Module Algorithmes et Raisonnement
S8 - AR - 2A - 2023-2024
BE2 : exercices sur les Listes

1 Introduction

• Selon votre groupe, l'enseignant fera la transition avec le BE précédent.
• Nous réaliserons quelques exercices sur les listes puis utiliserons ces acquis dans ce BE.
• Le cours prochain fera un bilan sur les listes et autres structures composées de Prolog.

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

2 Les Listes

• Vous pouvez en apprendre davantage sur les termes Prolog (Annexes, section 12.2) et sur les listes en Annexes (section 12.5).

• 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 (comme les listes en Python) .

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]]	

☞ 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"**.
    ☞ 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'.
  - — N.B. SWI-Prolog permet de présenter une chaîne de caractères par ses caractères si la chaine est construite avec des anti-apostrophes :
    Ex : **write(‘Ecole Centrale‘)** % notez bien les anti-apostrophes
    ➛ Affichera la liste des codes ASCII [69,99,111,108,101,32,67,101,110,116,114,97,108,101]

- 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 !

3 Manipulation des listes

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

• 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 connaître le nombre d'éléments d'une liste.

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

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++).

4 Exemples et exercices

Nous étudions ci-dessous les prédicats suivants :

- **est_liste/1** (équivalent au prédéfini is_list/1)
- **somme/2** (équivalent au prédéfini sum_list/2)
- **somme1/2** (qui est la version avec les contraintes de **somme/2**)
  ➛ Le prédéfini SWI Prolog équivalent à *somme1/2* est **sum/3**
- **membre /2** (équivalent au prédéfini **member/2**)
- **concat/3** (équivalent au prédéfini **append/3**)
- **affiche_liste/1** : affichage du contenu d'une liste avec numérotation

4.1 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( [] ).
est_liste( [_Tete | Reste] ) :- est_liste(Reste).

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").
    ?- L = [a, b, c], est_liste(L), writeln(L), display(L) .      % comparer ces 2 affichages
```

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

N.B. : Pour écrire / afficher, on peut aussi utiliser "**format**" ((voir Doc SWI Prolog)). Par exemple :

```
X="Ecole Centrale", format("~s \n",[X]).        % ou
format("Ecole Centrale \n").                    % ou
format("~s \n",["Ecole Centrale"]).

X="Ecole Centrale", Y=12, format("~s ~d \n",[X,Y]).
```

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

4.2 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 calculs numériques utilisant l'évaluateur **is** dans les expressions arithmétiques, Prolog permet de manipuler 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 puis on utilise log(10).

☞ En fit, ce ne sont pas des prédicats, ce sont simplement une facilité d'écriture (dans les expressions arithmétiques) et le mot clef **is** provoque les évaluations.

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

```
somme_qui_ne_marche_pas([], 0 ).
somme_qui_ne_marche_pas([Tete | Reste], Tete + Total) :- somme_qui_ne_marche_pas(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 *somme_qui_ne_marche_pas/2* à votre code et essayer :
?- somme_qui_ne_marche_pas([3,7,2],S)
et expliquer le résultat.

4.3 Refaire la somme avec des contraintes

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

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

☞ L'utilisation de # permet d'exploiter les capacités de Prolog à résoudre des équations (ici linéaires) sur des entiers relatifs ($\mathbb{Z}$).
Par contre, elle souffre de certaines erreurs d'évaluation si la liste contient autre chose que des nombres ; 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 12.1 une version de somme avec accumulateur.

• Le prédéfini **sum/3** de SWI-Prolog réalise le même effet que notre **somme1/3** ci-dessus.

```
?– use_module(library(clpfd)).   % (ou :– use_module(library(clpfd)). si placé dans un fichier)
?– sum([1,5,2], #=, X).       % "#=" : que X soit "égal" à la somme.   On aura X=8,
?– sum([1,5,2], #>, X).       % "#>" que la somme soit supérieure à X. On aura X in 4..sup, 4#=<X. :
?– sum([1,5,a], #>, X).       % Erreur d'évaluation (sur la constante 'a' ? !)
```

☞ Préférez l'utilisation des contraintes :

☞ On utilise # avec d'autres opérateurs pour imposer une contrainte (vs. faire un test).
➙ # est utilisé dans les opérateurs : #=, #>, #<, #>=, #=<, ...

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

5 Autres exercices

Les exercices suivants peuvent être réalisés en séance. L'enseignant vous en dira davantage.

5.1 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 12.3) 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.

5.2 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]). $\rightarrow X \in 2..10$ pourquoi ?

ordonne3([3,2,X]). $\rightarrow X \in -2^{28}..2$ pourquoi ?

5.3 *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 Prolog (la Doc Prolog).

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
```

➷ Le prédéfini *max_list(Liste, LeMax)* donne dans la variable LeMax le maximum d'une liste d'entiers.

6 Exercices supplémentaires sur les listes

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

Écrire et tester les prédicats suivants :

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

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

Voir aussi en section 11.4 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 11.3 où taille est revisité.

6.1 Prédicat membre

• Le prédicat membre/2 :

```
membre(X, [X| Reste]).
membre(X, [Tete, Reste]) :– membre(X, Reste).
```

☞ Remarques sur les warning des variables non utilisées (cf. variables anonymes.)

• Exemple :

```
?– membre(1, [1,2,3]).
?– membre(X, [1,2,3]). % cas d'énumération
```

☞ Voir le prédéfini **member/2.**

6.2 Prédicat concat

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).
```

☞ Remarques sur les warning des variables non utilisées (cf. variables anonymes.)

Posons la question simple :

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

☞ Voir le prédéfini **append/3.**

7 Quelques autres prédéfinis sur les listes

- length/2
- nth0 / 3 et nth1/3
- sort/2
- ... voir la documentation SWI Prolog

• Quelques exemples :

```
?– length([1, a, "ECL", 12.5], Taille).      % donne Taille=4
?– nth0(1,[a,b], X).                         %donne  X = b.
?– sort([z,a,b], L).                         % donne L = [a, b, z].
```

8 Exercices en séance & à rendre

8.1 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).

8.2 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.

8.3 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 ?
Méthode 1- Cette comptabilité peut se faire à travers les passages des paramètres : on dispose en paramètre une liste de quantités restante de chaque ingrédient.

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 apparaître 1 unité restante pour le lait.

Pour ce faire, on "enregistre" (on comptabilise dans les paramètres) 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 continue avec la préparation d'un autre mets M2 (ou peut être même encore M1).

En dehors de gérer/comptabiliser les quantités disponible via les paramètres (ci-dessus), une autre solution 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.

Méthode 2- Une autre manière de réaliser le même effet est d'utiliser des *variables globales* en Prolog.
☞ Attention : cette solution n'est pas standard et ne sera peut-être pas comprise dans d'autres Prologs.

• Voir également la page 254 de la documentation Prolog. Il s'agit des prédicats :

Ces deux prédéfinis utilisent les retours arrières ("b" pour "back-track" = REVERSIBLE)
- ◦ **b_setval(nom, valeur)** % les 2 paramètres doivent être précisés
- ◦ **b_getval(nom, valeur)** % le 1er paramètre est le nom de la "variable globale", le 2e = sa valeur

Ces deux prédéfinis n'utilisent pas les retours arrières ("nb" pour "no back-track" = IRREVERSIBLE)
- ◦ nb_setval(nom, valeur) % les 2 paramètres doivent être précisés
- ◦ nb_getval(nom, valeur) % le 1er paramètre est le nom de la "variable globale", le 2e = sa valeur

Autres :
- ◦ **nb_current(nom, valeur)** % énumérer toutes les variable globales
- ◦ **nb_delete(nom)** % supprimer la variable globale "nom"

Voir un exemple et les détails ci-dessous.

Méthode 3- Enfin et en se plaçant sur le domaine de l'optimisation, une autre solution est de créer
◦ (comme ressources nécessaires) la matrice $Recettes : Mets \times Ingredients$ (tous les mets préparables, tous les ingrédients nécessaires)
◦ (comme ressources disponibles) le vecteur $Disponibles$ contenant les quantités initialement disponible de chaque ingrédient.
◦ un vecteur de mets V_m où $V_m[i] \geq 0$ représentera (dans la solution) combien de fois le mets m pourra être préparé (en compagnie d'autres mets) étant données nos ressources données dans le vecteur $Disponibles$,
◦ et de poser un système d'équations-inéquations à optimiser selon un critère (par exemple, nombre de mets). Dans ce système, on devra avoir :
➛ on a le nécessaire pour préparer $V_m[i]$ fois le met i :
$\forall i : V_m[i] * Recettes[i, j] \leq Disponibles[j] \; \forall j : ingredients$
◦ Pour la fonction objective, on pourra par exemple maximiser $somme(V_m)$

☞ Prolog est capable d'établir et de résoudre ce type de systèmes. On pourra développer cet exemple quand on aura étudié les contraintes.

Méthode 4- avec **assert/1**, **recorda/3** et **retract/1**. Voir la documentation SWI Prolog.

8.4 Méthode 2 : détails de la gestion des quantités

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

SWI Prolog permet de manipuler des variables *globales* (quasi similaire aux langages tels que Python/C/C++).
Il suffit pour cela d'affecter la valeur initiale de la quantité Q1 à l'ingrédient I1 par le prédéfini **b_setval(I1, Q1).**

Par exemple, on écrira **b_setval(lait,3).** pour associer la valeur 3 à l'identifiant 'lait'.

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

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

☞ Vous l'avez remarqué : on n'écrit pas *b_setval(I1, Q1 - Q1_1)*
mais **Le_reste #= Q1 - Q1_1** ou **Le_reste is Q1 - Q1_1** suivi de **b_setval(I1, Le_reste)**
car Prolog n'évalue pas les arguments d'appel d'un prédicat.

Pour la cohérence, on fera attention à utiliser *b_getval* après avoir fait *b_setval*.

Vous pouvez utiliser **b_setval/2** ou **nb_setval/2** pour affecter une valeur à un identifiant (on dit 'atome' en Prolog). De même pour **b_getval/2** ou **b_getval/2**

☞ Le "b" de **b_setval/**2 veut dire "back-trackable". La version **nb_setval/2** (et nb_getval/2) ne supporte pas les retours arrières : en cas de retour arrière, l'ancienne valeur associée n'est pas restaurée.

• Un intérêt avec *b_setval* est qu'en cas d'échec ou de retour arrière, *b_setval* peut restituer l'ancienne valeur de la variable. Par contre, avec *nb_setval* (qui va de paire avec *nb_getval*), en cas de retour arrière, la valeur de l'identifiant n'est pas restaurée.

☞ Dans le sujet des recettes avec la gestion des quantités, si vous utilisez *b_setval* et *b_getval*, ne faites 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 repas en une seule fois et pour tous les ingrédients d'un repas, et non pas au fur et à mesure des utilisations de chaque ingrédient.

➛ Voir les exemples en page suivante.

☞ Voir aussi *setarg/3* (et *getarg/3*) ainsi que *nb_setarg* et *nb_getarg*. Ces prédéfinis servent à réaliser le même effet mais différemment.

8.5 Important

La documentation SWI-Prolog précise à propos de *b_setval/2* :
b_setval(+Name, +Value) associate the term Value with the atom Name or replace the currently associated value with Value. On backtracking the assignment is reversed.
If the variable Name did not exist before calling b setval/2, backtracking causes the variable to be deleted.

Ce qui veut dire : si vous utilisez des variables globales avec *b_setval/2*, bien noter que par le mécanisme-même de retours-arrières, ces variables n'existeront plus à la fin des retours arrières.

Par contre, les variables globales existeront après leur exploitation si *nb_setval/2* est utilisé.
L'exemple ci-dessous montre ce propos.

```
?– b_setval(toto, 12), b_getval(toto, X).
==> 12
?– b_getval(toto, X).
==>  ERROR: variable 'toto' does not exist
```

Par contre :

```
– nb_setval(tata, 12), b_getval(tata, X).
X = 12.

?– b_getval(tata, X).
X = 12.
```

☞ Pour éviter l'erreur dans le cas de *b_setval/2*, on peut au préalable faire exister la variable à l'aide de *nb_setval/2*. Par exemple :

```
?– nb_setval(tata, []), b_setval(tata, 12), b_getval(tata,N). % tata existera après retour arrière
N = 12.

?– b_getval(tata, X).
X = [].
```

Ci-dessous, deux exemples d'utilisation des variables globales.

- Exemple 1 : un exemple simple d'utilisation des variables globales.
- Exemple 2 : utilisation des variables globales dans le problème de sac à dos.

8.6 Exemple 1 de gestion des variables globales

• Un exemple de manipulation des variables globales (la Méthode 2 ci-dessus) :

Dans cet exemple, **go1/1** montre que si un prédicat modifie la valeur d'une variable globale (par *b_setval(Var_globale, New_val)*) puis échoue, on récupère l'ancienne valeur de la variable globale (on est revenu en arrière) puisqu'on avait demandé à Prolog (par le "b" devant les noms des prédicats de gestion des variables globales) que ces modifications soient RÉVERSIBLES.

```
go1 :-
    b_setval(toto, 250),                    % toto=250
    b_getval(toto, Valeur1),                % toto vaut 250
    format("toto vaut actuellement ~d \n", [Valeur1]),

    predicat_qui_change_la_var_globale_avec_backtrack_puis_echoue(toto),
    format("Après un échec, voyons la valeur de toto (normalement, la modification de toto est annulée) \n"),
    b_getval(toto, Valeur2),
    format("toto vaut maintenant ~d \n", [Valeur2]).

% Ce prédicat va d'abord modifier la valeur de la variable globale mais de manière REVERSIBLE puis échouera

predicat_qui_change_la_var_globale_avec_backtrack_puis_echoue(Var_globale) :-
    b_getval(Var_globale, Valeur),
    New_val is Valeur + 10,
    b_setval(Var_globale, New_val),
    fail.
predicat_qui_change_la_var_globale_avec_backtrack_puis_echoue(_).
```

De manière symétrique, dans **go2/0**, on modifie la valeur de notre variable globale mais de manière IRREVERSIBLE (en utilisant "nb" devant les noms des prédicats concernés). Dans ce cas, lors des retours arrières, la variable globale NE RÉCUPÈRE PAS son ancienne valeur.

```
%-------------------------------------------------------------------
go2 :-
    nb_setval(titi, 150),
    nb_getval(titi, Valeur1),
    format("titi vaut actuellement ~d \n", [Valeur1]),

    predicat_qui_change_la_var_globale_sans_backtrack_puis_echoue(titi),
    format("Après un échec, voyons la valeur de titi (normalement, la modification de titi persiste) \n "),
    nb_getval(titi, Valeur2),
    format("titi vaut maintenant ~d \n", [Valeur2]).

% Ce prédicat va d'abord modifier la valeur de la variable globale mais de manière IRREVERSIBLE  puis échouera

predicat_qui_change_la_var_globale_sans_backtrack_puis_echoue(Var_globale) :-
    nb_getval(Var_globale, Valeur),
    New_val is Valeur + 10,
    nb_setval(Var_globale, New_val),
    fail.
 predicat_qui_change_la_var_globale_sans_backtrack_puis_echoue(_).

%-------------------------------------
% Essais : par go1, on récupère la valeur initiale de la variable globale dont le nom est "toto".
% Mais par go2, on ne récupère pas la valeur initiale de la variable globale dont le nom est "titi".
?- go1.
toto vaut actuellement 250
Après un échec, voyons la valeur de toto (normalement, la modification de toto est annulée)toto vaut maintenant 250
true.

?- go2.
titi vaut actuellement 150
Après un échec, voyons la valeur de titi (normalement, la modification de titi persiste) titi vaut maintenant 160
true.
```

8.7 Exemple 2 de gestion des variables globales

1Il s'agit du problème de remplir un sac à dos d'un certain volume avec des objets qui ont chacun un volume.
Le principe est simple : on prend un objet non encore choisi (les objets déjà sélectionnés sont gérés par la liste "deja_ramasse" qui est une variable globale), le volume de cet objet doit être inférieur ou égal au volume restant (géré par la variable globale "v_disponible").
Le méthode du remplissage est une méthode Gloutonne. On ne cherche donc pas à optimiser le volume, on prend un objet quelconque dès que possible.

```
:– use_module(library(clpfd)).

volume(chandelier, 5).
volume(collier, 1).
volume(television, 25).
volume(ordinateur, 20).

% remplir le sac de volume Volume_limite
% On créer des vars globales :
% v_disponible pour le volume encore V_disponible
% deja_ramasse : ce qu'on a déja ramassé
remplir(Volume_limite, Liste) :–
    b_setval(v_disponible, Volume_limite),
    b_setval(deja_ramasse, []),
    remplir(Liste).

% On prend un nouvel objet :
% Le nouvel objet doit avoir un volume inférieur au volume restant et ne doit pas être déjà ramassé
% Si ok, mettre à jour les 2 variables globales.
remplir([Objet | Objets]) :–
    b_getval(v_disponible, V_disponible),
    b_getval(deja_ramasse, Deja_ramasse),
    volume(Objet, V_objet),
    not(member(Objet, Deja_ramasse)),
    V_disponible #>= V_objet,
    Vol_restant #= V_disponible – V_objet,
    b_setval(v_disponible, Vol_restant),
    append(Deja_ramasse, [Objet], New_Deja_ramasse),
    b_setval(deja_ramasse, New_Deja_ramasse),
    remplir(Objets).  % remplir le reste
remplir([]).

/*
?– remplir(50, L).
L = [chandelier, collier, television] ;
L = [chandelier, collier, ordinateur] ;
L = [chandelier, collier]
...
*/
```

9 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'.

9.1 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.

• 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) :

N.B : Pour certains des question sci-dessous, vous aurez besoin des prédicats prédéfinis *sort/2* et *findall*/3 (voir ci-dessous),

1- Les auteurs ayant écrit un titre identique

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

3- Les livres d'un auteur dont le prix est inférieur à un prix donné.

4- Les livres d'un auteur dont le prix est supérieur à un prix donné.

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

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

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

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

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

10- 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.

9.2 Quelques prédicats utiles

☞ Voir également le fichier Details-predicats-utiles.pdf.

• **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_qui_trouve_des_réponses,
        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).
```

☞ Si le but donné en 2e argument de *findall/3* n'a pas de solution, la liste (3e argument) sera vide.
☞ Si le but donné en 2e argument de *findall/3* n'existe pas, une erreur est provoquée.
Exemple 3 :

```
foo(a, b, c).   foo(a, b, d).
foo(b, c, e).   foo(b, c, f).   foo(c, c, g).

?– findall(C, foo(A,B,C), L).
L = [c, d, e, f, g].

?– findall(C, foo(x,B,C), L).
L = [].

?– findall(C, foo(A,B,C,D), L).  ==> Erreur     foo/4 n'existe pas.
```

• **bagof/3** et **setof/3** : comportement similaire à *findall/3*.
bagof/3 accepte le quantificateur existentiel noté par ^. La présence d'une variable avec ce quantificateur devant une variable permet de ne pas donner de valeur à cette variable.
setof/3 est comme *bagof/3* ; simplement, les résultats sont un ensemble au lieu d'une liste.
Exemple :

```
?– bagof(C, foo(A, B, C), Cs).
A = a, B = b, C = G308, Cs = [c, d] ;
A = b, B = c, C = G308, Cs = [e, f] ;
A = c, B = c, C = G308, Cs = [g].

?– bagof(C, A^foo(A, B, C), Cs).
A = G324, B = b, C = G326, Cs = [c, d] ;
A = G324, B = c, C = G326, Cs = [e, f, g].
```

Voir https ://www.swi-prolog.org/pldoc/man ?predicate=bagof/3 pour davantage d'exemples de *bagof/3*.

• **between/3** : de la forme **between(binf, bsup, Var)** qui signifie : Var prend les valeurs $binf..bsup$

$between(binf, bsup, Var) \Rightarrow$ Var prend des entiers dans [inf,sup]

Exemple :

```
?– between(1,3, X).
    X = 1   puis X=2 et enfin X=3.
```

Les deux méta-prédicats suivants ont un comportement **assez similaire**.
• **forall/2** : de la forme **forall(p(X), q(X,Y))** dont la signification est :

$$forall(p(X), q(X,Y)) \quad \equiv \quad \forall X \exists Y p(X) \implies q(X,Y)$$

• **foreach/2** : de la forme **foreach(p(X), q(X,Y))** dont la signification est :

$$forall(p(X), q(X,Y)) \quad \equiv \quad \exists Y \forall X p(X) \implies q(X,Y)$$

Exemples de *forall/3*

```
% pour tout X, Il existe Y,  (X dans  1..3  –>  (Y : 1..3 et X =< Y))    '–>' veut dire "implique"
% Autrement dit : Pour tout X , X :: 1..3 –> Il existe Y :: 1..3, Y >= X

?– forall(between(1, 3, X), (between(1, 3, Y), X=<Y, writeln([X, Y]))).
[1,1]
[2,2]
[3,3]
true
```

☞ Ici, on a : $\forall X, X \in 1..3 \implies \exists Y \in 1..3, Y \geq X$
et là, $X = 1$ conduit à trouver $Y = 1$ telle que $Y \geq X$ puis idem pour $X = 2$ puis $X = 3$.

Exemples de *foreach/3*

```
% Il existe Y, pour tout X (X dans  1..3  –>  (Y : 1..3 et X =< Y))    '–>' veut dire "implique"
% Autrement dit : Il existe Y, Pour tout X , X :: 1..3 –>  Y :: 1..3, Y >= X

?–foreach(between(1, 3, X), (between(1, 3, Y), X=<Y, writeln([X,Y]))).
[1,1]
[_26920,2]
[2,2]
[_26920,3]
[2,3]
[3,3]
Y = 3.
```

☞ Dans *forall(between(1, 3, X), (between(1, 3, Y), X=<Y))*., on a :
$\forall X, X \in 1..3 \implies \exists Y \in 1..3, Y \geq X$
et là, $X = 1$ conduit à trouver $Y = 1$ telle que $Y \geq X$

☞ Voir le supplément sur ces deux prédicats dans le fichier Details-predicats-utiles.pdf.

9.3 Circuits en Prolog

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

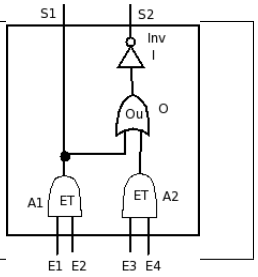
Modélisation et évaluation des circuits :
on modélisera des circuits logiques simples composé de portes ET, OU, OUX, OR, INV.

Ci-dessous, on a les tables de vérités des portes élémentaires utilisés :

```
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]).
```

On se propose de modéliser le circuit donné à droite de la figure ci-dessous par :

```
:- use_module(library(clpfd)).

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]).
```



Quelques questions à poser :

```
circuit('C1', [1, 1, 0, 1], [0, 1]).        ➞ false
circuit('C1', [1, 1, 0, 1], [1,0]).         ➞ true
circuit(X, [ 1, 1, 0, 1], [S1,S2]).         ➞ X ='C1', S1 =1, S2 =0
circuit(X, [1, 1, 0, 1],S).                 ➞ X ='C1', S=[1,0]
circuit(X, [E1, E2, E3, E4], [0, 1]).       ➞ Plusieurs réponses
circuit('C1', [E1, E2, E1, E4], [0, 1]).    ➞ Plusieurs réponses
circuit('C1', [E1, 0, E1, E4], [0, 1]).     ➞ Plusieurs réponses
circuit('C1',E, [0, 1]).                    ➞ ....
```

A noter : Les prédicats *and/3, or/3* et *inv/3* ci-dessus n'utilisent pas les contraintes. Ils peuvent être écrits à l'aide des contraintes (*clpfd*) de la manière suivante :

```
and(_,[E1,E2], [S]) :-
    [E1,E2] in 0..1,
    S #= E1 #/\ E2.
or(_,[E1,E2], [S]) :-
    [E1,E2] in 0..1,
    S #= E1 #\/ E2.
inv(_, [E], [S]) :-
    E in 0..1,
    S #= #\(E).
```

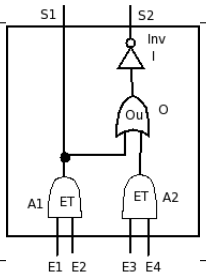
• **<u>Exercice 1 :</u>**
Supprimer les prédicats "and", "or" et "inv" et modéliser ce même exercice avec les contraintes booléennes.

Indications :
L'utilisation des contraintes booléennes permet ici de ne plus avoir besoin des tables de vérités.
Les opérateurs logiques définis dans la librairie *clpb* sont différents de ceux de *clpfd*.
Consulter la page *https ://www.swi-prolog.org/pldoc/doc/_SWI_/library/clp/clpb.pl*.

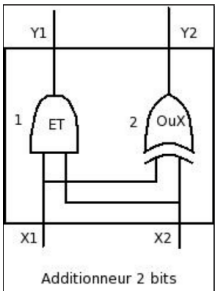
On remplacera le ET logique (noté dans $\mathbb{Z}$ par $\#/\backslash$) par "*", le OU logique (noté dans $\mathbb{Z}$ par $\#\backslash/$) par "+", l'équivalence $A \Leftrightarrow B$ par **sat(A = := B)** et l'implication $A \Rightarrow B$ devient **sat(A =< B)**.

Vous pouvez donc utiliser ce prédicat qui modélise notre circuit 'C1' :

```
:- use_module(library(clpb)).

circuit('C1', [E1,E2,E3,E4], [S1,S2]) :-      %'C1' est le nom
  sat(S1 = := E1 * E2),                       % S1 ⇔ (E1 ∧ E2)
  sat(S3 = := E3 * E4),                       % idem
  sat(S4 = := S1 + S3),                       % S4 ⇔ (E1 ∨ E3)
  sat(S2 = := ~ S4).                          % S2 ⇔ ¬S4
```



Interroger ce prédicat comme ci-dessus et vérifiez / notez les réponses.

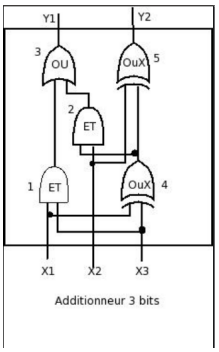


Exercice 2 : Soit l'additionneur 2 bits :

1- Présenter la spécification exécutable de ce circuit en Prolog.
2- A l'aide de ce circuit, réaliser un additionneur 3 bits en montant en série deux additionneurs 2 bits.

N.B. : dans l'exercice suivant, on réalisera un additionneur 3 bits sans monter en série 2 additionneurs de 2 bits.

Exercice 3 : Soit l'additionneur 3 bits directement réalisé avec des portes élémentaires :
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.

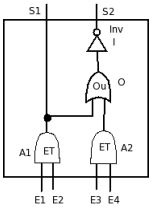
➙ Pour réaliser ces circuits, vous avez le choix d'utiliser *clpfd* ou *clpb*.

Exercice 4 (bonus) :
Le prédicat suivant permet de détecter une pannée unique dans un circuit (circuit 'C1' ci-dessus). Pour ce faire, on a utilisé *clpfd*. Lire aussi l'encadré qui suit le prédicat ci-dessous.

```
:- use_module(library(clpfd)).

panne_circuit('C1', [E1,E2,E3,E4 ], [S1,S2 ], [P1,P2,P3,P4]) :-  %'on utilise l'équivalence (<=> )
   #\P1   # <=> (S1 #<=> E1 #/\ E2),     % le ET #/\
   #\P2   # <=> (S3 #<=> E3 #/\ E4),
   #\P3   # <=> (S4 #<=> S1 #\/ S3),     % le OU #\/
   #\P4   # <=> (S2 #<=>  #\(S4)).       % la négation  #\
```

☞ Que veut dire l'expression
$\#\backslash P1 \quad \# <=> (S1 \#<=> E1 \#/\backslash E2)$?

Elle dit : si P1 n'est pas en panne (exprimée par $\#\backslash P1$), alors elle réalise sa fonction qui est (S1 #<=> E1 #/\ E2).



1- Tester ce prédicat avec :
`panne_circuit('C1', [1,1,1,0 ], [0,1], [P1,P2,P3,P4]).`
et déduire que P1 est en panne.

2- Transformer ce prédicat en version **clpb**.

9.4 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

9.4.1 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*.

10 Coupe choix

10.1 Prédicat concat et le coupe choix

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

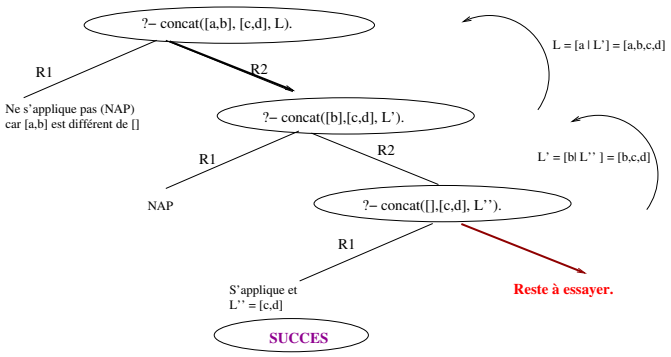
```
(R1)    concat([], L, L).
(R2)    concat( [X | Reste], L, [X | L2] ) :-        %
              concat(Reste, L, L2).

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=[]

10.2 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.

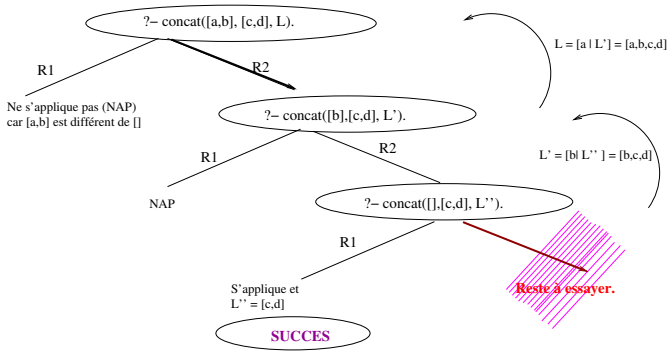
• *Se contenter d'une seule réponse dans concat/3 :*

➙ 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.

10.2.1 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,!.*

11 Pour aller plus loin

11.1 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.*

11.1.1 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.

11.1.2 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 Prolog, la contrainte négation s'écrit "#\" et implique s'écrit "#=>" ("#<=>" est équivalent)

11.2 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).
```

11.3 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).

11.4 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.

11.5 Nème élément d'une liste

Le prédéfini **nth1(Indexe , Liste, Element)** permet de trouver l'élément du rang Indexe (en commençant par 1) dans L.
nth0(Indexe , Liste, Element) commencera à 0.
nth1(Indexe , Liste, Element, Reste) commencera à 1 et mettra dans "Reste" le reste de la liste (après le Nème élément)
Exemple : ?-nth1(2, [a,b,c,d], X). % → X=b.

?- nth1(2,[1,2,3], V, R).
➙ V = 2,
➙ R = [1, 3].

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

12 Annexes

12.1 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).

12.2 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 Prolog 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

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

12.3 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.

12.4 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 Prolog)

☞ **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

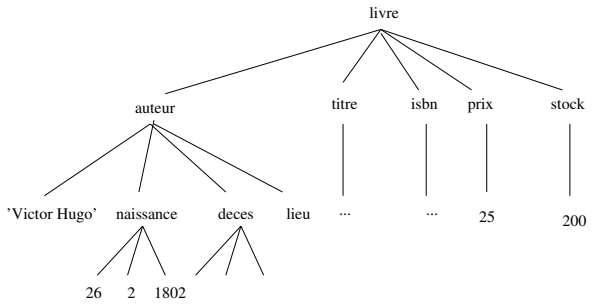
☞ Voir la doc. Prolog pour plus de détails.

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

12.5 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), ...
).



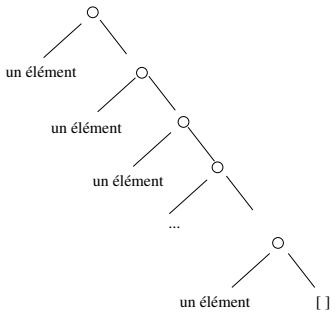
➙ cet arbre est quinternaire.

• 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.

12.6 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.