-choix exécuté (donc *franchi*), la clause DISQUALIFE les suivantes.

☞ De facto, avec le coupe-choix, le OU entre (R1) et (R2) devient 'Si-Alors-Sinon'.

Ce changement est belle et bien opérationnel (a un effet lors de l'exécution et ne change pas la définition logique de la concaténation) et n'a pas une interprétation logique fondée.

Les plus curieux se diront que OuX est bien un opérateur de la logique et ils auront raison dans le cas présent (ceux-là mêmes savent que "si-alors-sinon" n'est pas un opérateur logique !).
On verra dans ces BEs que le coup-choix pourra être placé n'importe où dans une clause, là où son interprétation logique devient plus délicate (cf. *plus_grand* ci-dessous).

Remarques :

Le prédéfini **append/3** est équivalent à notre *concat/3*.

Si on décide d'utiliser un coupe choix dans une requête (telle que *concat(L1,L2,[a,b,c,d]), !.*), on peut exprimer la même chose par **once(concat(L1,L2,[a,b,c,d]))**, c'est à dire : une seule réponse à ...

➙ *once(P)* est traduit par *P, !.*

VI Exercices en séance & à rendre

VI.1 Recettes

1) On définit une base de données (*bd_recettes1.pl* fournie) concernant des repas et les ingrédients nécessaires pour les préparer.

Le nombre d'ingrédients pour préparer un mets est inconnu d'avance.

Si pour préparer un mets **m**, l'on a besoin des ingrédients **a, b et c,** on écrira :

```
% recette(nom, liste d'ingrédients)

recette(m, [a, b, c]).
```

Exemples : pour préparer un gâteau, il faudra du lait, de la farine et des oeufs ; ce que l'on peut exprimer par le fait :

```
recette(gateau, [lait, farine, oeufs]).

recette(the, [sachet_de_the, eau]).
```

2) Définir la relation *disponible(I)* qui décrit les ingrédients disponibles (dans la cuisine !).

Exemples :

```
disponible(eau).

disponible(lait).

disponible(oeufs).

....
```

QUESTIONS : définir les deux relations :

1) ***peut_prepare(R)*** qui est vraie pour le repas R si tous les ingrédients nécessaires pour préparer R sont disponibles.

Par exemple, on posera la question Prolog suivante :

?- peut_preparer(gateau).

2) ***a_besoin_de(R, I)*** qui est vraie pour un repas R et un ingrédient I si R contient I.

Par exemple, on posera la question Prolog suivante :

?- a_besoin_de(gateau, lait).

VI.2 Introduction des quantités

• Rappel de quelques exemples d'arithmétique (avec contraintes) :

```
A + B #=10.              % n'oubliez pas le '#' : c'est une équation sur les entiers positifs/nul
X + 2 * Y #= Z + 2.
A #> 5 - Y.              % Le '#' : c'est une inéquation sur les N+
U # >= 2, V #=< U-1.     % la place de '=' ici n'est pas comme en langage C !
B #\=5.                  % B différent de 5.
```

* On modifie les faits *recette* en introduisant les quantités disponibles de chaque ingrédient.

Par exemple, on modifie le prédicat disponible et on aura (*bd_recettes2.pl* fournie) :

```
disponible(lait,5).              % pour 5 litres (ou 5 unités)
```

* Modifier les prédicats ci-dessus pour tenir compte de ces quantités sachant qu'un mets **M** ne peut être préparé que si les ingrédients nécessaires existent en quantité suffisante.

* Poser la question *peut_preparer(X)* avec X = une variable pour savoir tout ce que l'on peut préparer.

N.B. : On ne gère pas ici le fait que préparer une *omelette* consomme des *oeufs* et empêchera peut-être de pouvoir préparer un *gâteau*.
➙ Cependant, vous pouvez en tenir compte (en bonus) si vous construisez une liste des ingrédients disponibles et que vous gérez une comptabilité des ces ingrédients, au fur et à mesure que vous réalisez différents mets.

Ci-dessous, une autre manière de gérer les quantités est donnée.

VI.3 Gestion des quantités

La gestion des quantités nécessite une comptabilisation des quantités.
Par exemple, si la base contient disponible(lait,3), l'utilisation de 2 unités (litres) de lait pour une *purée de pommes de terre* peut compromettre la préparation d'un *gâteau* qui a besoin de 3 unités de lait.
➙ Il est évident qu'on ne peut pas préparer les deux.

Comment réaliser cette comptabilité ?
1- Cette comptabilité peut se faire à travers les passages des paramètres : on dispose en paramètre une liste de quantités restante de chaque ingrédient.

2- Une autre manière de réaliser le même effet est d'utiliser des *variables globales* intermédiaires.

C'est à dire, ayant disponible(lait,3), après avoir consommé 2 litres de *lait* pour une *purée de pomme de terre*, modifier la quantité disponible de lait et faire apparaitre 1 unité restante pour le lait.

Pour ce faire, on enregistre les quantités disponibles de chaque ingrédient pour préparer un mets M1. Ensuite, quand un ingrédient I1 disponible en quantité Q1 est nécessaire à la préparation d'un mets M1, on consomme la quantité Q1_1 (quantité requise de I1 pour le mets M1) et on fait comme s'il ne nous restait (Q1-Q1_1) et on se met à la préparation d'un autre mets M2 (pourquoi pas encore M1).

Que veut dire : enregistrer la quantité initiale Q1 de l'ingrédient I1 ?

1- Une manière est de manipuler la base de données (prédéfinis assert/retract que vous pouvez étudier).

Pour cela, vous devez d'abord récupérer la liste L de tous les ingrédients ainsi que leur quantité. Cette liste est ensuite transmise aux prédicats qui comptabilisent les quantités si on décide de préparer un des Mets.

Sachant que la valeur d'une variable logique (de Prolog) n'est pas modifiable, vous devez donc avoir une telle liste (soit L_in des couples $Ingrdient - Quantite$) en entrée des prédicats concernés mais aussi une liste en sortie L_out qui contiendra les mêmes couples Ingrédient-Quantité que L_in en entrée sauf pour l'ingrédient dont la quantité a été modifiée.

2- Un autre moyen est d'utiliser l'opérateur @ := de BProlog qui permet la modification d'une valeur au sein d'une liste ou d'une structure. L'intérêt de cet opérateur est qu'il est sensible aux retours arrières (défait les modifications faites).

➙ Exemple : $S = f(1,2), S[1]$ @ := 10.
le terme S devient $S = f(10,2)$ et en cas de retour arrière, S retrouvera sa forme initiale.
Cet opérateur est (un peu) comme l'*affectation* dans les langages impératifs (ou comme les données mutables et *ref* en CAML) mais sensible aux retours arrières.

➙ Un autre exemple : on diminue la quantité de lait de 15 à 10 :
X=lait(15), X[1] @ := 10.
→ Réponse : X = lait(10)
→ X est devenu *lait(10)* et en cas de retour arrière, elle reprendra sa valeur précédente.

☞ Le principe de cette solution est proche du cas ci-dessus sauf que nous n'avons pas besoin d'utiliser la liste L_out ci-dessus et les modifications se font directement sur L_in.

➙ Ce mécanisme est une aide et a été implanté en BProlog à l'aide d'autres prédicats prédéfinis dont les détails ne sont pas utiles ici.

3- Une autre manière est de créer la matrice $Mets \times Ingredients$ (tous les mets, tous les ingrédients) et de poser un système d'équations-inéquations et optimiser selon un critère (par exemple, nombre de mets).

4- BProlog permet de manipuler des variables *globales* (quasi similaire aux langages tels que C/C++).
☞ mais pour cet exercice, cette technique sera relativement délicate à mettre en oeuvre.
➙ Voir en annexe IX.8, page 30 pour les détails.

VI.4 Récupération des résultats d'une exécution

Pour obtenir des résultats d'exécution dans un fichier, voir le fichier **utiles-bprol.pl**.

Voir aussi en Annexes (section IX.6, page 29) quelques prédicats prédéfinis .

VII Pour aller plus loin

VII.1 Max de deux termes et cut

• Essayons d'écrire le prédicat *plus_grand/3* qui fait (presque) la même chose que le prédéfini *max*.
➙ Remarquez que *max/2* est une fonction utilisable sous conditions, et que *plus_grand/3* un prédicat.

```
plus_grand(A, B, Max) :- A > B , Max = A.
plus_grand(A, B, Max) :- A =< B , Max = B.

Poser des questions (et expliquer les réponses) :
?- plus_grand(2, 3, M).
?- plus_grand(12, 3, M).
?- plus_grand(2, 3, 2).
?- plus_grand(X, 3, M).
?- X=10, Y=21, plus_grand(X, Y, M).
?- plus_grand(X, Y, M), X=10, Y=21.        % X et Y reçoivent des valeurs, mais trop tard !
```

➙ La dernière question montre que *plus_grand(A, B, C)* n'est pas une contrainte et n'impose pas à C d'être le plus grand de X et de Y.
➥ Par contre, le prédéfini max l'est si on l'utilise avec #=.
➥ Poser la requête *?- C #= max(A,B).*
➥ Ou *?- C #= max(A,B), A=2, B=3.*

VII.1.1 Cut et Max de deux termes

• On pourrait utiliser le coupe choix (cut, !) dans le prédicat *plus_grand/3* pour le rendre plus efficace (on l'appellera *plus_grand2/3*).

```
plus_grand2(A, B, Max) :- A > B , ! , Max = A.
plus_grand2(A, B, Max) :- Max = B.

Poser les questions (et expliquer les réponses) :
?- plus_grand2(2, 3, M).
?- plus_grand2(12, 3, M).
?- plus_grand2(2, 3, 2).
```

☞ Dans l'idée d'être encore plus efficace, pourquoi ne pas écrire *plus_grand2/3* sous la forme suivante (on l'appellera *plus_grand3/3*).
➙ La version ci-dessous traduit le raisonnement suivant (les *cuts* introduisent des "si-alors-sinon") :
si A > B alors le plus grand des deux est A sinon c'est B. «—notez-bien le schéma "si-alors-sinon"

```
plus_grand3(A, B, A) :- A > B , !.        % si A > B alors DISQUALIFIER la 2e clause.
plus_grand3(_A, B, B).

Poser les questions (et expliquer les réponses) :
?- plus_grand3(2, 3, M).
?- plus_grand3(12, 3, M).
?- plus_grand3(2, 3, 3).
?- plus_grand3(3, 2, 2).        % réponse aberrante, abus de cut.
```

➙ Pour corriger cette version, remplacer la 2e clause par la suivante (sans abandonner l'efficacité de cut) :

```
plus_grand3(A, B, B) :- A =< B.
```

N.B. : Vous pouvez remplacer le comparateur '<' par '@<' (ou '=<' par '@=<') pour pouvoir comparer tous termes, pas seulement les nombres.

VII.1.2 Contrainte 'Max des deux'

➽ La version 'contrainte' de plus_grand/3 (appelée *plus_grand4/3* ci-dessous) utilise les contraintes :
#> : contrainte plus grande (entiers naturels) ;
#=< : contrainte plus petit-ou-égale (entiers naturels) ;
#/\ : contrainte ET logique (booléen)
De plus, on doit supprimer la disjonction par les clauses en écrivant une seule règle (à la place de deux) qui utilise la contrainte booléenne OU (#\/).
On attendra le cours et le BE suivant pour en connaître plus de détails.

```
plus_grand4(A, B, C ) :- (A #> B #/\ C #= A) #\/ (A #=< B #/\ C #= B).
```

Interprétation (logique) : "(A>B AND C=A) OR (A =< B AND C=B)"

Poser les questions (et expliquer les réponses) :

```
?- plus_grand4(2, 3, M).
?- plus_grand4(12, 3, M).
?- plus_grand4(2, 3, 3).
?- plus_grand4(2, 3, 2).
?-plus_grand4(Y, 3, 4).
?-plus_grand4(3, X, 5).
?-plus_grand4(Y, X, Z), X=2, Y #= X+1 .
```

• Pour les plus curieux : la partie droite de *plus_grand4(A, B, C)* peut être lue de la manière suivante (AND et OR sont commutatifs) :
"¬ (A>B AND C=A) implique (A =< B AND C=B)"
Ce qui est la même chose que :
"¬ (A =< B AND C=B) implique (A>B AND C=A)"

On peut également remplacer chaque AND par implique (et vice versa) :
(A > B ==> C = A) AND (A =< B ==> C = B).

➙ En BProlog, la contrainte négation s'écrit "#\" et implique s'écrit "#=>" ("#<=>" est équivalent)

VII.2 Quelques utilisations de concat

Concaténation de 3 listes :

```
concat3(L1, L2, L3, Resultat) :
    concat(L1,L2,L1L2),          % concaténer les 2 premières listes dans L1L2
    concat(L1L2, L3, Resultat).          % concaténer L1L2 avec L3
```

Split :
Etant donné une liste L_noms de noms contenant 'jean', trouver la liste L1 contenant les noms qui viennent avant 'jean' dans L_noms, L2 contenant les noms après 'jean'.
➙ On pose la requête :

```
?- L_noms=[marie,paul,jean,jacques,helene, alex], concat(L1, L3, L_noms), L3=[jean|L2].
```

➙ L1 = [marie,paul], L3 = [jean,jacques,helene,alex], L2 = [jacques,helene,alex]
☞ On peut soumettre notre requête plus simplement :

```
?- L_noms=[marie,paul,jean,jacques,helene, alex], concat(L1, [jean|L2], L_noms).
```

VII.3 Utilisation de la taille d'une liste avec length

• Le prédéfini *length(Liste, Taille)* permet de connaitre la taille d'une liste.
?- length([a,b,c],N). → N = 3

En fait, comme la plupart des prédicats, *length(Liste, Taille)* établit une relation entre la Liste et l'entier Taille. C'est à dire, par sa nature réversible, length/2 permet de créer une liste (de variables) à partir de la Taille :
?- length(L, 3). → L=[X1,X2,X3].
➙ On pourra ensuite utiliser L pour à d'autres fins.

Exemple : soit une liste L de 9 éléments. transformer L en une matrice 3×3 (donc 3 listes L1,L2,L3) :
?- L=[a,b,c,d,e,f,g,h,i], length(L1, 3),length(L2, 3),length(L3, 3), concat3(L1,L2,L3,L).

On peut généraliser et créer un prédicat qui transformer une liste de taille L en une matrice de taille $L = M \times N$ (faire en exercice).

VII.4 Dernier d'une liste avec length et concat

Une manière de trouver le dernier élément d'une liste non vide L est de parcourir cette liste et s'arrêter sur le dernier (prédicat rappelé ici) :

```
dernier([Der], Der).          % un singleton est le dernier
dernier([_|Reste]) :- dernier(Reste).          % concaténer L1L2 avec L3
```

Une autre manière : trouver la taille N de la Liste, construire une liste L1 de taille N1=N-1, concaténer L1 et une liste L2 à un élément donnant L.

```
dernier1(L, Der) :-          % De plus, on contrôle la taille de L
   length(L,N), N #>0, N1 #= N -1, length(L1,N1), concat(L1,[Der],L).
```

Par exemple, si L=[a,b,c,d], on peut écrire :
?- dernier1([a,b,c,d], Der). % → Der=d.

VII.5 Nème élément d'une liste

Le prédéfini **nth(Indexe , Liste, Element)** permet de trouver l'élément du rang Indexe dans L.
Exemple : ?-nth(2, [a,b,c,d], X). % → X=b.

☞ Dans le domaine des contraintes, nous verrons element/3 une version particulièrement intéressante de *nth*.

VIII Exercices Bonus

Trois exercices sont proposées en option et en bonus. Ceux qui les rendent auront une meilleure note des BES.
➙ Il s'agit des 'livres', 'circuits' et 'équipes'.

VIII.1 Livres

• Exercice sur une base de données de Livres avec des listes.

Ce *travail à rendre* n'est pas obligatoire. Cependant , ceux qui le réaliseront amélioreront leur note.
A rendre sous 3 semaine ouvrables

• On se donne une base de données de livres où pour un écrivain, une liste de livres est présentée (cf. rangement par *auteur* dans une librairie).

Le format de chaque donnée (*livres* à deux arguments noté *livres/2*) est le suivant :

livres(auteur(prenom, nom), [(livre_1, prix), ..., (livre_n, prix)]).

Le nom/prénom/titre sont placés entre '...' à l'intérieur desquels les séparateurs sont acceptés. Si une apostrophe dans figurer dans un nom/titre, il faudra doubler ce dernier.
➙ Par exemple : *d'automne* s'écrira *'d''automne'*.

Chaque livre suivi de son prix est donné sous la forme d'un couple (x, y).
➙ Par exemple : *('Les Misérables', 35)*

La présence des parenthèses est obligatoire sans quoi il s'agirait de deux termes (et non un couple).

La BD livres

La base de livres (vous pouvez la compléter, attention aux accents).

livres(**auteur**('Victor', 'Hugo'), [('Juliette Drouet', 32), ('Notre Dame de Paris', 45), ('Les Misérables', 35),
('Quatre Vingt Treize', 24), ('Feuilles d automne', 30), ('Les Contemplations', 25)]).
livres(auteur('Léo', 'Ferré'), [('Testament Phonographe', 25), ('La méthode', 25), ('Benoit Misère', 30)]).
livres(auteur('Max', 'Weber'), [('Economie et Société', 24), ('Le savant et le Politique', 29),
('Théorie de la science', 34), ('La bourse', 25)]).
livres(auteur('Blaise', 'Pascal'), [('Pensées', 25), ('De l esprit Géométrique', 45)]).
livres(auteur('Confucius', 'Confucius'), [('Confucius', 35), ('La morale', 30), ('Les entretiens', 25)]).
livres(auteur('Jacques', 'Lacan'), [('D un autre à l autre', 30), ('Mon enseignement', 50)]).
livres(auteur('Sigmund', 'Freud'), [('Sur le rêve', 30), ('Totem et Tabou', 25), ('Métapsychologie', 40)]).
livres(auteur('Michel', 'Foucault'), [('Surveiller et punir', 34), ('Histoire de la folie', 25),
('L ordre du discours', 35)]).
livres(auteur('Jacques', 'Derrida'), [('Feu la cendre', 30), ('Mémoire d aveugle' , 20), ('Voiles', 25) ,
('Demeure' , 35), ('Position', 20)]).
livres(auteur('Michel', 'Serres'), [('Atlas, Philosophie des réseaux', 30), ('Tiers Instruit', 25)]).

.....

Écrire les prédicats répondant aux requêtes suivantes (utiliser les prédicats du sujet précédent sur les listes) :

1- Les auteurs dont le prénoms (ou les noms) sont identiques (compléter la base éventuellement).

2- La somme des prix des livres d'un auteur dont on précise le prénom et le nom.

3- Le nombre de livres d'un auteur dont on précise le prénom et le nom.

4- Le maximum des prix des livres d'un auteur dont on précise le prénom et le nom.

5- Les livres d'un auteur dont le prix est inférieur (ou supérieur) à un prix donné.

6- Les titres des livres dont le prix = un certain prix donné en paramètre.

7- Les auteurs dont le prénoms (ou les noms) sont identiques (compléter la base éventuellement).

8- Les auteurs ayant écrit un titre identique

9- Pour ceux qui veulent aller plus loin : en utilisant les prédicats prédéfinis *sort/2* et *findall/3* (voir ci-dessous),

9-1) Le titre et l'auteur du livre le moins (le plus) cher de cette librairie.

9-2) établir et afficher l'inventaire trié par titre des livres avec l'auteur en face. Pour récupérer les traces dans un fichier, voir ci-dessous.

Deux prédicats utiles pour les livres :

• **sort/2** : le prédicat prédéfini *sort(Liste, Résultat)* permet de trier *Liste* dans *Résultat*.
Exemple : sort([londres, ici, allo], L). =➙ L=[allo, ici, londres]

• **findall/3** : pour obtenir d'un seul coup toutes les réponses à une question :

```
findall(mise_en_forme_de_chaque_réponse,
        la_question,
        liste_de_tous_les résulats_cumulés).
```

Exemple 1 : pour obtenir la liste des noms des auteurs :

```
findall(Nom,
   livres(auteur(_prenom , Nom), _titres), % on veut seulement les Noms
   Resultat).
```

La liste *Resultat* contient la liste des noms.

La question posée demande à retenir seulement le nom dans chaque interrogation de livres (les autres variables sont dites anonymes et commencent par un '_').

Exemple 2 : si l'on veut une liste composée uniquement du premier titre de chaque auteur :

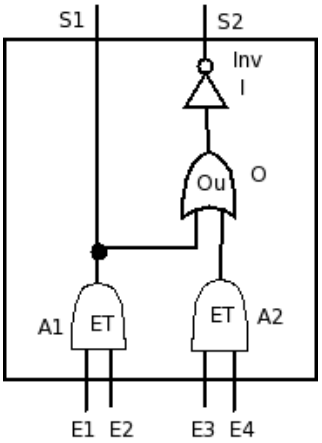
```
findall(Titre,
   livres(_auteur, [(Titre, _prix) | _reste]), % on veut seulement les Titres
   L).
```

VIII.2 Circuits en Prolog

• Etudier l'exemple ci-dessous puis répondre aux exercices proposés.

Modélisation et évaluation des circuits :
Le circuit ci-dessus se caractérise par :

```
circuit('C1',[E1,E2,E3,E4 ] ,[S1,S2 ]) :-     %'C1' est le nom
  and('A1',[E1,E2 ] ,[S1 ] ),          % il est mis entre ' 'pour être
  and('A2',[E3,E4 ] ,[S3 ] ),           % différencié d'une variable
  or('O',[S1,S3 ],[S4 ] ),
  inv('I',[S4 ],[S2 ]).
```



Les portes élémentaires (tables de vérité de and (Et), or (Ou) et inv (Non)) :

```
and(_,[1 ,1 ],[1 ]).      and(_,[0 ,1 ],[0 ]).
and(_,[1 ,0 ],[0 ]).      and(_,[0 ,0 ],[0 ]).

or(_,[1 ,1 ],[1 ]).       or(_,[0 ,1 ],[1 ]).
or(_,[1 ,0 ],[1 ]).       or(_,[0 ,0 ],[0 ]).

inv(_,[1 ],[0 ]).         inv(_,[0 ],[1 ]).
```

Quelques questions à poser :

```
circuit('C1',[1 ,1 ,0 ,1 ] ,[0 ,1 ]).      ➞ yes/no
circuit(X,[ 1 ,1 ,0 ,1 ],[S1,S2 ]).       ➞ X ='C', S1 =1, S2 =0
circuit(X,[1 ,1 ,0 ,1 ],S).                ➞ X ='C', S=[1,0 ]
circuit(X,[E1 ,E2 ,E3 ,E4 ],[0 ,1 ]).      ➞ 9 réponses
circuit('C',[E1 ,E2 ,E1 ,E4 ],[0 ,1 ]).    ➞ 5 réponses
circuit('C',[E1 ,0 ,E1 ,E4 ] ,[0 ,1 ]).    ➞ 3 réponses
circuit('C',E,[0 ,1 ]).                    ➞ ....
```

Exercice 1 :

(1) Réaliser ce même exercice avec les contraintes booléennes.

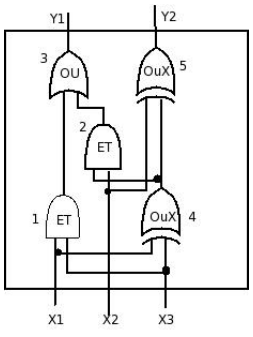
➞ L'utilisation des contraintes booléennes permet de ne plus avoir besoin des tables de vérité.

Exercice 2 : Soit l'additionneur 3 bits :

1- Présenter la spécification exécutable de ce circuit en Prolog.
2- A l'aide de ce circuit, réaliser un additionneur de deux nombres codés sur deux octets (8 bits).

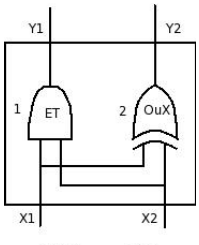
Remarque : pour éviter les problèmes de débordement, on supposera que les bits du poids le plus fort des deux nombres = 0.

➞ Aussi, utiliser plutôt les contraintes booléennes.



Additionneur 3 bits

Indication : à toutes fins utiles, un additionneur 2 bits est présenté ci-dessous.



Additionneur 2 bits

VIII.3 Equipes

Un exemple simple à étudier (travail optionnel et individuel)

En partant d'une liste de noms de joueurs, on désire créer N équipes de k joueurs.

Démarche : (on nomme les prédicats par A1, A2, B1, B2, ... pour explications)
• Entrées (prédicat *equipes*) : une liste de noms, un nombre d'équipes et un entier pour désigner le nombre de joueurs dans chaque équipe.
➙ On vérifie que le nombre total des joueurs est supérieur ou égal au *(nbr. d'équipes * taille de chaque équipe)*.
➧ La taille d'une liste est donnée par le prédéfini *length/2* : *length(Liste, Taille).*
➧ A noter : si le paramètre *Liste* de *length* est une variable, il deviendra une liste de *Taille* variables.
Par exemple : *length(L, 3)* donne *L = [Var1, Var2, Var3].*
➙ Après ces vérifications (dans *A1*), on s'emploie à la constitution des équipes (par *Bi*).
➙ En cas d'échec, soit dans A1 soit dans les prédicats appelés par A1, la clause A2 permet d'énoncer l'impossibilité de faire ces équipes.

Habituellement, un prédicat se contente de traiter le cas 'vrai' (succès) sachant qu'en logique binaire, le non-succès est implicitement un échec. Ici, on a voulu être plus explicite.

• Constitution des équipes :
Le prédicat *faire_les_equipes* (2 clauses nommées *B1, B2*) se charge de cette tâche.
Ses paramètres sont *Liste_joueurs*, *Nb_equipes* (restant à chaque appel), *Nb_joueurs_par_equipe*, *List_des_equipes* (qui s'enrichit par chaque appel) et *Reste_des_joueurs* (s'il reste des joueurs non sélectionnés).

On part de *Nb_equipes* à constituer. Chaque appel récursif à ce prédicat (dans *B1*) constitue une équipe ; supprime de la liste des joueurs ceux placés dans l'équipe qui vient d'être constituée ; fait "-1" (voir dans *B1*) sur *Nb_equipes* à chaque équipe constituée, refait un appel récursif pour faire les autres équipes .... jusqu'à 0 équipes à constituer.
Le test *Nb_equipes* = 0 est dans *B2*. Lorsque *Nb_equipes* devient 0, aucune autre équipe ne sera constituée et *Reste_des_joueurs* sera = ceux qui restent dans la liste des joueurs. Il se peut que *Reste_des_joueurs*=[] (si le compte était juste au départ dans A1). Pas de problème !

➧ Rappelez-vous : si un appel de prédicat échoue, le mécanisme de retour-arrière (Back Track) remet les variables à leur état précédent (on dit que l'on *défait* les unifications qui n'ont pas abouties) et on tente une autre solution. Ceci est une clef de base de la compréhension et la maîtrise de ce langage.

Dans les clauses *B1, B2* :
➙ *B2* ne s'applique que si *B1* ne s'applique pas. C'est l'objet des 2 premiers tests dans *B1* et le test *Nb_equipes* = 0 dans *B2*.
➙ C'est une solution pour éviter d'employer un cut dans *B1* : à l'aide de ces tests, on rend *B1* et *B2* mutuellement exclusives.

➧ N'abusez pas du coupe-choix. Ils suppriment (souvent) la capacité **réversible** d'un prédicat (et le transforment en une *fonction* à la C/C++ ; ce serait dommage !).

• Constitution d'une équipe : prédicat *faire_une_equipe/4*
Les paramètres sont : *Liste_joueurs, Nb_js_par_equipe, Reste_joueurs, Une_equipe*
➙ On commence par créer une liste de *Nb_js_par_equipe* variables que l'on unifiera avec des joueurs pris dans *Liste_joueurs*.
Cette unification est demandée au prédéfini *sublist(SS_liste, Liste)* qui teste si *SS_liste* est sous-liste de *Liste*, OU qui énumère les différentes sous-listes d'une liste (testez *?- sublist(SS_liste, [a, b, c]).*).
Lorsque l'unification demandée a lieu, le paramètre *Une_equipe* aura récupéré des noms de joueurs et il faut retrancher ces joueurs de la liste des joueurs. C'est l'objet du prédicat *difference_liste*.

• *difference_liste* enlève de *Liste_joueurs* les *Nb_js_par_equipe* joueurs placés dans *Une_equipe* à l'aide du prédéfini *select/3* (qui fait cela pour un élément) ; les appels récursifs à *difference_liste* complèteront

ces suppressions.

➞ Tester *select/3* (et *delete/3* qui supprime *toutes* occurrences d'un élément dans une liste) ; ils pourront vous être utiles plus tard.

... les autres solutions

VIII.3.1 Autres solutions

Il y a d'autres solutions à ce problème. Voici quelques indications sur une autre solution.

Les entrées du prédicat *equipes2* : une liste de noms de joueurs, un nombre d'équipes et un entier pour désigner le nombre de joueurs dans chaque équipe. Un dernier paramètre contiendra le reste des joueurs : les non engagés dans l'une des équipes constituées.

• **La démarche :**

— On vérifie que le nombre total des joueurs est supérieur ou égal au *(nbr. d'équipes * taille de chaque équipe)*.

— Après ces vérifications, on envisage cette fois une liste des équipes (par *length*) dont on connait le nombre.

— Pour couvrir toutes les combinaisons possibles, on calcule une permutation de la liste des noms des joueurs (par le prédéfini *permutation/2*)

— Puis on exige que le résultat de la concaténation de toutes les équipes + le reste des joueurs (non engagés) soit égale à une permutation donnée de la liste initiale des joueurs.

➞ Le prédéfini *permutation/2* donne toutes les permutations possibles d'une liste.

➞ La concaténation de *Nb_equipes* listes de joueurs s'effectue par un prédicat qui utilisera le prédéfini *append/3*.

IX Annexes

IX.1 Somme avec accumulateur

• La version *somme2/2* suivante utilise la technique dite d'*accumulateur*.

Une technique dite d'*enrobage* (wrapping) permet de conserver deux paramètres (pour somme2) mais cache une version (*somme/3*) à 3 paramètres.

```
somme2(L, S) :- somme(L, 0, S).        % on appelle somme/3 en initialisant l'accumulateur.

somme([], Accu, Accu).                 % La liste est vide : Accu contient la somme totale.
somme([Tete | Reste], Accu, Somme_finale) :-
    number(Tete) ,                     % Pour éviter certaines erreurs (voir ci-dessus)
    Accu1 is Accu + Tete, ,            % Ajouter Tete à Accu : Accu = la somme partielle.
    somme(Reste, Accu1, Somme_finale).
```

Poser des questions (et expliquer les réponses) :

```
?- somme2([12, 15, 23], S).
?- somme2(a, S).
?- somme2([X], S).
?- somme2(X, S).
?- somme2([12, 15, a], S).
```

➙ On remarque que Accu contient à tout moment la somme des nombres déjà rencontrés. Elle est passé d'appel en appel, ce qui permet de disposer de la somme finale lorsqu'on arrive à la fin de la liste.

La technique d'accumulateur employée dans somme2/2 permet d'utiliser une récursivité terminale (plus facilement optimiseable) à la place d'une récursivité non-terminale dans somme/2 ci-dessus.

En outre, on dispose à chaque étape de la somme partielle. C'est précisément cette propriété (de disposer de la solution partielle à chaque étape) qui rend la technique d'accumulateur intéressante (utilisée par exemple dans les parcours de graphes).

IX.2 Compléments Termes en Prolog

Un terme = un objet manipulé par un programme Prolog.

Prolog distingue trois sortes de termes : variables, termes atomiques et termes composés.

— Les **variables** = les objets inconnus de l'univers : une suite alphanumérique (mélange de lettres et de chiffres) commençant par une majuscule ou par '_' :

➙ Exemples : X, Y12, Variable, _toto , _15

➙ Une variable commençant par '_' est une variable **anonyme** : celle dont la valeur ne nous intéresse pas mais que sa présence est nécessaire pour satisfaire un nombre d'arguments donné.

➙ Une variable Prolog est soit **libre** soit **liée**.

Si elle est libre, elle peut prendre n'importe quel terme pour valeur (peut être liée à n'importe quel terme).

Si elle est liée, soit elle est liée à une autre variable (par exemple via *X=Y* qui lie X à Y mais les deux sont des variables), soit elle a reçu une valeur (on dit qu'elle est **instanciée**), comme dans *X=f(12)*.

Si une variable est instanciée (a reçu une valeur), elle ne pourra pas changer de valeur le long de sa durée de vie.

— Les termes **atomiques** (= élémentaires, non décomposables) représentent les objets simples connus de l'univers.

Les termes atomiques se déclinent en :

✍ les nombres : entiers ou réels,

✍ les identificateurs : une chaîne alpha-numérique commençant par une minuscule

➞ Par exemple : *toto, aX12, jean_Paul_Durand*

✍ les chaînes de caractères entre ""

➞ *"Ecole Centrale"*, *"1265"*

— Les termes **composés** construits sous la forme $foncteur(t_1, ..., t_n)$ où *foncteur* est un identificateur (ne peut pas être une variable) et t_i un terme quelconque. La valeur n est appelée l'arité du terme composé.

➞ Exemple : *personne(jean, adresse(12, lyon)).*

➞ Une liste est un terme composé (voir ci-dessous).

• Notez qu'une variable peut être liée à un terme composé qui contient des variables.
Par exemple, dans **X=f(Y)**, X restera pendants sa durée de vie associée à *f(.)* même si Y reçoit une valeur. Par exemple : *X=f(Y), ..., Y=12* donnera X=f(12).

• Pour être complet, une constante identificateur est assimilée à un terme composé sans argument ($n = 0$)

• Une programme Prolog est en soit un terme composé.

• On présente une chaîne de caractères comme une constante atomique mais la plupart des environnements Prolog (c'est le cas de BProlog utilisé ici) traite une telle chaîne comme une liste de ses caractères. De ce fait, une chaîne de caractère serait un terme composé.

• On appellera terme "appelable" (callable) si celui-ci est un terme composé (identificateur compris) qui correspond à un prédicat du programme. Par exemple, si le programme contient le fait *pere(jean, pierre)*, alors *X=pere(jean, pierre)* devient un terme appelable (on peut écrire : *X=pere(jean,pierre), call(X)*).

• Le Prolog fournit des prédicats prédéfinis pour tester ces catégories :
var/1, *nonvar/1* : variable ou non-variable
atom/1, *integer/1*, *float/1*, *number/1*, *atomic/1* : tester les constantes
compound/1, *callable/1* : termes composés et appelable

➞ BProlog ajoute le prédicat (*list/1*).

IX.3 Comparaison littérale

• **Sur la relation d'ordre :**
Les opérateurs '==', '≡', '@<', '@=<', '@>', '@>=', comparent deux termes selon la convention suivante :

➙ Comparaison de deux variables : la plus ancienne d'abord
➙ Variables domaine fini : la plus ancienne d'abord
➙ nombres : ordre numérique
➙ Atomes (constantes non décomposables) : ordre alphanumérique (table code ASCII)
➙ Termes composés : d'abord l'arité (nb. d'arguments) puis nom du foncteur puis les arguments
➙ Une liste est traitée comme un terme composé.

IX.4 Termes arithmétiques évalués en Prolog

• Dans une expression arithmétique, les expressions (E, E1, E2) suivantes sont évaluées et représentent un nombre (E_i est une expression arithmétique), par exemple dans X is E où E doit être évaluable (donc non variable) :

+E , **-E** : expression signée

E1 + E2 , **E1 - E2**, **E1 * E2** : addition, soustraction et multiplication

E1 / E2 : Division de nombres (entiers, réels)

E1 // E2 : partie entière du quotient E1 / E2 (comme en C)

E1 div E2 : partie entière du quotient E1 / E2

E1 $/<$ **E2** : partie entière du quotient E1 / E2 arrondie par défaut.

E1 $/>$ **E2** : partie entière du quotient E1 / E2 arrondie par excès.

E1 rem E2 : le reste arrondi de la division E1 / E2

E1 mod E2 : la partie entière du reste de la division E1 / E2

abs(E) : valeur absolue de E

sign(E) : le signe de E (-1, 0 et 1)

min(E1,E2) : le minimum de E1 et de E2

max(E1,E2) : le maximum de E1 et de E2

E1 ** E2 : $E_1^{E_2}$

☞ Il y a des opérations sur les bits (comme en C, voir la doc. de BProlog)

☞ **integer(X), float(X), round(X)** : conversion en entier / réel / arrondi.

sqrt(E) :

atan(E) :

cos(E) , **sin(E)** :

acos(E) , **asin(E)** :

exp(E) : e^E

log(E) : log naturel

ceiling(E) : arrondi par excès de E

floor(E) : arrondi par défaut de E

round(E) : arrondi de E au plus proche entier

truncate(E) : partie entière de E

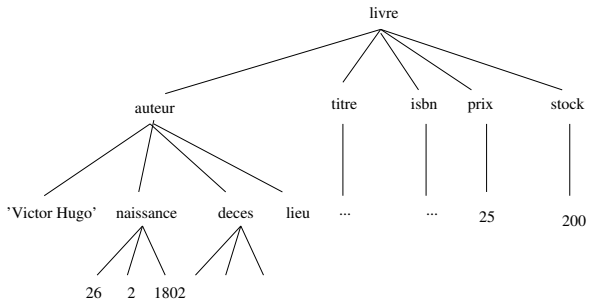
☞ Voir la doc. BProlog pour plus de détails.

• Quelques autres opérations au niveau des bits existent également (consulter la doc de BProlog).

IX.5 Détails des Listes en Prolog

• En Prolog, une liste est un terme composé.
➙ Un exemple de terme composé (un arbre) en Prolog :

```
livre (auteur ( 'Victor Hugo', naissance(26,2,1802),
                deces(22,5,1885), lieu(paris)),
       titre('les misérables'),
       isbn(1234567),
       prix(25),
       stock(200), ...
       ).
```



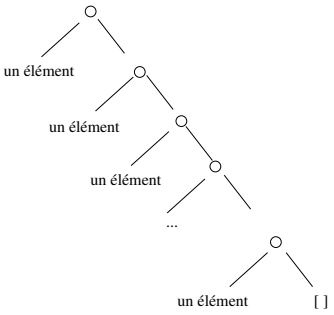
➙ cet arbre est quinternaire.

• En Prolog, une liste est un *arbre binaire*.

• En tant que arbre binaire, une liste dispose d'une structure contenant un nombre théoriquement illimité d'information, organisées en arbre binaire dégénéré.

• Le symbole [] désigne la liste vide.

• Avant de voir de plus prêt la syntaxe particulière des listes, voyons comment peut-on présenter en Prolog un arbre binaire.
➙ On doit décider comment désigner un arbre vide (le symbole particulier '[]' dans le cas d'une liste) . Soit la constante 'rien' pour représenter un arbre vide.



Voici quelques exemples d'arbres binaire représentant les noms d'élèves d'une classe dont on ne connait pas d'avance le nombre :

1. rien — arbre (classe) vide
2. classe(jean, rien) — un seul élève
3. classe(jean, classe(pierre, rien)) — deux élèves
4. .....
5. classe(jean, classe(pierre, classe(helene, classe(marie, classe(jacques, ..., classe(dominique, rien))....)

Pour faire simple, les exemples ci-dessus s'écrivent sous des formes plus simples suivantes.

1. [] — arbre (classe) vide
2. [jean] — un seul élève
3. [jean, pierre] — deux élèves
4. [.....]
5. [jean, pierre, helene, marie, jacques, ..., dominique]

Remarquons que de la même manière que l'expression '2+3+5' est une forme apparente de '+(2, +(3, 5))', une liste Prolog est une forme apparente d'un arbre binaire. Pour compléter la comparaison avec l'exemple d'addition, ajoutons :
➙ L'opérateur homologue d '+' de l'addition est le symbole '.' (ou alternativement l'opérateur '[]') ;
➙ En plus, on a besoin de représenter 'une liste vide' par [] (alternativement par 'nil').
➙ A la lumière de ces équivalences, la liste de 2 élèves '**[jean, pierre]**' est la forme externe de '**.(jean, .(pierre,[]))**'.

• Exécuter la requête **display("Ecole Centrale").** Interprétez.

IX.6 Autres prédicats prédéfinis utiles

• Rappel :
pour créer un terme (un arbre), il n'y a aucune action particulière à faire : il suffit de l'écrire pour qu'il soit crée.

➙ Exemple : le terme *personne(jean, 21, adresse(145, thiers, lyon))*

• **Quelques prédéfinis :**

\+ P : (ou *not provable(P)*) réussit si le but P échoue et, échoue si P réussit.

X = Y : réussit si les termes X et Y s'unifient.

X \= Y : ou \+(X=Y) réussit si les termes X et Y ne s'unifient pas.

X == Y : réussit si les termes X et Y sont littéralement identiques.

X \== Y : réussit si les termes X et Y ne sont pas littéralement =

atom(X) : réussit si le terme X est un atome.

integer(X), **float(X)** : réussit si le terme X est un entier (ou un réel).

atomic(X) : réussit si le terme X est un atome ou un entier.

var(X), **nonvar(X)** : réussit si le terme X est une variable (ou non variable).

write(X) : réussit et affiche le terme X sur l'unité de sortie (écran). Voir aussi **format**.

nl : passage à la ligne

read(X) : réussit et lit le terme X sur l'unité d'entrée (clavier). Voir aussi **get(X)** et **get0(X)**.

• **Opérateurs arithmétiques** (autres que les contraintes)

*is , + , - , * , /, mod, abs*, < , =< , >, >=

?- *A is 12 mod 5* ➙ A = 2

?- *Y = 32, A is abs(-3) + 5 * Y.* ➙ A = 163 (Y=32 est équivalent à Y is 32)

• **Comparateurs** (après évaluation)

?- *1 >= 2 .* ➙ non

?- *1 =< 2 .* ➙ oui

?- *3+7 > 5+2* ➙ oui % évaluation puis comparaison

• **Test d'égalité après évaluation :** = := , =\=

?- *123 + 34 mod 2 = := 12 * 3 - 6 .* ➙ non (les expressions sont d"'abord évaluées, puis comparées).

*?- 123 + 34 mod 2 =\= 12 * 3 - 6-5 .* ➙ oui

• **Identité littérale :** == , \==

?- X == Y. % X,Y variables libres

➙ No

alors que *?-X=Y.* réussit.

IX.7 Rediriger les sorties de BProlog

Pour pouvoir rendre une trace de vos test, vous aurez besoin de savoir rediriger les sorties de Prolog vers un fichier.
➙ Le fichier 'utiles-bprol.pl' fourni contient ce code.

Code

```
rediriger_sorties(Fic) :-
    open(Fic,write,F),      % on ouvre le fichier de sortie
    current_input(I),      % Quel est le fichier d'entrée actuel ?
    current_output(O),      % et le fichier de sortie ?
    asserta(streams___(F,I,O)),      % on les sauvegarde
    write('la sortie redirigée vers '),
    write(Fic), nl,
    set_output(F).       % Maintenant, les sorties iront vers Fic

remettre :-
    retract(streams___(F,I,O)),  % Récupération des noms sauvegardés
    set_output(O),      % Remettre l'ancienne sortie
    set_input(I),        % Et entrée
    flush_output(F),      % on écrit tout
    close(F),
    writeln('la sortie est rétablie').
```

Exemple d'utilisation :
?- rediriger_sorties('toto.txt'), writeln('Bonjour'), append([a,b],[c,d], L), writeln(L), remettre.

On aura dans le fichier 'toto.txt' deux lignes : Bonjour puis [a,b,c,d].
☞ Notons que les écritures n'ont lieu que par *write* + *ln* ou *writeln*.

IX.8 Gestion des quantités

A propos de la gestion des quantités dans l'exercice des ingrédient à l'aide des variables globales en BProlog.

BProlog permet de manipuler des variables *globales* (quasi similaire aux langages tels que C/C++).
Il suffit pour cela d'affecter la valeur initiale de la quantité Q1 à l'ingrédient I1 par le prédéfini **global_set(I1, Q1).** Par exemple, **global_set(lait,3).**

On peut consulter la valeur de la variable I1 (par exemple lait) par **global_get(lait, Qte)** et récupérer la valeur actuelle dans Qt.

Ensuite, chaque fois que l'on consomme la quantité Q1_1 de cet ingrédient pour un met, on affecte à I1 le reste (Q1 - Q1_1) par **global_set(I1, Le_reste)**.

Vous l'avez remarqué : on n'écrit pas *global_set(I1, Q1 - Q1_1)*
mais **Le_reste #= Q1 - Q1_1,** **global_set(I1, Le_reste)**
car BProlog n'évalue pas les arguments d'appel d'un prédicat.

Pour la cohérence, on fera attention à utiliser *globale_get* après avoir fait *global_set*..

☞ Ne faite la modification de la quantité d'un ingrédient que si vous pouvez préparer le mets qui le contient. En plus, vous devez faire la comptabilité des quantités pour un met en une seule fois, c'est à dire, non pas au fur et à mesure des utilisations.

• Exemple d'utilisation des variables globales :
➞ global_set(toto, 5), ... global_get(toto,X), ...

• Une difficulté avec global_set (affectation à une variable globale) et est qu'en cas d'échec ou de retour arrière, global_set ne peut restituer l'ancienne valeur de la variable. Une solution sera d'utiliser un prédicat qui permet de tenir compte de cet effet (voir aussi les BEs suivants).

Exemple : si un retour arrière a lieu sur une global_set, on remet l'ancienne valeur en place.

```
% PRINCIPE de gestion des variables globales de BProlog avec back-track
   b_global_set(Id, Val) :-
      (is_global(Id)          % est-ce une variable globale ?
            -> global_get(Id, Old)          % Si oui, prend son ancienne valeur
            ;   Old=0          % sinon lui donner 0 (jamais utilisée)
      ),
      b_global_set(Id, Old, Val).          % Il y a maintenant une ancienne valeur pour Id

   b_global_set(Id, _Old, Val) :-          % A la 1e utilisation, utiliser global_set(Id, Val).
      global_set(Id, Val).

% on revient ici en cas de retour arrière
   b_global_set(Id, Old, _Val) :-
      global_set(Id, Old).          % Remettre l'ancienne valeur de Id.
```

➞ On utilisera b_global_set au lieu de global_set si on veut être sensible aux retours arrières.
