PROLOG

I - INTRODUCTION

Le langage PROLOG est basé sur le calcul des prédicats du premier ordre (avec quelques extensions et restrictions). C'est une implémentation du *Principe de Résolution* (Robinson 1965) avec des stratégies particulières et des restrictions[1].

PROLOG a été conçu par A.Colmerauer à Marseille. Depuis, de nombreux *interpréteurs* PROLOG ont été écrits, avec des syntaxes différentes.
Dans ce qui suit, on utilisera la syntaxe et les conventions les plus couramment utilisées maintenant. Ce sont celles utilisées SWI-Prolog (utilisé dans ce module) ainsi que par, entre autres, C-Prolog, Sixtus-Prolog.

PROLOG peut être vu comme un langage d'expression de connaissances. Il peut être vu aussi comme un langage de programmation adapté au calcul symbolique et à la manipulation d'objets, de prédicats et de relations.
Il n'y a pas d'instruction, pas d'affectation, pas de boucles explicites, mais uniquement des *clauses* exprimant des faits, des règles et des questions.

Exemples de faits :

```
pere(alain, jeanne).
pere(alain, michel).
pere(michel, robert).
mere(sylvie, robert).
mere(sylvie, luc).
pere(georges, sylvie).
```

Exemples de questions :

	Réponses :	
`:- pere(alain, jeanne)`	oui	
`:- pere(alain, robert)`	non	
`:- pere(alain, X)`	oui	`X = michel`
		`X = jeanne`
`:- mere(X, robert)`	oui	`X = sylvie`
`:- mere(X, michel)`	non	
`:- pere(michel, X), mere(sylvie, X)`	oui	`X = robert`
`:- pere(alain, X), pere(X, robert)`	oui	`X = michel`
`:- grand_pere(alain, robert)`	non	

Exemple de règle, permettant de définir[2] la relation *grand-père* :
X est grand-père de Y si X est le père de quelqu'un qui est le père ou la mère de Y, c'est-à-dire s'il existe Z tel que X est le père de Z et Z est le père de Y, soit, en logique (calcul des prédicats du premier ordre) :

$$\exists Z \left(\text{père}(X,Z) \wedge \left(\text{père}(Z,Y) \vee \text{mère}(Z,Y)\right)\right) \Rightarrow \text{grand-père}(X,Y)$$

que l'on peut aussi écrire :

$$\forall Z \left(\left(\text{père}(X,Z) \wedge \text{père}(Z,Y) \Rightarrow \text{grand-père}(X,Y)\right) \wedge \left(\text{père}(X,Z) \wedge \text{mère}(Z,Y)\right) \Rightarrow \text{grand-père}(X,Y)\right)\right)$$

[1] restriction aux clauses de Horn, utilisation de la stratégie LUSH, recherche en profondeur, essai des clauses d'entrée dans l'ordre où elles se présentent, utilisation d'une simplification de l'unification

[2] Il ne s'agit en fait que d'une condition suffisante

Cette dernière formulation est celle qui sera utilisée en PROLOG, avec une quantification universelle implicite de toutes les variables :

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

On peut alors interroger sur la relation `grand_pere` :

```
:- grand_pere(alain, robert)  oui
:- grand_pere(alain, X)       oui    X = robert
:- grand_pere(X, robert)      oui    X = alain
                                     X = georges
:- grand_pere(X,Y)            oui    X = alain    Y = robert
                                     X = georges  Y = robert
                                     X = georges  Y = luc
```

Fonctionnement :

`grand_pere(X,Y)` ?

1. pere(X,Z), pere(Z,Y)
   - 1.1. X=alain, Z=jeanne, pere(jeanne,Y) ? échec
   - 1.2. **X=alain, Z=michel,** pere(michel,Y) ? oui, **Y=robert**
   - 1.3. X=michel, Z=robert, pere(robert,Y) ? échec
   - 1.4. X=georges, Z=sylvie, pere(sylvie,Y) ? échec
2. pere(X,Z), mere(Z,Y)
   - 2.1. X=alain, Z=jeanne, mere(jeanne,Y) ? échec
   - 2.2. X=alain, Z=michel, mere(michel,Y) ? échec
   - 2.3. X=michel, Z=robert, mere(robert,Y) ? échec
   - 2.4. **X=georges, Z=sylvie,** mere(sylvie,Y) ? oui, **Y=robert**
     Y=luc

II - QUELQUES DEFINITIONS

Une **constante** est une chaine de caractères commençant par une lettre minuscule, ou un nombre.

Exemples : `a, sylvie, 2000`

Une **variable** est une chaine de caractères commençant par une lettre majuscule ou un _ .

Exemples : `X, A, _a, Sylvie, _`

Un **terme** est une constante, ou une variable, ou un symbole fonctionnel appliqué à une liste de termes.

Exemples : `X, a, f(a), a+3, 3+4`

Un **littéral** est un symbole de prédicat (relation) appliqué à une liste (pouvant être vide) de termes, et exprimant une propriété qui peut être vraie ou non.

Exemples : `p(a), q(X), pere(alain,Y), true`

Remarque: il ne doit pas y avoir de blanc entre le prédicat et la parenthèse .

Un **clause** PROLOG[1] est une suite ordonnée de littéraux composée

- soit d'un littéral dit positif (facultatif), puis du symbole :- , puis de littéraux dits négatifs séparés par des virgules, et terminée par un point.

  Exemples :
  ```
  grand_pere(X,Y) :- pere(X,Z), pere(Z,Y).
  :- pere(alain,Z), pere(Z,Y).        (clause négative)
  :- pere(alain,Y).                   (clause négative unitaire)
  ```

- soit d'un littéral positif suivi d'un point.

  Exemple : `pere(alain, jeanne).` (clause positive)

[1] restriction par rapport à la définition des clauses dans le calcul des prédicats

la clause `p :- n1, n2, … , np.` (resp. `:- n1, n2, … , np.`)
représente la formule logique
$n_1 \wedge n_2 \wedge ... \wedge n_p \Rightarrow p$ (resp. $\neg (n_1 \wedge n_2 \wedge ... \wedge n_p)$)
ou $\neg n_1 \vee \neg n_2 \vee ... \vee \neg n_p \vee p$ (resp. $\neg n_1 \vee \neg n_2 \vee ... \vee \neg n_p$)

`p` est la **tête** de la clause, $n_1, n_2, \ldots, n_p$ la **queue** de la clause

Un **paquet de clauses** est un ensemble de clauses ayant le même nom de littéral de tête (littéral positif).

Une **demande de résolution** ou **question** ou **liste de buts** est une clause négative (c'est-à-dire sans littéral positif)

`:- b1 , b2 ,... , bp .`

Un **programme** PROLOG est un ensemble de *clauses* non négatives.

III - FONCTIONNEMENT DE L'INTERPRETEUR

L'interpréteur prend le premier but b_1, qu'il essaie de *résoudre*, c'est-à-dire il essaie d'unifier b_1 avec une tête de clause.
S'il réussit, il cherche à résoudre la queue de la clause, instanciée par l'unification, en résolvant, dans l'ordre, chacun des littéraux de cette queue; puis il essaie de résoudre le prochain but en attente. En cas d'échec, il y a *retour arrière* au dernier choix effectué.
Quand une solution a été trouvée, on peut, soit arrêter la recherche, en tapant [Entrée] (Retour-Charriot), soit demander les autres, en tapant ; suivi de [Entrée]. (Dans certains PROLOG, on a systématiquement toutes les solutions.)
La résolution est terminée s'il n'y a plus de littéraux à résoudre et plus de choix à traiter.

Exemple:

Programme:

```
grand_pere(X,Y) :- pere(X,Z) , pere(Z,X) .
grand_pere(X,Y) :- pere(X,Z) , mere(Z,X) .
pere(jean, pierre) .
pere(jean, simon) .
pere(pierre, jacques) .
pere(jules, robert) .
pere(jules, jeanne) .
mere(jeanne, pierre) .
mere(jeanne, simon) .
mere(pierrette, jacques) .
mere(pierrette, alain) .
mere(julie, robert) .
mere(julie, anne) .
```

Demande:

```
:- grand_pere(GP,PE),write(GP),write(' est le grand-père de '),write(PE),nl.
```

Le premier but

```
grand_pere(GP,PE)
```

est unifié avec le premier littéral de la première clause, et remplacé par la liste de buts

```
pere(GP, Z) , pere(Z, PE)
```

La première question de cette nouvelle demande donne la substitution

`GP` $\rightarrow$ `jean` ; `Z` $\rightarrow$ `pierre`

qui, reportée dans la deuxième question donne

`pere(pierre,PE)`
et `PE` → `jacques`
Le deuxième but de la demande initiale
`write(GP)` devenu `write(jean)`
écrit la constante `jean`.
Le troisième but
`write(' est le grand-père de ')`
écrit la chaîne de caractères donnée en argument.
Le quatrième but `write(jacques)` écrit `jacques`.
Enfin, `nl` fait aller à la ligne.
L'interpéteur va donc écrire
`jean est le grand-père de jacques.`
Si l'on désire les autres solutions (; [Entrée]), l'interpréteur PROLOG effectue des retours arrière, et la demande
`pere(GP, Z) , pere(Z, PE)`

donne alors successivement

GP	Z	PE
jean	simon	échec
pierre	jacques	échec
jules	robert	échec
	jeanne	échec

Puis, la demande
`pere(GP, Z) , mere(Z, PE)`

donne

GP	Z	PE
jean	pierre	échec
	simon	échec
pierre	jacques	échec
jules	robert	échec
	jeanne	pierre
		simon

En fin de compte, l'interpréteur aura écrit
```
jean est le grand-père de jacques
jules est le grand-père de pierre
jules est le grand-père de simon
```

III - POUR TRAVAILLER EN SWI-PROLOG

Taper `pl` suivi d' [Entrée]

Le chargement de clauses (non négatives), à partir d'un fichier se fait par la commande
`:- consult(nom_du_fichier) .`
ou `:- [nom_du_fichier] .`
Si l'on veut écrire les clause au terminal, on tape
`:- consult(user).`
ou `:- [user].`
puis les clauses, puis [CTRL] D.

On ne peut pas charger une clause négative à partir d'un fichier. Pour ne pas devoir toujours taper les demandes directement à la console, on peut ranger dans un fichier la clause
`q(_) :-` < demande > .
par exemple

```
q(_) :- grand_pere(GP,PE),
        write(GP),write(' est le grand-père de '),write(PE),
        nl.
```

et demander seulement à la console `q(_).` si on veut une solution
`q(X).` si on veut toutes les solutions.

Pour lister le paquet de clauses de prédicat de tête p, taper `listing(p).`
Pour lister toutes les clauses, taper `listing.`
Pour exécuter une commande shell, taper `sh.`
on peut alors exécuter n'importe quelle commande shell, pour revenir sous PROLOG, taper [CTRL] D

On peut écrire des commentaires : tout ce qui suit % dans une ligne est un commentaire, ainsi que tout ce qui est encadré entre `/*` et `*/`.

Pour sortir, taper `halt.` (Ne pas oublier le point.)

IV - DECLARATIF VS PROCEDURAL

Un *programme* PROLOG peut être vu sous deux aspects: *déclaratif* ou *procédural*

On peut d'abord le penser d'une façon déclarative pour le concevoir, mais il sera ensuite interprété comme un programme, d'où un grand DANGER, mais aussi quelques possibilités d'améliorer l'efficacité.

Dans l'aspect déclaratif, la clause

```
p :- q , r .
```

est interprétée comme
p est vrai si q et r sont vrais
ou *q et r impliquent p*
Dans l'aspect procédural, la même clause est interprétée comme
Pour résoudre (vérifier) p, résoudre (vérifier) d'abord le sous-problème q,
puis le sous-problème r.
On définit ainsi l'*ordre* dans lequel les buts sont considérés.

De plus, d'un point de vue déclaratif, l'ordre des clauses n'a pas d'importance. Mais, d'un point de vue procédural, l'interpréteur considérant les clauses les unes après les autres, dans l'ordre où elles se trouvent, celui-ci a de l'importance.

Exemple: quatre versions du programme d'*ancêtres*

```
ancetre(X, Z) :- parent(X, Z) .
ancetre(X, Z) :- parent(X, Y) , ancetre(Y, Z) .
parent(michel, bernard) .
parent(thomas, bernard) .
parent(thomas, lise) .
parent(bernard, anne) .
parent(bernard, pierre) .
parent(pierre, jean) .

ancetre2(X, Z) :- parent(X, Y) , ancetre2(Y, Z) .
ancetre2(X, Z) :- parent(X, Z) .

ancetre3(X, Z) :- parent(X, Z) .
ancetre3(X, Z) :- ancetre3(X, Y) , parent(Y, Z)  .
```

```
ancetre4(X, Z) :- ancetre4(X, Y) , parent(Y, Z)  .
ancetre4(X, Z) :- parent(X, Z) .
```

Déroulement des différents *programmes*

On va poser les quatre questions correspondant aux quatre variantes de ce progamme

1. :- ancetre(thomas, pierre) .

La résolution donne

1. parent(thomas, pierre) ; échec
2. parent(thomas, Y) , ancetre(Y, pierre)
   - 2.1. Y = bernard ; but = ancetre(bernard, pierre) .
     - 2.1.1. parent(bernard, pierre) ; **succès**

       on peut alors s'arrêter ou demander de continuer

     - 2.1.2. parent(bernard,Y') , ancetre(Y', pierre)
       - 2.1.2.1 Y' = anne ; but = ancetre(anne, pierre)
         - 2.1.2.1.1. parent(anne,pierre) ; échec
         - 2.1.2.1.2. parent(anne, Y") , ancetre(Y", pierre)

           échec
       - 2.1.2.2. Y' = pierre ; ancetre(pierre, pierre)
         - 2.1.2.2.1. parent(pierre, pierre) ; échec
         - 2.1.2.2.2. parent(pierre, Y") , ancetre(Y", pierre)

           Y" = jean ; but = ancetre(jean, pierre)
         - 2.1.2.2.2.1. parent(jean, pierre) ; échec
         - 2.1.2.2.2.2. parent(jean, Y"') , ancetre(Y"', pierre)

           échec
   - 2.2. Y = lise ; but = ancetre(lise, pierre)
     - 2.2.1. parent(lise, pierre) ; échec
     - 2.2.2. parent(lise, Y') , ancetre(Y', pierre)

       échec

La solution est trouvée rapidement, puis, si on demande de continuer, la recherche, sans nouvelle solution, est plus longue.

2. :- ancetre2(thomas, pierre) .

1. parent(thomas, Y) , ancetre(Y, pierre)
   - 1.1. Y = bernard ; but = ancetre2(bernard,pierre)
     - 1.1.1. parent(bernard, Y') , ancetre2(Y', pierre)
       - 1.1.1.1. Y' = anne ; but = ancetre2(anne, pierre)
         - 1.1.1.1.1. parent(anne, Y") , ancetre2(Y", pierre)

           échec
         - 1.1.1.1.2. parent(anne, pierre) ; échec
       - 1.1.1.2. Y' = pierre ; but = ancetre2(pierre, pierre)
         - 1.1.1.2.1. parent(pierre, Y") , ancetre2(Y", pierre)

           Y" = jean ; but = ancetre2(jean, pierre)

           1.1.1.2.1.1.parent(jean, Y"'), ancetre2(Y"', pierre);

           échec

           1.1.1.2.1.2. parent(jean, pierre) ; échec
         - 1.1.1.2.2. parent(pierre, pierre) ; échec
     - 1.1.2. parent(bernard, pierre) ; '**succes**'

       on peut alors s'arrêter ou demander de continuer

   - 1.2. Y = lise ; but = ancetre2(lise,pierre)
     - 1.2.1. parent(lise, Y') , ancetre2(Y', pierre) ;

       échec
     - 1.2.2. parent(lise, pierre) ; échec
2. parent(thomas, pierre) ; échec

La solution est trouvée, *après* une recherche assez longue.

3. :- ancetre3(thomas, pierre) .

1. parent(thomas, pierre) ; échec
2. ancetre3(thomas, Y) , parent(Y, pierre)
   - 2.1. parent(thomas, Y'), parent(Y', pierre)
     - 2.1.1 Y' = bernard ; but = parent(bernard, pierre) ; '**succes**'
       on peut alors s'arrêter ou demander de continuer

     - 2.1.2 Y' = lise ; but = parent(lise, pierre) ; échec
   - 2.2. ancetre3(thomas, Y") , parent(Y", Y'), parent(Y', pierre)
     - 2.2.1. parent(thomas, Y") , parent(Y", Y'), parent(Y', pierre)
       - 2.2.1.1.Y" = bernard ; but = parent(bernard,Y') , parent(Y',pierre)
         - 2.2.1.1.1. Y' = anne ; but = parent(anne, pierre) ; échec
         - 2.2.1.1.2. Y' = pierre ; but = parent(pierre, pierre) ; échec
       - 2.2.1.2. Y" = lise ; but = parent(lise, Y'), parent(Y', pierre)
         échec
   - 2.2.2.ancetre3(thomas,Y'"),parent(Y'",Y"),parent(Y",Y'),parent(Y',pierre)
     - 2.2.2.1. parent(thomas, Y'") , parent(Y'", Y") , parent(Y", Y') ,
       parent(Y', pierre)
       etc ... (branche finie) ...
     - 2.2.2.2. ancetre3(thomas,Y""), parent(Y"", Y'"), parent(Y'", Y") ,
       parent(Y", Y'), parent(Y', pierre)
       etc ... (branche infinie) ...

La solution est trouvée, mais, si on demande de continuer, il y a alors bouclage. Le programme ne s'arrêtera pas.

4. :- ancetre4(thomas, pierre) .

1. ancetre4(thomas, Y) , parent(Y, pierre)
   - 1.1. ancetre4(thomas, Y') , parent(Y', Y) , parent(Y, pierre)
     - 1.1.1.ancetre4(thomas,Y") ,parent(Y",Y') ,parent(Y',Y) , parent(Y,pierre)
       - 1.1.1.1 etc ...
         ... (branche infinie) ...

Ici, la situation est catastrophique, on ne trouve même pas la solution.

Conclusion

Sur le plan *déclaratif*, les quatre versions du programme sont équivalentes. Elles définissent la même fonction. Mais les interprétations qui en sont faites ne sont pas les mêmes. Elles dépendent de l'ordre des clauses, et de l'ordre des littéraux dans une clause. Sur le plan procédural, seuls `ancetre` et `ancetre2` sont corrects. Ils donnent toutes les solutions et s'arrêtent. Si l'on ne cherche qu'une solution, `ancetre` est meilleur, car il la trouve plus rapidement. Les deux autres variantes bouclent, `ancetre3` après avoir trouvé toutes les solutions, `ancetre4` avant même d'en avoir trouvé une!

Le comportement de ces programmes peut s'expliquer par la remarque suivante: il vaut mieux essayer d'abord ce qui est simple : `parent` plutôt que `ancetre`. Le meilleur est celui qui donne la priorité à `parent`, à la fois pour ordonner les clauses, et pour ordonner les littéraux. Le plus mauvais est celui qui donne la priorié à `ancetre` dans les deux cas.

Pour écrire un programme PROLOG, il est conseillé de le *penser* d'abord d'une manière déclarative pour sa conception. Puis, pour l'exécuter, on cherchera à améliorer son efficacité, par l'ordre des clauses et des littéraux, et par l'utilisation de coupes.

V - UTILISATION DE LA COUPE

Il est possible d'empêcher des retour-arrière au moyen d'un prédicat prédéfini *coupe* (*cut*), noté "`!`". Si on a une telle coupe dans la clause suivante

$p :- n_1, n_2, \ldots , !, \ldots , n_p.$

tous les choix mémorisés depuis l'appel à la tête de clause jusqu'à l'exécution du `!` sont supprimés.
La coupe peut servir à interdire l'exploration de certaines branches et à améliorer l'efficacité d'un programme.

1. Exemple

Soit `f` définie par $f(x) = \begin{cases} 0 & \text{si } x<3 \\ 2 & \text{si } 3\leq x<6 \\ 4 & \text{si } x\leq 6 \end{cases}$

On définit un prédicat

$p(x, y) \Leftrightarrow y = f(x)$

par l'ensemble de clauses suivant:

```
p(X,0) :- X<3 .
p(X,2) :- X>=3 , X<6 .
p(X,4) :- X>=6 .
```

(*Remarque*: les paramètres d'entrée de `<` et `>=` doivent être instanciés quand on les résout, sinon il y a échec.)

Cet ensemble de clauses est déclaratif et équivalent à la définition mathématique de `f`. On peut changer l'ordre des clauses.

Exemple d'exécution:
Résoudre `:- p(5,Y).`
1. 5<3 ; échec
2. 3=<5 , 5<6 ; **succès** ; avec Y = **2**
3. 6=<5 ; échec

On remarque que les trois cas sont exclusifs. On peut donc s'arrêter en cas de succès. Ceci sera fait avec des coupes:

```
p(X, 0) :- X<3 , ! .
p(X, 2) :- X>=3 , X<6 , ! .
p(X, 4) :- X>=6 , ! .
```

Ce programme est encore déclaratif, et équivalent à sa définition mathématique, à condition de préciser la propriété d'exclusivité.
L'exécution devient
1. 5<3 ; échec
2. 3=<5 , 5<6 ; **succès** ; avec Y = **2**

On remarque également que, après avoir vérifié que 5 n'est pas plus petit que 3, il est inutile de vérifier que 3 est inférieur ou égal à 5. On va donc supprimer les conditions redondantes:

```
p(X, 0) :- X<3 , ! .
p(X, 2) :- X<6 , ! .
p(X, 4) :- ! .
```

Ce programme n'est plus déclaratif. Si on change l'ordre des clauses, on ne calcule plus la même fonction. De plus, les coupes sont indispensables, sinon la relation ne serait plus fonctionnelle, `p(5,Y)` donnerait `Y = 2` puis `Y = 4`

2. Autre exemple: maximum de deux nombres

max(X, Y, Z) $\Leftrightarrow$ Z est le maximum de X et Y

Le programme suivant est déclaratif:

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

Celui-ci est plus efficace, mais non déclaratif:

```
max(X, Y, X) :- Y=<X , !  .
max(X, Y, Y) .
```

3. Coupes et Logique

L'ensemble de clauses suivant est déclaratif:

```
p :- a , b .
p :- c .
```

L'ordre des clauses est indifférent et p vérifie $p \Leftrightarrow (a \wedge b) \vee c$

Les ensembles suivants sont non déclaratifs. L'ordre est important

```
p :- a , ! , b .
p :- c .
```

p vérifie $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

```
p :- c .
p :- a , ! , b .
```

p vérifie $p \Leftrightarrow c \vee (a \wedge b)$

VI - LISTES- CONCATENATION

1. Listes

On peut utiliser des listes, comparables à celles de LISP, notées `[a,b,c,d]` ou `.(a,.(b,.(c,.(d,[]))))` ou `[a|[b,c,d]]`. `[]` est la liste vide, `[b,c,d]` est la queue de la liste `[a,b,c,d]`.
L'unification de `[a,b,c,d]` et `[a|L]` donne `L = [b,c,d]`.

2. Concaténation

On peut définir la concaténation de deux listes par le programme suivant:

```
concatenation([],L,L) .
concatenation([X,L1],L2,[X,L3]) :- concatenation(L1,L2,L3)
```

Alors la demande

```
:- concatenation([a,b,c], [d,e], R) .
```

donne `R = [a,b,c,d,e]`

On a en effet:

1. échec
2. X = a, L1 = [b,c]; L2 = [d,e]; but = concatenation([b,c], [d,e], L3)
   - 2.1. échec
   - 2.2. X' = b; L1'=[c]; L3 =[b|L3']; but = concatenation([c], [d,e], L3')
     - 2.2.1. échec
     - 2.2.2. X" = c; L1" = []; L3' =[c|L3"]; but = concatenation([], [d,e], L3")

2.2.2.1. L3" = [d,e]; **succès** avec L3' = [c,d,e]
L3 = [b,c,d,e]
R = [a,b,c,d,e]

On peut aussi appeler cette fonction en donnant le résultat de la concaténation, et en cherchant quelles listes donnent ce résultat. Ainsi la demande

```
:- concatenation(P, Q, [a,b,c,d]).
```

donne

```
P = []        Q = [a,b,c,d]
P = [a]       Q = [b.c.d]
P = [a.b]     Q = [c.d]
P = [a.b.c]   Q = [d]
P = [a,b,c,d] Q = []
```

En effet, on a

1. **P = [] , Q = [a,b,c,d]; succès** ;
2. X = a ; P = [a∧L1]; L3 = [b,c,d]; but = concatenation(L1,Q,[b,c,d]);
   2.1. L1 = []; **Q = [b,c,d] succès et P = [a]**
   2.2. X' = b; L1 = [b∧L1']; L3 = [c,d]; but = concatenation(L1', Q, [c,d])
      2.2.1. L1' = []; **Q = [c,d] succès et P = [a,b]**
      2.2.2. X" = c; L1' = [c, L1"]; L3" = [d]; but = concatenation(L1", Q, [d])
         2.2.2.1. L1" = []; **Q = [d] succès et P = [a,b,c]**
         2.2.2.2. X"'= d; L1"=[d∧L1"']; L3"'=[]; but =concatenation(L1"',Q,[])
            2.2.2.2.1. L1"' = []; **Q = [] succès et P =[a,b,c,d]**
            2.2.2.2.2. échec

VII - ECHEC - NEGATION PAR ECHEC

`fail` est un prédicat prédéfini qui produit un *échec* (prédicat *toujours faux*). Il permet d'exprimer une phrase comme "Marie aime tous les animaux sauf les escargots":

```
aime(marie,X) :- escargot(X) , ! , fail .
aime(marie,X) :- animal(X) .
```

L'ordre est important et la coupe indispensable.

La *négation* , notée `not` ou `\+` est un prédicat prédéfini, défini comme:

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

On pourra exprimer la phrase précédente par:

```
aime(marie,X) :- animal(X) , not escargot(X) .
```

On peut définir ainsi un prédicat `different` qui exprime que `X` et `Y` ne peuvent pas s'unifier:

```
different(X,X) :- ! , fail .
different(X,Y) .
```

ou

```
different(X,Y) :- not X=Y .
```

ou

```
different(X,Y) :- X=Y,!,fail ; true .
```

c'est-à-dire

```
different(X,Y):- X=Y, ! , fail.
different(X,Y) .
```

où `X=Y` est vrai si `X` et `Y` peuvent être unifiés et `;` exprime une disjonction.

Attention
`not p(X)` ne veut pas dire "p(X) faux" mais "on échoue quand on essaie de déduire p(X)". Les deux propriétés ne sont équivalentes que dans le cas d'un monde *fermé*.

Exemple: Soit l'ensemble de clauses:

```
r(a) .
q(b) .
p(X) :- not r(X) .
```

La question `:- q(X) , p(X).` donne `X = b`
La question `:- p(X) , q(X).` échoue.

En effet, dans le premier cas, la résolution de `q(X)` donne `X = b`. Il faut ensuite résoudre `p(b)`, soit `not r(b).` La première clause définissant `not` donne :

`r(b) ...` soit échec.

La deuxième clause réussit.
Dans le deuxième cas, on résout d'abord `p(X)` soit `not r(X).` La première clause définissant `not` donne :

```
r(X) , ! , fail .
```

`r(X)` donne `X = a` ; on continue et `fail` échoue. Mais, cette fois, il n'y a pas de retour arrière à cause du `!`, que l'on a traversé, et la deuxième clause définissant `not` n'est pas essayée.

VIII - PREDICATS ARITHMETIQUES

Le terme `1+2` représente `+(1,2)` en notation infixe, possible en SWI-Prolog, et non le nombre `3`. En particulier, il ne peut s'unifier avec `3`.
Pour évaluer des expressions arithmétiques, on dispose de prédicats spéciaux. Ils doivent évaluer des expressions arithmétiques et échouent s'ils ne peuvent pas le faire.
Ainsi,

```
X is 1+2
```

renvoie `X=3.`
Plus généralement,

```
X is Y
```

évalue `Y` et le résultat est unifié avec `X`. Seul `Y` est évalué.
Dans les comparaisons suivantes, `X` et `Y` sont évalués.

```
X =:= Y
```

est vrai si les valeurs de `X` et `Y` sont égales. Ne pas confondre avec

```
X = Y
```

qui unifie `X` et `Y`, ni avec

```
X == Y
```

qui est vrai si `X` et `Y` sont instanciés par des termes identiques.

```
X =\= Y
```

est vrai si les valeurs de `X` et `Y` sont différentes.

```
X < Y [resp. X >Y, X =<Y, X >=Y]
```

est vrai si la valeur de `X` est < [resp. >, ≤, ≥] la valeur de `Y`.

IX - MODIFICATION DE PROGRAMMES

Il est possible, à l'intérieur d'un programme, de sélectionner des clauses, d'en ajouter, supprimer, modifier.

```
assert(C)
```
ajoute la clause C. Sa position dans la liste des clauses dépend de l'implémentation.
```
asserta(C)
```
ajoute C *par le haut*, c'est-à-dire avant la première clause du paquet correspondant.
```
assertz(C)
```
ajoute C *par le bas*, c'est-à-dire après la dernière clause du paquet correspondant.
```
clause(P,Q)
```
cherche une clause de tête P et de corps Q. (Q = true pour une clause positive).
```
retract(C)
```
supprime la première clause qui s'unifie avec C.
```
abolish(N, A)
```
supprime toutes les clauses de nom N et d'arité A.

Remarques syntaxiques:
- on n'écrit pas le . de fin de clause, soit `assert(p(a));`
- en raison des priorités des opérateurs, on devra écrire `assert((p:-q,r))` et non `assert(p:-q,r) .`

Exemple d'utilisation: suite de Fibonacci

Le programme déclaratif suivant traduit la définition mathématique:
```
fibo(0,0) .
fibo(1,1) .
fibo(N,F) :- N>1 ,  N1 is N-1, fibo(N1,F1) ,
                    N2 is N-2, fibo(N2,F2) ,
                    F is F1+F2 .
```

Remarque: la clause
```
fibo(N,F1+F2) :- N>1 , fibo(N-1,F1), fibo(N-2,F2)
```
serait incorrecte car `N-1` ne pourrait pas s'unifier avec une valeur numérique, mais seulement avec une différence.

Ce programme a les inconvénients de tout programme récursif simpliste qui calcule les valeurs de la suite un nombre catastrophique de fois.

Pour éviter cela, une solution en PROLOG consiste à écrire un programme, non déclaratif, qui mémorise les valeurs calculées en ajoutant les clauses correspondantes, soit
```
fibo(0,0) :- ! .
fibo(1,1) :- ! .
fibo(N,F) :- N1 is N-1, fibo(N1, F1) ,
             N2 is N-2, fibo(N2, F2) ,
             F is F1+F2 ,
             asserta((fibo(N, F) :- !)) .
```

Ainsi, si on demande
```
:- fibo(4,Y) .
```
Y prendra la valeur 3, après que les clauses suivantes
```
fibo(4,3) :- ! .
fibo(3,2) :- ! .
fibo(2,1) :- ! .
```
aient été ajoutées. Aucune valeur n'est calculée deux fois. Les coupes, indispensables, arrêtent la recherche quand on a trouvé la valeur mémorisée. Le test `N>1` devient inutile.
De plus, si on demande maintenant
```
:- fibo(6,Y) .
```
seules les valeurs pour 5 et 6 seront calculées, les autres étant toujours mémorisées par les clauses.

X - ENTREES - SORTIES

`see(Fic)` : lecture sur fichier `Fic ;`
`seeing(Fic)` : donne l'entrée courante dans `Fic;`
`seen` : ferme l'entrée courante;
`tell(Fic)` : sortie sur fichier `Fic ;`
`telling(Fic)` : donne la sortie courante dans `Fic;`
`told` : ferme la sortie courante;

Lectures - Ecritures

`read(X)` : lecture dans l'entrée courante d'un terme (qui doit être suivi d'un point, d'un blanc ou d'un [Entrée]) qui est unifié avec `X`
`write(X)` : écriture de `X` sur la sortie courante
`nl` : passage à la ligne
`tab(N)` : saute `N` espaces
`get0(N)` : renvoie dans `N` le code ASCII du caractère lu
si on rencontre la fin de fichier, renvoie [CTRL] Z et le fichier est fermé
`get(N)` : même chose mais pour le premier caractère non blanc
`put(N)` : écrit le caractère de code N

XI - QUELQUES AUTRES PREDICATS PREDEFINIS

`;` exprime une disjonction. Ainsi
`P :- A , (B;C) , D.` est une abréviation pour

```
P :- A , B , D.
P :- A , C , D.
```

Attention: il doit y avoir un blanc devant la parenthèse dans `P :- (A;B), C.`

`P → Q ; R .` signifie *si P alors Q sinon R* et est défini par

```
P → Q;R  :-  P,!,Q.
P → Q;R  :-  R .
```

`repeat` permet de gérer une boucle efficacement. Il est défini par

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

`X =.. Y` est vrai si `X` s'unifie avec un terme de la forme f(X1, … Xn) et `Y` avec [f, X1, …, Xn]

`call(X)` exécute la clause instantiant `X`.

Il existe de nombreux autres prédicats prédéfinis.

BIBLIOGRAPHIE

I.BRATKO, Programmation en Prolog pour l'Intelligence Artificielle, Intereditions, 1988
W.F.CLOCKSIN, C.S.MELLISH, Programming in PROLOG, Springer Verlag 1981
M.CONDILLAC, PROLOG: fondements et applications, Dunod 1986
F.GIANNESI, H.KANOUI, R.PASERO, M.VAN CANEGHEM, PROLOG , Interéditions 1985
R.KOWALSKI, Logic for problem solving, North Holland, 1979