Introduction à la

Programmation en Logique

Eléments de Programmation

en Prolog

A. S.Saidi

Dépt. M.I.S.

Ecole Centrale de Lyon

1991..2006

Avertissement :

Ce polycopié est une introduction au langage Prolog.

Cette présentation se limite volontairement à un noyau "standard" du langage, appelé C-Prolog ou encore Prolog-DEC10.

Différentes implantations de Prolog proposent des prédicats supplémentaires qui élargissent ce noyau au détriment de la portabilité.

Dans sa version initiale, ce polycopié présentait un support de TD sous la forme de transparents présentant le langage VM/Prolog sous VM/CMS de la machine IBM 43-81. La révision actuelle conserve cette forme de copies de transparents.

Conseils de lecture :

Afin d'en faciliter l'utilisation, ce polycopié comporte une indication de degré de difficulté sur certains paragraphes ainsi que sur certains exercices. Ces paragraphes sont entourés entre ❞ ❝. Ces sections peuvent être lues dans une deuxième lecture.

Table des matières

1.Introduction

Comparaison succincte des paradigmes impératif, fonctionnel et relationnel :

❏ *Programmation impérative*

Ce paradigme est caractérisé par l'affectation remettant à jour l'état de la mémoire. Un exemple caractéristique est l'instruction X:=X+1.

- Pour calculer des résultats, on décrit les objets (le "Quoi") ainsi que les traitements (le "Comment") portant sur ces objets.
- L'exécution d'un programme réalise une fonction des entrées vers les sorties. Cette fonction est spécifiée par le programme qui décrit la façon dont il faut calculer les valeurs des sorties. Pour un tuple de valeurs en entrée, on obtient toujours le même tuple de valeurs en sortie.

- Une variable est initialisée par affectation. Elle est modifiée au cours du calcul pour aboutir à une valeur finale.

Des exemples de langages impératifs sont ADA, Pascal, C, Modula ...

❏ *Programmation fonctionnelle:*

- Le composant de base est la fonction (au sens mathématique du terme)
- La structure de contrôle essentielle est l'application de fonctions
- Comme dans le cas de la programmation impérative, on décrit le traitement.
- Les paramètres des fonctions étant en "entrée", il n'y a pas d'affectation au sens mise-à-jour des paramètres. La seule sortie est la valeur délivrée par la fonction.
- On peut manipuler les fonctions comme des données; les passer en paramètre; écrire une fonction qui renvoie une fonction...
- L'exécution d'un programme réalise une fonction des entrées vers la sortie.
  Cette fonction est spécifiée par le programme qui décrit la façon dont il faut calculer les valeurs des sorties.

Exemple: ML, Lisp...

❏ *Programmation relationnelle (ou déclarative, mise en oeuvre en Prolog) :*

- On décrit le "Quoi". Le "comment" est figé par un certain algorithme fixe (cet algorithme implante le principe de résolution).

- Une variable est une inconnue algébrique dont on cherche la valeur.
  Lorsqu'une variable recoit une valeur, on dit qu'elle est instanciée.
- On ne peut pas modifier la valeur des variables instanciées.
- Il n'y a pas d'affectation (au sens mise à jour des variables ou des paramètres par des opérations telles que X:=X+1).
- L'exécution calcule une instance de la relation décrite par le programme.

Exemples de relations (*écritures Prolog à quelques détails syntaxiques près*)

Relation	*Interprétation*
pere(jean, paul).	*jean est le père de paul*
fils(X, Y) :- pere(Y, X).	*X est le fils de Y si Y est le père de X*
origine((X,Y)) :- X=0 , Y =0.	*le couple (X,Y) est l'origine si X=Y=0*
dans_cercle((X,Y), R) :- $X^2 + Y^2 < R^2$.	*le point (X,Y) est dans le cercle de rayon R si $X^2 + Y^2 < R^2$*
fermat(A,B,C,N) :- **enum(A) , enum(B) , enum(C) ,** **$A^N + B^N = C^N$.** **enum(X) :- X∈ {1,2,... }.**	*trouver les entiers A,B,C,N tels que $A^N + B^N = C^N$*
entier(0).	*0 est un entier*
entier(succ(N)) :- entier(N).	*si N est un entier alors son successeur l'est également*

Prolog (Début des 70 , Marseille)

- Description des propriétés/relations sur les objets du domaine de discours par :
  - la description des connaissances simples par des faits avérés;
  - la description des connaissances déductives par des règles

- Soumission de questions relatives à cette description produisant des réponses calculées selon un algorithme fixe pré défini (moteur Prolog)

Exemple:

derive(X+Y, Z, U) :- derive(X, Z, A) , derive(Y, Z, B) , U=A+B.

❾ Deux lectures possibles de cette description:

1• Lecture Déclarative:
La dérivée de la somme X+Y est A+B, la somme des dérivées de chacun des composants X et Y

2• Lecture Procédurale:
Pour calculer la dérivée de X+Y, calculer A la dérivée de X puis calculer B la dérivée de Y et construire le terme A + B représentant la somme des deux.

❾ De même, on peut avoir deux lectures de :

entier(succ(N)) :- entier(N).

1• *Si N est un entier alors son successeur est un entier*
2• *Pour calculer un entier, calculer N, son prédécesseur.*

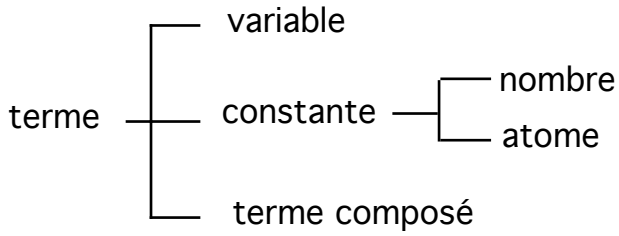
Grâce au dispositif calculatoire de Prolog, le programmeur utilisant ce langage est grandement libéré des considérations procédurales pour se concentrer sur l'aspects déclaratif des descriptions. Ainsi, Prolog est un formalisme exécutable de haut niveau où la résolution d'un problème consiste à présenter une théorie (par des axiomes) et de demander (par les questions) la démonstration des théorèmes dans cette théorie. Le domaine d'applications de Prolog est assez large. Il est notamment utilisé en Intelligence Artificielle, Traitement de langues naturelles, Bases de données déductives... L'étude de différentes extensions du langage constitue un domaine de recherche très développé notamment en France.

2.Eléments du langage Prolog

Un programme Prolog manipule des objets appelés *termes*. Un terme Prolog dénote différentes formes lexicales.

Terme :

I - constante
II - variable
III - terme composé



I- Constante

Atomes et nombres

• I.1 Atome

- Objet individuel qui ne représente que lui même. Il ne peut être égal à un autre atome que s'il s'écrit exactement de la même façon.

- Le nom d'un atome commence par une lettre minuscule.

Exemples : **pierre** **x24** **toto** **'ceci est un atome presente entre quotes'**

☞ ⇒ deux atomes de même nom représentent le même objet.

• I.2 Nombres

Entier ou réel

Entier: **123**
Réel: **3.14**

Remarque : La constante chaîne de caractères n'existe pas en Prolog standard. On peut cependant mettre entre guillemets une séquence de caractères comme dans "ecole". Cette chaîne est en fait équivalente à une liste d'entiers notée entre [] (ici [101,99,111,108,101]) représentant les codes ASCII des caractères de la chaîne.

II- Variables

- Inconnue algébrique pour laquelle on cherche une valeur.
- Tout atome qui commence par une lettre majuscule ou par '_' est une variable.
- La variable réduite à _ est une variable dite *anonyme*. La valeur d'une telle variable n'est pas accessible.

Exemples: **X _toto _12056 _**

III- Terme composé

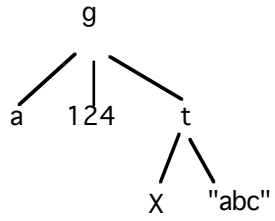
Construit à partir :

- d'un atome représentant un symbole fonctionnel (appelé foncteur)
- de 0 à n arguments qui sont eux mêmes des termes

Représentation canonique d'un terme composé :

Un terme composé décrit un arbre dont la racine est le foncteur et dont les fils sont les arguments.

g(a,124,t(X,"abc"))



Exemples de termes composés :

- **pere(jean,paul)**
- **succ(succ(X))**
- **personne(dupont,**
  adresse(lyon,thiers,32),
  fonction(mis, enseignant),
  salaire(2300, 23008))

Remarque syntaxique : *%texte jusqu'à la fin de ligne* est un commentaire.
/ texte sur plusieurs lignes */* est un commentaire.

3.Faits et questions élémentaires

Les connaissances sur le problème décrit sont exprimées sous forme de **faits** et de **règles**. Un **programme** est constitué de l'ensemble de faits et de règles utilisés pour décrire ce problème.

Un **fait** est une affirmation simple et sans condition préalable notée sous la forme d'un terme fonctionnel. Ce qui donne à ce terme fontionnel le status d'un fait est le sens attaché à ce terme ainsi que l'utilisation que l'on en fait dans le programme. Par exemple, l'on exprime la connaissance "jean est le père de pierre" par *pere(jean, pierre)* .

De même, *entier(succ(0))* est un fait qui exprime la connaissance " le successeur de 0 est un entier". Ce fait utilise le terme fonctionnel *succ(0)* qui lui, n'est pas un fait dans ce contexte.

Une **question** simple est un terme fonctionnel que l'on fait précéder de "?-" ou de ":-".
Par exemple, *?-entier(0)* pose la question : "est-ce que 0 est un entier ?".

On se donne la base de faits suivante :

```
entier(0).

pere(jean, paul).
pere(jean, helene).
pere(jean, pierre).
pere(pierre, vincent).
pere(jacques, marie).
pere(olivier, mark).

epouse(sylvie, jean).

il_pleut.

homme(jean).

patron(emile, henri).
patron(bordet, jean).

eleve(durand, x25).
```

❒ *Questions simples (question dont la réponse est oui-non) :*

?- pere(jean,pierre) ❷ succès

Explication du résultat : on cherche dans la base un fait (un terme fonctionnel) dont la racine est "pere" avec deux arguments. L'on vérifie ensuite que les arguments de ce fait correspondent (sont identiques) à ceux de la question posée.

?- pere(jean,marie) ❷ échec

Explication : Comme ci-dessus.

?- il_pleut(maintenant) ❷ échec

Explication : On cherche dans le programme un fait (un terme fonctionnel) à un argument dont la racine est "il_pleut". Cette recherche échoue. Le fait "il_pleut" (sans argument) n'est pas sélectionné car celui-ci est sans argument.

4.Questions complexes

❒ *Conjonction*

Une conjonction est vraie si tous ses composants le sont.

?- pere(jean,paul) , pere(jean,pierre). ❷ succès

Explication : On procède de la même manière que précédemment mais on ne conclut sur un succès que si tous les éléments de la conjonction sont évalués vrai.

❒ *Disjonction*

La disjonction sera vraie si l'un de ses composants est vrai.

?- pere(jean,paul) ; homme(jean). ❷ **➔ succès**
⇑ ⇑
vrai vrai

?- pere(jean,carlos) ; homme(jean). ❷ **succès**
⇑ ⇑
faux vrai

?- pere(jean,carlos) ; homme(marie). ❷ **échec →**
⇑ ⇑
faux faux

Explication : On procède de la même manière que précédemment pour traiter complètement le premier composant de la disjonction puis, on traite le second composant puis le suivant...

5.Idée intuitive de l'algorithme du moteur Prolog

Formalisons le comportement du moteur Prolog pour trouver les réponses à une question sans variable traitées dans les exemples précédents.

Soit **P** la base de faits (le programme)
B=b_1 , ... , b_n la question (le but)

La fonction suivante est activée pour répondre à la question.

```
Fonction resoudre (B : but) retourne booléen =

Si B = vide alors retourne (vrai)      -- n=0 dans B=b1 , .... , bn. Plus rien à prouver
Sinon
      considérer le premier sous-but (b1) dans B; b1 de la forme f(a1,...,am)
①     LC = l'ensemble des faits de P de la forme f(a'1, ...,a'm)
      Tant que LC <> vide faire
          choisir C le premier élément de LC
          LC = LC - {C}
②         Si ai = a'i  , i = 1...m alors
              B' = b2 , ... , bn
              Si resoudre(B') alors retourne (vrai) Fin si
          Fin si
      Fin Tant que
      retourne (faux)
Fin si
Fin resoudre
```

L'algorithme est non-déterministe; c'est à dire qu'il trouve toutes les réponses à la question posée. Ce non-déterminisme est mise en oeuvre par la boucle Tant que. Ainsi, si la base contient plusieurs fois le même fait, on aura autant de fois de réponses. Nous verrons l'intérêt de ce non-déterminisme lors que nous considérerons les variables.

La confrontation des arguments au point ② est une forme très simplifiée de l'unification que nous développerons plus loin.

La constitution de LC au point ① est une action de "pré-unification" où l'on constitue un ensemble de faits candidats pouvant conduire à une réponse.

6.Utilisation des variables

❒ *Variables dans les questions*

L'utilisation des variables dans les questions permet de demander toutes les valeurs pour celles-ci telles que les réponses à la question soient vraies.

?- pere(jean, X).

③ X= paul (première réponse)

③ X= helene

③ X= pierre

Le sens de la question:

- peut on trouver une valeur pour X telle que la formule résultante soit dans la base ?

ou encore

- existe-t-il une valeur C pour X telle que pere(jean, C) soit dans la base ?

Explication : On cherche dans la base un fait (un terme fonctionnel) dont la racine est “pere” avec deux arguments. L’on vérifie ensuite que le premier argument de ce fait est égal à "jean". Dans ce cas, X prend la valeur du second argument de ce fait. On dit que X est **instancié** à cette valeur.

Le moteur Prolog essaie de trouver toutes les valeurs pour C telles que *pere(jean, C)* soit dans la base. C’est comme si l’on reposait la même question tant qu’il y a un succès. Les autres réponses s'obtiennent en considérant les autres faits de la base et en suivant la même démarche ci-dessus.

Une variable figurant dans une question est quantifiée existentiellement

❒ *Utilisation des variables anonymes*

Pour savoir si l'individu "vincent" a un père sans s'intéresser à celui-ci :

?- pere(_, vincent) ❷ succès

Le sens de la question:

Existe-t-il un individu C (dont le nom ne nous intéresse pas) tel que pere(C, vincent) soit dans la base ?

❒ *Variables dans les faits*

pere(adam, X). % Pour tout X, adam est le père de X.

patron(bordet, _).

On peut considérer un fait avec variable représentatif d'un ensemble (potentiellement infini) de faits.

Une variable figurant dans un fait est quantifiée universellement

❒ *Variables dans les conjonctions*

?- pere(X, pierre) ,pere(X, helene). ❷ X= jean

↑ ↑

X représente le même individu

Le sens de la question:

Existe-t-il une valeur c pour X telle que pere(c, pierre) et pere(c, helene) soient dans la base ?

Selon le principe dit de "remplacement uniforme", les variables identiques dans une conjonction doivent prendre la même valeur.

❒ *Variables dans les disjonctions*

?- pere(X, Y) ; patron(X, _).

La question ci-dessus obtient toutes les valeurs pour X et Y telles que *pere(X, Y)* réussisse puis, toutes les valeurs pour X telles que *patron(X ,_)* réussisse.

Dans une disjonction telle que $D_1 ; D_2 ;... ; D_n$, il n'y a aucun lien entre les variables syntaxiquement identiques qui apparaissent dans chaque partie D_i .

On peut donc reformuler la question ci-dessus par **?- pere(X, Y) ; patron(Z, _).**

7.Résolution avec variables

Lorsqu'une question telle que *?-pere(X,Y)* est posée, on cherche dans la base de faits un fait de la forme *pere(α,β)* auquel cas X est *instancié* à l'argument α et Y à l'argument β. Ces liens sont établies dans une table de symboles θ constituée d'un ensemble de couples *t*/*t*' où *t* est une variable et *t*', soit une variable différente de t soit, une constante. Dans ce cas, on dit que *t* est **unifié** avec *t'*. Pour la question posée ci-dessus, on obtiendrait la table θ={X/jean, Y/paul}.

Si une question obtient un succès, les valeurs des variables figurant dans la question sont extraites de θ puis, éventuellement affichées à l'écran. C'est le processus d'*extraction des réponses*.

On peut donc dire que la démonstration de B=pere(X,Y) réussit et produit θ={X/jean, Y/paul}. La table de symboles θ est appelée une **substitution**. Si l'on applique θ à B que l'on note par θ(B), on obtient *pere(jean,paul)* qui est une **instance** de *pere(X,Y)*. L'application θ(t) remplace uniformément chaque variable de *t* par sa valeur donnée dans θ.

Si l'on recherche une autre réponse pour *pere(X,Y)*; la substitution θ est mise à vide et l'on considère dans la base de faits, le fait suivant celui qui avait donné la réponse précédente.

7.1- Unification

Principe :

Unifier deux termes t1 et t2 revient à trouver des valeurs pour les variables de ces termes telles que t1 et t2 deviennent identiques. Nous avons vu que ce même principe s'applique aux faits et questions; c'est à dire, on unifiera les termes tels que *pere(jean, X)* et *pere(Y, pierre)*. On peut présenter le principe de l'unification par :

- Deux termes identiques s'unifient ;
- Une variable s'unifie avec n'importe quel autre terme ;
- Deux constantes différentes ne s'unifient pas
- Pour t_1 de la forme $\mathbf{f(a_1,...,a_n)}$ et t_2 de la forme $\mathbf{g(a'_1,.., a'_m)}$
  si f≠g ou n≠m alors t_1 et t_2 ne s'unifient pas
  sinon unifier a_i et a'_i pour i=1..m

Pour unifier deux termes t1 et t2 en Prolog, on écrit **t1=t2**.

Nous pouvons maintenant proposer l'algorithme de résolution en tenant compte des variables.

7.2- Algorithme du moteur Prolog

❝ L'algorithme suivant décrit le principe de résolution de Prolog avec variables (sans les termes fonctionnels). Les paramètres de la fonction sont (toujours) en entrées.

Appel : resoudre(B,{}); résultat dans γ

```
θ, θ', σ, γ : substitution
Fonction resoudre (B : but; θ:substitution) retourne booléen =

  Si B = vide alors                -- n = 0 dans B=b1 , .... , bn. Plus rien à prouver
⑩   γ=θ; Retourne (vrai);          -- γ est la solution
  Sinon
    considérer le premier sous-but (b1) dans B;
    LC = l'ensemble des faits de P
    Tant que LC <> vide faire
□     choisir C le premier élément de LC
      LC = LC - {C} ;
      σ= {};
      Si unifier(b1 , C)           -- mise à jour de σ
⑦     Alors
        θ' = θ ° σ;
        B' = θ'(b2 , ... , bn) ;
        Si resoudre(B', θ') alors retourne (vrai)  Fin si
      Fin si
    Fin Tant que
    Retourne (faux)
  Fin si
Fin resoudre
```

Notons les points suivants sur la fonction *resoudre* :

- C'est en ⑩ que l'on peut mettre en place les retours arrières conduisant aux différentes solutions ;
- En □, si C possède des variables communes avec b_1 alors on renomme les variables de C. Ce renommage supprimera la confusion entre la variable X dans la question *pere(X, jean)* et la même variable dans le fait *pere(adam, X)*.

La substitution σ est résultat de la fonction unifier que nous décrivons ci-dessous :

7.3- Algorithme d'unification

σ : substitution
Fonction unifier(t1,t2 : terme) retourne booléen =
Cas
- t1 est identique à t2 ⇒ retourne (vrai)
- t1 est variable ⇒ σ = σ ∘ {t1 / t2}; retourne (vrai)
- t2 est variable ⇒ σ = σ ∘ {t2 / t1}; retourne (vrai)
- t_1 de la forme $\mathbf{f(a_1,...,a_n)}$ et t_2 de la forme $\mathbf{g(a'_1,..,a'_m)}$

⇒ Si f≠g ou n≠m Alors retourne (faux)
Sinon Pour i=1..m
Si Non unifier($\sigma(a_i), \sigma(a'_i)$) Alors retourne (faux);
Fin si
Fin Pour
retourne (vrai)
Fin si
- autres ⇒ retourne (faux)

Fin cas
Fin unifier

• L'operateur "∘" est un opérateur de composition de substitutions. On peut simplifier en disant que l'opération *θ °{t1 / t2}* ajoute le couple {t1/t2} à θ en respectant certaines conditions :

- θ ne contient pas de couples tels que *t/t'* et *t/t"* (les *t* identiques). Ainsi, une substitution définit une fonction où pour t, on obtient une seule valeur t' (ou t").

- θ ne contient pas de couples tels que {X/X}.

- Généralement, une variable t' figurant à droite d'un couple t/t' ne doit pas figurer à gauche d'un autre couple t'/t". Nous reviendrons sur cette condition dans les algorithmes en annexe. ❞

Exemples :

12 =? 15 => échec
X =?eleve => succès, θ={X/eleve}
X =?Y => succès, θ={X/Y}
X =?X => succès, θ={}
Ecole =?"ecole" => succès, θ={Ecole/"ecole"}
'ecole' =?"ecole" => écheccar l'atome 'ecole' ≠ la chaîne "ecole"
pere(jean, X) =?pere(Y, pierre) => succès, θ={X/pierre, Y/jean}
parents(jean, X) =?parents(Y, pierre, marie) => échec pour le nombre d'arguments

Remarque sur les exercices : voir la section 13 pour quelques éléments de programmation.

Exercice-1 : Editer la base de faits suivant, le charger, le modifier et poser des questions.....

Soit la base de faits :

```
entree(salade).
entree(oeufs).
entree(pate).
entree(melon).

plat(poisson).
plat(poulet).
plat(viande).

dessert(glace).
dessert(pomme).
dessert(raisin).
dessert(gateau).
dessert(fromage).
```

Formuler des questions Prolog pour les interrogations suivantes :

- Quelles sont les entrées, plats et desserts ?
- Y a-t-il du fromage en entrée ?
- Est-ce que le gateau est un plat ?
- Est-ce que le menu propose des frites ?
- Quels sont tous les repas possibles que l'on peut composer à partir de ce menu?
- Y a-t-il des légumes dans ce menu ?
- Le melon, est une entrée ou un dessert ?
- D'autres questions ....

8.Règles

Les faits simples permettent de représenter les connaissances de base du monde que l'on décrit.

Les **règles** Prolog permettent de représenter de nouveaux faits déductibles à partir de ces derniers. Une règle est exprimée sous la forme :

f(a_1, a_2, ..., a_n) :- g(b_1, b_2, ...,b_m), ..., h(p_1, p_2, ..., p_l).

tête de la règle *SI* *corps de la règle*

Si le corps de la règle est vrai alors la tête de la règle est vraie. On peut donc lire une règle de la manière (procédurale) suivante : **conclusion SI conditions.**

C'est à dire, si les conditions exprimées par le corps de la règle sont vraies, alors la conclusion exprimée dans la tête de la règle est vraie.

La tête de la règle a la forme d'un terme fonctionnel; le corps de la règle est une conjonction de termes fonctionnels. Ces termes fonctionnels ont un statut patrticulier : ils peuvent être vrais ou faux.

Terminologie : On appelle un tel terme fonctionnel un **littéral**. D'une manière générale, on appelle **clause** une règle ou un fait.

Exemples de règles :

Pour exprimer la connaissance "l'individu X est le grand père de l'individu Y si X est le père d'un individu Z et Z est le père de Y", on écrit la règle

```
grand_pere(X, Y)    :- pere(X, Z) , pere(Z, Y).
```

Cette règle (qui utilise 3 littéraux) nous permet de retrouver les couples <X,Y> tels que "X est le grand père de Y" à partir de la relation "pere" de la base et d'éviter ainsi d'énumérer tous les faits grand_pere(α,β) dans notre programme.

Les règles ci-dessous expriment la donnée suivante : "est considéré comme personne tout élève, enseignant, travailleur, père ou enfant".

```
personne(X)     :- eleve(X).
personne(X)     :- enseignant(X).
personne(Y)     :- travailleur(Y).
personne(Z)     :- pere(Z, _).      % un père est une personne
personne(Z)     :- pere(_, Z).      % un enfant est une personne
```

Un paquet de règles dont les têtes ont le même symbole fonctionnel suivi du même nombre d'arguments est appelé un **prédicat**. Les règles ci-dessus définissent le prédicat personne à un argument (désigné par personne/1).

Dans un **programme** Prolog constitué d'un ensemble de prédicats, l'ordre de définition des prédicats n'a pas d'importance.

❒ *Disjonction dans les règles*

```
parent(X,Y) :- pere(X,Y) ; mere(X,Y) .
```

Dans cet exemple, les variables X et Y de la partie droite de la règle sont liés à celles de la partie gauche (tête), mais pas entre elles.

Remarque:

La disjonction est uniquement destinée à faciliter l'écriture des règles. Par exemple, on peut réécrire :

```
parent(X,Y) :- pere(X,Y) ; mere(X,Y) .
```

En

```
parent(X,Y) :- pere(X,Y) .
parent(X,Y) :- mere(X,Y) .
```

8.1- Règles récursives

La récursivité est la structure de contrôle de base en Prolog permettant la mise en oeuvre du raisonnement par induction.

Intuitivement, un prédicat récursif est un prédicat qui intervient dans sa propre définition. On peut écrire une telle définition en suivant le principe général suivant :

- *Ecrire un cas particulier d'ordre fini k*
- *Ecrire une définition générale de la propriété à l'ordre n > k en supposant la propriété bien définie à l'ordre m < n.*

L'ordre de la propriété est un entier positif, par exemple la longueur d'un chemin dans un graphe, la taille d'une liste, la profondeur d'un arbre...

• Exemple-1 : définition de la fonction factorielle :

- La factorielle de 0 est 1
- La factorielle de n > 0 est n fois la factorielle de n-1.

Dans le calcul de 5!, l'ordre de la propriété est 5.

• Exemple-2 : définition des entiers (axiomes de Peano) :

Dans cette définition algébrique, on se sert de la constante *0* et de l'opérateur *succ* pour générer les termes d'une algèbre représentant les entiers : 0, succ(0), succ(succ(0)) ...

- 0 est un entier
- succ(N) est un entier si N est un entier.

D'où le prédicat *entier/1* :

```
entier(0).
entier(succ(N)) :- entier(N).
```

On peut utiliser ce prédicat pour savoir si un terme est un entier ou bien pour générer tous les entiers (attention, il y a une infinité de solutions).

8.2- Récursivité et induction

❝ La récursivité est une "traduction" algorithmique du raisonnement par un schéma d'induction. Ce schéma a pour but de décrire l'ensemble E d'éléments vérifiant une certaine propriété p : E={x | p(x)} .

Dans un raisonnement inductif, on décrit :

• **la base *B*** : on décide que certains objets appartiennent à E. Ces objets peuvent eux-mêmes constituer un ensemble décrit par un autre schéma d'induction.

• **les règles *R*** : ce sont des règles d'induction (ou de production). Une règle d'arité k (c'est à dire à k arguments) décrit comment, à partir de k objets appartenant à E sous certaine hypothèse, on peut former sans ambiguïté un objet appartenant nécessairement à E. Ces règles peuvent être décrites par un autre schéma inductif.

Un schéma d'induction comporte également une description de clôture qui vérifie que E est le plus petit ensemble (pour l'inclusion) contenant la base et, stable par les règles de production R (on montre l'existence et l'unicité de E).

On peut également vérifier par construction qu'un objet appartient à E si et seulement s'il est obtenu à partir de la base en appliquant les règles en un nombre fini d'étapes.

Cette phase de clôture est systématique et supposée implicite.

On note par *(B, R)* la description d'un schéma inductif définissant les éléments de E. ❞

• Exemple-3 : la généalogie :

Définissons la relation "ancêtre" à partir de la relation "parent" ci-dessus. On veut donc définir E = {<x,y> | ancêtre(x,y)}

• Les éléments de la base sont énumérés par la relation "parent" :

X est un ancêtre de Y si X est parent de Y (père ou mère).
D'où B = {<x,y> | parent(x,y)}

• L'ensemble R comporte une seule règle :

X est un ancêtre de Y si X est le parent d'un ancêtre de Y.

On obtient le prédicat *ancètre/2* :

```
ancetre(X, Y) :- parent(X, Y) .
ancetre(X, Y) :- parent(X, Z) , ancetre(Z, Y).
```

Remarque : L'ensemble R peut églament être défini par :

X est un ancêtre de Y si X est l'ancetre d'un parent de Y.

Ainsi, on obtient une nouvelle version (équivalenteà) du prédicat *ancètre/2* :

```
ancetre(X, Y) :- parent(X, Y) .
ancetre(X, Y) :- parent(Z,Y) , ancetre(X, Z).
```

Résumé et remarques :

Les deux démarches exposées ci-dessus sont analogues. Pour l'exemple-3, l'ordre que l'on doit choisir peut être le nombre de générations séparant un individu de son ancêtre. Ainsi, dans la règle générale, on exprime la relation *ancêtre* à la génération *n* en fonction de la même relation à la génération *n-1*.

Cependant, pour concevoir un algorithme récursif efficace, on s'appuie dans la pratique sur une définition inductive bien fondée de l'espace des données E en vérifiant que l'algorithme définit les éléments de B d'une manière simple. On définit ensuite les règles en vérifiant que pour une donnée x (hors de B), l'algorithme sur x est une combinaison simple des résultats des évaluations sur les antécédents de x.

Le passage d'un schéma inductif à un programme destiné à être exécuté sur une machine nécessite parfois des adaptations qui tiennent compte des possibilités du langage.

Dans l'exemple-3, on peut proposer la règle

```
ancetre(X,Y) :-  ancetre(Z,Y), parent(X,Z).
```

L'exécution du but *:-ancetre(A,B)* produira les couples de E avant de s'enfermer dans une boucle infinie. Il y a deux raisons à cette non terminaison :

• cette règle destinée à définir un couple <x,y> hors de B s'appuie sur la définition d'un autre couple hors de B qui lui même s'appuie sur un autre couple hors de B...

• le moteur Prolog traite la conjonction comme un "puis" c'est à dire, la commutativité de la conjonction disparaît dans l'interprétation procédurale d'une définition déclarative.

Nous sommes donc amenés à tenir compte de ce comportement lors de la traduction d'un schéma inductif en un algorithme récursif.

D'autres formes d'induction :

Dans une démarche, dirigée par les données, de raisonnement et d'écriture des schémas inductifs, on distingue un autre type de schéma appelé l'induction (récursivité) structurelle. Dans cette démarche, le raisonnement s'appuie sur la définition syntaxique de la donnée manipulée pour produire des schémas récursifs. Ainsi, lorsque la structure de données est récursive, et c'est le cas en Prolog, la structure des prédicats est calquée sur la structure de données. Nous étudierons des exemples de ces schémas lors d'écriture des prédicats de traitement des listes et des arbres.

Exemple-4 : Définition de la relation *plus_grand* entre des individus.

1- solution par un schéma inductif :

• Les éléments de la base sont énumérés par les connaissances simples. Par exemple, les faits suivants explicitent la relation "X est immédiatement plus grand que Y" :

```
p_grand(jean, pierre).
p_grand(pierre, jacques).
p_grand(jacques, helene).
```

Nous pouvons donc exprimer les éléments de la base par :

X est plus grand que Y si X est immédiatement plus grand que Y.

• L'ensemble R de règles définissant la relation plus_grand :

X est plus grand que Y si X est plus grand que Z et Z plus grand que Y.

D'où le prédicat *plus_grand/2* :

```
plus_grand(X,Y) :- p_grand(X,Y).
plus_grand(X,Y) :- p_grand(X,Z), plus_grand(Z,Y).
```

Exemple de question :

?- plus_grand(jean, X).

③ X = pierre, X = jacques, X = helene

2- Une solution alternative utilisant la même base:

```
plus_grand(X,Y) :- p_grand(X,Z), plus_grand_ou_egal(Z,Y).
plus_grand_ou_egal(X,X).
plus_grand_ou_egal(X,Y) :- plus_grand(X,Y).
```

Exemple de question : ?- plus_grand(jean, X).

③ X = pierre, X = jacques, X = helene

3- La solution utilisant la transitivité directe. La seule relation P-grand définit la base et l'ensemble R :

```
p_grand(jean, pierre).
p_grand(pierre, jacques).
p_grand(jacques, helene).
p_grand(X,Y) :- p_grand(X,Z), p_grand(Z,Y).      %la transitivité
```

Exemple de question : ?- p_grand(jean, X).

③ X = pierre, X = jacques, X = helene

③ Débordement dû à une boucle infinie

Discuter des problèmes posés par cette solution.

9.Algorithme de Prolog : traitement des règles

❝ Etendons l'algorithme précédent pour tenir compte des règles. Nous considérons les faits commes des règles dont la partie droite est vraie. Soit :

P= le programme = un ensemble de règles

B = b_1,...,b_n ($n \geq 0$) = le but

```
θ, θ', σ, γ : substitution
Fonction resoudre (B : but; θ:substitution ) retourne booléen =
   Si B = vide alors                -- n = 0 dans B=b1 , .... , bn. Plus rien à prouver
   γ=θ; Retourne (vrai);            -- γ est la solution
   Sinon
      considérer b1 dans B;
⑩     LC = l'ensemble des règles de P de la forme  A :- a1,..., ak
      Tant que LC <> vide faire
         choisir C le premier élément de LC
         LC = LC - {C}
¶     Renommer C tel que C et b1 n'aient pas de variable commune
         σ= {};
         Si unifier(b1 , C)              -- mise à jour de σ
         Alors θ' = θ ° σ;
①                B' = θ' (a1,..., ak , b2 , ... , bn)        -- application de θ'
                 Si resoudre(B', θ') alors retourne (vrai) Fin si
         Fin si
      Fin Tant que
      Retourne (faux)
   Fin si
Fin resoudre
```

On constate qu'en ①, le corps de la règle considérée en ⑩ est insérée en tête de B'. Ce qui explique le comportement du moteur Prolog dans le traitement de certaines règles récursives à gauche.

Pour éviter les conflits de noms de variables, on renomme C (en ¶) en remplaçant ses variables par des nouvelles, de sorte qu'il n'y ait pas de variable commune entre C et b_1. Les variables d'une règle étant locales à la règles, ce renommage ne modifie rien au sens de la règle.

Il n'y a pas d'autre changement par rapport à l'algorithme précédent. L'algorithme d'unification précédent reste inchangé. Nous verrons en annexe l'ensemble des algorithmes d'un interprète Prolog ainsi qu'une version complète de la fonction *resoudre* traitant les termes fonctionnels.❞

10. Résumé des définitions

• Littéral : un terme prédicatif
Exemple: **p** , **q** , **r**, **pere(jean, X)**

Un littéral *négatif* est un littéral précédé du symbole de négation *not*

• Règle (clause) : *un_littéral_positif :- conjonction_de_littéraux*
Exemple: **p :- q(Z) , r , not t(Y) .**

• Tête de règle: la partie gauche d'une règle
Exemple: p dans la règle **p :- q.**

• Corps de règle: la partie droite d'une règle
Exemple: q dans la règle **p :- q.**

• Fait: règle sans corps
Un fait est une règle dont le corps est égal à *true*.

• Question (but) : règle sans tête
Un but est une règle dont la tête est égale à *false*.

• Prédicat : paquet de règles de même constante prédicative
et de même nombre d'arguments

• Programme : paquet de prédicats

Portée des noms

Hormis les constantes entières qui représentent leurs valeurs (et certains noms prédéfinis e.g. not), les noms en Prolog n'ont pas de signification particulière.

La règle de la portée des noms est :

- Les variables sont locales à l'intérieur d'une question, règle ou fait.
- Les constantes, les noms de fonctions et de prédicats sont connus dans tout le programme.

10.1- Terme fonctionnel et prédicat

Nous avons vu que le seul objet manipulé en Prolog est le terme. Nous avons également remarqué la différence entre un terme fonctionnel et un littéral. Les définitions ci-dessous précisent plus formellement ces distinctions.

Soit D le domaine de discours (contenant des termes).

• Un terme composé $F(t_1,t_2,...,t_n)$ est un *terme fonctionnel* si F est une constante fonctionnelle et les t_i des termes:

F : $\mathbf{D^n}$ **-->** **D**

$t_1,t_2,..,t_n$ |--> $F(t_1,t_2,..,t_n)$

Exemple :

D = {f, a, b, 1, "ecole", jean, pere, pierre, ... , f(a,1,"ecole")}

f : D^3 --> D

a × 1 × "ecole" |--> **f(a,1,"ecole")**

Un tel terme est identique à une structure de Pascal. Il n'a donc en soi aucune interprétation. C'est à l'utilisateur de se donner un "sens" à un terme fonctionnel (c'est-à-dire, à l'interpréter). Ainsi, on peut par exemple considérer *f(2)* égal à *3* dans une interprétation associant *f* à *succ* et, égal à *1* dans une autre interprétation associant *f* à *pred*.

L'algorithme de §7.3 s'applique à l'unification de termes composés.

• Un terme composé $P(t_1,t_2,...,t_n)$ est un *terme prédicatif* si P est une constante prédicative (un atome) et les t_i des termes:

P : $\mathbf{D^n}$ **-->{vrai,faux}**

$t_1,t_2,..,t_n$ |--> {vrai, faux}

Exemple :

Avec la définition D de ci-dessus et, en supposant que jean est le pèere de pierre :

père : D^2 --> {vrai, faux}

jean × pierre |--> **vrai**

Exemple:

```
entier(0).
entier(succ(N)) :- entier(N).
```

constantes: {0}
constantes fonctionnelles : {succ}
constantes prédicatives : {entier}
variables : {N}

p Un terme fonctionnel est l'expression *syntaxique* de l'application d'une fonction à ses arguments. A ce titre, il ne représente que lui-même. Il n'y a donc aucune évaluation associée à un terme fonctionnel dans un programme Prolog.

Ainsi, *succ(succ(0))* ne représente pas la valeur 2 mais un arbre unaire. De même, *sin(1)* ne représente qu'un arbre unaire dont la racine est l'atome *sin* avec la seule feuille *1*.

L'application d'un symbole fonctionnel à ses arguments est abordée dans "Evaluation des termes".

10.2- Quantification des variables

❝ • Une variable figurant dans un fait, dans une tête de clause ou à la fois dans une tête de clause et dans le corps de la même clause est quantifiée universellement

Par exemple, la formule :

$\forall X, vole(X) \wedge a\_des\_plumes(X) \Rightarrow oiseau(X)$

peut être traduite par

```
oiseau(X) :- vole(X) , a_des_plumes(X).
```

• Les variables figurant dans un but et dans le corps d'une clause sans avoir d'occurrence dans la tête de la même clause sont quantifiées existentiellement.

Par exemple :

$\forall X \ \forall Y \ \exists Z \ pere(Z,X) \wedge pere(Z,Y) \Rightarrow frere(X,Y)$

peut être traduit par

```
frere(X,Y) :- pere(Z,X) ,  pere(Z,Y)
```
❞

10.3- Notion d'interprétation et de modèle

❝ Considérons le prédicat *pere(pierre, paul)*. Ce prédicat traduit la relation "père" entre "pierre" et "paul". On peut interpréter cette relation par la phrase "pierre est le père de paul". Cette interprétation n'est pas unique. On peut par exemple convenir d'associer le deuxième argument au "père" et le premier au fils, pour avoir l'interprétation "paul est le père de pierre".

Il est important de convenir d'une *interprétation* claire pour chaque relation et de la respecter dans le programme Prolog, sans quoi on risque des incohérences.

Pour une formule logique F, un *modèle* est une interprétation qui associe à F la valeur de vérité "vraie".

• Exemple : Considérons la formule $\forall x \exists y\ p(x,y)$ et l'interprétation I suivante. Soit D le domaine des entiers non-négatifs et la relation '<' que l'on assigne au symbole p. Ainsi, l'on peut interpréter la formule ci-dessus par :

"Pour tout entier non-négatif, il existe un entier non-négatif qui est strictement plus grand que le premier."

On dit que cette interprétation est un *modèle* pour cette formule.

Par contre, la même interprétation I n'est pas un modèle pour la formule $\exists y \forall x\ p(x,y)$ pour laquelle on peut considérer l'interprétation "épistémologique" suivante :

Soit D le domaine des homes et la relation 'est_pere_de' que l'on assigne au symbole p. Ainsi, l'on peut interpréter la formule $\exists y \forall x\ p(x,y)$ par :

"Il existe un individu y (Adam) tel que pour tout individu x, y est_pere_de x"

Cette interprétation est un modèle pour la formule ci-dessus mais ne l'est pas pour $\forall x \exists y\ p(x,y)$.

Notons enfin que $\exists y \forall x\ p(x,y)$ est équivalente à $\forall x\ p(\alpha,y)$ où α est une constante (par exemple Adam) alors que $\forall x \exists y\ p(x,y)$ est équivalente à $\forall x\ p(x, h(x))$ où h est une fonction délivrant une valeur selon x (par exemple, un entier non-négatif strictement plus grand que x).

La notion de modèle est étendue aux axiomes d'une théorie. Ici, nous en avons donné une idée simple et intuitive. On constate qu'il peut y avoir une infinité d'interprétations et donc une infinité de modèles mais, il en existe un seul "intéressant" décrit par un programme Prolog (sans négation) appelé le modèle minimal de Herbrand) (voir la bibliographie). ❞

11. Pratique de l'unification

L'unification est le dispositif de base en Prolog. Il convient donc de bien maîtriser son fonctionnement.

L'algorithme suivant décrit le principe de fonctionnement de l'unification en Prolog dans un formalisme proche du langage Pascal.

Soit deux termes T1 et T2 à unifier. Selon la structure de ces termes, l'un des cas ci-dessus est appliqué. Lors que T1 et T2 sont unifiables, cet algorithme produit une substitution σ telle que σ(T1) = σ(T2). A l'appel de la fonction unifier, σ={}.

Pour unifier les termes T1 et T2, on écrite en Prolog **T1=T2**.

σ : substitution
Fonction unifier(T1,T2 : termes) retourne booléen =
Cas
- T1 est identique à T2 : retourne(vrai)
- T1 est une variable : $\sigma = \sigma \circ \{T1/T2\}$; retourne(vrai)
- T2 est une variable : $\sigma = \sigma \circ \{T2/T1\}$; retourne(vrai)
- T1 est une constante ou T2 est une constante : retourne(faux)
- T1 et T2 sont des termes composés ayant le même symbole fonctionnel et le même nombre d'arguments. Soit :

  T1 = f(t1, t2,..., tn) et T2 = f(t'1, t'2,..., t'n).

  Il faut unifier un par un les arguments de T1 et de T2 et, à chaque étape, appliquer θ aux termes restants. Ce qui est effectué par la boucle suivante :

```
i=0
Repéter
    i := i+1
    appliquer σ à t_i et à t'_i
    test = unifier(t_i, t'_i)
Jusqu'à (i>n) ou non test
retourne(test)
```

- Autre : retourne(faux)

Fin cas
Fin unifier

p Du point de vue syntaxique, les littéraux sont des termes fonctionnels. On peut donc appliquer l'algorithme d'unification ci-dessus lors du choix des clauses pendant la résolution.

Exemple-1 :

T1 = f(a, g(X, b))
T2 = f(Y, g(b, X))

L'unification de ces deux termes revient aux unifications simples suivantes :

1- **Y =?a** à lire, Y est-il unifiable avec a?

Le résultat est θ={Y/a}.

Appliquons θ aux termes g(X, b) et g(b, X). On a :

- θ(g(X, b)) = g(X, b) car X n'est pas lié dans θ
- θ(g(b, X)) = g(b, X) idem

2- **g(X, b) =? g(b, X)**

L'unification de ces deux termes revient à :

21- **X =?b**

Le résultat combiné avec θ donne : θ={Y/a, X/b}

Appliquons θ à X et à b. On a : θ(X) = b, θ(b) = b.

22- **b =?b**

Cette unification ne produit pas de lien et la substitution finale est :

θ={Y/a, X/b}

Exemple-2 :

T1= p(X, a, f(b, f(Y, n)))
T2= p(s, Y, f(Z, f(X, n)))

On a :

1- **X =? s** => θ={X/s}

2- θ(Y) =? θ(a)

C'est à dire **Y =? a** => θ={X/s, Y/a}

3- θ(f(b, f(Y, n))) =? θ(f(Z, f(X, n)))

C'est à dire **f(b, f(a, n)) =? f(Z, f(s, n))**

31- **Z =? b** => θ={X/s, Y/a, Z/b}

32- **f(a, n) =? f(s, n))**

321- **a =? s** □ Echec

Exemple-3 :

T1= entier(suc(suc(suc(0))))

T2= entier(suc(suc(1))) □ Echec

Extraction des réponses :

Lors de l'unification des termes, la substitution θ contient un ensemble de couples variable/terme. Ainsi, la valeur d'une variable peut être obtenue en suivant une "chaîne" de couples dans θ. Par exemple, étant donné le terme T=f(X, g(Y)) et θ={X/Z1, Z1/p(R), R/a, Y/f(Z1)}, les valeures des variables X et Y s'obtiennent en suivant les chaînes :

X -> Z1 -> p(R) -> p(a)
Y -> f(Z1) -> f(p(R)) -> f(p(a))

On a θ(T) = f(p(a), g(f(p(a)))).

Exercice-2 : Etudier les résultats d'unification des termes suivants :

f(a, X) et f(a, X, X).
f(X) et X
f(X, Y, X) et f(g(X), g(Y), Y)
personne(jean, adresse(45, marseille, lyon), profession(vendeur)) et
personne(X, adresse(_, _, lyon), profession(Y)).

Exercice-3 : Soit la base de connaissances suivante :

livre(auteur(fredric, dard), titre('fleur de nave'), prix(40)).
livre(auteur(victor, hugo), titre('les misérables'), prix(300)).
livre(auteur(victor, hugo), titre('fantasia chez les ploucs'), prix(200)).
livre(auteur(eugene, chang), titre('le silence'), prix(180)).

Formuler en Prolog les questions suivantes :

- Quel est le nom d'un auteur d'un livre prénommé victor?
- existe-t-il un livre valant 40 francs ?
- quel est le titre d'un livre de frederic dard ?
- quel est le prix du livre "le silence" ?
- y a-t-il des livres dans la base ?

12. La résolution

Dans ce qui suit, nous étudions plus en détail la stratégie du moteur Prolog et les problèmes liés à celle-ci. Cette présentation décrit l'unique instruction de ce que l'on appelle "la machine abstraite Prolog".

12.1- La stratégie de Prolog : effacement sans variable

Cette stratégie est modélisée par le modèle suivant de la machine Prolog (règle Modus Tolens):

(1) B = :- A_1 , A_2 ,..., A_k
(2) R = A_1 :- B_1 , ..., B_n
(3) B' = B_1 , ..., B_n , A_2,..., A_k

C'est à dire :

Si le but courant est B = :- $\mathbb{A}_1$, A_2 ,..., A_k et
il existe une règle R = $\mathbb{A}_1$:- B_1 , ..., B_n
(dont la tête est identique à A_1)

Alors effacer le couple <$\mathbb{A}_1$,$\mathbb{A}_1$> et
construire le nouveau but B' en remplaçant A_1 dans B par la partie droite de R; c'est à dire

B' = B_1 , ..., B_n , A_2,..., A_k

Dans le but B = A_1 , ... , A_k, si k =0 alors B est démontré.

Pour simplifier l'écriture, on exprime ce principe (appelé le principe de résolution dans sa forme générale) par la règle :

:- A_1 , A_2 ,..., A_k $\quad$ A_1 :- B_1 , B_n		*Si l'on a ceci*
———————————	à lire	———————
:- B_1 , B_n , A_2 ,..., A_k		*On en déduit celà*

Exemple:

B = :- ski , balade
R = ski :- montagne , neige
③ *B' = :- montagne , neige , balade*

12.2- Retour arrière

Comme nous l'avons vu dans les algorithmes, la résolution choisit à chaque étape une règle dans la base. Il peut y avoir plusieurs règles candidates. Pour assurer la complétude (donner toutes les réponses, démontrer un but de toutes les manières possibles), il faut prévoir de revenir en arrière pour essayer les autres règles.

Ce retour arrière est provoqué :

1 - un échec de démonstration
2 - un échec explicite par l'utilisateur

Remarque :
Dans les Prolog standard, la frappe d'un point-virgule (**;**) à la suite d'un succès est interprétée comme un échec explicite provoquant la recherche de la solution suivante.

Il faut noter que le retour arrière remet en cause ce qui a été fait dans l'ordre inverse.

Pour mieux comprendre ce principe, on peut imaginer la recherche d'un chemin entre un point de départ et une arrivée dans un labyrinthe.
Le cas (1) ci-dessus est analogue à la situation produite lors que l'on arrive à une impasse. On revient en arrière au point de choix le plus récent où l'on avait la possibilité d'emprunter plusieurs couloirs. On tente ainsi un autre chemin....

Le cas (2) peut être envisagé lorsque par une balise "n'allez pas plus loin" (*fail*), on signale qu'il n'y a aucun chemin que l'on puisse emprunter depuis l'endroit où l'on se trouve aboutissant à un succès.
Enfin, la remarque ci-dessus est analogue à la situation où l'on doit trouver tous les chemins possibles pour aller du point de départ au point d'arrivée. On procède de la même manière pour revenir sur nos pas et examiner les autres possibilités dans l'ordre chronologique des couloirs empruntés.

12.3- Représentation de la résolution par l'arbre de recherche

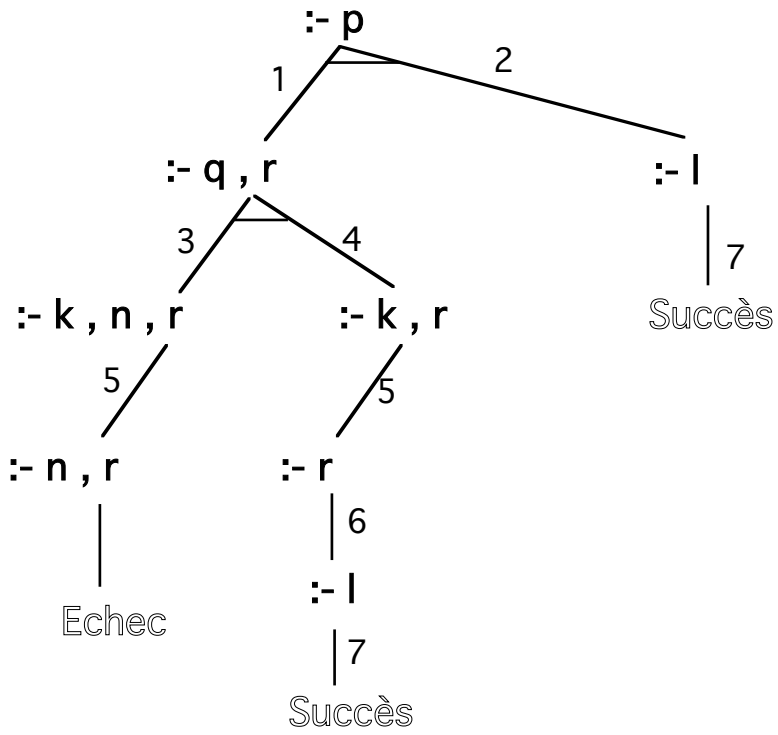
Soit le programme sans variable :

(1) p :- q , r.
(2) p :- l .
(3) q :- k , n.
(4) q :- k.
(5) k.
(6) r :- l.
(7) l.

Et la question

:- p.

L'arbre suivant illustre comment on obtient deux réponses positives à la question:



12.4- Représentation par la pile de retours arrières

❝ Cette représentation permet de comprendre le principe de fonctionnement des retours arrières. Nous utiliserons ce dispositif lors de la présentation du coupe-choix.

La pile des retours arrières est représentée par une séquence de littéraux placés entre ⟦ ⟧.

• Au départ, la pile est vide (notée ⟦⟧)

• A une étape i, l'état de la machine est représenté par

:- ⟦A_i, B_j , C_k⟧ E , F , G

Ce qui veut dire que les règles <A_i, B_j, C_k> ont été utilisées et la liste actuelle des buts à démontrer est <E , F , G>. La notation X_i représente la $i^{ème}$ clause du programme dont la tête est le littéral X.

• L'étape i+1 sera :- ⟦A_i, B_j , C_k, E_m⟧ H , F , G si l'on efface E avec la clause numéro *m* de la forme E :- H.

• La machine s'arrête à l'étape n et conclut sur :

- un **succès** si à l'étape n, on a :- ⟦.....⟧. C'est à dire qu'il ne reste plus rien à démontrer et le contenu de la pile représente les clauses utilisées)

- un **échec** si à l'étape n, on a :- ⟦.....⟧ F et F ne peut plus s'effacer à l'aide d'une clause du programme.

Exemple-1 : soit le programme sans variable

(1) p :- q , r.
(2) p :- l .
(3) q :- k , n.
(4) q :- k.
(5) k.
(6) r :- l.
(7) l.

Application à l'exemple précédent avec la question **:- p.**

:- [ ] p.
  :- [p1] q , r.
    :- [p1, q3] k , n , r.
      :- [p1, q3, k5] n , r.
        Echec sur n. On retourne en arrière en rétablissant l'état précédent

    :- [p1, q3] k , n , r.
        k ne peut pas être effacé par d'autre clause. On retourne en arrière.

  :- [p1] q , r.
    :- [p1, q4] k , r.
      :- [p1, q4 , k5] r.
        :- [p1, q4 , k5, r6] l.
          :- [p1, q4 , k5, r6, l7] .
          Plus rien à démontrer. On obtient un ***succès****.*
          Le contenu de la pile représente les clauses utilisées.

    Si un retour en arrière demandé par un point-virgule :

        :- [p1, q4 , k5, r6] l. *=> pas d'autre possibilité pour l*
      :- [p1, q4 , k5] r. *=> pas d'autre possibilité pour r*
    :- [p1, q4] k , r. *=> pas d'autre possibilité pour k*
  :- [p1] q , r. *=> pas d'autre possibilité pour q*
:- [ ] p.
  :- [p2] l. *=> Ici, la second clause est utilisée*
    :- [p2, l7] . *Plus rien à démontrer =>* ***succès***

    Si retour en arrière demandé:
  :- [p2] l. *=> pas d'autre solution pour l*
:- [ ] p. *=> pas d'autre solution pour p =>* ***Echec***

12.5- La stratégie de Prolog : effacement avec variable

(1) B = :- A_1 , A_2 ,..., A_k
(2)R = A'_1 :- B_1 , ..., B_n
(3)B' = B_1 , ..., B_n , A_2,..., A_k

C'est à dire :

Si le but courant est B = :- $\mathbb{A}_1$, A_2 ,..., A_k et
il existe une règle R = $\mathbb{A}'_1$:- B_1 , ..., B_n
dont la tête s'unifie avec A_1 et produit la substitution θ

Alors effacer le couple <$\mathbb{A}_1$, $\mathbb{A}'_1$> et
construire le nouveau but B' en remplaçant A_1 dans B par la partie droite de R; puis appliquer θ à B'. C'est à dire

$$B' = \theta(B_1 , ..., B_n , A_2,..., A_k)$$

Dans le but B = A_1 , ... , A_k, si k =0 alors B est démontré.

Exprimons ce principe par la règle :

$$\frac{:- A_1 , A_2 ,..., A_k \qquad A'_1 :- B_1,..., B_n \qquad A_1=\theta(A'_1)}{:- \theta(B_1,..., B_n , A_2 ,..., A_k)}$$

Exemple :

B = :- oiseau(canard) , produit_foie_gras(canard).
R = oiseau(X) :- vole(X), a_des_plumes(X)

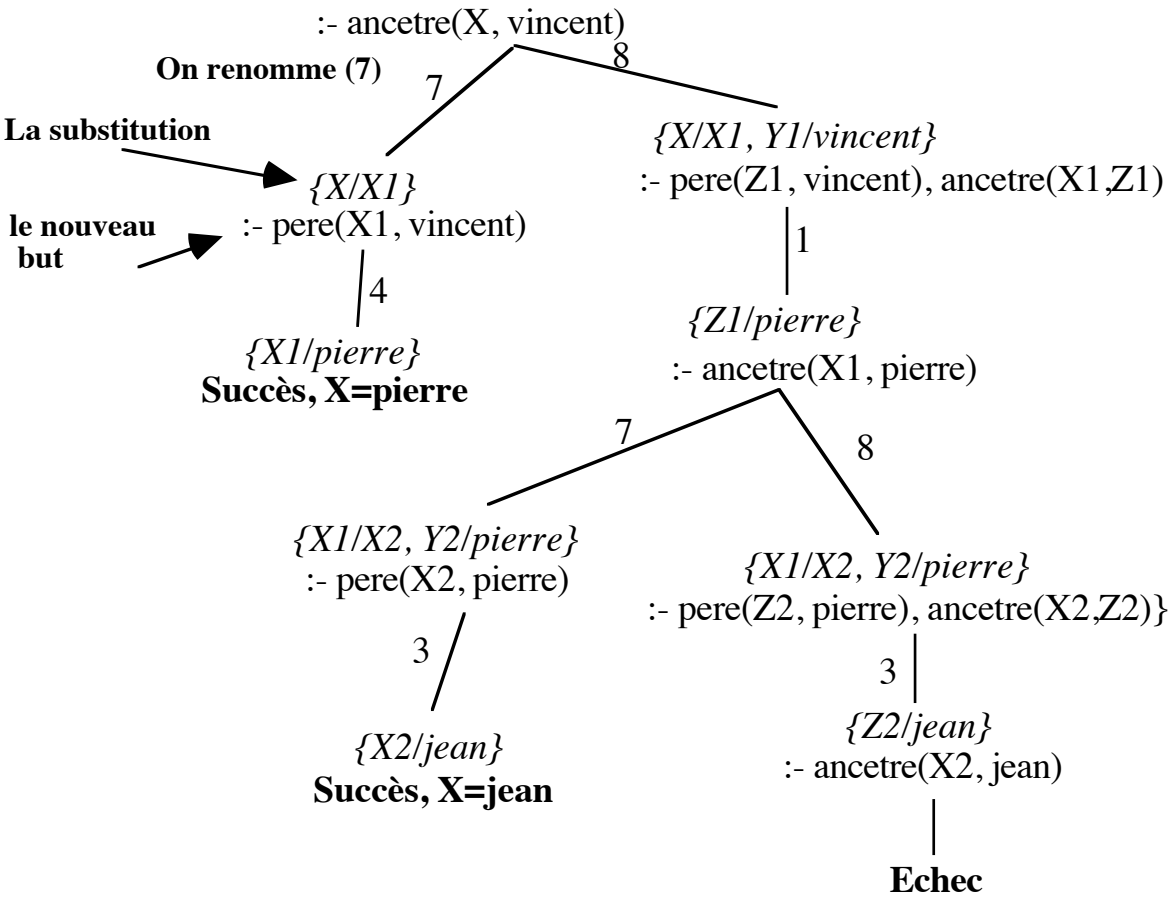
③ B' = :- vole(canard), a_des_plumes(canard), produit_foie_gras(canard). ❞

Exemple 2 : représentation par arbre de résolution avec variable

(1)pere(jean, jacques).
(2)pere(jean,helene).
(3)pere(jean,pierre).
(4)pere(pierre,vincent).
(5)pere(jacques,marie).
(6)pere(jacques,michel).
(7)ancetre(X,Y) :- pere(X,Y).
(8)ancetre(X,Y) :- pere(Z,Y) , ancetre(X,Z).

La question :
:- ancetre(X,vincent).

L'arbre de résolution :



❝ **Représentation par pile avec variable de la même question :**

(1)pere(jean, jacques).
(2)pere(jean,helene).
(3)pere(jean,pierre).
(4)pere(pierre,vincent).
(5)pere(jacques,marie).
(6)pere(jacques,michel).
(7)ancetre(X,Y) :- pere(X,Y).
(8)ancetre(X,Y) :- pere(Z,Y) , ancetre(X,Z).

:- ancetre(X,vincent).

:- [ ] ancetre(X, vincent).
:- [ancetre7(X, vincent)] pere(X, vincent).
:- [ancetre7(X, vincent), pere4(X, vincent)].

=> Succès avec X = pierre

Si le retour arrière demandé

:- [ancetre7(X, vincent)] pere(X, vincent).

=> Echec

aucune autre valeur C pour X telle que pere(C,vincent)

=> Retour en arrière : on remonte dans la pile

:- [ ] ancetre(X,vincent).
:- [ancetre8(X, vincent)] pere(Z, vincent) , ancetre(X,Z).
:-[ancetre8(X, vincent) , pere4(Z, vincent)] ancetre(X,pierre).
:- [ancetre8(X, vincent) , pere4(Z, vincent) , ancetre7(X,pierre)] pere(X,pierre).
:- [ancetre8(X, vincent) , pere4(Z, vincent) , ancetre7(X,pierre), pere3(X,pierre)] .

=> Succès avec X = jean

Si retour arrière demandé

Les rétablissements successifs des états n'aboutiront à aucun autre succès ❞

12.6- Hypothèse du monde fermé et négation

N'est vrai que ce que l'on peut prouver

non P est vrai si **P** est faux
non P est faux si **P** est vrai

Exemple:

femme(X) :- ***not*** *homme(X).*

Etant donné cette règle, un individu I pour lequel la propriété homme(I) n'est pas démontrable est considéré comme femme.

Par exemple, étant donné les individus {jean, pierre, jacques} et les faits :

homme(jean).
homme(pierre).

La question **:- femme(jacques).** réussit car le fait homme(jacques) est absent de la base.

En Prolog, un littéral négatif (tel que *not p*) ne peut figurer que dans le corps d'une règle ou dans une question. Il n'est donc pas possible d'avoir des faits niés où des règles dont la tête est un littéral négatif.

Remarque sur la syntaxe : On peut écrire *not p* par **\+p**.

12.7- Problèmes liés à la stratégie de Prolog

A chaque étape de la résolution, l'interpréteur Prolog doit faire deux choix :

1- L'un pour choisir un but dans une suite de buts à effacer. C'est le premier élément de la suite qui est toujours choisi.

③ Pour effacer une suite Q_0 , Q_1, ... , Q_n , c'est Q_0 qui est choisi en premier;

2- L'autre pour choisir la règle qui sert à effacer le but Q_0.

③ C'est la première règle du programme dont la tête s'unifie avec B qui est choisie.

Cette stratégie de développement d'arbre de résolution "en profondeur d'abord" rend la récursivité à gauche quasiment inexploitable (sauf sous certaines conditions). De plus, l'opérateur de conjonction n'est plus considéré commutatif et l'ordre des littéraux dans le corps d'une règle ainsi que l'ordre entre les clauses d'un prédicat deviennent importants.

Bien que plus efficace en calcul, cette stratégie peut parfois échouer dans le calcul des réponses d'un programme.

Pour comprendre ces problèmes, on distingue les "conséquences logiques" d'un programme (les littéraux logiquement déductibles) des réponses effectivement obtenues d'un programme par la stratégie de résolution.

Exemple-1:

⑩ **p :- p , q.**
❾ **p .**
① **q.**

On peut aisément constater que "p" est une connaissance acquise (car le fait p est présent). Néanmoins, la question :

:- p.

Nous enferme dans une "boucle" infinie sans démontrer p car (1) est choisi avant (2).

Il existe des algorithmes de construction des conséquences d'un programme Prolog remplissant certaines conditions. Pour en donner une idée simple sur l'exemple-1 :

- on commence avec l'ensemble E_0 des faits du programme : $E_0 = \{p,q\}$
- on construit $E_1 = E_0 \cup \{$tous les littéraux dont le corps est dans $E_0\}$: $E_1 = \{p,q\}$
-....
- on construit $E_n = E_{n-1} \cup \{$tous les littéraux dont le corps est dans $E_{n-1}\}$: n=1 et $E_n = \{p,q\}$

On montre que ce processus s'arrête (il existe un point fixe pour l'opérateur de conséquence immédiate que l'on applique à chaque étape). L'ensemble E_n est minimal (le plus petit ensemble contenant toutes les conséquences) et contient les réponses aux questions pour lesquelles Prolog calculerait une réponse.

Exemple-2:

Soient les spécifications suivantes et leurs codages en Prolog:

S'il pleut ou il neige alors il y a une précipitation
S'il gèle et il y a une précipitation alors il neige
S'il ne gèle pas et il y a une précipitation alors il pleut

Fait: *il neige*
Questions: *Y a-t-il une précipitation?*
Est-ce qu'il neige?

Codage en Prolog :

Notons
pr : il y a une précipitation
pl : il pleut
ne : il neige
ge : il gèle

Le programme Prolog correspondant sera :

```
pr :- pl ; ne.
ne :- ge , pr.
pl :- not ge , pr.
ne.
```

Soient les questions :

(1) :- pr.
(2) :- ne.

On peut constater que la question (2) est conséquence immédiate du programme. "ne" étant vrai, "pr" l'est également par la première clause. Pourtant, la démonstration ci-dessous montre la difficulté d'obtenir une réponse à la question (1).

- Démonstration de la question-1

:- pr , pr :- pl ; ne
———————————————

:- pl ; ne , pl :- not ge , pr
———————————————————————

:- not ge , pr ❷ *not ge* réussit car *ge* étant absent de la base, *ge* échoue
———————————

:- pr ❷ **On boucle.**

La démonstration de *pr* nous a conduit à *pr*.

- Démonstration de la question-2

:- ne , ne :- ge , pr
———————————————

:- ge , pr ❷ Echec: *ge* est absent de la base
(donc non démontrable)

③ Retour arrière

:- ne , ne
———————————

❷ **succès**

13. Quelques élements de programmation

La programmation dans un environnement Prolog s'effectue dans un cycle édition-chargement-exécution. La phase d'édition s'effectue à l'aide d'un éditeur habituellement indépendant de l'environnement Prolog. On invoque ensuite l'interprète Prolog et l'on soumet un but (un prédicat prédéfini) afin de charger les clauses du programme dans l'espace du travail. On peut ensuite poser des qustions relatives à ce programme, vérifier les résultats et éventuellement reprendre l'édition, recharger le programme...

Gestion de la base :

Ces prédicats prédéfinis permettent de charger (ou de recharger si un premier chargement a déjà eu lieu) une base de clauses. Ils permettent également de manipuler dynamiquement les clauses de l'espace de travail.

consult(F) ou **[F]** :
Lire le fichier F et les insérer dans l'espace de travail. Le nome du fichier F est un atome. Par exemple, consult('monfile.pl') charge le fichier monfile.pl.

reconsult(F) ou **[-F]** :
Comme *consult* mais on remplace les prédicats qui sont définis dans F.

abolish(Sym_pred, Arité) :
Effacer de l'espace de travail le paquet de clauses dont le symbole prédicatif est Symb_pred et dont l'arité est Arité.

clear: efface le contenu de la base.
listing : Lister les clauses de l'espace de travail.

listing(Sym_pred) :
Lister les clauses dont le symbole prédicatif est sym_pred.

Scénario d'utilisation :

- Produire le texte du programme à l'aide d'un éditeur
- Faire vérifier le programme par un analyseur statique si cela ne se fait pas au chargement
- Activer l'interprète Prolog (par exemple par la commande "prolog")
- Charger le programme (par exemple par "consult")
- Poser les questions

Mise au point :

Les environnement Prolog disposent également d'un certains nombre de prédicats prédéfinis permettant de suivre pas-à-pas l'exécution d'un programme.

spy(symb_pred) :
Mettre un point d'arrêt sur les prédicats dont le symbole prédicatif est Symb_pred.
spying : Quels sont les points d'arrêts ?
nospy(symb_pred) :
Retirer le point d'arrêt mis sur les prédicats dont le symbole prédicatif est symb_pred.
trace : Activer le mode trace. Arrêt sur tous les prédicats
notrace : Désactiver le mode trace.
debug : Activer le mode debug. Arrêt sur les prédicats que l'on veut étudier (par spy).
nodebug : Désactiver le mode debug.
abort : Abandonner la résolution.

Remarque : le mode trace est utilisé pour une mise au point globale où l'on peut suivre les réponses de tous les prédicats. Pour une mise au point sélective, on peut choisir le mode debug. Ensuite, on peut mettre des points d'arrêts sur certains prédicats et suivre uniquement l'exécution de ces derniers.

Quelques prédicats utiles :

Les prédicats prédéfinis ci-dessus seront utiles pour réaliser les exercices. Ils seront développés plus en détail par la suite.

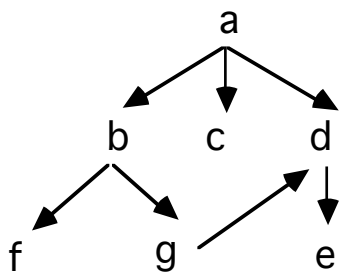
X = Y : réussit si les termes X et Y s'unifient.
X \= Y : ou **not(X=Y)** réussit si les termes X et Y ne s'unifient pas.
X == Y : réussit si les termes X et Y sont littéralement identiques.
X \== Y : réussit si les termes X et Y ne sont pas littéralement identiques.
atom(X) : réussit si le terme X est un atome.
integer(X) : réussit si le terme X est un entier.
float(X) : réussit si le terme X est un réel.
var(X) : réussit si le terme X est une variable.
write(X) : réussit et affiche le terme X sur l'unité de sortie (écran).
display(X) : réussit et affiche le terme X sous sa forme canonique sur l'unité de sortie (écran).
read(X) : réussit et lit le terme X sur l'unité d'entrée (clavier).

13.1- Exercices

Exercice-1: recherche de chemins dans un graphe

Représenter le graphe orienté suivant par des faits Prolog. On peut noter par **arc(X,Y)** le fait qu'il y a un chemin direct de X à Y.

Proposer un raisonnement inductif puis écrire le prédicat **chemin(X,Y)** représentatnt la fermeture réflexo-transitive de la relation **arc**.



Coder ce programme en Prolog. Poser des questions et vérifier les réponses en traçant la résolution.

• Dessiner l'arbre de recherche (résolution) pour la question :

?- chemin(a,e).

et vérifier les réponses de Prolog.

Exercice 2: Base de données familiale:

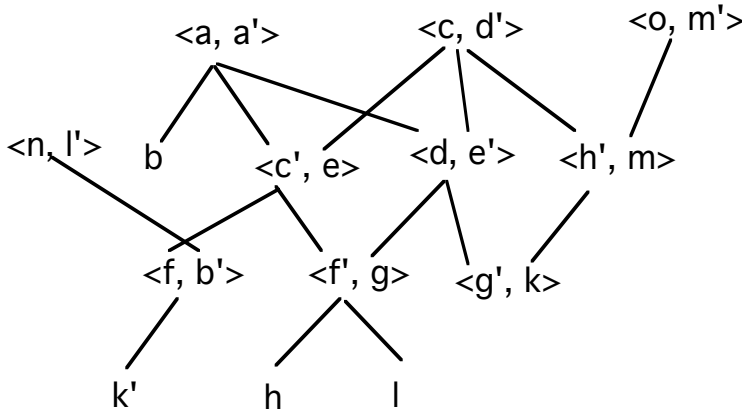
On définit trois relations élémentaires (X ,Y: individus)

pere(X, Y) :	X est père de Y
epouse(X, Y) :	X est épouse de Y
homme(X) :	X est un homme

On dispose également du prédicat femme/1.

femme(X) :- not homme(X).

On se donne le graphe suivant représentant une famille. <X, Y'> dénote un couple où Y' est l'épouse de X. De même, <X', Y> dénote que X' est l'épouse de Y. Tous les X sont définis comme homme. Les X' seront déduits.



Définir une collection de faits *pere/2*, *epouse/2* et *homme/1* d'après cet arbre puis, donner d'une manière incrémentale, les définitions des relations suivantes :

- epoux(X,Y)
- mere(X,Y)
- parent(X,Y)
- enfant(X,Y)
- fils(X,Y)
- fille(X,Y)
- parent(X,P,M)
- ancetre(X,Y)
- gdpere(X,Y)
- gdmere(X,Y)
- gdparent(X,Y)
- frere_ou_soeur(X,Y)
- frere(X,Y)
- soeur(X,Y)
- oncle_ou_tante(X,Y)
- oncle(X,Y)
- tante(X,Y)
- cousin_germain/germaine

- neveux/niece

- beau_frere/belle_soeur/beau_pere/belle_mere/...

Exercice 3:

Définir une base de données concernant des repas et les ingrédients nécessaires pour les préparer. Le nombre d'ingrédients pour préparer un met est inconnu d'avance.

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

ingrédient(m, il_faut(a, il_faut(b, il_faut(c, rien)))).

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

ingrédient(gâteau, il_faut(lait, il_faut(farine, il_faut(oeufs, rien)))).

De même, la base contiendra le fait :

ingrédient(thé, il_faut(sachet_de_thé, il_faut(eau, rien))).

• Définir une relation *disponible(I)* qui décrit les ingrédients disponibles.

• Définir les deux relations :

- *peut_préparer(R)* qui est vraie pour le repas R si tous les ingrédients nécessaires sont disponibles.

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

14. Contrôle de la résolution

Mécanismes pour la maîtrise de l'exécution :

- Echec explicite (*fail*)
- Coupe-choix ou cut (*!*)

Coupe-choix (cut)

Prolog est un langage non-déterministe : il permet de décrire des "ou" logiques et d'énumérer les différentes solutions associées à une question.

Chaque "ou" peut être vu comme un choix : l'interprète Prolog mémorise chaque "ou" dans un "point de choix". Le coupe-choix est utilisé pour couper les “ou” et d’élaguer ainsi l'arbre de résolution. Du point de vue opérationnel, lors que l'interpréteur Prolog rencontre un cut, il "oublie" tous les choix possibles précédant ce cut. Le coupe-choix s'écrit en Prolog par "!".

```
a :- b , c , ! , d , f.
a :- ....
```

Effets du coupe-choix :

- Permet de réduire l'espace de recherche par l'élagage de l'arbre de résolution
- Est toujours effacé (il réussit toujours)
- Lorsqu'il est franchi, contrôle le retour en arrière par:
  - La suppression des points de choix sur les prédicats figurant à sa gauche (sur “b, c” de l’exemple ci-dessus)
  - La suppression des points de choix sur la tête de la clause qui le contient(sur “a” de l’exemple ci-dessus)
- N'a pas d'effet sur sa droite (sur “d, f” de l’exemple ci-dessus)

Classification des cuts

• Coupe choix **vert** : il peut être supprimé. Sa présence améliore les performances et sa suppression ne modifie pas les résultats obtenus.

• Coupe choix rouge : ne peut être supprimé sans modifier les réponses.

Exemples d'utilisation de cut:

```
age(vieux).
age(jeune).
taille(grand).
taille(petit).

choix1(X,Y) :- age(X) , taille(Y).

choix2(X,Y) :- ! , age(X) , taille(Y).

choix3(X,Y) :- age(X) , ! , taille(Y).

choix4(X,Y) :- age(X) , taille(Y) , !.
```

Questions :

:- choix1(X,Y) .
- ❷ X=vieux , Y = grand
- ❷ X=vieux , Y = petit
- ❷ X=jeune , Y = grand
- ❷ X=jeune , Y = petit

:- choix2(X,Y) . ❷ les mêmes réponses

:- choix3(X,Y) . ❷ X=vieux , Y = grand
❷ X=vieux , Y = petit

:- choix4(X,Y) . ❷ X=vieux , Y = grand

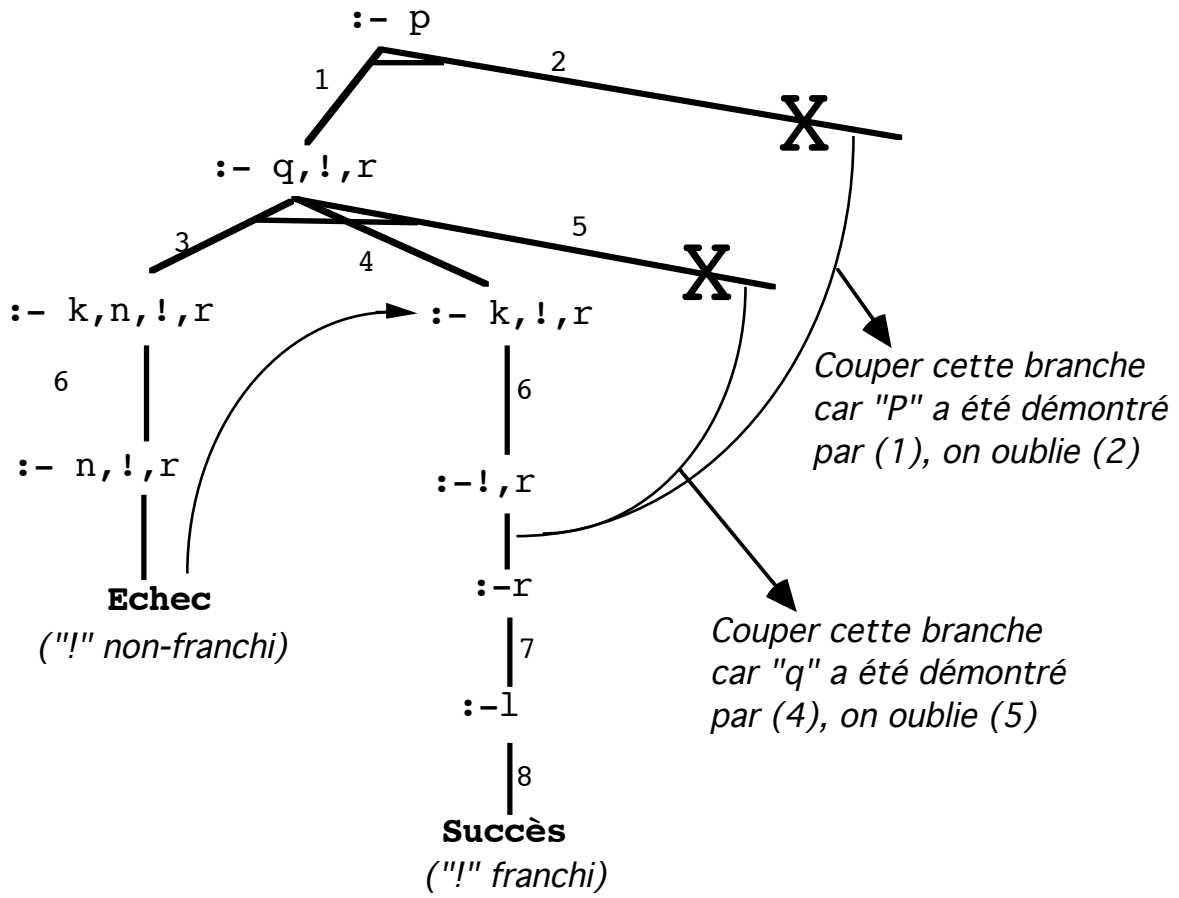
• Le "!" dans choix2 est *Vert*. Sa présence évite de rechercher une autre clause choix2 à deux paramètres. La suppression de ce cut ne modifie donc rien dans les résultats obtenus.

• Le "!" dans choix3 (et dans choix4) est *Rouge*. Sa suppression modifie les résultats obtenus car, sans ce cut, on obtiendrait les mêmes réponses que ceux de choix1.

Effets de cut dans l'arbre de résolution

Exemple:

(1) **p :- q , ! , r.** % le 1er succès de "q" suffit
(2) **p :- l .** % si le "!" de (1) est franchi, on n'essayera pas (2)
(3) **q :- k , n.**
(4) **q :- k.**
(5) **q.**
(6) **k.**
(7) **r :- l.**
(8) **l.**



Effet de coupe-choix dans l'arbre de résolution

❝ *Illustration de l'effet de cut par la pile des retours arrières*

(1) p :- q , ! , r.
(2) p :- l .
(3) q :- s , n.
(4) q :- s.
(5) s.
(6) r :- l.
(7) l.

Question : :- p.

Notation : $!_x$ dénote le (!) figurant dans la règle dont la tête est x.

:- [] p.
  :- [p1] q , $!_{p1}$, r. par(1)
    :- [p1, q3] s , n , $!_{p1}$, r. par(3)
      :- [p1, q3 , s5] n , $!_{p1}$, r. par(5)
        ③ **Echec** sur n, retour en arrière :

    :- [p1, q3] s , n , $!_{p1}$, r.
      ③ d'autres s? non => retour en arrière
  :- [p1] q , $!_{p1}$, r.
    :- [p1, q4] s , $!_{p1}$, r. par(4)
      :- [p1, q4 , s5] $!_{p1}$, r. par(5)

❺ **$!_x$** réussit et vide la pile de droite vers la gauche jusqu'au premier **x** inclue. Ici, on vide la pile jusqu'à p1

:- [] r. par(5)
  :- [r6] l. par(6)
    :- [r6, l7] . par(7)
    plus rien à démontrer => **succès**
    si retour en arrière demandé:

  :- [r6] l. => pas d'autre solution
:- [] r. => pas d'autre solution
la pile est vide, on ne peut plus rien démontrer ==> **Echec**

Exemple avec (!) dans un but:

(1) p :- q , r.
(2) p :- l .
(3) q :- s , n.
(4) q :- s.
(5) s.
(6) r :- l.
(7) l.
-: p , !.

On peut proposer la règle :
$:- p , !.
et poser la question :
:- $.

:- [] $
:- [$] p , !$.
:- [$, p_1] q , r , !$. par(1)
:- [$, p_1, q_3] s , n , r , !$. par(3)
:- [$, p_1, q_3, s_5] n , r , !$. par(5)
Echec sur n, retour en arrière :

:- [$, p_1, q_3] s , n , r ,!$.
d'autres s? non => retour en arrière

:- [$, p_1] q , r , !$.
:- [$, p_1, q_4] s , r , !$. par(4)
:- [$, p_1, q_4 , s_5] , r , !$. par(5)
:- [$, p_1, q_4 , s_5, r_6] l , !$. par(6)
:- [$, p_1, q_4 , s_5, r_6, l_7] !$. par(7)

On vide la pile jusqu'à $:
:- [].
Plus rien à démontrer ==> **succès**
Si retour en arrière demandé ==> **Echec ❞**

Exemple : dérivation formelle

Spécification:

derive(f_X , x , R) calcule R, la dérivée symbolique de f_X par rapport à X où X est atomique et représente symboliquement une variable.

Dans l'expression f_X, tout symbole qui est syntaxiquement différent du paramètre x est considéré comme une constante.

E.g. $f_X = y^2 + a$ est une expression constante.

Du point de vue mathématique, dans *derive(f_X , x , R)*, on fait varier X et on observe f_X.

Les programmes simplifiés ci-dessous traitent les cas simples et l'addition.

➔1ère version : version déclarativement correcte mais ...

```
derive(X, X, 1).
derive(Y, X, 0) :- atomic(Y).          /* constante entière ou atome */
derive(X + Y, Z,  A + B) :-
    derive(X, Z, A) , derive(Y, Z, B).

:- derive(y, x, R).  R = 0              ❷ une seule réponse
:- derive(x, x, R).  R = 1 ; R = 0      ❷ deux réponses
```

Il y a une erreur dans la deuxième réponse. Elle provient de l'application de la seconde règle.

➔ 2ème version : on corrige en rendant les règles 1 et 2 mutuellement exclusives

```
derive(X, X, 1).
derive(Y, X, 0) :- Y \== X , atomic(Y).
derive(X + Y, Z, A + B) :-
    derive(X, Z, A) , derive(Y, Z, B).
```

➔ 3ème version : on utilise le coupe-choix pour améliorer les performances

```
derive(X, X, 1) :- !.                  /* Coup-choix ROUGE */
derive(Y, X, 0) :- atomic(Y) , !.      /*Coup-choix VERT */
derive(X + Y, Z, A + B) :-
    derive(X, Z, A) , derive(Y, Z, B).
```

15. Opérateurs et termes composés

Rappels:

Un terme composé est construit à partir :

-d'une constante (atome non numérique)

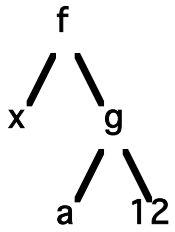
-de n arguments (n≥0) entre parenthèses.

Un atome seul est considéré comme un terme composé à 0 argument.

Tout terme peut être représenté par un arbre dont la racine est le foncteur et dont les fils sont les arguments.

Exemple de représentation canonique

f(x,g(a,12))



L'arbre associé est une représentation canonique (abstraite) et ne dépend pas du mode d'écriture des termes.

Il serait parfois plus agréable de pouvoir représenter les termes composés par une notation préfixée (comme pour *not*) ou en infixée (comme pour ',' , ';').

Pour ce faire, on donne aux constantes atomes un **statut d'opérateur**.

15.1- Opérateurs

Les termes Prolog sont habituellement donnés sous la forme :
$f(arg_1,...,arg_n)$ où f est un atome et $n \geq 0$.

Pour les termes fonctionnels, f est considéré comme l'opérateur et les arg_i les opérandes (rappelons qu'il n'y a pas d'évaluation de ces termes).

Pour rendre les programmes plus lisibles, il serait intéressant de pouvoir par exemple écrire "X + 12" au lieu de '+'(X,12).

Syntaxe de la déclaration d'un opérateur :

OP(Précédence , Associativité , Nom).

Pour déclarer un opérateur, il faut préciser trois informations :

● **Nom de l'opérateur :**

Le nom est un atome non numérique. On encadre les atomes particuliers entre apostrophes.
Exemples de noms d'opérateurs : aime, '+', '==>', si, ...

● **Précédence de l'opérateur :**

Une valeur entière entre 1 et 1200. Lors de la construction de la forme canonique, elle permet de lever l'ambiguïté lorsque les termes se présentent sans parenthèse. C'est donc une valeur relative.

Par exemple, si *op1* est de précédence 100 et *op2* de précédence 200, alors *op2* précède *op1* dans l'écriture (de gauche à droite) des termes faisant intervenir les deux opérateurs. C'est à dire que dans la représentation arborescente de ces termes, le terme construit avec *op1* est sous-terme de celui construit avec *op2*.

Ainsi, sous réserve d'une définition correcte de l'associativité, si *op1* et *op2* sont binaires, le terme "*X op1 Y op2 Z* " est donné par la forme canonique *op2(op1(X,Y), Z)*.

On peut aborder ce point en considérant la valeur de la précédence comme l'inverse de celle de la priorité.
Dans l'exemple ci-dessus, *op1* est plus prioritaire que *op2*; c'est pourquoi le premier terme construit (le terme le plus interne) est *op1(X,Y)*.

● **Associativité :**

L'associativité permet de préciser la position d'un opérateur. Des exemples de positions sont :

infixe :	X **+** 5	p **,** q	a aime b
préfixe :	**\+** oiseau(X)	**not** p	**-**3
postfixe :	X 5 **+**	Y**!**	....

L'associativité permet également de lever l'ambiguïté dans le cas où un terme comporte plusieurs opérateurs de même précédence.

L'associativité s'exprime par les spécifications suivantes où **f** représente l'opérateur et **x** et **y** les arguments.

- Pour la notation infixée : xfx, xfy, yfx
- Pour la notation préfixée : fx, fy
- Pour la notation postfixée : xf, yf

Le choix de **x** et de **y** permet d'indiquer l'associativité. En l'absence de parenthèses, un **y** indique que l'argument à cette place peut contenir des opérateurs de précédence égale ou inférieure à celle de l'opérateur; un **x** indique que tous les opérateurs de l'argument doivent avoir une précédence strictement inférieure à celle de l'opérateur considéré **f**.

Exemple1 : considérons l'opérateur + déclaré *yfx*.

L'expression *a+b+c* a deux interprétations possibles : *(a+b)+c* et *a+(b+c)*. Dans la deuxième écriture, on a à droite du premier + un opérateur de précédence égale; or la présence de *x* dans *yfx* contredit cette situation. La première écriture est donc choisie par la présence d'*y*.

En d'autres termes, *yfx* étant associatif à gauche; *(a+b)+c* associatif à gauche est la forme correcte.

p un **y** à droite de **f** précise une associativité à droite.

un **y** à gauche de **f** précise une associativité à gauche.

Définir l'associativité d'un opérateur :

spécification	*signification*
fx	préfixe, non associatif (comme le moins unaire)
fy	préfixe, associatif à droite (comme not)
xf	postfixe, non associatif
yf	postfixe, associatif à gauche
xfx	infixe, non associatif (comme =)
xfy	infixe, associatif à droite (comme ;)
yfx	infixe, associatif à gauche (comme +)

Exemple2 :

Avec l'opérateur *not* déclaré fy, on peut écrire *not not p*. Ce qui serait illégal si *not* était déclaré fx.

Exemple 3 : avec

:- op(500, yfx, +) , op(400, yfx, *).

L'expression *a+b*c* est équivalente à *a+(b*c)* ou encore à *+(a,*(b,c))*.

Exemple 4:

Pour pouvoir écrire *A aime B et C* aussi bien que *aime(A, et(B,C))* ; on déclare :

:- op(80, xfy, et).
:- op(100, xfx, aime).

On peut demander la forme canonique par *display* :
:- display(jean aime marie et paul). ➋ aime(jean, et(marie, paul)).

Si les déclarations de ces opérateurs devaient donner à *aime* une précédence inférieure à celle donnée à *et*, on écrirait :

:- op(110, xfy, et).
:- op(100, xfx, aime).

:- display(jean aime marie et paul). ➋ et(aime(jean, marie), paul).

p *Lorsqu'un symbole est déclaré comme un opérateur, les termes composés construits avec ce symbole peuvent être représentés à la fois sous la forme préfixée et sous la forme externe conforme à la déclaration que l'on aura faite.*

Par exemple, on peut écrire *jean aime marie* aussi bien que *aime(jean, marie)*.

Déclaration de quelques opérateurs importants :

Les opérateurs suivants sont prédéclarés en Prolog. On peut déclarer plusieurs opérateurs ayant les mêmes caractéristiques en les présentatnt entre [] (sous la forme d'une liste).

```
op(500, yfx,  ['+', '-']).              % le '-' et le '+' binaires
op(400, yfx,  ['*', '/', mod]).         % le '*', le '/' et l'opérateur modulo
op(500, fx, ['-', +]).                  % les '-' et '+' unaires
op(1200,   xfx,  ':-').                 % comme dans tête :- corps.
op(1100,   xfy,  ';').                  % le disjonction (le ou)
op(1050,   xfy,  '->').                 % le conditionnel
op(900, fy, ['\+', not]).               % les deux formes de négation
op(1000,   xfy,  ',').                  % le et
op(1200,   xfx,  '-->').                % opérateur de règle de production
op(1200,   fx,   [':-', '?-']).         % comme dans :- but ou dans ?-but
op(200, xfy,  '^').                     % opérateur 'il existe'
op(700, xfx,  ['=..']).                 % opérateur de passage arbre ⇔ liste
op(700, xfx,  ['@>=', '@=<', '@>', '@<']).      % comparateurs de termes
op(700, xfx,  ['\==', '==', '\=', '=']).        % opérateurs de coïncidence
op(700, xfx,  ['=\=', '=:=', '<', '>', '>=', '=<', is]). % opérateurs arithmétiques
```

Exemple de définition d'opérateurs

Pour définir un opérateur, il faut considérer deux points :

- les caractéristiques de l'opérateur (nom, position, associativité)
- les caractéristiques des autres opérateurs

En effet, on doit considérer comment les termes que l'on construit avec les nouveaux opérateurs inter-agissent avec ceux déjà définis.

Pour ce faire, on considère la forme canonique des termes à construire.

Exemple : on veut définir les opérateurs **et** et **ou** pour pouvoir écrire :

(1)	a et b	dont la forme canonique est	et(a,b)
(2)	a ou b	dont la forme canonique est	ou(a,b)
(3)	a et b et c	dont la forme canonique est	et(a, et(b,c))
(4)	a ou b ou c	dont la forme canonique est	ou(a, ou(b,c))
(5)	a et b ou c	dont la forme canonique est	ou(et(a,b), c))
(6)	a ou b et c	dont la forme canonique est	ou(a, et(b,c))

(1) et (2) nous informent que **et** et **ou** sont binaires et infixés (choix entre xfx, xfy ou yfx).

(3) et (4) précisent que **et** et **ou** sont associatifs à droite (donc xfy).

Enfin, (5) et (6) nous informent que **et** est plus prioritaire que **ou** (c'est à dire, la précédence de **ou** est supérieure à celle de **et**).

D'où les déclarations :

```
:- op(15, xfy, ou).
:- op(10, xfy, et).
```

```
:- display(a et X ou b et Y).
   ③ ou(et(a, X),et(b,Y))
```

Les déclarations ci-dessus suffisent pour construire les termes qui ne font intervenir que les opérateurs **et** et **ou**. Par contre, si l'on doit écrire des termes tels que *not a et b*, on doit prendre en compte les caractéristiques de *not*.

Etant donné la déclaration :

```
:-  op(900, fy, not).
```

on obtient :

```
:- display(not a et b).
   ③  not(et(a,b)).
```

Si l'on veut obtenir la forme canonique *not(a) et b*, la précédence de **et** (et de **ou**) doit être supérieure à 900.

On peut par exemple écrire :

```
:- op(920, xfy, ou).
:- op(910, xfy, et).

:- display(not a et b).
   ③  et(not(a), b).
```

Nous traitons en exercice un exemple complet de déclaration d'opérateurs utilisé dans la définition d'un petit langage de programmation.

16. Listes

- Le seul objet syntaxique manipulé par Prolog est le **terme** regroupant les constantes, les variables et les termes composés. Tout terme peut être représenté par un arbre (la représentation canonique).

- Une liste est un terme composé construit à l'aide de l'opérateur binaire point (‘.’) dont les deux arguments sont des termes. Le premier argument de cet opérateur est appelé **tête**; le second **queue**.

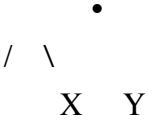
- Par exemple, *‘.’(a, ‘.’(f(g), ‘.’(c, 1)))* est une liste dont la tête est ‘a’ et dont la queue est la liste *‘.’(f(g), ‘.’(c, 1))*.

- Pour simplifier l’écriture, on peut placer les éléments d’une listes entre [et]. Dans cette écriture, on sépare les éléments de la liste par une virgule, le dernier terme de la liste étant précédé du symbole ‘|’. Par exemple, la liste donnée ci-dessus peut être représentée par *[a, f(g), c | 1]*. Ce symbole ne peut être utilisé qu’une seule fois dans une liste.

- Par convention, la constante *nil* notée *[]* représente la liste vide. Elle peut être omise lors qu’elle est le dernier terme de la liste dans la notation entre [et]. Ainsi, une écriture telle que *[a, b, c | []]* est équivalente à *[a, b, c]*.
  Pour harmoniser les écritures, il est préférable de terminer une liste par [] (comme dans [a, b, c]) sans quoi l’écriture des prédicats de manipulation de listes sera plus complexe.

Exemples de listes :

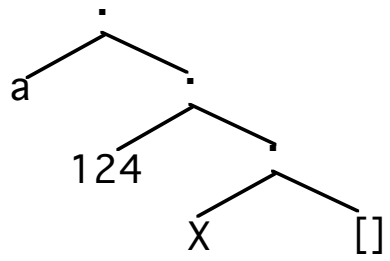
- [X , Y] : tête= X, queue = [Y]
- [a,b,c]: tête= a, queue = [b,c]
- [f(X), g([a])] : tête= f(X), queue = [g([a])]
- [[a,b(12)],c] : tête = [a,b(12)] , queue = [c]
- [X | Y] : tête= X , queue = Y
- [a | b] : tête= a , queue = b
- [a,b|c] : tête= a, queue = [b|c]



- [a , b] = [a, b | []] : tête= a , queue = [b]
- [a, b, []] : tête = a, queue = [b, []]

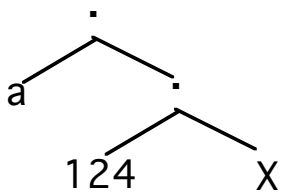
- Représentation canonique de **[a , 124 , X]**:

. (a , (. (124 , . (X , [])))



- Représentation canonique de **[a , 124 | X]**:

. (a , (. (124 , X)))



- Dans les exemples suivants, on utilise le prédicat d'unification '='

  ?- [a , b | c] = [X|Y]. ➋ X = a, Y = [b|c]

  ?- [a , b] = [X|Y]. ➋ X = a, Y = [b]

  ?- [a | b] = [X|Y]. ➋ X = a, Y = b

  ?- [X, a | Y] = [b,a,c,d] ➋ X = b, Y = [c,d]

- Définition récursive d'une liste en Prolog :

```
liste([]).
liste([Tete | Queue]) :- liste(Queue).
```

Exemples de manipulation de listes

- Exemple-1:

ajouter(ELE, LISTE, LISTE1) vrai si LISTE1 est une liste dont le terme en tête est ELE et dont la queue est LISTE

```
ajouter(X, L, [X|L]).
```

Exemples d'interrogation :

```
:- ajouter(a, [b,c], [a,b,c]).    =>    succès
:- ajouter(X, [a], [X,a]).        =>    succès
:- ajouter(a,L, [a,b]).           =>    L= [b]
:- ajouter(a, [b], [b,a]).        =>    échec
:- ajouter(X, Y, [a,b,c]).        =>    X=a, Y=[b,c]
```

Détails de résolution pour la question :- ajouter(a, [b], [b,a])

L'unification de *ajouter(a, [b], [b,a])* et de ajouter(X, L, [X|L]) revient aux trois unifications suivantes :

- X =? a => X est unifié à 'a' => θ = {X/a}
- L =? [b] => L est unifié à [b] => θ = {X/a, L/[b]}
- [X|L] =? [b, a] => on remplace X et L dans [X|L],
  c'est à dire : θ([X|L]) = [a, b]

La troisième unification revient donc à : [a, b] =? [b, a]
Les deux termes étant différents, l'unification échoue.

Détails de résolution pour la question :- ajouter(a, [b,c], [a,b,c]) :

L'unification de *ajouter(a, [b,c], [a,b,c])* et de ajouter(X, L, [X|L]) revient aux trois unifications suivantes :

- X =? a => X est unifié à 'a' => θ = {X/a}
- L =? [b, c] => L est unifié à [b, c] => θ = {X/a, L/[b, c]}
- [X|L] =? [a, b, c] => on remplace X et L dans [X|L],
  c'est à dire : θ([X|L]) = [a, b, c]

  La troisième unification revient donc à : [a, b, c] =? [a, b, c]
  Les deux termes étant identiques, l'unification réussit.

- Exemple-2:
  membre(Ele, Liste) vrai si Ele appartient à Liste

Raisonnement récursif structurelle :

Nous avons vu qu'une liste était définie par :

Liste : vide ou bien un terme suivi d'une liste.

Exprimé en Prolog, on obtient :

liste(Liste) :Liste=[] ; Liste=[Tete|Queue] , liste(Queue).

Pour savoir si un élément X appartient à une liste, l'on doit décider des traitements dans chaque cas de cette définition. On aura :

- Aucun élément n'appartient à la liste vide. Ce qui donne en Prolog
  (1) membre(X, L) :- L=[], fail.

- Lorsque la liste n'est pas vide, l'élément X appartient à la liste L si la tête de la liste L est égale à X (en Prolog : s'unifie avec X). D'où
  (2) membre(X, L) :- L=[X | Q].

- Pour obtenir toutes les réponses, lorsque la liste n'est pas vide, l'élément X peut également figurer dans la queue de liste. D'où :
  (3) membre(X, L) :- L=[Y|Q], membre(X, Q).

Améliorations :

Grâce au mécanisme d'unification, on peut réécrire les clauses ci-dessus en :

(1') *membre(X,[]) :- fail.*
(2') *membre(X, [X | Q]).*
(3') *membre(X, [Y|Q]) :- membre(X, Q).*

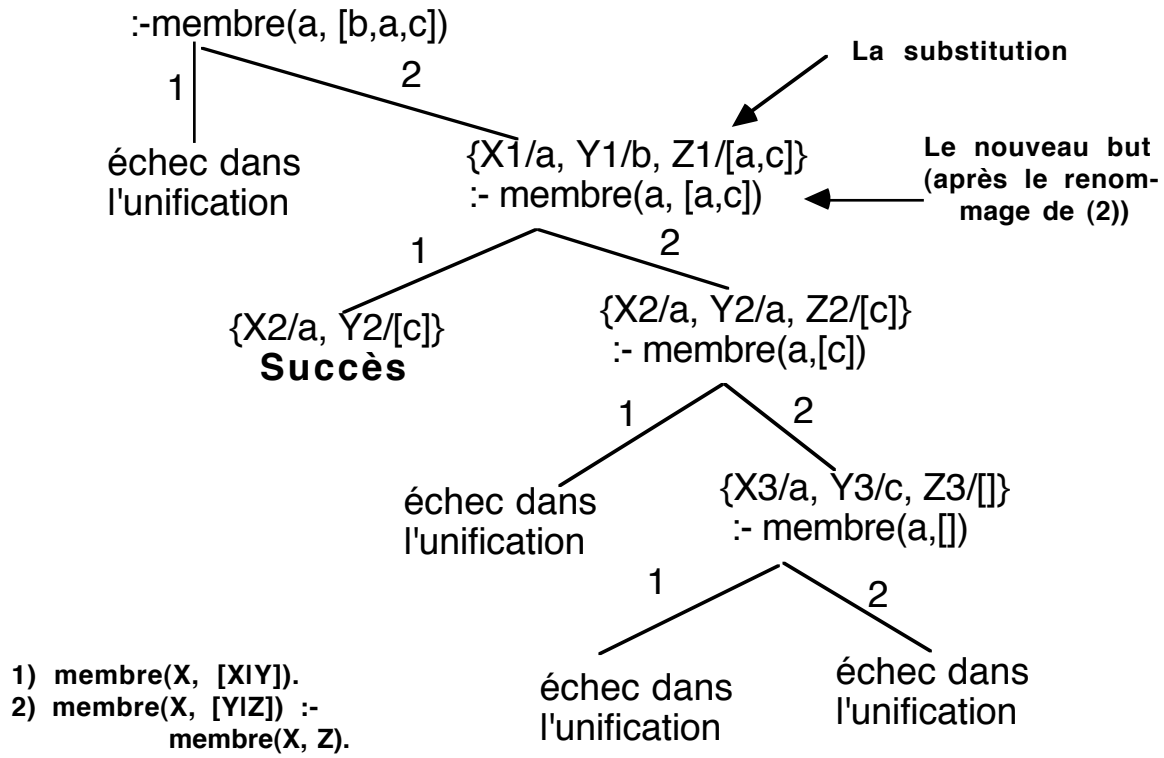
De plus, on constate que si la liste est vide, les clauses (2') et (3') suffisent pour provoquer un échec car ces clauses ne seront pas appliquées et l'on obtiendrait par conséquence un échec; ce qui est exprimé par (1'). On peut donc faire l'économie de la clause (1'). La version finale de membre sera :

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

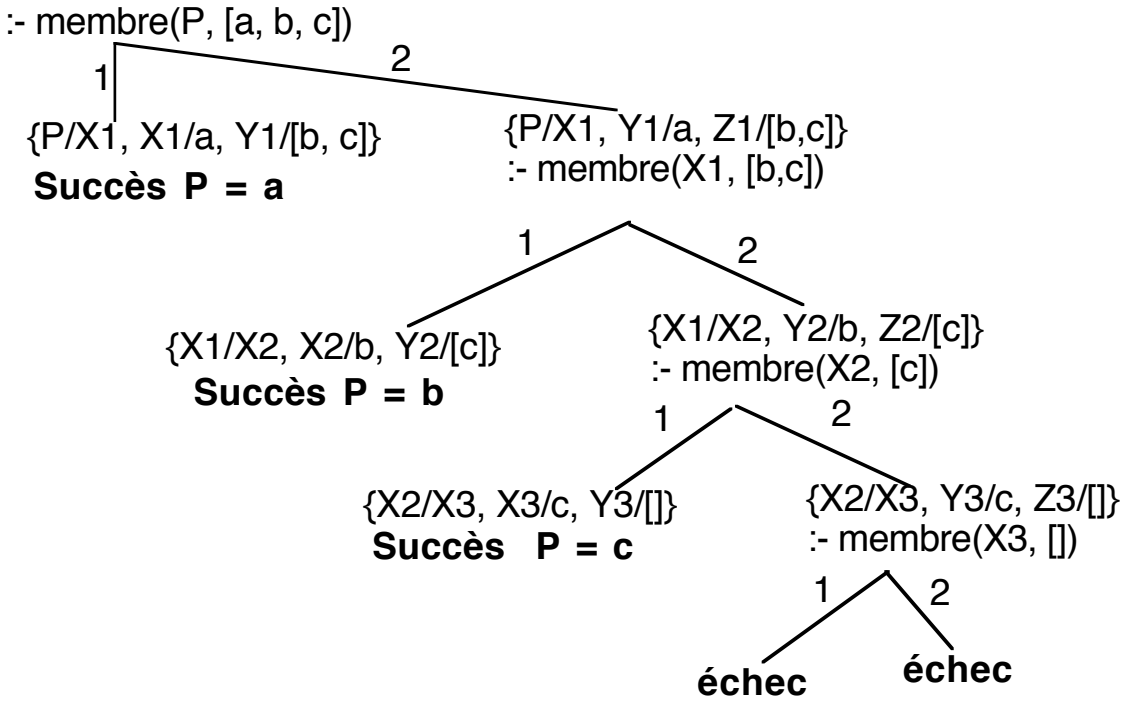
Exemples d'interrogation :

:-membre(a , [b,a,c]) => succès
:-membre(a , [b,a,c,a]) => 2 succès
:-membre(a , [b,d,c]) => échec
:-membre(X , [b,a,c]) => X = b ; X = a ; X= c
:-membre(a , [b|X]) => X= [a|X1]; X= [X2,a|X1]; X= [X2,X3,a|X1]; ...
une infinité de succès
:-membre(b,[a|b]) => échec *car dans [a|b], la queue n'est pas une liste*
:-membre(a,[a|b]) => succès

Détails de résolution pour la question :-membre(a , [b,a,c])



Détails de résolution pour la question :-membre(P , [a, b, c])



Si l'on veut obtenir une seule réponse du prédicat membre:

- On pose la question avec un cut (rouge).

  :-membre(X , [b,a,c]) , !. => X = b

  ;échec

  :-membre(a , [b,a,c,a]) , !. => un seul succès

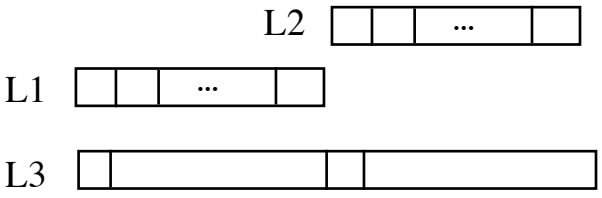
- On ajoute '!' (cut rouge) dans la première clause :

```
membre(X,[X|Y]) :- !.
membre(X,[Y|Z]) :- membre(X,Z).
```

:-membre(X , [b,a,b]) => X = b
:-membre(a , [b,a,c]) => succès
:-membre(a , [b,d,c]) => échec
:-membre(a , [b|X]) => X=[a | X1]

Exemple-3 : concaténation de deux listes

concat(L1, L2, L3) vrai si L3 est le résultat de la concaténation de L1 et de L2.



Raisonnement :

L1 = [] ==> concat(L1, L2, L2).
L2 = [] ==> concat(L1, L2, L1).
L1 et L2 ≠ [] ==> L1 = [X| Q1], concat(Q1, L2, Q2), L3 = [X| Q2].

D'où la vesrion améliorée de Prolog :

(1) concat([], L, L).

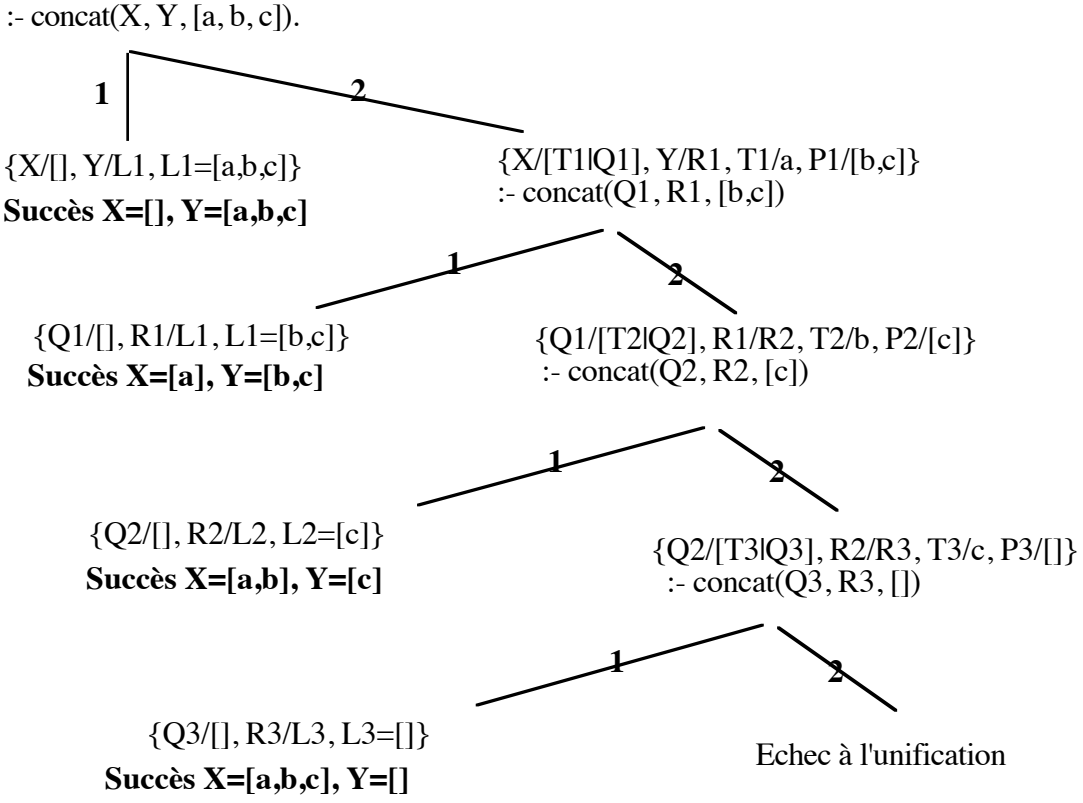
(2) concat([X|Y], L1, [X|L2]) :- concat(Y, L1, L2).

:- concat([a,b], [c,d], L). => L= [a,b,c,d]

:- concat(X, [a,b], Y). => X = [], Y = [a,b] ;
X = [X1], Y = [X1,a,b] ;
X = [X1,X2], Y = [X1,X2,a,b] ;.....

:- concat(X, Y, [a,b,c]).=>X = [] , Y = [a,b,c] ;
X = [a] , Y = [b,c] ;
X = [a,b] , Y = [c] ;
X = [a,b,c] , Y = []

Détails de résolution de :- concat(X, Y,[a,b,c]



Exemple 4: Intersection de deux listes ordonnées.

intersec(L1, L2, L3) vrai si L3 est l'intersection de L1 et de L2

```
intersec([], _, []).
intersec([X|Y], Z, [X |T]) :-
    membre(X, Z) , intersec(Y, Z, T).
intersec([X|Y] , Z , T) :-
    not membre(X, Z) , intersec(Y, Z, T).
```

not membre recalcule *membre* pour déduire sa négation.

A l'aide de "!", on peut éviter de recalculer *membre* .

```
intersec([], _, []) :- !.                    /* !1 : cut vert */
intersec([X|Y], Z, T) :-                        /* !2 : cut rouge */
    not membre(X, Z) , ! , intersec(Y, Z, T).
intersec([X|Y], Z, [X |T]) :- intersec(Y, Z, T).
```

Remarque :

Dire que la présence d'un cut ne modifie pas un programme, c'est connaître et examiner tous les cas de figure et d'utilisation de ce programme; ce qui est rarement une réalité. C'est pourquoi il faut éviter l'utilisation abusive ou non certaine de cut dans les programmes qui risquent de ne plus répondre à leur spécification non déterministe.

Le prédicat ci-dessus doit être utilisé pour calculer une intersection ou bien, pour tester si L3 est effectivement l'intersection de L1 et de L2.

De plus, les listes considérées sont ordonnées :

intersec([a,b],[b,a],[a,b]) réussit alors que intersec([a,b],[b,a],[b,a]) échoue.

Une utilisation incohérente de ce prédicat peut produire des résultats abhérents.

Classification des "!" utilisés dans l'utilisation habituelle du prédicat intersec :

- $!_1$ est **vert**. Sa présence améliore le comportement opérationnel du programme
- $!_2$ est **rouge**. Sa suppression laisse une définition ne correspondant plus à la spécification de l'intersection

16.1- Quelques exemples de manipulation de listes

❏ Relation binaire shift_gauche : shift_gauche(Liste, Liste_décalée)

Exemple : shift_gauche([a,b,c,d],L) ==> L= [b,c,d,a]

```
shift_gauche([X|Y], Z) :- concat(Y, [X], Z).
```

❏ *efface(Ele, Liste, Liste_restante)* : vrai si *Liste_restante* est *Liste* sans *Ele*.

```
efface (X, [X|Q], Q).
efface (X, [Z|Q], [Z|Q1]) :- efface (X, Q, Q1).
```

:- efface(a, [a,b,c], L).

③ L = [b, c]

:- efface(X, [a,b,c], L).

③ X = a, L = [b, c]

X = b, L = [a, c]

X = c, L = [a, b]

❏ *dernier(Ele, Liste)* : vrai si *Ele* est le dernier éément de *Liste*.

```
dernier (X, [X]).
dernier (X, [T|Q]) :- dernier (X, Q).
```

ou

```
dernier (X, L) :- concat (_,[ X], L).
```

❏ *prefixe(Pref, Liste)* vrai si *Pref* est préfixe de *Liste*

```
prefixe ([], _).
prefixe ([X|Q], [X|Q1]) :- prefixe (Q, Q1).
```

:- prefixe(X, [a,b,c]).

③ X = [] ;

X = [a] ;

X = [a,b] ;

X = [a,b,c]

❏ *suffixe(Suff, Liste)* vrai si *Suff* est un suffixe de *Liste*

```
suffixe (X,X).
suffixe (X,[Y|Q]) :- suffixe (X,Q).
```

:- suffixe(X, [a, b, c]).

③ X = [a,b,c] ;
X = [b,c] ;
X = [c] ;
X = []

❏ Recherche de sous-listes : *sous_liste(Sous_liste, Liste)*

```
sous_liste (X, Y) :- prefixe (X, Y).
sous_liste (X, [Y|Z]) :- sous_liste (X, Z).
```

:- sous_liste(X,[a,b,c]).

③ X = [] ; X = [a] ; X = [a,b] ; X = [a,b,c] ;
X = [] ; X = [b] ; X = [b,c] ;
X = [] ; X = [c] ; X = [] ;

Autres solutions pour sous_liste :

```
sous_liste (X,Y) :- prefixe (P,Y) , suffixe (X,P).
```

ou

```
sous_liste (X,Y) :- concat (_,L2,Y) , concat (X, _,L2).
```

Remarque : améliorer sous_liste pour éviter les solutions X=[] redondantes.

❏ Permutation des éléments d'une liste : *permute(Liste, Liste_permutée)*

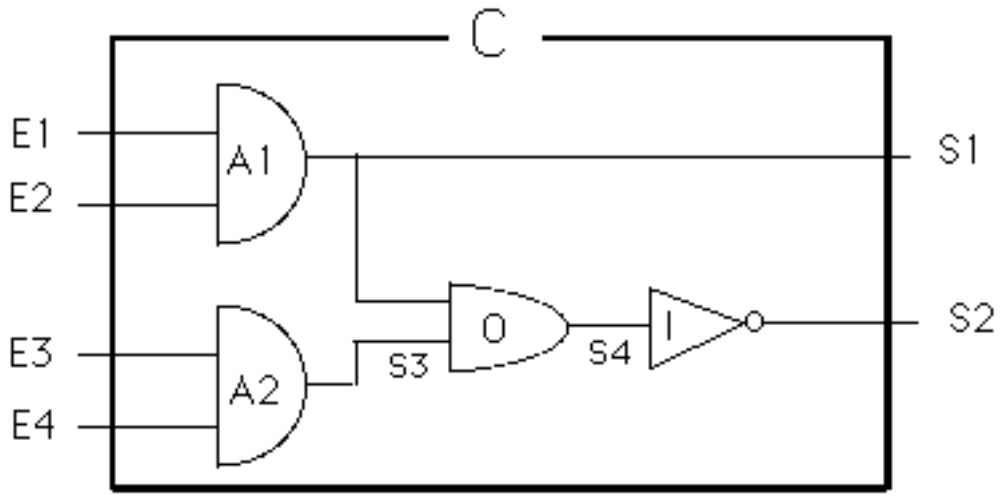
```
permute ([], []).
permute (L, [X|Q]) :-
    efface (X, L, L1) , permute (L1, Q).
```

:- permute([a,b,c],X).

③ X = [a,b,c] ;X = [a,c,b] ; X = [b,a,c] ;
X = [b,c,a] ; X = [c,a,b] ; X = [c,b,a]

Remarque : donner une trace de résolution pour la question ci-dessus.

Exemple : Description de circuits avec des listes:




```
circuit('C', [E1,E2,E3,E4] , [S1,S2]) :-        % l'atome 'C' est le nom du circuit
        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 ]).
```

- Test : avec entrées et sorties connues.
  :- circuit('C',[1 , 1 , 0 , 1], [0 , 1]). ❷ échec

- interrogation :
  - Entrées connues, sorties demandées
    :- circuit(X, [1 , 1 , 0 , 1], [S1, S2]).
    ③ X = 'C' , S1 = 1 , S2 = 0

    :- circuit(X,[1 , 1 , 0 , 1], S).
    ③ X = 'C' , S = [1, 0]

    :- circuit(X,[1 , 1 , 0 , 1], [1,S2]).
    ③ X = 'C' , S2 = 0

  - Sorties connues, entrées demandées
    :- circuit(X,[E1 , E2 , E3 , E4], [0 , 1]).
    ③ 9 réponses

    :- circuit('C',[E1 , E2 , E1 , E4], [0 , 1]).
    ③ 5 réponses

    :- circuit('C',[E1 , 0 , E1 , E4], [0 , 1]).
    ③ 3 réponses

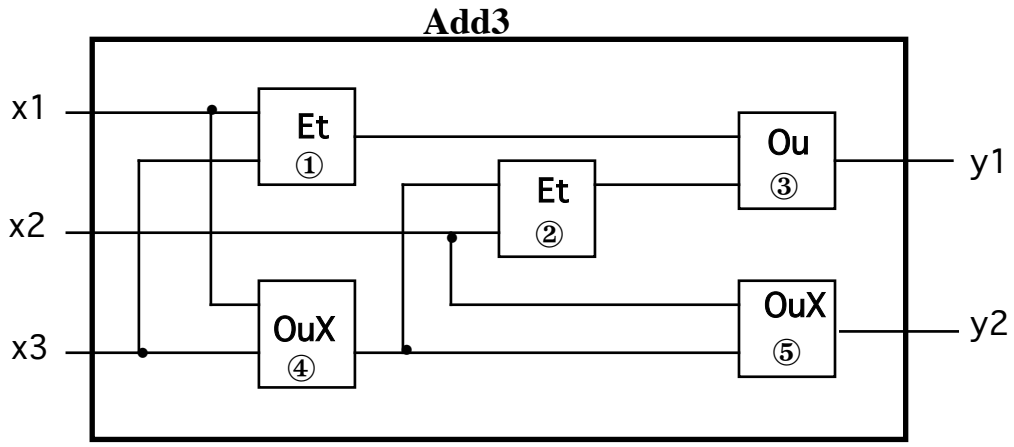
    :- circuit('C',E, [0 , 1]).
    ③ 9 réponses

    :- X = [0 , 1] , circuit('C', E , X).
    ③ 9 réponses

  - génération : entrées et sorties inconnues
    :- circuit('C', E, S).
    .....

Exercice 4:

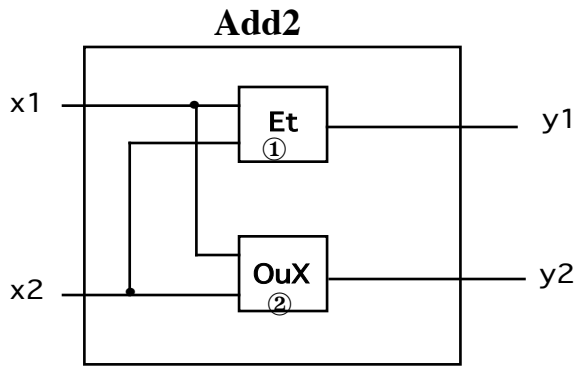
- Décrire l'additionneur 3 bits ci-dessous.



NB: *OrX* est un OU exclusif dont la table de vérité est :

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

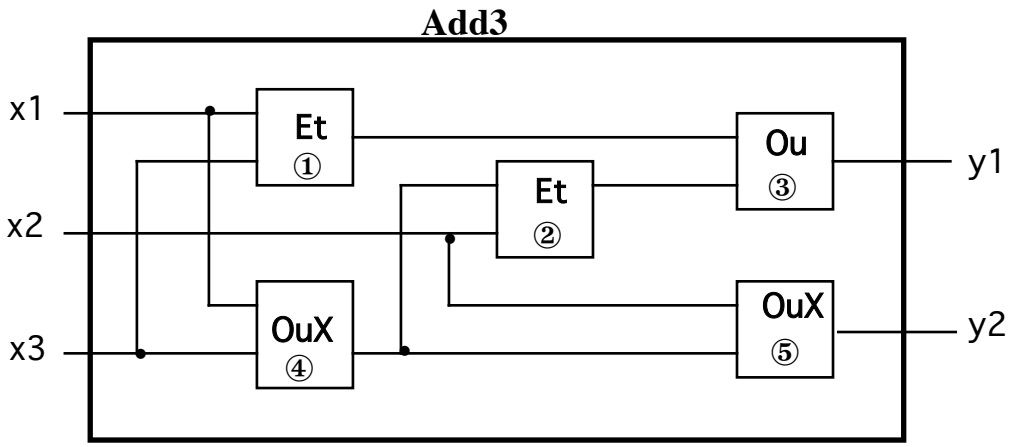
- A l'aide de l'additionneur de deux bits ci-dessous, réaliser un additionneur de 8 bits.



Indication : créer d'abord un additionneur 4 bits à l'aide de ceux ci-dessus.

Utilisation de Prolog en détection de pannes de circuits :

Considérons le circuit



En fonctionnement normal, ce circuit réalise l'addition de 3 bits. Par exemple, si le vecteur d'entrée <X1,X2,X3>=<1,1,1>, on obtient le vecteur de sortie <Y1, Y2> = <1,1>, ce qui représente le nombre décimal 3, la somme des entrées.

Si pour un vecteur d'entrée donné, le résultat, n'est pas celui escompté, on conclut sur une panne du circuit. Pour simplifier le problème, on supposera qu'une panne dans le circuit ne peut provenir que d'une seule des portes à la fois. Par exemple, si <X1, X2, X3> = <1, 1, 0> et <Y1, Y2> = <0, 1>, on doit conclure que (seule) la porte ↘ est peut être en panne. De même, pour <X1, X2, X3> = <1, 0, 1> et <Y1, Y2> = <0, 0>, on conclut que soit la porte ♣, soit la porte ⑦ est peut être en panne.

Le programme ci-dessous décrit le principe de détection de pannes sous l'hypothèse de panne unique pour le circuit spécifique add3.

- **Description des portes élémentaires :**

```
and([0,0],[0]).   and([0,1],[0]).   and([1,0],[0]).   and([1,1],[1]).
or([0,0],[0]).    or([0,1],[1]).    or([1,0],[1]).     or([1,1],[1]).
orx([1,1],[0]). orx([0,1],[1]).   orx([1,0],[1]).    orx([0,0],[0]).
```

- **Description du circuit pour le test de bon fonctionnement**

```
circuit([X1,X2,X3], [Y1,Y2], [P1,P2,P3,P4,P5]) :-
    and([X1,X3],[U1]) , and([X2,U3],[U2]), or([U1,U2],[Y1]),
    orx(_,[X1,X3],[U3]), orx(_,[X2,U3],[Y2]) , ! ,
    write('aucun problème') , nl.
```

- **Prédicat de détection de panne spécifique au circuit étudié**

```
circuit([X1,X2,X3], [Y1,Y2], [P1,P2,P3,P4,P5]) :-
    au_plus_un_vrai([P1,P2,P3,P4,P5]),
  (P1=0 -> and([X1,X3], [U1]);     P1=1 ) ,
  (P2=0 -> and([X2,U3], [U2]);     P2=1) ,
  (P3=0 -> or([U1,U2], [Y1]);   P3=1) ,
  (P4=0 -> orx([X1,X3],[U3]);  P4=1) ,
  (P5=0 -> orx([X2,U3],[Y2]);  P5=1),
    write([P1,P2,P3,P4,P5]) , nl , fail.          %fail pour chercher d'autres pannes
```

• Respect de la contrainte de panne unique.

• Générer une liste de 0/1 dont un seul élement est à 1

```
au_plus_un_vrai(P) :- au_plus_un_vrai(P,_).
au_plus_un_vrai([],0).
au_plus_un_vrai([P|Ps], R) :-
            au_plus_un_vrai(Ps, Q),or([P,Q],[R]), and([P,Q],[0]).
```

Exemples d'interrogations :

```
   :- circuit([1,1,0], [0,1], [P1,P2,P3,P4,P5]).
      ==>  [0,0,0,1,0]      C'est à dire, la 4ème porte peut être en panne

   :- circuit([1,0,1], [0,0], [P1,P2,P3,P4,P5]).
      ==> [1,0,0,0,0]   C'est à dire, la 1ère porte peut être en panne
      ==> [0,0,1,0,0]   C'est à dire, la 3ème porte peut être en panne
```

Exercice-5

•1 Soit la liste des mois Mois=[jan, fev, ..., dec]. Répondre aux questions suivantes à l'aide du prédicat de concaténation :

- quel est le mois qui précède le mois de juin ?
- quel est le mois qui suit le mois de septembre ?
- quel est le mois qui précède le mois de mars et celui qui le suit ?

•2 Ecrire le prédicat **est_triee(Liste)** qui réussit lorsque les éléments de la liste Liste sont ordonnés (dans l'ordre croissant).

Remarque:

Utiliser '@<' (plus petit) et '@=<' (plus petit ou égal) pour comparer deux élements.
Par exemple, X @< Y.

•3 Ecrire le prédicat **merge(Liste1, Liste2, Liste3)** qui fusionne les deux listes ordonnées Liste1 et Liste2 pour donner la liste ordonnée Liste3.

•4 Calculer la longueur (nombre d'éléments) d'une liste. On utilisera les symboles *0* et *succ* pour representer les entiers.

•5 Ecrire le prédicat quick-sort puis, le modifier en évitant d'utiliser la concaténation.

17. Opérations sur les termes

17.1- Unification et comparaison de termes

❑ Unification :

- **X = Y** X et Y s'unifient t produisent une substitution θ.

  :- f(a,X,Z) = f(X, a , b). ==> θ = {X/a, Z/b}
  :- X = X. ==> θ = {}
  :- X = g(b, [c|Y]). ==> θ = {X/g(b, [c|Y])}

- **X \= Y** X et Y ne s'unifient pas (*X \= Y* est équivalent à *not(X=Y)*)

  :- f(a,X,Z) \= f(X, b , b). ==> succès

❑ Comparaison et Coïncidence:

Les comparaisons se font suivant l'ordre défini sur les termes.

Lorsqu'une variable intervient dans une comparaison, elle est remplacée par sa valeur (si elle est instanciée).

L'ordre défini sur les termes est l'ordre **croissant** suivant :

- Les variables par leur ordre d'ancienneté (e.g. l'ordre d'entrée au clavier)
  (e.g. le but **:- X@<Y** réussit ainsi que le but **:- Y @< X**.)

- Les entiers par leur ordre numérique
- Les réels par leur ordre numérique
- Les atomes classés dans l'ordre alphanumérique (ASCII)
- Les termes composés classés par leur artié (numérique) puis par leur foncteur (alpha-numérique) et enfin par leurs arguments (alpha-numérique) et de gauche à droite.

- **X == Y** X et Y littéralement identiques

  :- X == X. => succès
  :- a == a => succès

- **X \== Y** X et Y littéralement différents

  :- X \== Y => succès
  :- a \== b => succès

p :- X=Z , Y= K, X = Y => succès car X s'unifie avec Y
:- X=Z , Y= K , X == Y => échec car X est littéralement différent de Y

- **X @< Y** X est inférieur à Y dans l'ordre standard (défini ci-dessus)
- **X @=< Y** X est inférieur ou égal à Y dans l'ordre standard
- **X @> Y** X est supérieur à Y dans l'ordre standard
- **X @>= Y** X est supérieur ou égal à Y dans l'ordre standard

- **Compare(Opérateur, Terme1,Terme2)**
  Prédicat de comparaison général. Les opérateurs autorisés sont :
  '=' (pour ==), '>' (pour @>) et '<' (pour @<).

Hors mis son utilisation pour comparer des termes, *compare* est utilisé comme un prédicat d'ordre supérieur. Ainsi, on peut demander la relation d'ordre (un opérateur parmi '=', '>' et '<') qui existe entre deux termes :

:- compare(X, a, b). ==> X= '<'

17.2- Arithmétique

- Les expressions arithmétiques sont construites à l'aide des opérateurs suivants:

+ : addition des nombres
- : soustraction (et le - unaire préfixé)
***** : multiplication
/ : division
mod : reste de la division
abs : valeur absolue (unaire)

Les expressions construites avec ces opérateurs sont évaluées par les opérateurs suivants :

- Les expressions peuvent être évaluées puis être comparées entre elles par les opérateurs infixés suivants :

< : inférieur
=< : inférieur ou égal
> : supérieur
>= : supérieur ou égal

NB : '=' et \=' ne provoquent pas d'évaluation de leur paramètre.

- Pour tester l'égalité (et l'inégalité) de deux expressions arithmétiques, on dispose de **=:=** et **=\=** :

E1 =:= E2 ❷ teste l'égalité de E1 et de E2 après les avoir évalué.
E1 =\= E2 ❷ teste l'inégalité de E1 et de E2 après les avoir évalué.

Exemple :

X=20, 3*4-2 =:= X/2 ==> succès

- Enfin, l'opérateur infixé IS permet de confronter deux expressions arithmétiques. Dans "X is Y", l'expression arithmétique Y est évaluée puis "affectée" à X (si X est variable) ou confrontée à la valeur de X.

Exemples :

:- X is 2 , Y is X+1. ❷ X=2, Y=3

:- X= 2, Y is X+1. ❷ idem
:- X is 2, Y= X+1. ❷ X=2, Y=2+1 (2+1 ce n'est pas = 3).
:- X = 2, Y = X+1. ❷ idem
:- 4+3 is 10-3 ❷ échec car on n'évalue pas à gauche de 'is'
:- 4+3 =:= 10-3 ❷ succès
:- 2 is 5-3 ❷ succès

Exemples d'utilisation :

- Les entiers par les axiomes de Péano :

```
entier(0).
entier(N) :- entier(M) , N is M+1.
```

:- entier(5). ==> succès
:- entier(N). ==> 0; 1; ...; infini

- PGCD de deux nombres :

```
pgcd(X,X,X).
pgcd(X,Y,D) :-
   X < Y , Y1 is Y - X, pgcd(X,Y1,D).
pgcd(X,Y,D) :-
   Y < X , pgcd(Y,X,D).
```

?-pgcd(13,78,J). ❷ J = 13.

- Valeur absolue :

```
vabs(X,X) :- X > 0 ,!.
vabs(X,Y) :- Y is -X.
```

NB : *vabs(X,Y)* est équivalent à *Y is abs(X)*.

- minimum(X,Y, Z) : Z est min(X,Y)

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

Solution avec coupure :

```
minimum(X,Y,X) :- X =< Y, !.          /*coupure verte*/
minimum(X,Y,Y) :- Y > X.
```

p simplification abusive:

```
minimum(X,Y,X) :- X =< Y, !.          /*coupure rouge*/
minimum(X,Y,Y) .
```

:- minimum(2,5,5). ❷ réussit à l'aide de la deuxième clause !!!!

La bonne solution :

```
minimum(X,Y,Z) :-   X =< Y, ! , Z=X.      /*cut rouge*/
minimum(X,Y,Y) .
```

- Le Nième élément d'une liste :

```
nieme(1,[Xl_],X) :- !.
nieme(N,[_lXs],X) :- N > 1, M is N - 1, nieme(M,Xs,X).
```

- Les N premiers éléments d'une liste :

```
nfirst(1,[Xl_],[X])    :- !.
nfirst(N,[XlXs],[XlYs]) :- N > 1, M is N-1, nfirst(M,Xs,Ys).
```

17.3- Résumé et remarques sur l'évaluation des termes

En Prolog, les termes composés ne sont pas évalués:

succ(0) n'est pas égal à 1
5 + 6 n'est pas égal à 11

L'évaluation peut se faire par une demande explicite (par l'utilisation des opérateurs =:=, =\=, <, >=, >, =<, is).

- Evaluation d'expression arithmétique demandée par **is**

X is 3 + 2 * 15 - 8 ==> X =25
X = 6 , Y is X + 15. ==> Y = 21

p X is X +1 impossible (au sens d'incrémentation)

- **=** et **IS**

X = 2+3 ==> X = 2+3
X is 2+3 ==> X = 5

p X = X +1 (Problème d'occurrence)

- **IS** et **=:=**

Dans *X is Expr*, si X est variable, la valeur d'*Expr* est unifiée avec X; sinon l'égalité de X et d'*Expr* est testée.

Dans *Expr1 =:= Expr2,* les deux expressions sont évaluées puis, l'égalité de leur valeur est vérifiée. Il n'y a pas d'effet d'affectation avec =:=.

17.4- Exercices

Exercice-6 Spécifier complètement le prédicat *dérivée* vue précédemment.

Exercice-6-bis : On présente l'heure sous la forme HH:MM. Donner les déclarations des opérateurs ainsi que les prédicats pour réaliser l'arithmétique des termes du type heure:

H := Expr Affectation.

Expr est une expression arithmétique de la forme

H1 :+: H2, *H1 :-: H2*, *H1 :*: constante*, *H1 :/: constante*

Les opérations booléennes sont

H1 :=: H2 (égalité), *H1 :<: H2* (plus petit), *H1 :>: H2* (plus grand)

Exercice-7 On dispose d'une base de données de trafic ferroviaire où les faits sont présentés sous la forme :

train(Origine, Destination, H_dep, H_arr).

H_dep et H_arr sont de la forme *HH:MM*.

Ecrire le prédicat *chemin(Lieu1, Lieu2, Gares, Durée)* qui calcule la liste des gares et le temps nécessaire pour aller de lieu1 à lieur2.

Exercice-8 Soit un ensemble de faits *lien(noeud1, noeud2, capacité)*. Ces faits représentent un réseau d'irrigation.

- Ecrire un prédicat *chemin(A,B)* qui vérifie s'il y a un chemin entre deux noeuds données A et B.
- Ecrire un prédicat *chemin(A,B,Liste)* qui vérifie s'il y a un chemin entre deux noeuds données A et B sans passer par aucun des noeuds donnés dans *Liste*.
- Ecrire un prédicat *chemin(A,B,Capa)* qui vérifie s'il y a un chemin entre deux noeuds données A et B avec une capacité d'au moins égale à Capa.

Exrcice-9 Calculer le capital final après N années obtenu avec C francs palcé à un taux T.

Exercice-10 Calculer le nombre d'années N nécessaire pour doubler (au moins) un capital de C francs placé à un taux T.

Exercice-11 Modifier l'exercice sur le repas et les ingrédients en ajoutant à la base la quantité disponible de chaque ingrédient ainsi que la quantité nécesssaire de chaque ingrédient pour chaque repas. Modifier *peut_préparer* pour tenir compte des quantités d'ingrédients nécessaires (et disponibles) pour préparer un repas.

Exercice-12 : Définition d'un petit langage de programmation

A l'aide des déclarations d'opérateurs, définir un petit langage de programmation permettant de construire des phrases avec les mots clés. Ces mots clés sont les opératuers :

et, ou (sur des booléens), si , alors, sinon, vaut (test d'égalité), :=, puis (le point-virgule de pascal), lire, écrire ....

Par souci de simplification, nous laissons cette définition volontairement informelle. Les sens associés à ces constructions sont laissés libres. On peut par exemple s'appuyer sur des définitions analogues en Pascal.

A toute fin utile, les constructions peuvent être basées sur les règles suivantes :

<programme> ::= <instructions>.
<instructions> ::= <instruction> .
<instructions> ::= <instruction> "**puis**" <instructions>.
<instruction> ::= <affectation>.
<instruction> ::= "**lire**" <variable>.
<instruction> ::= "**écrire**" <variable>.
<instruction> ::= <conditionnel>.
<affectation> ::= <variable> "**:=**" <expr_arith>
<conditionnel> ::= "**si**" <condition> "**alors**"
<instructions> <suite_condition>.
<suite_condition> ::= "**sinon**" <instructions>
<suite_condition> ::= .
<condition> ::= <expr_arith> "**vaut**" <expr_arith>.
<expr_arith> ::= <variable>.
<expr_arith> ::= <nombre>.
<expr_arith> ::= {une expression simple avec **+**, **-**, ***, / ...}
<variable> ::= {un atome commençant par une lettre minuscule}
<nombre> ::= {une constante entière}

Décider de la forme canonique de chaque construction et déclarer les opérateurs en conséquence. Associer un programme Prolog permettant d'exécuter les programmes écrits dans ce petit langage.

Exercice-12 bis : puzzle simple.

construire un nombre N contenant une seule occurrence tous les chiffres 1,2,3,4,5,6,7,8,9 tel que les premiers k chiffres de N soient divisibles par k, pour k=1..9.

Par exemple, le nombre N= 381654729 a cette propriété car pour

k=1 3 est divisible par 1

k=3 381 est divisible par 3...

18. Prédicats extra-logiques

Ce sont des prédicats qui n'ont pas de définition (ou de justification) logique mais qui permettent d'utiliser Prolog comme un langage de programmation.

18.1- Prédicats de test de type

integer(X) et **float(X)** X est un entier/réel?
var(X) X est une variable?
nonvar(X) X est pas instanciée? (équivalent à not var(X))
atom(X) X est un atome?
atomic(X) X est un atome ou un entier?

18.2- Entrées/Sorties - Fichiers

read(X) lecture d'un terme (il faut terminer avec '.')
get0(X) lecture du prochain caractère (X = le code ASCII)
get(X) lecture du prochain caractère imprimable différent du blanc;
X est le code ASCII de ce caractère.
skip(Code) lecture jusqu'à rencontrer un caractère dont le code ASCII est égal à Code.
see(F) fichier en entrée devient F
seeing(F) quel est le fichier en entrée?
seen ferme le fichier actuel d'entrée;
'user' devient le fichier d'entrée courant

write(X) écriture d'un terme en tenant compte des déclarations des opérateurs
writeq(X) écriture avec des quotes
display(X) écriture d'un terme sous la forme canonique
displayq(X) écriture de la forme canonique avec les quotes
put(X) écrire le caractère dont le code ASCII est X.
nl passage à la ligne
tab(N) tabulation de N colonnes
tell(F) fichier en sortie devient F
telling(F) quel est le fichier en sortie ?
told ferme le fichier actuel de sortie;
'user' devient le fichier de sortie courant
exists(F) le fichier F existe-t-il?

Exemple de manipulation de fichiers

Copier le contenu d'un fichier texte dans un autre. Le fichier en lecture contient des termes Prolog.

```
copy_fichier :-
    write('quel est le fichier à lire ?'), read(Fin), suite_copy(Fin).

suite_copy(Fin) :-
    exists(Fin), !,   % le fichier existe sur le disque
    write('quel est le fichier de sortie ?'), read(Fout),                see(Fin),tell(Fout),
    read(T),          % la première lecture
    proceed(T,Fin,Fout).
suite_copy(Fin) :-
    write('le fichier '), writeq(Fin), write('n existe pas '), nl.

proceed('end_of_file') :- seen, told.
proceed(T) :- write(T), nl, read(T1), proceed(T1).
```

Exercice-13 : copier le contenu d'un fichier texte dans un autre en transformant les lettres minuscules en majuscules.

Exercice-14 On dispose de deux fichiers contenant des entiers ordonnés. Créer un troisième fichier ordonné résultant de la fusion de ces deux fichiers.

18.3- Mise au point

spy symb_pred :
Mettre un point d'arrêt sur les prédicats dont le symbole prédicatif est Symb_pred.

spying :
Quels sont les points d'arrêts ?

nospy symb_pred :
Retirer le point d'arrêt mis sur les prédicats dont le symbole prédicatif est symb_pred.

trace : activer le mode trace.
notrace : désactiver le mode trace.

debug : activer le mode debug.
nodebug : désactiver le mode debug.

abort : abandonner la résolution

18.4- Gestion de la base

consult(F) :
Lire les clauses du fichier F et les insérer dans l'espace de travail.
consult(F) est équivalent à *[F]*.

reconsult(F) :
Comme *consult* mais on remplace les prédicats qui sont définis dans F.
reconsult(F) est équivalent à [-F].

assert(Clause) :
Ajouter Clause dans l'espace de travail à la fin du paquet de clauses du même symbole prédicatif que celui de Clause.

asserta(Clause) :
Ajouter Clause dans l'espace de travail au début du paquet de clause du même symbole prédicatif.

assertz(Clause) : comme assert(Clause).

rectract(Clause) :
Effacer la première occurrence de Clause de l'espace de travail.

abolish(Sym_pred, Arité) :
Effacer de l'espace de travail le paquet de clauses dont le symbole prédicatif est Symb_pred et dont l'arité est Arité.

clause(Tete, Corps) :
Unifier Tete avec la première clause de l'espace de travail.
Corps sera unifier au corps de cette clause.

current_predicate(Sym_pred,Terme) :
Terme est le prédicat de l'espace de travail dont le symbole prédicatif est Sym_pred.

listing : Lister les clauses de l'espace de travail.

listing(Sym_pred) :
Lister les clauses dont le symbole prédicatif est sym_pred.

Exemples:

- **Un petit méta-interpréteur Prolog en Prolog.**

```
effacer((B ,BS)) :-
    ! , effacer_un_but(B) , effacer(BS).
effacer(B) :-
    effacer_un_but(B).

effacer_un_but(true) :-
    !.              % pour accélérer
effacer_un_but(B) :-
    clause(B,Corps), effacer(Corps).
```

- **Insertion de clauses par *assert* :**
  Par assert, on ajoute à la base le prédicat *liste_de_un* :

```
:-assert((liste_de_un([]))).
:-assert((liste_de_un([1|L]) :- liste_de_un(L))).

:-listing(liste_de_un).
```

③ liste_de_un([]).
liste_de_un([1|A]) :-liste_de_un(A).

```
:- liste_de_un(X).
```

③ X = [] ;
X = [1] ;
X = [1,1] .....

- **Copie d'un terme :**

Il est parfois nécessaire de copier un terme (pour démontrer un but sans instancier ses variables, renommer un terme...). La copie du terme "t" est identique à "t" avec des variables fraîches :

```
copy(X,Y) :-
    asserta('$copier'(X)),        % insérer
    '$copier'(Y),                 % récupérer
    retract('$copier'(X)).        % supprimer
```

Remarque :

l'utilisation de nom d'atome particulier (ici '$copier') est une pratique courante en Prolog. Ainsi, on diminue le risque d'ajouter ou de supprimer accidentellement des clauses "ordinaires".

Exemple d'utilisation :

?- copy(f(A,B), La_copie).

③ La_copie = f(C,D)

Exemple d'application :

Pour tester si le prédicat "P" est démontrable sans unifier ses variables à une valeur :

```
tester(P) :- copy(P, Q), Q, !.
```

Ainsi, si "Q" réussit, on déduit que "P" est démontrable sans modifier les variables de "P".

?- tester(membre(X, [a,b,c]).

③ succès sans modifier X.

Remarque : la double négation que nous étudirons plus loin produit le même effet.

- Un prédicat qui s'auto détruit quand il est appelé :

On veut insérer le prédicat :

p(1) :- write('coucou'), retract((p(1):-A)).

Par assert :

assert((p(1) :- write('coucou') , retract((p(1) :- W)))).

?- listing.

③ p(1) :-

write(coucou), retract((p(1):-A)).

?- p(X).

③ coucou

X = 1

?- listing.

③ le prédicat "p" n'est plus dans la base.

Exemple d'utilisation :

On utilise cette technique de programmation pour mémoriser l'utilisation d'une clause. La suppression de la clause en question élague une branche de l'arbre de résolution développée lors du premier passage.

Exercice-15 : écrire deux prédicats "p" et "v" remplissant les actions suivantes:

- si "p" est appelé, il s'efface et ajoute le prédicat "v" à la base.
- si "v" est appelé, il s'efface et ajoute le prédicat "p"à la base.

Remarque : "p" et "v" simulent un sémaphore booléen.

Exercice-16 Etant donné un ensemble de faits *age(nom,age)* et *sexe(nom,sexe)*; produire et insérer dans la base l'union de ces deux relations i.e. les faits *ags_sexe(nom, age, sexe)* à partir des deux premiers faits.

19. Méta-variables et méta-prédicats

L'unique structure de données définie en Prolog est le terme.

Une clause de la forme *':-'(Tete, Corps)* est un terme.

Sachant qu'un programme Prolog est une conjonction de clauses, on peut considérer un programme comme un terme, c'est à dire, construire un terme dont la valeur est un programme.

L'idée force est que d'un point de vue opérationnel, il n'y a pas de séparation entre données et traitement et on peut manipuler, syntaxiquement, un programme comme une donnée.

Donnée <=> Programme

Définition : un méta-prédicat est un prédicat de manipulation de prédicat.

Nous énumérons ci-dessous quelques uns des méta-prédicats pour montrer comment ils sont construits. On notera que les interpréteurs Prolog refusent une redéfinition de ses méta-prédicats par l'utilisateur.

❑Le méta-prédicat prédéfini *call* :

```
call(P) :- P.
```

?- call(membre(a, [b,a])) équivaut à ?- membre(a, [b,a])

On peut par exemple construire le terme *pere(X,jean)* à partir de *pere*, jean et X puis activer ce terme par call.

Lorsqu'une variable *X* apparaît dans le corps d'une clause, elle est remplacée par *call(X)*.

❑Prédicat "solution unique" : à utiliser lors que l'on s'intéresse à un seul succès

```
once(P) :- P , !.
```

:- once(membre(X,[a,b,c])). ==> X=a.

Exercice: Ecrire le prédicat *loop(N,P)* qui exécute N fois le prédicat P.

```
loop(1,P) :- P, !.
loop(N,P) :- P, M is N-1, loop(M,P).
```

❑ *La négation*

L'évaluation de la négation est basée sur le fait que l'univers du discours est défini seulement par ce que l'on décrit. Par conséquent, ce qui n'est pas défini (ou ce qui n'est pas démontrable) est faux.

En Prolog, on ne peut pas décrire des faits négatifs et on admet qu'ils sont déduits par le complément des faits positifs décrits :

not P réussit si P ne réussit pas (et inversement).

Exemple : pour savoir si jean est-il orphelin

:- not pere(_, jean).

Implantation de la négation par l'échec:

```
not P :- P , ! , fail.
not P .
```

Remarques :

\+P est syntaxiquement équivalent à **not P**.

Comme on peut le constater, not P se contente du premier succès de P.

Si *not P* réussit et que P contient des variables, ces variables ne seront pas instanciées.

Exemple:

```
homme(jean).
homme(pierre).
femme(marie).
?-not homme(jacques). %prouver que jacques n'est pas un homme
```

Puisque l'on peut pas démontrer *homme(jacques)*, on déduit que *not homme(jacques)* est vrai. (l'interprétation que l'on peut faire de cette déduction est une affaire de *modèle*)

❑La disjonction :

```
(P ; _) :- P.
(_ ; Q) :- Q.
```

❑L'itération : le prédéfini *repeat*

```
repeat.
repeat :- repeat.
```

Exemple : traduction du schéma impératif :

```
read(X)
Tant que X<> fini
    traiter(X)
    read(X)
Fin Tant que
```

```
:- repeat , read(X) , (X = fini, write(fin) ; traite(X) , fail).
```

❑Si-alors-sinon , si-alors:

Le schéma "si P alors Q sinon R" se traduit par *P -> Q ; R* donné par :

```
(P -> Q ; _) :-    P , ! , Q.
(_ -> _ ; R) :-    R.
```

Et le schéma "si P alors Q " se traduit par *P -> Q* donné par :

```
(P -> Q ) :-   P , ! , Q.
```

p Notons que la définition du méta-prédicat (P -> Q ; R) évite que la forme (P -> Q ; R) soit traitée comme (P->Q) ; R.

Exemple : le même schéma impératif ci-dessus dans une version récursive :

```
foo                 :- read(X) , process(X).
process(fini)   :- ! , write(fin).
process(X)      :- traite(X) , fail.
process(_)      :- foo.
```

Et enfin une autre manière d'écrire le prédicat *foo* à l'aide de si-alors-sinon :

```
foo :- read(X) , (X=fini -> write(fin) ; (process(X)).
```

❑ *Métaprédicats toute-solutions*

- **bagof(Terme, But, Liste_Résultat).**
  Liste_Résultat est une liste de réponses à But mises sous forme spécifiée par Terme.

- **setof(Terme, But, Ens_Résultat).**
  Ens_Résultat est une liste de réponses à But mises sous forme spécifiée par Terme. La différence par rapport à bagof est qu'Ens_Résultat est trié selon l'ordre entre les termes et ne contient pas de doublon.

- **X ^ P.**
  Il existe X tel que P réussit. Utilisé en dehors de setof/bagof, X ^ P a le même effet que call(P).

Exemple1 : étant donné la base

salaire(jean,5000).
salaire(jacques,6500).
salaire(jacques,6500).
salaire(marie,4000).
salaire(marie,4000).
salaire(helen,3500).

?- bagof(S, salaire(Z,S), M).
③ M = [3500] ;
M = [6500,6500] ;
M = [5000] ;
M = [4000,4000] ;
?- setof(S, salaire(Z,S), M).
③ M = [3500] ;
M = [6500] ;
M = [5000] ;
M = [4000] ;

?- bagof(Y, Z^ salaire(Z,Y) ,M).
③ M = [5000,6500,6500,4000,4000,3500]

Exemple2 :

A partir des faits instances de la relation *pere(X,Y)* et ceux instances de la relation *plus_grand(X,Y)*, on peut trouver la LISTE des *freres*:

```
?- bagof(   frere(X,Y),
            (pere(Z,X) , pere(Z,Y) , plus_grand(X,Y)),
            Liste).
```

③ Liste = la liste de termes de la forme *frere(X,Y)*
qui satisfont le but *pere(Z,X) , pere(Z,Y) , plus_grand(X,Y)*
pour un Z donné.

On obtient autant de listes que de couples de frères.

```
?- setof( frere(X,Y),
            (pere(Z,X) , pere(Z,Y) , plus_grand(X,Y)),
            Ensemble).
```

③ Ensemble = l'ensemble (sans répétition) de termes de la forme *frere(X,Y)*
qui satisfont le but *pere(Z,X) , pere(Z,Y) , plus_grand(X,Y)*
pour un Z donné.

On obtient autant de liste que de couples de frères sans doublon.

```
p ?- bagof(frere(X,Y),
            (Z^(pere(Z,X) , pere(Z,Y)) , plus_grand(X,Y)), Liste).
```

③ Liste = la liste de termes de la forme *frere(X,Y)*
qui satisfont le but *pere(Z,X) , pere(Z,Y) , plus_grand(X,Y)*
pour tout Z.

Cette fois, on obtient la liste de tous les couples de frères; par exemple :

③ Liste = [frere(paul,pierre),frere(mark,alex)]

Exemple-3 :

Transformer une liste L en un ensemble E (liste sans doublon) par le but :

?- L=[a,b,c,a,c,d,e,f], setof(X, membre(X,L), M).

③ M = [a,b,c,d,e,f]

Exemple-4 :

Calcul de l'intersection de deux listes L1 et L2 dans la liste L3 par le but :

?- L1=[a,b,c,a,c,d,e,f], L2=[a,n,t,e,d,c,f],
setof(X,(membre(X,L1),membre(X,L2)),M).

③ M = [a,c,d,e,f] ;

Exemple-5 :

Etant donné les faits

arc(a, b).arc(a, c). arc(b, c).arc(b, d).

Et les questions :

1 - bagof(X, arc(X,Y), M).

③ Y = b , M = [a]
Y = c , M = [a,b]
Y = d , M = [b]

La présence de variable libre (celle qui n'est pas citée dans le premier paramètre de *bagof* mais qui figure dans le second conduit aux résultats dispersés. Ces variables sont quantifiées par ∀ (par exemple Y ci-dessus). La présence de "^" permet de quantifier ces variables par ∃ et d'obtenir l'ensemble des réponses)

2 - bagof(X, Y ^ arc(X,Y), M).

③ M = [a,a,b,b]

3 - bagof(Y, X ^ arc(X,Y), M).

③ M = [b,c,c,d]

Exercice-?

Reprendre l'exercice7 et écrire le prédicat *le_plus_court(Lieu1, Lieu2, Durée)* qui calcule le temps le plus court entre le deux lieux.

❑ Schéma générateur / testeur

Certaines implantations de Prolog proposent le méta-prédicat *forall* qui vérifie une propriété P sur un ensemble de valeurs de x : $\forall x\ P_X$

On peut écrire : $\forall x\ P_X = \neg(\exists x\ \neg P_X)$

Soit Q_X générateur des valeurs de x; on peut alors écrire

$$\forall x\ P_X = \neg(\exists x\ \neg P_X) = \neg(Q_X \wedge \neg P_X)$$

Ce qui donne en Prolog :

```
pourtout(Q,P) :- not (Q , not P).
```

Exemple1: Vérifier que tous les éléments d'une liste L sont impaires :

```
:- pourtout(membre(X,L) , impaire(X)).
```

Exemple2**:** Contrôle de cohérence dans une base de données

Soit une base de données du personnel:

```
salaire(jean,5000).
salaire(jacques,6500).
salaire(marie,4000).
homme(jean).
homme(jacques).
femme(marie).
```

- vérifier que tout le monde gagne au moins 4000:

```
:- pourtout(salaire(_,N) , N >= 4000).
```

- vérifier que les hommes gagnent plus que les femmes:

```
:- pourtout(
        (homme(H) , salaire(H,N) , femme(F) , salaire(F,M)),
        N > M).
```

p Etant donné la négation dans *not P*, une utilisation correcte du schéma *forall* doit veiller à ce que *P* soit sans variable libre lors de sa démonstration.

❝ ❑ *La double négation :*

Comme en logique formelle, on considère que **not not P** et **P** sont équivalents: *not not P* est vrai si *P* est vrai.

La différence procédurale (dans leur traitement en Prolog) entre *not not P* et *P* réside dans le fait que le succès de *not not P* n'instancie pas les variables de *P*.

Exemple : pour savoir si un terme fonctionnel a une structure de liste :

```
est_liste([]).
est_liste([_|_]).
type_liste(X) :-  not not est_liste(X).
```

Lors de l'utilisation de "type_liste", si X n'est pas une variable libre alors on vérifie qu'elle a une structure de liste; si X est une variable, l'on vérifie qu'elle peut être une liste sans l'instancier à [].

Par contre, l'absence de *not not* unifierait X à [].

Utilisation :

?- type_liste(Z). ==> succès sans la modification de Z

?- type_liste([a,b]). ==> succès

Remarque :

Nous avions obtenu le même résultat avec le prédicat *tester* qui utilise le prédicat *copy* (Section : "gestion de la base").

p **Problèmes avec la négation par échec**

Dans une utilisation correcte de *not P*, P ne doit pas contenir de variable libre lors de l'appel de *not P*.

Exemple-1 : Considérons les faits suivants qui expriment la relation "qui-aime-qui" parmi les individus "marie", "jean" et "paul" :

```
aime(marie, jean).    % marie aime jean
aime(jean, paul).     % jean aime paul
```

et la question "y a-t-il quelqu'un qui n'aime pas jean" est posée par :

?- not aime(X, jean).

On conclut (selon le principe de la négation par échec) que *not aime(X, jean)* échoue car *aime(X, jean)* réussit avec X=marie. Pourtant, on ne peut pas dire qu'il n'y a personne qui n'aime pas jean puisque pierre n'aime pas jean.

D'où vient le problème?

Le problème réside dans le fait que le principe de la négation par échec ne produit pas de réponse (des valeurs pour les variables); autrement dit, *not aime(X,jean)* ne nous informe pas sur les individus qui n'aiment pas jean.

Ce n'est donc qu'en utilisant *not* avec des littéraux sans variables que la *correction* de la négation par échec est respectée (théorème de la correction de la résolution avec la négation par échec; voir la bibliographie)

Exemple-2 : dans la question *?- not (X = 1) , X = 2 .* où X est une variable libre, on obtient un échec alors que la question a une réponse; c'est à dire X=2.

Comme pour l'exemple précédent, *not* a été utilisé avec une variable libre. Il aurait suffit de poser la question sous la forme *?- X = 2, not (X = 1).* pour obtenir la réponse attendue.

Afin d'éviter cette situation, certaines implantations de Prolog (cf. Mu-Prolog) "retardent" l'effacement des littéraux négatifs en attendant que toutes leurs variables aient une valeur. Ce retardement modifie en quelque sorte la règle de calcul utilisée en Prolog pour effacer les buts de gauche à droite. Ainsi, le succès d'une démonstration est conditionné par l'absence de but retardé.❞

20. Manipulation des termes composés

Un terme composé est un arbre. Les prédicats de manipulation d'arbres sont :

- =.. permet le passage d'une liste à un arbre (et inversement). Le nombre d'éléments de la liste doit être connu à l'exécution de =..

Format : terme_fonctionnel =.. liste

Exemples : ?-f(a,b,c) =.. [f,a,b,c]

?-f(x,[b,c],g(Y)) =.. [f, x, [b,c], g(Y)]

Exemple d'utilisation : construction et appel d'un prédicat

?- P =..[pere, jean, X] , P.

- **arg(N,Terme, Nème_arg) :**

Permet l'extraction des éléments d'un arbre (l'arité de Terme ≥ N>0)

Exemples :

?-arg(2, f(x,[b,c],g(Y)), Deuxieme_terme).

③ Deuxieme_terme = [b,c]

?- arg(3, f(x,[b,c],g(Y)), Troisieme_terme).

③ Troisieme_terme= g(Y)

?- arg(0, f(x,[b,c],g(Y)), Arg). ❷ échec (N=0)

?- arg(5, f(x,[b,c],g(Y)), Arg). ❷ échec (N > arité)

- **functor(Terme, Atome, Arité) :**
  Construction d'un arbre ou extraction de sa racine et de son arité

Exemples :

?- functor(racine(1,b,[a]), Atome, Arite).

③ Atome = racine , Arite = 3

?- functor(Arbre, racine,3).

③ Arbre = racine(X, Y, Z)

?- functor(Arbre, racine_toute_seule,0).

③ Arbre = racine_toute_seule

?- functor(X,1,0).❷ X = 1

?- functor(1,X,Y). ❷ X = 1 , Y = 0

- **name(Atome, Liste_caractères)**

  Liste_caractères est la liste de codes ASCII des caractères de l'Atome

?-name(centrale, X).
③ X = [99,101,110,116,114,97,108,101]

?-name('bla bla', X).
③ X = [98,108,97,32,98,108,97]

?-name(X, "chaine"). ❷ X = chaine

- **length(Liste, Taille)**
  La longueur de la liste Liste unifiée à Taille.

?- length([a,b,c],N). ❷ N=3.
?- length("toto",N). ❷ N=4.

Exemple : parcours d'arbres binaires

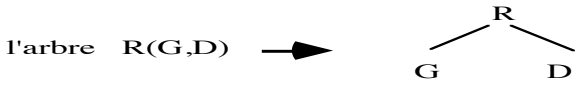
Recherche et insertion dans un arbre binaire ordonné horizontalement

Convention:

- L'arbre vide est noté []
- Il n'y a pas de doublon dans l'arbre
- La relation d'ordre sur chaque noeud est:

  max(sag) < info(racine) < min(sad).

- La représentation de différentes configurations:



Les feuilles (n'ont pas de descendant) :

F([],[]) □ F

Recherche dans un arbre binaire ordonné horizontalement :

```
cherche(Ele, Ele) .
cherche(Ele, Arbre) :- Arbre =.. [Ele, _, _] .
cherche(Ele, Arbre) :-
    Arbre =.. [Racine, Gauche , _ ] ,
    plus_petit(Ele , Racine) , cherche(Ele, Gauche).

cherche(Ele, Arbre) :-
    Arbre =.. [Racine , _ ,Droite ],
    plus_petit(Racine, Ele) , cherche(Ele, Droite).
```

Exemples de questions:

:- N= i(b(a, f),z) , cherche(z,N) ❷ succès
:- N= i(b(a, f),z) , cherche(b,N) ❷ succès
:- N= i(b(a, f),z) , cherche(k,N) ❷ échec

Exercice : Introduire cut dans ce programme pour améliorer les performances.
Que faut-il modifier dans le prédicat cherche pour pouvoir énumérer les feuilles de l'arbre ?

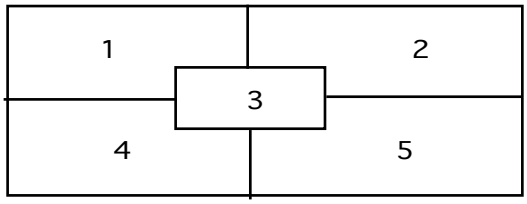
Exercice-17 : Ecrire le prédicat *feuilles(Arbre, Liste_feuilles)* qui renvoie dans une liste les feuilles d'un arbre n-aire. Améliorer la solution pour éviter d'utiliser la concaténation.

Exercice-17bis : Ecrire les prédicats d'insertion dans un ABOH.

Exercice-18 : Modifier ces programmes pour implanter un dictionnaire sous forme d'un arbre binaire ordonné horizontalement dont chaque élément est de la forme *noeud(cle, info)* où
- *noeud* est un atome
- *clé* est un atome représentatnt un nom d'élève
- *info* est un terme composé représentant des informations concernat l'élève.

21. Introduction à la méthodologie

Coloration de cartes



Enoncé:

colorier la carte ci-dessus par les trois couleurs R, B et V en respectant la contrainte : deux régions voisines ne doivent pas avoir la même couleur.

21.1- Méthode constructive

A partir d'une configuration initiale S_0, on génère la configuration partielle S_1 successeur de S_0 dont la conhérence est testée dès que possible (dès sa génération). Ainsi, la cohérence de la solution est vérifiée lorsqu'à l'étape n, la configuration partielle S_n (successeur de Sn-1) est la configuration finale recherchée.

Le schéma général de la méthode constructive est donné par :

p(...) :-

générer puis vérifier une succession de configuration partielle jusqu'à la configuration finale

Dans l'exemple ci-dessous, les couleurs déjà affectées à chaque étape ne sont pas en contradiction entre elles mais de nouvelles viendront mettre en cause leur cohérence.

```
coloration(C1, C2, C3, C4, C5) :-
        voisin(C1, C2) , voisin(C1, C3) ,voisin(C1, C4) ,
        voisin(C2, C3) , voisin(C2, C5) ,
        voisin(C3, C4) ,voisin(C3, C5) ,
        voisin(C4, C5) .
        couleur(r).  couleur(b). couleur(v).

        voisin(X,Y) :- couleur(X) , couleur(Y) , X \==Y.
```

Réponses obtenues:

C1	C2	C3	C4	C5
r	b	v	b	r
r	v	b	v	r
b	r	v	r	b
b	v	r	v	b
v	r	b	r	v
v	b	r	b	v

21.2- Méthode génération/test

Le schéma général de cette méthode est

p(...) :-
Génération des configurations totales suivie des
Contraintes à vérifier.

Pour l'exemple de coloration, on construit une combinaison de cinq couleurs puis on vérifie les contraintes.

```
coloration(C1,C2,C3,C4,C5) :-
    cinq_couleurs(C1,C2,C3,C4,C5) ,          % génération d'une configuration
    different(C1,C2) , different(C1,C3) , different(C1,C4),
    different(C2,C3) , different(C2,C5),
    different(C3,C4) , different(C3,C5) ,
    different(C4,C5) .

    couleur(r). couleur(b). couleur(v).

    different(X,Y) :- X \== Y.

    cinq_couleurs(C1,C2,C3,C4,C5) :-
        couleur(C1) , couleur(C2) , couleur(C3) ,
        couleur(C4) , couleur(C5).
```

21.3- Méthode contraindre et générer

Dans cette méthode, on pose les contraintes puis on génère les combinaisons. Les contraintes posées ayant un effet global sur l'environnement, les combinaisons invalides seront rejetées immédiatement et ne seront pas générées. De plus, les contraintes étant testées avant toute génération, on évite de procéder à la génération dès lors que les contraintes sont jugées inconsistantes.

La mise en place de cette méthode nécessite la modification de la règle de calcul de Prolog pour disposer du mécanisme de retardement.
(C.f. PrologII, PrologIII, D-Prolog, BNR Prolog, Clp(R), Chip...).

Cette possibilité n'est pas offerte par les Prologs standard.

Le schéma général de cette méthode est

p(...) :-
Contraintes à vérifier
Génération des configurations totales.

Une solution en PrologII à titre indicatif:

```
color(C1,C2,C3,C4,C5) :-
    dif(C1,C2) , dif(C1,C3) , dif(C1,C4) ,
    dif(C2,C3) , dif(C2,C5) ,
    dif(C3,C4) , dif(C3,C5) , dif(C4,C5),
    couleur(C1) , couleur(C2) ,couleur(C3) ,
    couleur(C4) , couleur(C5).
```

dif(X,Y) pose la contrainte suivante :

les deux termes X et Y doivent être différents dès que leur valeurs sont connues.

Exemple: Génération d'une séquence de 4 chiffres différents.

- Une solution par la méthode constructive:

```
permutation(X1, X2 , X3 , X4 ) :-
    chiffre(X1) , chiffre(X2) , pas_dans(X1, [X2]) ,      % les 2 premiers différents
    chiffre(X3) , pas_dans(X3,[X1,X2]),                   % les 3 premiers différents
    chiffre(X4) , pas_dans(X4, [X1, X2, X3]).             % tous différents

pas_dans(X,[]).
pas_dans(X,[Y|L]) :- X \== Y , pas_dans(X, L).

chiffre(1).   chiffre(2).   chiffre(3).      chiffre(4).   chiffre(5).
```

?- permutation(X,Y,Z,K).

③ X=1, Y=2, Z=3, K=4

X=1, Y=2, Z=3, K=5

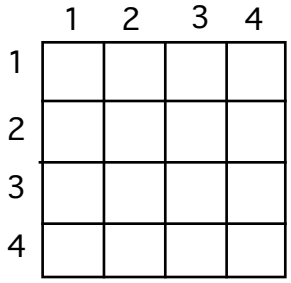
.....

- Une solution par la méthode génération/test :

```
permutation(X1, X2 , X3 , X4 ) :-
    chiffre(X1) , chiffre(X2) ,                  % génération
    chiffre(X3) , chiffre(X4) ,                  % d'une configuration
    tous_differents([X1, X2, X3, X4]).   % vérification des contraintes

pas_dans(X,[]).
pas_dans(X,[Y|L]) :- X \== Y , pas_dans(X,L).

tous_differents([]).
tous_differents([X|L]) :- pas_dans(X,L) ,   tous_differents(L).


chiffre(1).   chiffre(2).   chiffre(3).      chiffre(4).   chiffre(5).
```

Exercice-19 : Problème des N-reines

Il s'agit de placer N (ici N=4) pions sur un échiquier (N x N) suivant :



En respectant les contraintes :

(1) - deux pions ne peuvent pas être sur la même ligne
(2) - deux pions ne peuvent pas être sur la même colonne
(3) - deux pions ne peuvent pas être sur la même diagonale

Si P1 et P2 sont deux pions de coordonnées (c1,l1) et (c2,l2), les contraintes ci-dessus sont exprimées en fonctions de ces coordonnées:

(1) - $c_1 \neq c_2$
(2)- $l_1 \neq l_2$
(3) - $|l_2-l_1| \neq |c_1-c_2|$

Proposer une solution par les méthodes constructive et génération / test.

Exercice-20 : Etendre l'exercice des n-reines au cas général (N passé en paramètre).

Exercice-21 : Problème du cavalier.

Il s'agit de déplacer un cavalier du jeux d'échec sur un échiquier (N x N) (ici N = 5). Pour simplifier, les cases sont numérotées de 1 à N^2 (voir figure-1)·

Les mouvements du cavalier sont de la forme d'un 'L' . Par exemple, lorsque le cavalier est en case 13, il peut se déplacer en 6, 2, 16,.... (maximume 8 positions possibles) tout en restant sur l'échiquier.

	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3			13		
4	16				20
5		22		24	25

Figure 1

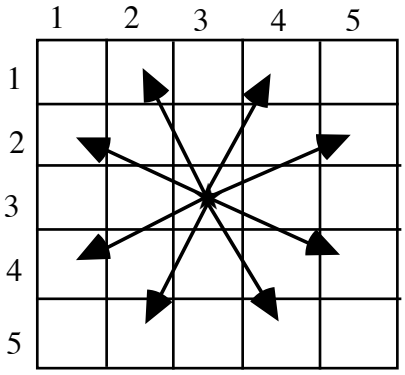


Figure 2

Indication : Si le cavalier se trouve en <L, C>, c'est à dire en case numéro L*C, il peut avoir au plus 8 destinations possibles données par <L±2, C±1>, <L±1, C±2> tel que toutes ces valeurs soient inférieures à N+1 et supérieures à 0 afin de rester sur l'échiquier.

1- On suppose donné le prédicat **move(X,Y)** permettant de connaître les cases Y que l'on peut emprunter à partir de la case X. Par exemple, pour X=1 et N=5, on aura :

```
move(1,8).
move(1,12).
```

Ecrire le prédicat *path(départ, arrivée, liste_des_cases)* qui réussit s'il est possible de déplacer le cavalier de la case *départ* jusqu'à la case *arrivée*. En cas de succès, *liste_des_chemins* contient la liste des cases empruntées.

- 2- Ecrire le prédicat *cavalier(départ, liste_des_cases)* qui, en partant de la case *départ*, doit visiter toutes les cases de l'échiquier en ne passant qu'une seule fois dans chaque case.

22. Annexes : algorithmes de résolution en Prolog

❝ Cette section peut être lue ultérieurement ❞

Les algorithmes suivants décrivent le fonctionnement du moteur Prolog. Ils aideront à comprendre d'une manière précise comment la résolution basée sur l'unification fonctionne.

Auparavant, étudions quelques définitions utilitaires.

22.1- discordance

Ensemble θ de discordances de deux termes T_1 et T_2 est un ensemble de couples X_i/Y_i où X_i est une variable et Y_i un terme.

Exemples:

T_1 = t(X,Y)
T_2 = t(f(Y), Z)
③ θ = {X/f(Y) , Y/Z}

T_1 = f(X,Y, g(b,T,12))
T_2 = f(a , Z, Y)
③ θ = {X/a, Z/Y ,Y/g(b,T,12)}

L'ensemble θ doit remplir un certain nombre de conditions :

- Dans le couple X_i/Y_i , X_i est différent de Y_i
- Dans le couple X_i/Y_i , X_i n'a pas d'occurrence dans Y_i
- Dans θ , les couples sont tels qu'il n'y a pas de chaîne X_i/Y_i ... Z_i/X_i.

22.2- substitution

La substitution notée θ(T) (ou Tθ) consiste à appliquer θ au terme T pour obtenir une *instance* de T.

Exemple:

T = f(a,X,Y)

θ = {X/a, Y/g(Z,"toto",12) , Z/T}

③ Tθ = f(a,a,g(T,"toto",12))

L'idée importante à retenir est que les variables identiques doivent prendre des valeurs identiques. Ce que l'on appelle la substitution uniforme.

Fonction substitution(T:terme,θ: substitution) retourne terme =

Cas

- constante(T) : retourne(T)
- variable(T) :
  S'il existe dans θ un couple T/T_1
  Alors retourne(substitution(T_1,θ))
  Sinon retourne(T)
  Fin si
- terme_composé(T) :
  Soit T = **f**(arg_1,...,arg_n)
  Retourne(**f**(substitution(arg_1, θ), ..., substitution(arg_n, θ)))
- autre: retourne(T)

Fin cas

fin substitution

Remarques :

La substitution définit une fonction; c'est à dire qu'il n'existe pas deux couples t/t' et t/t" dans θ. par exemple, {X/b, X/y} est illégale.

Si l'on applique deux fois une substitution à un terme, on doit obtenir le même résultat. : (Tθ)θ = Tθ. On dit alors que θ est idempotent.

22.3- Renommage

Remplacement de façon uniforme des variables d'un terme par de nouvelles variables "fraîches".

Le renommage est opéré pour éviter les conflits de nom entre les variables syntaxiquement identiques qui n'ont rien de commun.

Exemple:

Soit T = f(a,X,Y,X).

Un renommage de T donne :

T' = f(a,X1,Y1,X1).

Le renommage est une sorte de substitution.

Exemple:

T = f(a,X,Y)
θ = {X/X1 , Y/Y1}

③ Tθ = f(a, X1, Y1).

22.4- Composition de substitutions

La composition de substitutions est notée $\theta_1 \circ \theta_2$. On a pour un terme T et les substitutions θ_1 et θ_2 : $(T\theta_1)\theta_2 = T(\theta_1 \circ \theta_2)$

L'opération de composition de substitutions est suivie d'une simplification comme l'on peut la constater par l'algorithme suivant :

Fonction Composition(θ_1, θ_2 : substitution) retourne substitution =

soient:

$\theta_1 = \{u_1/s_1 , ..., u_m/s_m\}$

$\theta_2 = \{v_1/t_1 , ..., v_n/t_n\}$

construire θ_3 par les trois étapes suivantes :

a) $\theta_3 = \{u_1/s_1\ \theta_2, ..., u_m/s_m\ \theta_2, v_1/t_1 , ..., v_n/t_n\}$

b) supprimer de θ_3 les $\mathbf{u_i/s_i\ \theta_2}$
pour lesquels $\mathbf{u_i = s_i\ \theta_2}$ et

c) supprimer de θ_3 les $\mathbf{v_j/t_j}$ (présents dans θ_2)
si $\mathbf{v_j}$ est dans $\{u_1 , ..., u_m\}$

retourne(θ_3)

Fin Composition

Exemple: (X,Y,Z variables et f,a,b constantes)

$\theta_1 = \{X/f(Y) , Y/Z\}$

$\theta_2 = \{X/a, Y/b , Z/Y\}$

a) $\theta_3 = \{X/f(b) , Y/Y , X/a, Y/b , Z/Y\}$ puis

b) $\theta_3 = \{X/f(b) , X/a, Y/b , Z/Y\}$ puis

c) $\theta_3 = \{X/f(b) , Z/Y\}$

22.5- Unification

L'algorithme suivant donne la fonction d'unification de deux termes t1 et t2: (t1 et t2 en entrée, θ en entrée-sortie).

Cet algorithme est identique à celui de §7.3

θ : substitution;
Fonction unifier(t1,t2 : terme) retourne booléen =
cas
1) t1 = t2: retourne(vrai)
2) var(t1) :
si t1 figure dans t2 alors retourne(faux); finsi -- test d'occurrence
2') var(t1) :
θ := θ ∘ {t1 / t2} ; retourne(vrai)
3) var(t2) :
θ := θ ∘ {t2 / t1} ; retourne(vrai)
3') var(t2) :
si t2 figure dans t1 alors retourne(faux); finsi -- test d'occurrence
4) const(t1) ou const(t2) : retourne(faux)
5) t1 et t2 des termes composés de même foncteur et de même arité:
soient $t_1 = \mathbf{f}(arg_1,....,arg_{\mathbf{n}})$ et
$t_2 = \mathbf{f}(arg'_1,....,arg'_{\mathbf{n}})$
alors
retourne($\Lambda_{i=1..n}$ unifier(substitution(arg_i,θ), substitution(arg'_i,θ)))
6) autre: retourne(faux)
Fincas
Fin unifier

Remarque sur la notation :

$\Lambda_{i=1..n}$ E_i dénote une conjonction de toutes les expressions E_i.

Exemples:

- t1= f(a,g(X,b))
  t2=f(Y,g(b,X) θ={Y/a, X/b}

- t1= p(X, a, f(b,f(Y,n)))
  t2= p(s,Y, f(Z,f(X,n))) Echec

- t1= entier(suc(suc(suc(0))))
  t2= entier(suc(suc(1))) Echec

Remarque :

Qu'obtient-on lors de l'unification de X et de f(X) ?

L'algorithme précédent produira la discordance {X/f(X)}; ce qui représente l'arbre infini f(f(f(f(...)...))). le couple {X/f(X)} n'est pas en accord avec les conditions de bonne forme de construction de l'ensemble de discordance. Pour éviter ce type de construction, il faut ajouter aux points (2) et (3) de l'algorithme précédent un test (dit le test d'occurrence) qui vérifie qu'une variable n'est pas liée à un terme qui contient cette variable. Cependant, étant donné la fréquence d'utilisation de l'unification dans les moteurs Prolog, ceux qui effectuent ce test voient leur performance diminuée de près de 50%. A ceci, les Prolog modernes donne plusieurs réponses :

1- mettre en place des algorithmes différents (cf. PrologII et son système de réécriture de termes);

2- provoquer un échec à la détection de boucle lors de l'unification (cf. exemple ci-dessous)

3- permettre la construction d'arbres infinis (cf. PrologIII, VM/Prolog...)

4- demander au programmeur de veiller à ne pas écrire des programmes nécessitant le test occurrence.

Exemple :

- t1= f(X,Y,X)
  t2= f(g(X), g(Y), Y).

Comme on peut le constater, l'algorithme d'unification peut rentrer dans une boucle lors de l'unification de t1 et de t2.

22.6- Algorithme général de résolution

P= le programme = un ensemble de clauses
B = b_1,..., b_i ,...,b_n (n ≥ 0) = le but

Appel initial**: Si résoudre(B,{}) alors γ est la substitution-réponse.**

θ', σ, γ : substitution

Fonction RESOUDRE(B : liste_de_buts ; θ : Substitution) retourne booléen =

```
Si B = vide alors      -- n=0 ; θ contient les réponses.
        γ = θ;         -- Ici, on peut mettre en place les retours arrières.
        retourne(vrai) -- Extraire les réponses puis si d'autres solutions demandées
                       -- alors retourne(faux) pour forcer l'exploration de tous les C
Sinon
❿   choisir bi dans B
    LC = ensemble de clauses de P
    Tant que LC < > vide faire
¶       choisir C dans LC, C de la forme a0 :- a1,....,ak (k ≥ 0)
        renommer(C)
        LC = LC - { C}
        σ = {}
        Si unifier(a0 , bi )
        Alors
            θ' = θ ∘ σ
            B' = θ'(b1,.., bi-1 , a1,....,ak ,bi+1, .....bn)
            Si      Resoudre(B' , θ')
            Alors   retourne(vrai)
            Fin si
        Fin si
    Fin Tant que
    retourne(faux)
Fin si
fin Résoudre
```

Remarque :

L'algorithme ci-dessus est plus général que celui de Prolog. Nous discutons ci- après de sa spécialisation en Prolog et des problèmes posés par cette adaptation.

- au point ❿, on peut considérer i=1. C'est ce qui est fait dans les moteurs Prolog. Un tel choix permet à l'utilisateur de mieux "maîtriser" ses programmes au détriment d'une perte du caractère "logique" de celui-ci, notamment la commutativité de l'opérateur de conjonction. Ce choix procure un comportement séquentiel au moteur Prolog et présente d'autres inconvénients (plus graves en présence des variables), en particulier le problème de boucle.

- au point ❡, on peut choisir le premier C dans LC. Les moteurs standard Prolog font un tel choix. Ainsi, on considère un ordre entre les clauses d'un même prédicat. Un tel choix permet à l'utilisateur de mettre en place un raisonnement procédural au détriment de la déclarativité de la spécification. En outre, on ne bénéficie plus de la commutativité de l'opérateur de disjonction.

Notons cependant que ces restrictions permettent d'avoir un langage de programmation de haut niveau pour écrire des spécifications exécutables avec des performances acceptables.

Exemple d'application de l'algorithme de résolution sur des listes

Soit le prédicat de concaténation que nous avons déjà étudié :

(r1) concat([] , L, L).

(r2) concat([T|Q] , L , [T|L1]) :-

concat(Q , L, L1).

Soit le but B= :- concat([a] , [b,c] , L)et la substitution initiale

$\theta = \{\}$

- b_1 = concat([a] , [b,c] , L), LC= {r_1, r_2}

  -r_1 renommée ne s'unifie pas avec b_1

  -r_2 renommée donne r'_2 :

  r'_2 = concat([T_1 | Q1] , L1 , [T1|L2]) :- concat(Q1 , L1 , L2).

  -l'unification de b_1 et r'_2 produit :

  σ = {T1/a , Q1/[] , L1/[b,c] , L / [T1|L2]}

  $\theta' = \theta \circ \sigma$= {T1/a , Q1/[] , L1/[b,c] , L / [a|L2]}

  B' = concat(Q1 , L1 , L2)

  $B'\theta'$ = concat([] , [b,c] , L2).

  on rappelle la fonction Résoudre

- b_1 = concat([] , [b,c] , L2)

  -r_1 renommée donne r'_1:

  r'_1 = concat([] , L3, L3)

  -l'unification de b_1 et r'_1 produit :

  σ = {L3/[b,c] , L2/L3}

  $\theta' = \theta \circ \sigma$= {T1/a , Q1/[] , L / [a,b,c] , L3/[b,c] ,L2/[b,c]}

  B' = vide

  on rappelle la fonction résoudre

- la fonction Résoudre renvoie vrai car B=vide
  on extrait de θ les valeurs des variables du but initial (ici la variable L)
  on a L = [a,b,c]

  <u>REPONSE</u> : L = [a,b,c]

23. Quelques références en Prolog

• *Computing with logic*
D. Maier, D.S. Warren
Benjamin/Cumming Pub. Co. 1988

• *Fondements de la programmation logique*
J.W. Lloyd
Eyrolles 1988

• *L'art de Prolog*
L. Sterling, E. Chapiro
MIT press 1986

• *Logic, Programming and Prolog*
U. Nilsson, J. Maluszynski
John Willey 1990

• *Programmation en logique*
C.J. Hogger
Masson 1987

• ***Programmer en Prolog*** (accessible et bon marché)
W.F. Clocksin , C.S. Mellish
Eyrolles 1981

• *Programmer en Prolog pour l'intelligence Artificielle*
I. Bratko
InterEditions 1989

• *Prolog programming in depth*
M.A. Covington, D. Nute, A.Vellino
Scott, Foresman & Co. 1988

24. Solutions des exercices

- Exercice1 : recherche de chemins dans un graphe

Raisonnement inductif :

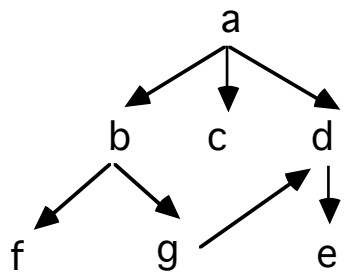
La base B : Il existe un chemin entre X et Y s'il existe un arc entre X et Y.

L'ensemble R : Il existe un chemin entre X et Y s'il existe un noeud Z et un arc entre X et Z et, il existe un chemin entre Z et Y.

(1)arc(a,b).
(2)arc(a,c).
(3)arc(a,d).
(4)arc(d,e).
(5)arc(b,f).
(6)arc(b,g).
(7)arc(g,d).

(8)chemin(X,Y) :- arc(X,Y).
(9)chemin(X,Y) :- arc(Z,Y) , chemin(X,Z).

Représentation par un graphe orienté



- Exercice2 : base de données familiale

```
epoux(X,Y)          :- epouse(Y,X).
mere(X,Y)           :- pere(Z,Y) , epouse(X,Z).
femme(X)            :- non homme(X).
parent(X,Y)         :- pere(X,Y) ; mere(X,Y).
enfant(X,Y)         :- parent(Y,X).
fils(X,Y)           :- enfant(X,Y) , homme(X).
fille(X,Y)          :- enfant(X,Y) , femme(X).
parent(X,P,M):-  % P et M sont les parents de X
        pere(P, X) , mere(M, X).
gdpere(X,Y)         :- parent(Z,Y) , pere(X,Z).
gdmere(X,Y)         :- parent(Z,Y) , mere(X,Z).
frere_ou_soeur(X,Y)    :- pere(Z,X) , pere(Z,Y) , not (X = Y) .
frere(X,Y)      :- frere_ou_soeur(X,Y) , homme(X).
soeur(X,Y)      :- frere_ou_soeur(X,Y) , femme(X).

oncle(X,Y)      :- parent(Z,Y) , frere(X,Z).
oncle(X,Y)      :- parent(Z,Y) , soeur(T,Z) , epoux(X,T).

tante(X,Y)      :- parent(Z,Y) , soeur(X,Z).
tante(X,Y)      :- parent(Z,Y) ,frere(X,Z) , epouse(X,T).

ancetre(X,Y)   :- parent(X,Y).
ancetre(X,Y)   :- parent(Z,Y) , ancetre(X,Z).
....
```

- Exercice-3 : Base de données ingrédients pour les repas

```
ingredient(gateau, il_faut(lait, il_faut(farine, il_faut(oeuf, rien)))).
ingredient(the, il_faut(sachet_de_the, il_faut(eau, rien))).
ingredient(oeuf_dur, il_faut(oeuf, il_faut(eau, rien))).

disponible(eau).
disponible(sachet_de_the).
disponible(oeuf).

peut_preparer(R) :- ingredient(R, Ingr), tous_disponibles(Ingr).

tous_disponibles(rien).
tous_disponibles(il_faut(X, Y)) :-
        disponible(X), tous_disponibles(Y).

a_besoin_de(R, I) :- ingredient(R, Ingr), est_dans(I, Ingr).

est_dans(X, il_faut(X, _)).
est_dans(X, il_faut(_, Z)) :- est_dans(X, Z).
```

```
:- peut_preparer(R).
   ③ R = the ;
          R = oeuf_dur ;

:- a_besoin_de(X, I).
   ③ X = gateau ,      I = lait ;
          X = gateau ,      I = farine ;
          X = gateau ,      I = oeufs ;
          X = the ,         I = sachet_de_the ;
          X = the ,         I = eau ;
          X = oeuf_dur, I = oeuf ;
          X = oeuf_dur, I = eau ;
```

• Exercice-4 : la description de l'additionneur 3 et 8 bits

```
and(_,[E1,E2],[S]) :- and([E1,E2],[S]).
and(_,[E1,E2|R],[S]) :- and([E1,E2],[S1]) , and(_,[S1|R],[S]).
and([0,0],[0]).   and([0,1],[0]).   and([1,0],[0]).   and([1,1],[1]).

or([0,0],[0]).   or([0,1],[1]).   or([1,0],[1]).   or([1,1],[1]).
or(_,[E1,E2],[S]) :-or([E1,E2],[S]).
or(_,[E1,E2|R],[S]) :-or([E1,E2],[S1]) , or(_,[S1|R],[S]).

nand(_,E,S) :- and(_,E,S1) , inv(_,S1,S).

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

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

add3bits(_,[X1,X2,X3],[Y1,Y2]) :-
  and(et1,[X1,X3],[U1]), and(et2,[X2,U3],[U2]), or(ou1, [U1,U2],[Y1]),
  orx(oux1, [X1,X3], [U3]),
  orx(oux2, [X2,U3], [Y2]).

add2bits(_,[X1,X2],[Y1,Y2]) :-
  and(et,[X1,X2],[Y1]),
  orx(oux, [X1,X2], [Y2]).

add4bits(_,[X1,X2,X3, X4],[Y1,Y2,Y3]) :-
  add2bits(a1, [X1,X2], [Z1,Z2]),
  add2bits(a2, [X3,X4], [K1,K2]),
  add2bits(a3, [Z2,K2], [L1,Y3]),
  add3bits(a4,[L1,Z1,K1],[Y1,Y2]).

add8bits(_,[X1,X2,X3, X4, X5,X6,X7,X8],[Y1,Y2,Y3,Y4]) :-
  add4bits(a1,[X1,X2,X3, X4],[P1,Z1,K1]),
  add4bits(a2,[X5,X6,X7, X8],[P2,Z2,K2]),
  add2bits(a3, [K1,K2], [L1,Y4]),
  add3bits(a4,[Z1,Z2,L1],[L2,Y3]),
  add3bits(a5,[P1,P2,L2],[Y1,Y2]).
```

- Exercice-5-1: le Quick Sort

cette solution évite d'utiliser la concaténation
pour une utilisation classique : qsort(à_trier, triée, []).
ou bien : qsort(à_trier, triée, fin_de_liste_triée)

```
qsort([X|L], R, R0) :-
    partition(L, X, L1, L2),
    qsort(L2, R1, R0),
    qsort(L1, R, [X|R1]).
qsort([], R, R).

partition([X|L], Y, [X|L1], L2) :-
    X=<Y,  !,   partition(L, Y, L1, L2).
partition([X|L], Y, L1, [X|L2]) :-
    partition(L, Y, L1, L2).
partition([], _, [], []).
```

Exemples :

?-qsort([3, 1, 5], R, []). ➋ R = [1, 3, 5]
?-qsort([3, 1, 5], R, [9, 19]). ➋ R = [1, 3, 5, 9, 19]

Exercice 5-3 : fusion de deux listes ordonnées.

```
merge(A, [], A).
merge([], B, B).
merge([A|As], [A|Bs], [A|Cs]) :- merge(As, Bs, Cs). % suppression des doublons
merge([A|RestAs], [B|RestBs], [A|Merged]) :-
   A @< B,
   merge(RestAs, [B|RestBs], Merged).
merge([A|RestAs], [B|RestBs], [B|Merged]) :-
   B @=< A,
   merge([A|RestAs], RestBs, Merged).
```

Exercice 5-4

```
length([], 0).
length([H|T], succ(Y)) :-  length(T, Y).
```

- Exercice-6 : la dérivation

```
d(U+V, X, DU+DV) :-              !,  d(U, X, DU), d(V, X, DV).
d(U-V, X, DU-DV) :-                 !,  d(U, X, DU), d(V, X, DV).
d(U*V, X, DU*V+U*DV) :-             !, d(U, X, DU), d(V, X, DV).
d(U/V, X, (DU*V-U*DV)/V^2) :- !, d(U, X, DU), d(V, X, DV).
d(U^N, X, DU*N*U^N1) :-             integer(N), N1 is N-1, d(U, X, DU).
d(-U, X, -DU) :-                    !, d(U, X, DU).
d(exp(U), X, exp(U)*DU) :-       !, d(U, X, DU).
d(log(U), X, DU/U) :-            !, d(U, X, DU).
d(X, X, 1) :-                       !.
d(C, X, 0).
```

Exemples :

- ?-d(x, x, N).
  ③ N = 1

- ?-d(y ^2 - 3 * x, x, M).
  ③ M = 0*2*y^1-(0*x+3*1) ;
  M = 0-(0*x+3*1) ;

- ?- d(exp(y ^2 - 3 * x), x, M).
  ③ M = exp(y^2-3*x)*(0*2*y^1-(0*x+3*1)) ;
  M = exp(y^2-3*x)*(0-(0*x+3*1)) ;

- ?-d(1, x, N).
  ③ N = 0 ;

• Exercice-7 On dispose d'une base de données de trafic ferroviaire où les faits sont présentés sous la forme :

train(Origine, Destination, H_dep, H_arr).

H_dep et H_arr sont de la forme *HH:MM*.

Ecrire le prédicat *chemin(Lieu1, Lieu2, Liste_Gares, Durée)* qui calcule la liste des gares et le temps nécessaire pour aller de lieu1 à lieur2.

```
:-op(15, xfx, ':').

% --------------------------- liste des trains ---------------------------
train(lyon,paris, 8:00, 10:0).
train(lyon,paris, 9:00, 11:05).
train(lyon,paris, 10:00, 12:10).
train(paris, bordeaux, 8:00, 12:0).
train(paris, bordeaux, 11:50, 16:30).
train(paris, bordeaux, 16:20, 22:40).
train(lyon,clermont, 10:00, 13:10).
train(lyon,clermont, 14:10, 18:10).
train(clermont, bordeaux, 10:50, 15:40).
train(clermont, bordeaux, 13:10, 18:30).
train(clermont, bordeaux, 18:20, 21:50).

% chemin(Lieu1, Lieu2, Liste_des_gares, durée total du voyage)
% Il n'y a pas de contrainte d'heure de départ, on fixe contrainte = 0:0
% Liste_des_gares est une liste de la forme :
%              [train(Lieu_dep, Lieu_arr, H_dep, H_Arr),..... ]

chemin(Lieu1, Lieu2, Liste_gares, Duree) :-
   chemin(Lieu1, Lieu2, 0:0, Liste_gares, Duree).

% chemin(Lieu1, Lieu2,Contrainte_d_heure,Liste_gares,durée_voyage)
% La contrainte (HH:MM) précise que l'on ne prend que les trains
% partant du Lieu1 après HH:MM
```

```
chemin(Lieu1, Lieu1, _, [], 0:0) :- !.
chemin(Lieu1, Lieu2, Contrainte, [train(Lieu1,Lieu3,Dep,Arr)|Reste], Duree) :-
    train(Lieu1,Lieu3,Dep, Arr),
    respect_contrainte(Dep, Contrainte),
    duree(Dep,Arr,Dur1),
    chemin(Lieu3, Lieu2, Arr, Reste, Durs),     %contrainte = Arr
    somme(Dur1, Durs, Duree).

% verifier que l'heure de départ A:B est ">" que la Contrainte C:D
respect_contrainte(A:B, C:D) :-
    A > C ; A = C, B >= D.

%calcul de la durée E:F en fonction de H.dep (A:B) et H. arr (C:D)
duree(A:B, C:D, E:F) :-
    D >= B ,!, E is C-A, F is D-B.
duree(A:B, C:D, E:F) :-
    E is C-A-1, F is D-B+60.

% addition de deux durées A:B + C:D => E : F
somme(0:0,X,X) :- !.
somme(X,0:0, X) :- !.
somme(A:B, C:D, E:F) :-
    E1 is A + C, F1 is B+D,
    (F1 > 60  ->  (F is F1 mod 60, E is E1 +1)
          ;    (F is F1, E is E1)).
```

TESTS

:- chemin(lyon, bordeaux, X, Y). % On veut partir n'importe quand

X = [train(lyon, paris, 8:0, 10:0), train(paris, bordeaux, 11:50, 16:30)]
Y = 6:40 ;

X = [train(lyon, paris, 8:0, 10:0), train(paris, bordeaux, 16:20, 22:40)]
Y = 8:20 ;

X = [train(lyon, paris, 9:0, 11:5), train(paris, bordeaux, 11:50, 16:30)]
Y = 6:45 ;

X = [train(lyon, paris, 9:0, 11:5), train(paris, bordeaux, 16:20, 22:40)]
Y = 8:25 ;

X = [train(lyon, paris, 10:0, 12:10), train(paris, bordeaux, 16:20, 22:40)]
Y = 8:30 ;

X = [train(lyon, clermont, 10:0, 13:10), train(clermont, bordeaux, 13:10, 18:30)]
Y = 8:30 ;

X = [train(lyon, clermont, 10:0, 13:10), train(clermont, bordeaux, 18:20, 21:50)]
Y = 6:40 ;

X = [train(lyon, clermont, 14:10, 18:10), train(clermont, bordeaux, 18:20, 21:50)]
Y = 7:30 ;

:- chemin(lyon, bordeaux, 12:0, X, Y). % On veut partir après 12:0

X = [train(lyon, clermont, 14:10, 18:10), train(clermont, bordeaux, 18:20, 21:50)]
Y = 7:30 ;

• Exercice-8 : le réseau d'irrigation

• Exercice-9,10 : calcul de capital

- Exercice-11 : extension de la base de données ingrédients pour les repas.

- Exercice12 : Définition d'un petit langage de programmation

Les définitions des opérateurs et des règles ci-dessus permettent d'interpréter un petit langage de programmation.

Ici, il n'y a pas d'analyse syntaxique et les erreurs syntaxiques produirons des messages d'erreur Prolog.

p dans ce langage, les variables commencent par une lettre minuscule.

```
:- op(1050, xfy, ou).
:- op(1000, xfy, et).
:- op(1100, xfy, alors).
:- op(1030, fx, si).
:- op(1150, xfy, sinon).
:- op(700, xfx, ':=').          % effactation : <variable := expression-arithmétique>
:- op(850, xfx, vaut).      % comparaison d'égalité de deux expression : <e1 vaut e2>
:- op(1150, xfy, puis).     % opérateur de séquencement
:- op(900, fx, ecrire).
:- op(900, fx, lire).

si X alors Y sinon Z :- !,  X -> Y ; Z.
si X alors Y :- !,  X -> Y.
X puis Y :- X, Y, !.
X puis Y .
X vaut Y :-  eval(X, K) ,  eval(Y, Z),  K=Z.
X := Y :-  eval(Y, Z),  (retract(compteur(X, N)); true),  assert(compteur(X, Z)).
ecrire X :- eval(X, N), !,  write(N),  nl.
ecrire X :- write(X),  nl.
lire X :- read(Y),  (retract(compteur(X, N)); true),  assert(compteur(X, Y)).
eval(X+Y,  Z) :-     !,  eval(X, X1),  eval(Y, Y1) ,  Z is X1+Y1.
eval(X-Y,  Z) :-          !,  eval(X, X1),  eval(Y, Y1) ,  Z is X1-Y1.
eval(X*Y,  Z) :-          !,  eval(X, X1),  eval(Y, Y1) ,  Z is X1*Y1.
eval(X/Y,  Z) :-          !,  eval(X, X1),  eval(Y, Y1) ,  Z is X1/Y1.
eval(X, Y) :-          compteur(X, Y),  !.
eval([X|Y],  Z) :-    !,  name(Z, [X|Y]). % cas de chaine
eval(X, X) :-          atomic(X),  !.
eval(X, Y) :-          Y is X.
```

Exemples de programmes interprétés dans ce langage:

x := 10 puis ecrire x.
③ 10

lire x puis ecrire x.
20.
③ 20

si x vaut 15 alors ecrire x sinon ecrire x-1.
③ 19

si x vaut 15 alors ecrire x sinon ecrire x-1 puis lire y puis ecrire y.
③ 19
10.
③ 10

si x vaut y alors ecrire yes sinon ecrire non.
③ non

ecrire x.
③ 20

si x+1 vaut x-1+2 alors ecrire hourra.
③ hourra

Exercice-12 bis : puzzle simple.

construire un nombre N contenant une seule occurrence tous les chiffres 1,2,3,4,5,6,7,8,9 tel que les premiers k chiffres de N soient divisibles par k, pour k=1..9.

Par exemple, le nombre N= 381654729 a cette propriété car pour

k=1 3 est divisible par 1

k=2 38 est divisible par 2

k=3 381 est divisible par 3

....

```
stageDiv([],Divisor,Result,NewNumber) :- %finished - just generate result number from string
    name(NewNumber,Result).
stageDiv(Digits,Divisor,DigitListSoFar,Result) :-
    select(NewDigit,Digits,RestOfDigits), %pick some digit
    append(DigitListSoFar,[NewDigit],NewDigitList),
    name(NewNumber,NewDigitList), %turn it into a number
    0 is NewNumber mod Divisor, %check against next divisor
    NewDivisor is Divisor+1,
    stageDiv(RestOfDigits,NewDivisor,NewDigitList,Result). %do the rest

append([],X,X).
append([X|Y],Z,[X|R]) :- append(Y,Z,R).

select(X,[X|R],R).
select(X,[Y|R],[Y|Z]) :- select(X,R,Z).
```

• Exercice-13 : copier le contenu d'un fichier texte dans un autre en transformant les lettres minuscules en majuscules.

• Exercice-14 : On dispose de deux fichiers contenant des entiers ordonnés. Créer un troisième fichier ordonné résultant de la fusion de ces deux fichiers.

• Exercice-15 **:** écrire deux prédicats "p" et "v".

• Exercice-16 **:** Etant donné un ensemble de faits *age(nom,age)* et *sexe(nom,sexe)*; produire et insérer dans la base l'union de ces deux relations i.e. les faits *ags_sexe(nom, age, sexe)* à partir des deux premiers faits.

• Exercice-17 : liste des feuilles d'un ABOH

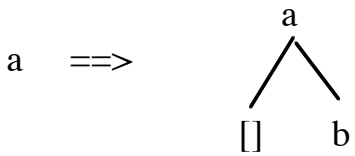
• *Exercice-17bis : Insertion dans un arbre binaire ordonné horizontalement :*

```
inser(Ele, Arbre, New_arbre) :-
    Arbre =.. [Racine, Gauche, Droite] ,
    inser_bis(Ele, Racine, Gauche, Droite,New_gauche, New_Droite) ,
    New_arbre =.. [Racine, New_gauche, New_Droite] .
inser(Ele, [], Ele) :- !.
inser(Ele, Feuille, New_arbre) :-
    Feuille \== [], Feuille =.. [Feuille],          % c'est une feuille <> []
    inser_bis(Ele, Feuille, [], [], Gauche, Droite),
    New_arbre =.. [Feuille, Gauche, Droite] .

inser_bis(Ele, Racine, Gauche, Droite,New_gauche, Droite)      :-
    plus_petit(Ele, Racine) ,
    inser(Ele, Gauche, New_gauche).
inser_bis(Ele, Racine, Gauche, Droite,Gauche, New_droite)      :-
    plus_petit(Racine, Ele) ,
    inser(Ele, Droite, New_droite).

plus_petit(X,Y) :-          /* selon le type des informations */
      X @< Y.               /* si le type est atome ou nombre*/
```

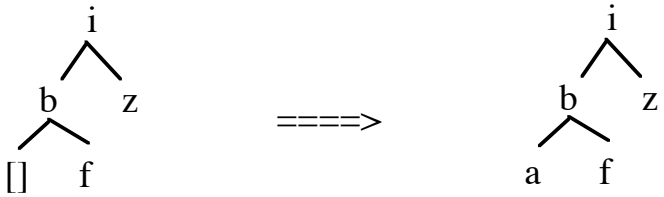
Exemples:

:- inser(a,[], N). ❷ N = a

:- N=a , inser(b,N,N1). ❷ N1 = a([], b).



:- N= i(b([], f),z) , inser(a,N,N1). ❷ N= i(b(a, f),z).



• Exercice-18 : le disctionnaire sous forme d'un ABOH

• Exercice-19 : le problème des n-reines

❺ Une solution simple pour N=4 par la méthode constructive.

La forme de la requête tient déjà compte de la seconde contrainte.

```
    nreines([]).
nreines([pion(L, C)|Reste]) :-
    nreines(Reste), membre(C, [1, 2, 3, 4]),  safe(pion(L, C), Reste).

safe(_, []).
safe(pion(L1, C1), [pion(L2, C2)|L]) :-
    L1 \== L2,  C1 \== C2,                    % la 1ère et la seconde contraintes
    CC is C1-C2,  LL is L1-L2,                   % 3ème contrainte
    CCabs is abs(CC),  LLabs is abs(LL),  LLabs =\=CCabs,
    safe(pion(L1, C1),  L).
```

?- nreines([pion(1, C1), pion(2, C2), pion(3, C3), pion(4, C4)]).

③ C1 = 3 , C2 = 1 , C3 = 4 , C4 = 2

③ C1 = 2, C2 = 4 , C3 = 1 , C4 = 3

❺ Une solution par la méthode génération/test consiste à générer une configuration totale puis de tester si celle-ci convient.

- Exercice-20 : Le problème des n-reines généralisé.

Principe de la solution (prédicat nreines(+N,?Liste)) :

- on génère une liste pion(Li,Ci) où les Li = N° de lignes avec i=N..1.
  Les Ci restent des variables.
  exemple : avec N=2, on génère [pion(2,X),pion(1,Y)].
- on appelle n_reines avec cette liste pour instancier les Ci
- le prédicat une_valeur_dans_n(+N, ?C) génère successivement les valeurs N..1 pour les Ci.
- le prédicat safe vérifie si une configuration donnée respecte les contraintes.

```
nreines(N, Liste) :-    generer(N, Liste),  n_reines(Liste, N).

generer(0, []).
generer(N, [pion(N, X)|Reste]) :- N > 0,  N1 is N-1 ,  generer(N1, Reste).

n_reines([], N).
n_reines([pion(L, C)|Reste], N) :-
    n_reines(Reste, N),  une_valeur_dans_n(N, C),
    safe(pion(L, C), Reste).

safe(_, []).
safe(pion(L1, C1), [pion(L2, C2)|L]) :-
    L1 \== L2,  C1 \== C2,              % Les deux premières contraintes
    CC is C1-C2,  LL is L1-L2,  CCabs is abs(CC),  LLabs is abs(LL),  LLabs =\= CCabs,
    safe(pion(L1, C1),  L).

une_valeur_dans_n(N, N) :- N > 0.
une_valeur_dans_n(N, M) :- N > 0 ,  N1 is N-1, une_valeur_dans_n(N1, M).
```

Exemples de requêtes :

?- nreines(4, X).

③ X = [pion(4, 2), pion(3, 4), pion(2, 1), pion(1, 3)] ;

X= [pion(4, 3), pion(3, 1), pion(2, 4), pion(1, 2)]

?-nreines(8, N). ==> ....

- Exercice-21 : Parcours de cavalier.

1- solution à la première question pour N=3.

Mouvements possibles à partir d'une case:

```
move(1,6). move(1,8).
move(2,7). move(2,9).
move(3,4). move(3,8).
move(4,3). move(4,9).
move(6,7). move(6,1).
move(7,6). move(7,2).
move(8,3). move(8,1).
move(9,4). move(9,2).
```

Prédicat principal

```
path(P,Q,R):-  path(P,Q,[P],R).
path(Z,Z,L,R) :- reverse(L,R).
path(X,Y,L,R) :-
    move(X,Z),    not(membre(Z,L)),   path(Z,Y,[Z|L],R).
```

utilitaires

```
membre(X,[X|T]).
membre(X,[Y|T]) :- membre(X,T).

reverse(L,R):-  reverse(L,[],R).
reverse([],R,R).
reverse([H|T],S,R):- reverse(T,[H|S],R).
```

Exemple de question : path(2, 4, R).

==> R = [2,7,6,1,8,3,4] et

R = [2,9,4]

Fonction resoudre (B : but ; θ : substitution) retourne booléen =

B : le but en entrée; θ : la substitution en entrée-sortie

Si B = vide alors retourne (vrai) -- n = 0 dans B=b_1 , ... , b_n. Plus rien à prouver

Sinon

considérer b_1 dans B; b_1 de la forme f(a_1,...,a_m)

LC = l'ensemble des faits de P de la forme f(a'_1,...,a'_m)

Tant que LC <> vide faire

choisir C le premier élément de LC

LC = LC - {C}

⑩ Si **unifier**(θ(a_i), θ(a'_i), θ) pour i=1..m

Alors B' = b_2 , ... , b_n

 Si resoudre(θ(B'), θ) alors retourne (vrai) Fin si

Fin si

Fin Tant que

Retourne (faux)

Fin si

Fin resoudre

7.1- Unification

Principe (sans le traitement des termes fonctionnels) :

Unifier deux termes t1 et t2 revient à trouver des valeurs pour les variables de ces termes telles que t1 et t2 deviennent identiques. On peut présenter le principe de l'unification par :

- Deux termes identiques s'unifient ;
- Une variable s'unifie avec n'importe quel autre terme ;
- Deux constantes différentes ne s'unifient pas

Pour unifier deux termes t1 et t2 en Prolog, on écrit $t_1 = t_2$.

Exprimons ce principe par la fonction suivante :

```
Fonction unifier(t1,t2 : terme; θ : substitution) retourne booléen =
   Cas
      t1 est identique à t2  ⇒ retourne (vrai)
      t1 est variable        ⇒ θ := θ °{t1 / t2}; retourne (vrai)
      t2 est variable        ⇒ θ := θ °{t2 / t1}; retourne (vrai)
      autres                 ⇒ retourne (faux)
   Fin cas
Fin unifier
```

• La notation *θ °{t1 / t2}* a pour effet d'ajouter le couple {t1/t2} à θ en respectant certaines conditions :

- θ ne contient pas de couples tels que *t/t'* et *t/t"* (les *t* identiques). Ainsi, une substitution définit une fonction où pour t, on obtient une seule valeur t' (ou t").

- θ ne contient pas de couples tels que {X/X}.

- Généralement, une variable t' figurant à droite d'un couple t/t' ne doit pas figurer à gauche d'un autre couple t'/t". Nous reviendrons sur cette condition trop restrictive dans les algorithmes en annexe.

Exemples :

```
12      =? 15          => échec
X       =?eleve        => succès,  θ={X/eleve}
X       =?Y            => succès,  θ={X/Y}
X       =?X            => succès,  θ={}
Ecole   =?"ecole"      => succès,  θ={Ecole/"ecole"}
'ecole' =?"ecole"      => écheccar l'atome 'ecole' ≠ la chaîne "ecole"
```

```
Fonction resoudre (B : but ; θ : substitution) retourne booléen =
   Si B = vide alors retourne (vrai)      % n = 0
   Sinon
      considérer b1 dans B; b1 de la forme f(a1,...,am)
      LC = l'ensemble des règles de P de la forme
⑩                f(a'1,...,a'm) :- a1,..., ak
      Tant que LC <> vide faire
            choisir C le premier élément de LC
            LC = LC - {C}
❡     Renommer C tel que C et b1 n'aient pas de variable commune
            Si unifier(θ(ai ), θ(a'i ), θ) pour i=1..m
            alors
⑦                B' = a1,..., ak , b2 , ... , bn
                 Si resoudre(θ(B'), θ) alors retourne (vrai) Fin si
            Fin si
      Fin Tant que
      Retourne (faux)
   Fin si
   Fin resoudre
```

θ : substitution

Fonction unifier(T1,T2 : termes) retourne booléen =

Cas

- T1 est identique à T2 : retourne(vrai)
- T1 est une variable : $\theta = \theta \circ \{T1/T2\}$; retourne(vrai)
- T2 est une variable : $\theta = \theta \circ \{T2/T1\}$; retourne(vrai)
- T1 est une constante ou T2 est une constante : retourne(faux)
- T1 et T2 sont des termes composés ayant le même symbole fonctionnel et le même nombre d'arguments. Soit :

  T1 = f(t1, t2,..., tn) et T2 = f(t'1, t'2,..., t'n).

  Il faut unifier un par un les arguments de T1 et de T2 et, à chaque étape, appliquer θ aux termes restants. Ce qui est effectué par la boucle suivante :

  Répéter Pour i=1..n
  appliquer θ à t_i et à t'_i
  test = unifier(t_i, t'_i)
  Jusqu'à non test
  retourne(test)
- Autre : retourne(faux)