Cours de PROLOG.

Ces cours ont été élaborés à partir des supports de madame Kassel.
Ils ont été remaniés de façon à pouvoir être consultés plus facilement.

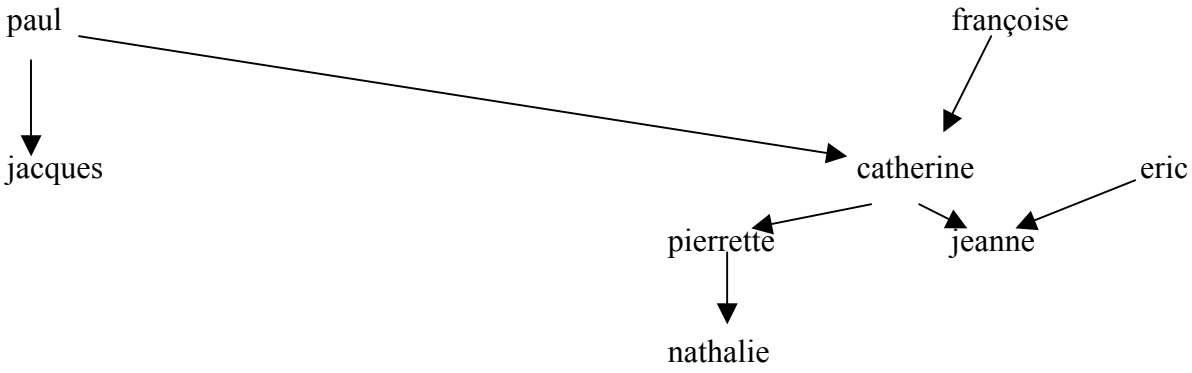
Version :
20/01/2002

Table des matières.

Chap.1 Un sous-ensemble de Prolog : les clauses de Horn sans symbole de fonction

1. Un premier programme Prolog :

Voici un arbre généalogique :



La connaissance contenue dans cet arbre se traduit en Prolog par un ensemble de faits :

```
pere(eric,jeanne).
pere(paul,jacques).
pere(paul,catherine).

mere(françoise, catherine).
mere(catherine,pierrette).
mere(catherine,jeanne).
mere(pierrette,nathalie).

femme(catherine).
femme(nathalie).
femme(jeanne).
femme(pierrette).
femme(françoise).

homme(paul).
homme(jacques).
homme(eric).
```

Le fait "pere(eric,jeanne)" représente la connaissance "Eric est le père de Jeanne" ; "pere" est un nom de relation ou prédicat, à deux arguments. Les arguments de la relation "pere" sont les

noms d'individus "eric" et "jeanne". Le fait "femme(catherine)". "femme" est un prédicat à un argument.

Il est possible de définir d'autres relations à l'aide des relations connues (père, mère, femme, homme), en utilisant des règles Prolog.

Ainsi, les règles :

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

La règle parent(X,Y):-pere(X,Y).
se lit : "Pour tout X et tout Y, si X est père de Y, alors X est parent de Y.

X et Y sont des variables logiques et représentent n'importe quel objet de l'univers.

Un programme Prolog est un ensemble des clauses ; une clause est un fait ou une règle. On note P l'ensemble des clauses d'un programme.

2. Chargement d'un programme :

On crée un fichier contenant un programme sous éditeur, on lance l'interpréteur Prolog.

?-consult(<nom_de_fichier>)
yes
?-

3. Utilisation d'un programme : les questions (on dit aussi résolution d'un but) :

Les questions peuvent être simples et composées.

- **Questions ou buts simples.**

?-parent(paul,jacques).
yes
?-

La connaissance contenue dans le programme permet de déduire que paul est parent de jacques.

?-parent(paul,eric).
no

?-

La connaissance contenue dans le programme ne permet pas de démontrer que paul est parent d'eric.

?-parent(X,jeanne).
X=eric
yes
X=catherine
yes
?-

Cette question se formule en utilisant des variables logiques : quels sont les X, tels que parent(X,jeanne) est vrai ?

- **Questions composées.**

?-parent(paul,jacques),parent(paul,catherine).
yes

La virgule représente la conjonction "et"

?-parent(paul,X),parent(X,jeanne).
X=catherine
yes

Ce sont les seules questions admises par Prolog.
Prolog n'admet pas le "ou".

4. Les faits :

Un fait ou formule atomique est la représentation d'une connaissance exprimant une relation entre des objets.

L'arité d'une relation (ou prédicat) est le nombre d'arguments d'une relation.

une relation r d'arité n est notée r/n.

Ex : La connaissance "Marie donne un bonbon à Pierre" est une relation à trois arguments, le donneur (Marie), l'objet donné (bonbon), le receveur (Pierre).

donne(marie,bonbon,pierre).

5. Les variables :

Pour tout X, aime(X,beautemps) est vrai, soit ∀Xaime(X,beautemps). On omet ∀(quelque soit).

Une variable logique représente un objet non déterminé.

Il ne faut pas confondre les variables logiques avec les variables des langages classiques, qui elles, représentent le contenu des cases mémoire.

La syntaxe des variables ne diffère de celle des atomes alphanumériques que par le fait que le premier caractère du nom d'une variable doit être une lettre majuscule ou '_' .

6. Les formules atomiques :

Définition : Un fait ou formule atomique est une expression de la forme r(t1,t2,…,tn) (n>=1) où r est un nom de relation à n arguments et t1, t2, ..., tn sont des noms d'objets ou des variables ; n est l'arité de r.

Un fait ne contenant pas d'occurrences de variables est un fait clos. Ex : femme(jeanne).

Si un fait contient des variables, alors il y a des quantificateurs universels implicites portant sur les variables. Par exemple aime(X,beautemps) est un fait non clos.

Remarque :
Une formule atomique ou fait est une expression de la forme r(t1,t2, …, tn) où r est un nom de relation d'arité n (n>=1) et les ti sont des noms d'objets ou des variables.

En réalité on peut utiliser des relations r d'arité 0. C'est un nom de proposition. Une proposition p est une formule atomique qui peut être soit vrai, soit faux.

7. Les règles :

Les règles permettent de définir des relations à l'aide d'autres relations. Une règle est une connaissance exprimable sous la forme suivante :

∀x1∀x2…∀xn (A1 et A2 et … Am→B) (1)

où les Aj et B sont des formules atomiques ; x1, x2, …xn sont les variables apparaissant dans les Ai et B.

La signification d'une telle règle est : pour tous x1, x2, …, xn, si les formules atomiques A1, A2, …, Am sont vraies, alors B est vraie. B est la conclusion de la règle ; les Ai sont les prémisses de la règle.

Un fait est une règle sans prémisse.

La notation (1) des règles est celle utilisée en logique mathématique.

En Prolog standard on note :
B:-A1,A2,…, Am.

Les quantificateurs universels sont implicites, et est remplacée par une virgule.

On emploie le mot clause pour désigner un fait ou une règle Prolog.

La conclusion d'une règle ou clause est la tête de la clause, l'ensemble des prémisses de la règle est la queue de clause.
Un fait est une clause de queue vide.

Exemples de clauses :

carnivore(X) :- humain(X).

parent(jean,paul) :-pere(jean,paul).

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

Pour tout X, tout Y et tout Z, si X est parent de Y et Y est parent de Z, alors X est un grand_parent de Z.

t:-p,q,r.

p,q et r étant les noms de propositions, cette règle énonce que si p et q et r sont simultanément vrais alors t est vraie.

Exemples de définitions de relations :

Définir la relation parent consiste à représenter la connaissance : un parent d'une personne est sa mère ou son père.

En langage de logique de premier ordre (plus puissant que Prolog) : $\forall x \forall y$ (parent(x,y) $\Leftrightarrow$(pere(x,y) ou mere(x,y)).

Cet énoncé se lit : x est parent de y ssi x est pere de y ou x est mere de y. Cette connaissance est la conjonction de deux énoncés :
1) Si x est parent de y, alors x est pere de y ou x est mere de y.
2) Si x est pere de y ou x est mere de y alors x est parent de y.

La connaissance 1) ne peut pas se représenter en Prolog, car la conclusion de la règle est un "ou" de formules atomiques.

La connaissance 2) contient un "ou" en prémisse.

Elle peut se représenter en Prolog à condition de la formuler différemment :

Si x est pere de y, alors x est parent de y.
Si x est mere de y, alors x est parent de y.

La connaissance de 2) est donc exprimable sous la forme des deux règles suivantes :

$\forall x \forall y(mere(x,y) \rightarrow parent(x,y))$ et $\forall x \forall y(pere(x,y) \rightarrow parent(x,y))$

soit en Prolog

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

Remarque :

Quoique toutes les variables d'une règle soient quantifiées universellement, nous parlerons quelques fois des variables qui ont des occurrences dans les prémisses mais pas dans la conclusion, comme si elles étaient quantifiées existentiellement à l'intérieur des prémisses.

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

peut se lire : pour tout X et tout Y, s'il existe Z, tel que parent(X,Z) et parent(Z,Y) sont vrais, alors grand_parent(X,Y) est vrai.

La logique mathématique permet de démontrer l'équivalence des deux formulations.

8. Les questions :

- **Les réponses correctes aux questions simples et closes.**

Une question simple et close est une formule atomique close, par exemple parent(paul,jacques).

La réponse correcte à une question simple et close F est "oui" si à partir de la connaissance du programme, que nous noterons P, on peut déduire que F est vrai, autrement dit si F est une conséquence logique de P ; la réponse correcte est "non" si F n'est pas une conséquence logique de P.

Ex : parent(eric,jeanne).

oui parce que pere(eric,jeanne). et parent(X,Y):-pere(X,Y);

La réponse correcte à la question parent(paul,eric) est non, parce que la connaissance du programme ne permet pas de déduire que paul est un parent d'eric.

La réponse "non" à la question F, signifie "la connaissance contenue dans le programme ne permet pas de conclure sur la valeur de vérité de F" et non que F est faux.

- **Questions simples non closes.**

Une réponse correcte à une question simple et non close F est un ensemble de valeurs associées aux variables apparaissant dans F, tel que, en remplaçant dans F les variables par ces valeurs, F soit une conséquence logique de P. (P|-F)

Ainsi {X=catherine} est une réponse correcte à la question parent(X,jeanne).

Une question simple et non close, peut admettre plusieurs réponses correctes.

Ainsi, {x=catherine} et {X=eric} sont des réponses correctes à la question parent(X,jeanne).

Répondre de manière correcte à une question simple et non close F consiste à trouver toutes les réponses correctes à F.

Définition :

Une substitution est un ensemble fini de paires de la forme Xi=ti, où Xi est une variable et ti un nom d'objet ou une variable ; Xi est différent de Xj si i est différent de j.

Un cas particulier de substitution est une substitution vide {} ; pour cette substitution, l'ensemble des couples Xi=ti est vide (il n'y a aucune variable à remplacer).

1 = {X=paul, Y=pierre} : les variables X et Y sont instanciées à paul et pierre respectivement.
2 = {X=U, Y=anne} est une substitution aussi.

Le résultat de l'application d'une substitution à une formule atomique F, noté $\overline{F}$, est la formule atomique obtenue en remplaçant chaque occurrence d'une variable Xi par ti, pour tout couple Xi=ti de $\overline{\ }$.

La formule atomique $\overline{F}$ est une instance de F.

Si on applique à F une substitution vide {}, on obtient évidemment F : toute formule atomique est donc instance d'elle-même.

Définition :

Soit F une formule atomique non close.
Si une substitution $\overline{\ }$ est telle que l'instance $\overline{F}$ de F est une conséquence logique de P, alors $\overline{\ }$ est une réponse correcte à F.

Si aucune instance de F n'est conséquence logique de P, alors "non" est la réponse correcte à F.

Répondre de manière correcte à une question F?, où F est une formule atomique non close, consiste à trouver toutes les instances $\overline{F}$ de F qui sont conséquences logiques de P.

S'il existe de telles instances $\overline{F}$, les réponses correctes sont les différentes substitutions $\overline{\ }$, sinon la réponse correcte est "non".

- **Questions ou buts composés.**

La forme la plus générale d'une question ou but Prolog est :
F1, F2, …, Fn? (n>=1), où les Fi sont des formules atomiques.

Un but ou question simple est un cas particulier de but composé (n=1); les Fi sont des sous-buts du but F1, F2, …, Fn.

Répondre de manière correcte à un but composé F1, F2, …, Fn? consiste à trouver les instances de F1, F2, …, Fn, soit F1 $\overline{\ }$, F2 $\overline{\ }$, …, Fn $\overline{\ }$, telles que chaque Fi est une conséquence logique de P.

Les questions composées sont intéressantes quand il y a des variables partagées.

parent(paul,X),parent(X,catherine)?

9. Les règles récursives :

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

La première clause n'est pas récursive, c'est la clause d'arrêt.

10. Sémantique d'un programme :

Définition :

Soit P un ensemble de clauses.
La signification ou sémantique logique du programme P est l'ensemble des formules atomiques closes F, telles que P|-F (F est une conséquence logique de P).
Cet ensemble, noté M(P), est le modèle minimal de Herbrand du programme P.

Exemples :

Si un programme P ne contient que des faits atomiques clos, alors sa sémantique M(P) est l'ensemble de ces faits.

Ainsi , sémantique du programme

```
mere(jeanne,paul).
mere(jeanne,marie).
pere(lucien,paul).
pere(lucien,marie).
```

est le programme lui-même.

En effet, les seules formules atomiques closes démontrables à partir de P, sont les formules atomiques du programme.

Si on adjoint au programme précédent les clauses :

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

Alors la sémantique du programme est l'ensemble M(P) = {mere(jeanne,paul), mere(jeanne,marie), pere(lucien,paul), pere(lucien,marie), parent(jeanne,marie), parent(jeanne,paul), parent(lucien,paul), parent(lucien,marie)}.

La sémantique d'un programme explicite les connaissances qu'un programme contient implicitement.

La sémantique d'un programme, c'est ce qu'on peut démontrer à partir des clauses du programme.

La sémantique d'un programme est indépendante de l'existence d'un interprète du langage, sa définition n'utilise que la notion de preuve.

Chap.2 Le langage Prolog pur

1. Les structures :

Prolog permet de représenter des objets contenant des informations, par une seule entité du langage, les structures, cette entité contenant toutes ces informations.

Ainsi, la date du 21 mai 1982 peut se représenter par la structure : date(21,mai,1982).

Le nom de la structure est date, ces arguments sont des valeurs du jour, du mois, de l'année ; la syntaxe des noms de structures est celle des atomes.

L'accès aux composantes de la structure date (jour, mois, année) peut se faire à l'aide des faits suivants :

```
jour(date(X,Y,Z),X).
mois(date(X,Y,Z),Y).
annee(date(X,Y,Z),Z).
```

En général, l'accès aux composantes d'une structure ne nécessite pas l'usage de faits et se fait le plus souvent de manière implicite.

Le livre "Les mots" de Jean-Paul Sartre, peut se représenter par la structure
livre(les_mots,jean_paul,sartre)

Le livre est perçu comme un objet à trois composantes : un titre, le nom et le prénom de l'auteur.

Une autre représentation possible est la structure

livre(les_mots,auteur(jean_paul,sartre))

Ici, un livre est perçu comme une structure à deux arguments, un titre et un auteur ; un auteur est une structure à deux arguments, un prénom et un nom.

Donc, les arguments d'une structure peuvent être des structures.

Dans **Prolog pur,** on dispose de trois moyens pour représenter des objets : des noms d'objets dont la syntaxe est celle des atomes ou des entiers, des noms de variables et des structures.

On appelle **termes** ces différents éléments du langage.

2. Définition des termes :

a) tout nom d'objet (atome, entier) est un terme.
b) toute variable est un terme.
c) Si s est un nom de structure à n arguments (n>=1), et si t1, t2, …, tn sont des termes, alors s(t1, t2, …, tn) est un terme.

La définition des termes est récursive. Les termes de type c) sont des structures. Un nom de structure a la syntaxe d'un atome.

Exemples :

Soit f est un nom de structure à deux arguments, alors jean, paul, X sont des termes d'après a) et b).
f(jean, paul), f(jean, X) sont des termes d'après c).
f(f(jean,paul),jean), f(f(jean,X),f(jean, paul)) sont aussi des termes d'après c).

Le nombre d'arguments d'une structure est l'arité de la structure.

3. Définition du langage Prolog pur :

a) Une formule atomique ou fait est une expression de la forme r(t1,t2, …, tn), ou r est un nom de prédicat d'arité n (n>=1) et les ti (1<=i<=n) sont des termes.

Si n=0, une formule atomique est un nom de proposition.

b) Une clause est une expression de la forme :

A. ou A:-B1, B2, …, Bn. (n>=1), où les Bi et A sont des formules atomiques.

Le Prolog pur ne se différencie du langage du chapitre 1 que par la définition des arguments possibles d'une relation ; ces arguments sont les termes, qui comprennent outre les noms d'objets et de variables, les structures.

Quoique la syntaxe des structures soit celle des formules atomiques, nous avons affaire à deux concepts très différents.

Une formule atomique est une relation entre des objets ; on peut parler de sa valeur de vérité, vraie ou fausse, contrairement aux structures qui elles représentent des objets.

La distinction entre les formules atomiques et les structures se fait en utilisant le contexte.

Par exemple, soit la clause : p(X,Y) :-h(X,v(X,Y)).

p(X,Y) est une formule atomique et non une structure, car c'est la conclusion d'une clause. De même h(X,v(X,Y)) est une formule atomique, car c'est la prémisse d'une clause ; v(X,Y) étant un argument de la formule atomique h(X,v(X,Y)), est une structure : en effet, une formule atomique ne peut être un argument d'une autre formule atomique.

On peut de la même manière faire la distinction entre :

a) un atome servant à nommer un objet.
b) un atome servant à nommer une relation.
c) un atome servant à nommer une structure.

Ex : a:-b(c,d(e,f)),h.

utilise les atomes a, b, c, d, e, f et h.

a et h sont des formules atomiques, donc des noms de propositions, b est un nom de relation binaire, c, e et f sont des noms de d'objets et d est un nom de structure d'arité 2.

Toutes les définitions du chap.1 (sémantique d'un programme, réponses correctes aux questions, etc…) restent valables dans le cas de Prolog pur.

4. Exemples de structures.

On se propose d'écrire un programme permettant de donner des informations sur un ensemble de cours. Chaque cours est caractérisé par sa matière, par le nom et prénom d'enseignant, par son horaire et la salle où le cours a lieu.

On a donc un ensemble d'assertions telles que : "le cours d'intelligence artificielle" est fait par Paul Durand et il a lieu les lundis de 9 à 11 dans le bâtiment Ampère, salle D", à représenter.

Voici 2 représentations possibles d'une telle assertion :

1) existe_cours(ia,paul,durand,lundi,9,11,ampere,d).
2) existe_cours(ia,enseignant(paul,durand),horaire(lundi,9,11),local(ampere,d)).

La deuxième est meilleure.
Prenons un ensemble de faits :

existe_cours(ia,enseignant(paul,durand),horaire(lundi,9,11),local(ampere,d)).
existe_cours(algo,enseignant(jean,dupont),horaire(mercredi,1,12),local(cochy,a)).

…

Cet ensemble de faits peut être perçu comme une base de données.

Voici quelques exemples de questions possibles :

Qui enseigne le cours d'IA ?

?-existe_cours(ia,Enseignant,Horaire,Local).

Enseignant=enseignant(paul,durand)
Horaire=horaire(lundi,9,11)
Local=local(ampere,d)
yes

Quels sont les jours où Paul Durand enseigne ?

?-existe_cours(Cours,enseignant(paul,durand),horaire(Jour,Hd,Hf),Local).

Cour=ia
Jour=lundi
Hd=9
Hf=11
Local=local(ampere,d)
yes

Les réponses aux questions ci-dessus contiennent l'information désirée, mais aussi des informations supplémentaires. Si on ne désire pas ces dernières, on peut poser les questions en utilisant des variables anonymes :

?-existe_cours(ia,Enseignant,_,_).

Enseignant=enseignant(paul,durand)
yes

?- existe_cours(_,enseignant(paul,durand),horaire(Jour,_,_),_).
Jour=lundi
yes

En général, on utilise des variables anonymes dans une clause, lorsqu'une variable a une seule occurrence dans la clause et qu'on ne s'intéresse pas à l'instanciation de cette variable.

Les entiers naturels sont habituellement nommés 0, 1, 2, …
La classe des entiers admet une construction récursive : un entier est l'entier 0 ou le successeur d'un entier.

Soit s un successeur, alors 0, s(0), s(s(0)), … représentent la suite des entiers.

Le prédicat entier(X) est vrai si l'objet X est un entier.

Définir ce prédicat revient à dire à quelle condition un objet X est un entier.

Un objet est un entier s'il est successeur d'un entier ou s'il est 0.

```
entier(0).
entier(s(Y)):-entier(Y).
```

A la question, quels sont les entiers naturels :

```
?-entier(X).
X=0
yes
X=s(0)
yes
X=s(s(0))
yes
etc…
```

Comme il y a une infinité des réponses possibles, Prolog les fournit. Le programmeur est obligé d'interrompre l'exécution du programme.
On peut définir l'addition de façon récursive :
voici la définition du prédicat(X,Y,Z) (X+Y=Z)

```
plus(0,Y,Y).
plus(s(X),Y,s(Z)):-plus(X,Y,Z).
```

```
?-plus(s(s(0)),s(0),X).
X=s(s(s(0))).
```

5. Représentation graphique des structures et des formules atomiques :

Une formule atomique r(t1,t2,…,tn) (respectivement une structure s(t1,t2,…,tn)) est représentée graphiquement par un arbre de racine r (respectivement s) ; les sous-arbres de la racine sont les représentations graphiques des termes t1, t2, …, tn.
Du point de vue syntaxique, il n'y a pas de différence entre les formules atomiques et les structures ; d'ailleurs, dans la terminologie usuelle Prolog, les formules atomiques sont qualifiées de termes. Cette terminologie est regrettable, car un terme représente un objet alors qu'une formule atomique est une relation entre des objets, ayant une valeur de vérité vraie ou fausse.

Un autre abus de langage que l'on fait consiste à identifier les concepts de noms de structure et de noms de relations avec leur syntaxe, c.à.d, des atomes.

Avec cette terminologie, le langage Prolog pur se définit comme suit :

6. Définition des termes :

a) Un atome ou un entier est un terme.

b) Une variable est un terme.

c) Si s est un atome et si t1, t2, …, tn sont des termes, alors s(t1,t2,…, tn) est un terme.

Les termes c) sont dit structures (ces structures recouvrent ce que nous avons appelé structures ainsi que les formules atomiques).

Une clause est une expression de la forme :
A. ou A :- B1, B2, …, Bn. (n>0) où A et les Bi sont des atomes ou des structures.

7. Les listes :

Une liste est une suite ordonnée de termes ; ces termes sont les éléments de la liste. Un cas particulier : la liste vide.

Ex :

L1 = []
L2 = [[]]
L3 = [jean,[paul,[pierre,eric]]]
L4 = [toto,titi]

La tête d'une liste non vide est le premier élément de la liste.

Ex :

[[a,b],c,d] : [a,b]
[X] : X

La queue d'une liste non vide est la liste obtenue en supprimant le premier élément de la liste. La queue d'une liste est toujours une liste.

Ex :

[[a,b],c,d] : [c,d]
[X] : []

8. Notation [X|Xs] des listes :

Une liste non vide est caractérisée par sa tête et sa queue. Le terme [X|Xs] représente une liste de tête X et de queue Xs.

Donc, [a,b,c,d]=[a|[b,c,d]]

9. Définition d'une liste :

```
liste([]).
liste([X|Xs]:-liste(Xs).
```

Les listes non vides sont des structures.

La syntaxe des listes non vides ne respecte pas la définition des termes Prolog. Cependant, de manière interne à l'interprète, les listes non vides sont des structures.

Une liste [X|Xs] est équivalente de point de vue logique à une structure d'arité 2 avec le premier argument X et le deuxième Xs et le nom '.'(un objet à deux composantes – sa tête et sa queue).
'.'(X,Xs).

Pour obtenir une représentation graphique d'une liste il faut l'écrire sous forme de structure.

10. Traitement de listes :

Comme nous disposons d'une représentation des listes ([X|Xs]) de nature récursive, on peut programmer de manière récursive les relations sur les listes.

- **La relation élément :**

element(X,Xs) qui signifie que X est un élément de Xs.

Formulation : X est élément de [Y|Ys] si X est le premier élément de la liste, soit X=Y, ou si X est élément de la queue de la liste, soit element(X,Ys).

Cette connaissance peut s'exprimer par les clauses :

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

Question : element(a,[b,d,a]).
yes

- **Concaténation :**

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

- **Le dernier élément :**

dernier([X],X).
dernier([X,Y|Zs],T) :- dernier([Y|Zs],T).

- **Inverser la liste :**

inverse([],[]).
inverse([X|Xs],Zs):-inverse(Xs,Ys),concatener(Ys,[X],Zs).

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

- **Longueur de liste**

longueur([],0).
longueur([X|Xs],s(N)):-longueur(Xs,N).

- **Suppression de la liste :**

Le programme suivant définit la relation supprimer(X,Xs,Ys), où Ys est la liste obtenue à partir de Xs en supprimant toutes les occurrences de X dans Xs.

supprimer(X,[],[]).
supprimer(X,[X|Xs],Ys):-supprimer(X,Xs,Ys).
supprimer(X,[Y|Ys],[Y|Zs]):-differents(X,Y),supprimer(X,Ys,Zs).

Chap.3 Sémantique opérationnelle de Prolog pur.

La sémantique opérationnelle de Prolog est la manière dont Prolog procède pour trouver les réponses à un but.

1. Importance :

Un progr. peut être logiquement correct, mais cependant l'interprète Prolog peut boucler lors de la résolution d'un but ; la connaissance de la sémantique opérationnelle permet de comprendre les raisons d'un tel bouclage.

On ne peut comprendre et utiliser les prédicats prédéfinis que si l'on connaît le mode de fonctionnement d'un interprète Prolog.

2. Interprétation procédurale des clauses, unification :

Soit la clause :

(1) **heureux(paul) :-fortune(paul,grande),sante(paul,bonne).**

Connaissance correspondante : "**Si la fortune de Paul est grande et si sa santé est bonne, alors Paul est heureux**".

L'énoncé de cette connaissance est la lecture déclarative de clause.

L'interprétation procédurale d'une clause est la façon dont la clause va être utilisée pour faire des démonstrations.

Dans Prolog, la clause (1) ne sera utilisée que si l'on cherche à démontrer le but heureux(paul).

3. Interprétation procédurale de la clause :

"pour démontrer **heureux(paul)**, je démontre le but **fortune(paul,grande)**, puis le but **sante(paul,bonne)**."

L'ordre choisi pour résoudre les buts **fortune(paul,grande)** et **sante(paul,bonne)** est leur ordre d'apparition dans la clause.

Prolog ne cherchera à résoudre le but **sante(paul,bonne)** qu'après avoir résolu le but **fortune(paul,grande)**.

Donc, le but **heureux(paul)** est réduit au but composé **fortune(paul,grande), sante(paul,bonne)**.

```
procedure démontrer(heureux(paul));
begin
        call démontrer (fortune(paul,grande);
        call démontrer (sante(paul,bonne);
end;
```

Les faits permettent de démontrer un but de manière immédiate.

Ex :
Le but "**fortune(jean,petite)**" est réduit à la clause vide, notée **{}**, par appel de la clause "**fortune(jean,petite)**".

Dans le cas le plus fréquent de clauses ayant des occurrences de variables, l'interprétation procédurale d'une clause nécessite l'utilisation de l'opération d'unification.

L'unification de deux termes **T1** et **T2** consiste à instancier les variables de **T1** et **T2**, de façon à faire coïncider **T1** et **T2**.

Par ex., pour unifier **parent(X,paul)** et **parent(jean,Y)**, on instancie X à jean et Y à paul ; les deux termes sont alors instanciés à **parent(jean,paul)**.

On dit que la substitution ~ = {X=jean, Y=paul} unifie parent(X,paul) et parent(jean,Y).

Dans les preuves Prolog, une clause n'est utilisée que pour résoudre les buts unifiables avec la tête de la clause.

Ex :
Soit la clause (2) **parent(X,Y):-mere(X,Y).** et le but **parent(paul,Qui).**

Pour pouvoir utiliser la clause (2) afin de résoudre le but **parent(paul,Qui)**, il faut unifier **parent(X,Y)** et **parent(paul,Qui)**. On applique ensuite la substitution ~ à toutes les variables de la clause (2). :
(2 bis) **parent(paul,Qui):-mere(paul,Qui)**.

La clause (2 bis) permet alors de réduire le but **parent(paul,Qui)** au but **mere(paul,Qui)**.

La clause (2 bis) est dite instance de la clause (2), puisqu'elle s'obtient en appliquant la substitution ~ **= {X=paul, Y=Qui}** aux variables de la clause.

Une instance d'une clause est vraie puisque la clause de départ est vraie quelles que soient les valeurs des variables X, Y.

4. Une autre façon de voir les choses :

```
procedure démontrer(parent(X,Y));
begin
        call démontrer(mere(X,Y));
end;
```

Pour démontrer le but **parent(paul,Qui)**, il y a appel de la procédure **démontrer (parent(X,Y))** ; "le passage des paramètres" se fait en utilisant l'opération d'unification.

En résumé, dans le cas de clauses avec variables, pour réduire un but **G** à l'aide d'une clause **A:-B1, B2, …, Bn**, il faut unifier le but **G** et **A**, la tête de la clause, à l'aide d'une substitution ~ ; après unification, le but **G** est réduit au but **(B1, B2,…, Bn)** ~.

5. Exemples :

```
1)
i.
b.
d.
h.
v.
a:-b,c,d.
a:-h,j.
c:-u,v.
u:-h,i.
```

Etant donné ce programme, on cherche à résoudre le but "a".

Pour démontrer un but, il faut le réduire progressivement à la clause vide à l'aide d'une suite de réductions.

L'ensemble des buts à réduire au départ est "a", G1=a.

```
G1=a   /* a est réduite à b, c, d grâce à a:-b,c,d. */
G2=b,c,d        /* b est réduite à la clause vide grâce à b. */
G3=c,d
G4=u,v,d
G5=h,i,v,d
G6=i,v,d
G7=v,d
G8=d
G9={}
```

Dans le cas des clauses avec variables, la modification à apporter aux preuves Prolog concerne l'utilisation de l'opération d'unification.

but : parent(paul,Qui)

programme :

```
pere(paul,jean).          /*clause 1*/

parent(X,Y):-mere(X,Y).       /*clause 2*/
parent(X,Y):-pere(X,Y).                /*clause 3*/
```

G1=parent(paul,Qui)
{clause 3. ~1={X=paul, Y=Qui}}

G2=pere(paul,Qui)

{clause 1. ~2={Qui=jean}}

G3={}

Par rapport au cas des propositions, une fois la preuve Prolog terminée, il faut chercher l'instanciation de la variable Qui, pour connaître la réponse obtenue à l'aide de cette preuve.

6. Recherche des preuves : le retour-arrière :

Dans une preuve Prolog, pour résoudre un but simple G, on utilise une clause C de tête unifiable à G.

Plusieurs clauses peuvent être unifiables avec le but G.

Ainsi, le but parent(paul,Qui) s'unifie à la clause 2 et à la clause 3.

En réalité, il est nécessaire d'utiliser toutes les clauses susceptibles de réduire un but pour les raisons suivantes :

afin d'obtenir toutes les réponses correctes à une question.

Si on se contente de choisir une clause quelconque pour réduire un but, on n'obtient pas toujours une preuve Prolog. Ainsi, dans le cas du programme précédent, si on choisit de réduire **G1** avec la clause **2**, le nouveau but **G2** est **mere(paul,Qui)** et on aboutit à un but qui ne peut être réduit. Il est nécessaire de pouvoir essayer aussi la clause **3** afin d'aboutir à une solution.

Donc, Prolog essaye toutes les clauses suivant l'ordre de leur apparition dans le programme en utilisant le mécanisme du retour-arrière (backtracking).

7. Algorithme d'unification :

Unificateur le plus général

- **Définition :**

Deux termes A et B sont unifiables, s'il existe une substitution ~, telle que **A~ = B~**. Une telle substitution ~ est un unificateur de A et B.

De manière plus intuitive, un unificateur ~ de A et B instancie les variables de A et B de façon à les rendre identiques.

Si **A=point(X,Y,12)** et **B=point(U,13,V)**, alors **~1={X=U,Y=13,V=12}** est un unificateur de A et B, puisque **A~1 = B~1=point(U,13,12)**.

De même **~2={X=15,Y=13,V=12}** est un autre unificateur de A et B, puisque **A~2=B~2=point(15,13,12)**.

Lorsque deux termes sont unifiables, ils admettent plusieurs unificateurs.

Il est normal de s'intéresser aux unificateurs qui donnent pour instance commune "l'expression la plus générale" possible.

Dans l'exemple ci-dessus, l'unificateur ~1 est préférable à ~2, le terme **point(U,13,12)** étant plus général que **point(15,13,12)**.

- **Définition:**

Un terme A est une instance du terme B, s'il existe une substitution ~ telle que A=B~.

A est plus général que B, si B est une instance de A.

A et B sont des variantes, si A est une instance de B et B est une instance de A.

Les variantes ne diffèrent pas que par le nom des variables utilisées.

- **Définition :**

~ est un unificateur le plus général (m.g.u. – most general unifier) de deux termes A et B, si :

a) ~ est un unificateur de A et B
b) pour tout unificateur ~~de A et B A~ est plus général que A~~.

8. Théorème :

Si deux termes A et B sont unifiables, alors ils ont des m.g.u.
De plus, si ~1 et ~2 sont deux m.g.u de A et B, alors A~1 et A~2 sont des variantes.

- **Un algorithme d'unification.**

Il détermine si deux termes sont unifiables et, si oui, calcule un m.g.u.

- **Algo de Robinson.**

Il y a 6 possibilités :

1) atome, atome
2) atome, variable
3) atome, structure
4) variable, variable
5) variable, structure
6) structure, structure

L' opération d'unification dans les cas 1) à 5) est immédiate :

1) Deux atomes sont unifiables ssi ils sont identiques ; un m.g.u. est la substitution vide.
2) Un atome b et un variable X sont unifiables ; un m.g.u. est ~={X=b}.
3) Un atome et une structure ne sont pas unifiables
4) Deux variables sont unifiables. Si elles sont identiques, par exemple X et X, un m.g.u. est la substitution vide. Deux variables de noms différents X et Y sont unifiables ; les m.g.u. possibles sont ~1{X=Y} ou ~2={Y=X}
5) Soit une variable X et une structure f(T1,T2,…,Tn) à unifier. Si la variable X apparaît dans la structure f(T1,T2,…,Tn), alors les deux termes ne sont pas unifiables. Dans le cas contraire, les deux termes sont unifiables et un m.g.u est ~={X=f(T1,T2,…,Tn)}

6) Pour que deux structures soient unifiables, il est nécessaire qu'elles aient le même nom et soient de même arité.

Soient donc les termes f(T1,T2,…,Tn) et f(S1,S2,…,Sn) à unifier.

L'unification de deux termes A et B peut être vue comme la résolution de l'équation A=B.

La résolution de cette équation est immédiate dans les cas de 1) à 5).

Dans le cas 6), la résolution de A=B, soit f(T1,…Tn)=f(S1,…Sn), se ramène à la résolution d'un système d'équations :

T1=S1

T2=S2
……..
Tn=Sn

Pour résoudre ce système d'équations, on commence par résoudre une des équations Ti=Si, par exemple T1 et S1.

Si T1 et S1 ne sont pas unifiables, les termes de départ ne le sont pas non plus ; si T1 et S1 sont unifiables, de m.g.u ~1, il faut appliquer cette substitution aux autres termes Ti et Si, puis essayer de résoudre une nouvelle équation Ti=Si.

On procède ainsi jusqu'à épuisement de tous les couples (Ti,Si).

Ex :

1) Pour unifier les structures r(paul,Y) et r(Y,jean), il faut résoudre le système d'équations :
paul=Y
Y=jean
On unifie paul et Y grâce à la substitution ~1 = {Y=paul}.
On applique cette substitution à l'équation Y=jean, soit :
paul=jean. paul et jean n'étant pas unifiables, les termes r(paul,Y) et r(Y,jean) ne sont pas unifiables.

2) Soit à unifier f(g(1),Y) et f(X,g(X)), d'où
g(1)=X
Y=g(X)
X et g(1) sont unifiables de m.g.u ~1 = {X=g(1)}.
On applique ~1 à l'équation Y=g(X) :
Y=g(g(1))
Y et g(g(1)) sont unifiables de m.g.u ~2={Y=g(g(1))}, donc les termes f(g(1),Y) et f(X,g(X)) sont unifiables, de m.g.u ~={X=g(1), Y=g(g(1))}.

Remarque :
On ne peut pas unifier une variable X à une structure la contenant ; par ex. X et pere(X).

9. Les preuves Prolog :

Rappel : En logique de proposition , si **G1 = b1, b2, …, bp** est le but à résoudre et si **C1 = (b1 :- a1, a2, …, aq)** et une clause du programme, alors cette clause permet de réduire **b1** (il y a appel de la clause **C1** par le but **b1**) et le nouveau but à résoudre est **G2 = a1, a2, …, aq, b2, …, bp** ; **G2** est dit résolvante du but **G1** et de la clause **C1**.

La modification dans le cas avec variables concerne l'utilisation de l'unification.

- **Déf :**

Soit **G = G1, G2, …, Gn** un but et **C=(A:-B1, B2, …, Bm)** une clause.
On suppose que **A** et **G1** sont unifiables, de m.g.u ~.

On appelle résolvante de **G** et **C**, le but composé : **B1~, B2~, …, Bm~, G2~, …, Gn~.**

La résolvante d'un but **G** et d'une clause **C** s'obtient en remplaçant le sous-but le plus à gauche dans le but **G**, par la queue de la clause **C** et en appliquant ensuite le m.g.u ~.

Le calcul de la résolvante de **G** et **C** correspond à la réduction de **G1** à l'aide de la clause **C**.

Chaque étape d'une preuve Prolog est le calcul d'une résolvante d'un but et d'une clause.

- **Remarque :**

La clause **C** choisie pour réduire un but **G**, ne doit pas avoir de variables communes avec le but, afin d'éviter des conflits de noms de variables.

La clause **entier(s(X)) :-entier(X)** ne permet pas de réduire le but **G = entier(X)** puisque **entier(s(X))** et **entier(X)** ne sont pas unifiables.

Cependant si on renomme les variables de la clause, soit

entier(s(Y)):-entier(Y).

le but **entier(X)** est réduit à **entier(Y)**.

Une variante d'une clause **C** est une clause **C'**, obtenue en renommant les variables de la clause **C**.

- **Ex :**

Une variante de la clause **grand_parent(X,Y) :- parent(X,Z)**, **parent(Z,Y)** est clause **grand_parent(X1,X2) :- parent(X1,X3), parent(X3,X2).**

Revenons au programme 1 :

1) grand_parent(X,Y) :- parent(X,Z), parent(Z,Y).
2) parent(X,Y):-pere(X,Y).
3) parent(X,Y):-mere(X,Y).
4) mere(françoise, catherine).
5) mere(catherine,pierrette).
6) mere(catherine,jeanne).
7) mere(pierrette,nathalie).
8) pere(eric,jeanne).
9) pere(paul,jacques).
10) pere(paul,catherine).

G1 = grand_parent(françoise,X).

{grand_parent(X1,X2) :- parent(X1,X3), parent(X3,X2).
~1 = {X1=françoise, X2=X}}

G2 = parent(françoise,X3),parent(X3,X)

{parent(X4,X5):-mere(X4,X5).

~2={X4=françoise, X5=X3}}

G3=mere(françoise,X3),parent(X3,X)

{mere(françoise,catherine).
~3={X3=catherine}}

G4=parent(catherine,X)

{parent(X6,X7):-mere(X6,X7).
~4={X6=catherine, X=X7}}

G5=mere(catherine,X)

{mere(catherine,pierrette).
~5={X=pierrette}}

G6={}

Pour résoudre un but, on construit une suite de buts : **G1, G2, …, Gn** ; chaque but (sauf le premier) est la résolvante du but précédent et d'une variante d'une clause du programme.

10. Théorème :

Pour tout programme P et pour tout but G, si la substitution ~ est la réponse correcte au but G, alors il existe une preuve Prolog de G avec ~ comme réponse associée.

Pour obtenir toutes les réponses correctes à une question, il suffit donc de construire toutes les preuves Prolog d'un but : s'il n'existe pas de telles preuves, la réponse correcte est non, sinon les réponses associées à ces preuves sont les réponses correctes à la question posée.

11. Un algorithme non déterministe de recherche des preuves :

Dans une preuve Prolog, si **G = G1, G2, …, Gn** est le but ou résolvante courante, il faut réduire **G1** à l'aide d'une clause unifiable à **G1**.

Plusieurs clauses peuvent être unifiables à **G1**, il est nécessaire de les essayer toutes, afin d'obtenir toutes les preuves Prolog et parce qu'un choix arbitraire d'une clause unifiable à **G1** n'aboutit pas toujours à une preuve.

Algo.

Soit R la résolvante courante initialisée au but initial G.

Tant que R <> {} Faire
début
Soit L le but le plus à gauche dans R ;
Choisir une (une variante d'une) clause C : (A :- B1, B2, …, Bn) (n >= 0) telle que L et A soient unifiables ;
R := résolvante de R et de C;
fin

12. Le retour-arrière :

Prolog essaye toutes les clauses unifiables à un but, suivant leur ordre d'apparition dans le programme, en utilisant le mécanisme de retour-arrière.

- **Ex :**

```
enseigne(paul,algo).   /*clause 1*/
enseigne(paul,ia).     /*clause 2*/
enseigne(pierre,ia).   /*clause 3*/
enseigne(pierre,reseaux).      /*clause 4*/
```

On cherche à savoir quelles sont les matières que paul et pierre enseignent en commun ; le but à résoudre est

G1 = enseigne(paul,X),enseigne(pierre,X).

Prolog réduit le but le plus à gauche enseigne(paul,X), en utilisant la clause 1.

G1 = enseigne(paul,X),enseigne(pierre,X). /*clause 1*/

{enseigne(paul,algo).
~1={X=algo}

G2=enseigne(pierre,algo).

Prolog garde l'information que la clause 1 a été essayée pour réduire le but G1.

Les clauses restant à essayer pour réduire enseigne(paul,X) sont les clauses 2, 3 et 4.

Pour réduire G2, on cherche à unifier enseigne(pierre, algo) avec une clause enseigne en partant de la premiere clause enseigne du programme. G2 n'est unifiable à aucune de ces clauses. Alors le but G2 est un echec.

Après un echec , Prolog va essayer les clauses non utilisées antérieurement en utilisant le mécanisme du retour-arrière.

G1 = enseigne(paul,X),enseigne(pierre,X). /*clause 2*/

{enseigne(paul,ia).
~2={X=ia}

G2=enseigne(pierre,ia). /*clause 3*/

{enseigne(pierre,ia).
~3={}

G3={}

- **Ex. complet :**

étant donné le programme 1, il faut résoudre le but grand_parent(françoise,X).

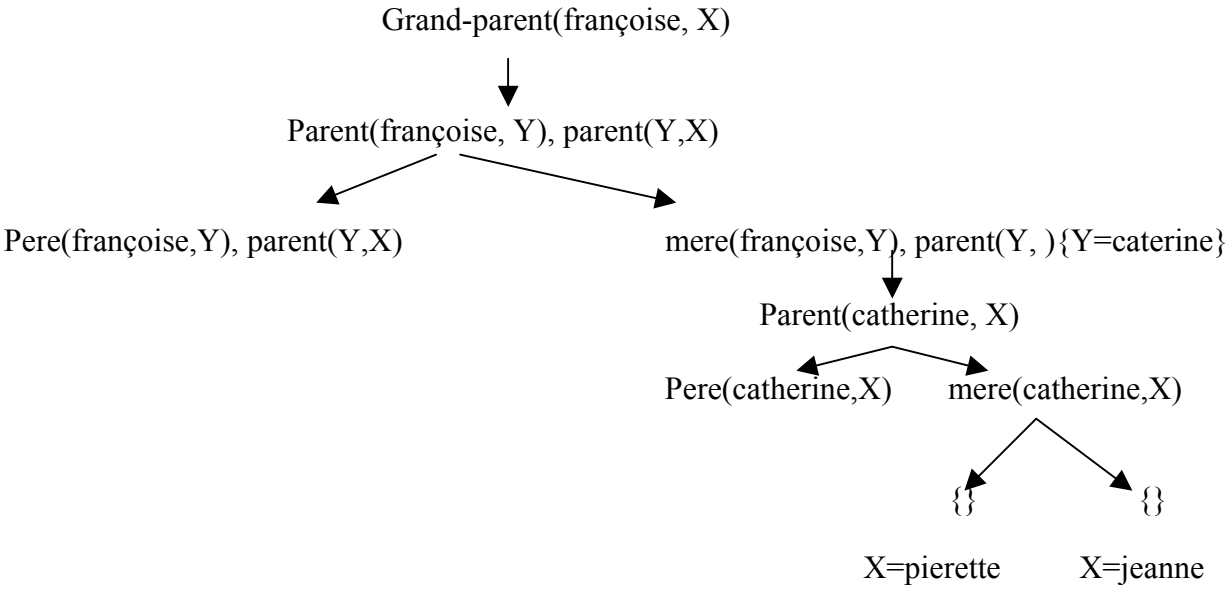
G1 = grand_parent(françoise,X) {1}
G2 = parent(françoise,X1),parent(X1,X) {2}
G3=pere(françoise,X1), parent(X1,X)
échec
retour à G2 :
G2 = parent(françoise,X1),parent(X1,X) {3}
G3=mere(françoise,X1), parent(X1,X) {4}

G4=parent(catherine,X) {2}
G5=pere(catherine,X)
échec
retour à G4 :
G4=parent(catherine,X) {3}
G5=mere(catherine,X) {5}
G6={}
succès X= pierrette
retour à G5 :
G5=mere(catherine,X) {6}
G6={}
succès X=jeanne
retour à G5 :échec;
retour à G4 :échec;
retour à G3 :échec;
retour à G2 :échec;
retour à G1 :échec;

Il existe un prédicat prédéfini "trace" qui permet d'afficher la suite des buts à réduire.

13. Arbre Prolog d'un but

L'arbre Prolog du but grand_parent(françoise,X).



La racine de l'arbre est le but à résoudre.

Pour calculer les fils d'un nœud, on cherche toutes les clauses unifiables avec le but à réduire, le but le plus à gauche dans le nœud ; les fils sont alors les résolvantes de ces clauses avec le but représenté par le nœud.
On essaye ainsi, de façon parallèle, toutes les clauses permettant de réduire un but.

Les branches sont des dérivations terminées.

Une dérivation, à partir d'un programme P et d'un but B1, est une suite de buts B1, B2, …, Bp telle que chaque but (sauf le premier) est une résolvante du but précédent avec une clause du programme.

Une dérivation B1, B2, Bp est une dérivation terminée si le but Bp est la clause vide ou est un but en échec.

Un but A1, A2, …, Am est en échec si A1 n'est unifiable à aucune tête de clause.

Une dérivation terminée B1, B2, …, Bp est soit une preuve si Bp={}, soit une dérivation terminée en échec.

Toute preuve Prolog correspond à l'exécution de l'algo décrit, mais la réciproque est fausse.

Chaque branche de l'arbre Prolog d'un but représente une exécution possible de l'algo.

Une exécution de l'algo construit soit une dérivation terminée (une preuve ou une dérivation en échec), soit une dérivation infinie.

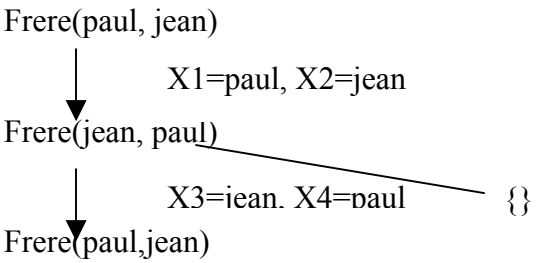
L'arbre sera infini ssi il existe une exécution de l'algo qui construit une dérivation infinie.

Ex.
```
frere(X,Y):-frere(Y,X).
frere(jean,paul).

but  : frere(paul,jean).

G1=frere(paul,jean)   {1}
G2 = frere(jean,paul) {1}
G3= frere(paul,jean)  {1}
G4= frere(jean,paul)  {1}
etc…
```

La dérivation infinie peut se lire : le but frere(paul,jean) se réduit à frere(jean,paul) qui se réduit à frere(paul,jean) qui se réduit à frere(jean,paul) etc. …



A la question frere(jean,paul). le programme : frere(X,Y):-frere(Y,X).
f frere(jean,paul).

ne répond pas et au bout d'un certain temps affiche un message de dépassement de capacité. En effet l'arbre est infini et la première dérivation qu'il construit est la dérivation infinie : (frere(jean,paul) → frere(paul,jean) → frere(jean,paul) etc.

Par contre, le programme : frere(jean,paul).
frere(X,Y):-frere(Y,X).

trouve la réponse "yes"

14. Résumé :

On obtient toutes les réponses correctes à une question F en construisant un arbre Prolog de F.

La racine de l'arbre est F.

Les fils d'un nœud sont les résolvantes du but représenté par le nœud avec toutes les clauses unifiables avec le but le plus à gauche dans le nœud.

S'il n'existe pas de telle résolvante, le nœud est un échec et est une feuille de l'arbre.

Un interprète Prolog construit l'arbre Prolog d'un but en utilisant un parcours en profondeur d'abord.

15. Semi-décidabilité :

Il n'existe pas d'algo capable de calculer toutes les réponses correctes à une question ; le calcul des réponses correctes à une question est un problème non décidable.

Même si on cherche à calculer une seule réponse correcte à une question F, ce problème n'est pas non plus décidable.

Il est semi-décidable.
C.à.d qu'il existe un algo tel que :

a) si le but F a des solutions, l'algo en trouve une en un temps fini.
b) si le but F n'a pas de solution, l'algo peut le reconnaître au bout d'un temps fini ou boucler.

La construction en largeur d'abord de l'arbre Prolog est un tel algo.

16. Réponses d'un interprète Prolog à une question (profondeur d'abord) :

Si l'arbre Prolog de F est fini, Prolog trouve toutes les réponses correctes à la question F ; si l'arbre est infini, Prolog boucle.

Toute réponse trouvée par Prolog est correcte.

1) si Prolog répond à la question F avec la substitution ~, alors Prolog a trouvé une preuve de F avec ~ comme réponse associée.
2) si Prolog répond non, alors F n'est pas conséquence logique de P. En effet, si Prolog répond non, c'est qu'il a construit tout l'arbre associé à F (en profondeur d'abord) et que cet arbre ne contient pas de preuve de F. Donc, F n'est pas une conséquence logique de P.

Le problème dans l'utilisation d'un interprète Prolog est la non terminaison ou bouclage qui peut correspondre aussi bien à un but qui devrait réussir qu'à un but qui devrait échouer.

Chap.4 Arithmétique et Syntaxe Prolog

Prolog permet l'utilisation des entiers et des réels avec leur notation usuelle : 3 6.75 1.5e-2, ainsi que les chaînes : "reines".

1. Les opérateurs :

Dans la notation utilisée pour les structures, le nom de la structure précède ses arguments : la notation préfixée.
Ce n'est pas toujours commode. Prolog permet d'utiliser d'autres notations dans le cas des structures d'arité 1 ou 2.

La notation infixée d'une structure d'arité 2 s(t1,t2), est l'expression "t1 s t2".

Pour pouvoir utiliser la notation infixée, il faut déclarer le nom s de la structure à l'aide d'un prédicat prédéfini op.
On dit que op est déclaré comme opérateur.
Le prédicat prédéfini op(P,A,N) a trois arguments :
N est le nom de la structure ou opérateur. A est l'associativité de l'opérateur. Dans le cas des structures d'arité 2, l'argument A a trois valeurs possibles :
yfx (associatif à gauche)
xfy (associatif à droite)
xfx (pas d'associativité).
P est un entier positif, dit précédence. Plus l'entier est petit, plus l'opérateur est prioritaire.

- **Ex :**

op(200,xfy,and).

Après cette déclaration, la clause **pierre and paul and marie** est acceptée et interprétée comme le terme **pierre and (paul and marie)**.
Le prédicat prédéfini display écrit sur la sortie le terme sous forme préfixée.

?- display(pierre and paul and marie).
and(pierre,and(paul,marie))

L'utilisation du mot "opérateur" peut prêter à confusion. On lui associe généralement la notion de calcul, alors qu'ici un opérateur est simplement un nom de structure qu'on s'autorise à écrire de manière infixée. Les opérateurs ne sont utilisés que pour écrire certains termes de manière plus agréable.

?-op(300,xfy,aime).

?-display(jean aime marie and paul)

aime(jean,and(marie,paul)).

Le terme "jean aime marie and paul" utilise deux opérateurs : aime et and. On commence par utiliser le plus prioritaire.

?- op(100,fy,not).
yes

not est déclaré comme opérateur unaire, préfixé, associatif.

?- op(300,yfx,or).
yes

or est déclaré moins prioritaire que le not et associatif à gauche. L'expression **a or b or c** est donc équivalente à **(a or b) or c**

?- display(a or not b and c).
or(and(not(b),c))

2. Les opérateurs prédéfinis :

Voici quelques exemples :

op(31,yfx,+).
op(31,yfx,-).
op(21,yfx,*).
op(21,yfx,/).
op(255,xfx,':-').
op(21,xfy,',').
op(21,xfy,';').

Les atomes symboliques ':-' et la virgule ',' sont aussi des opérateurs. Dans la clause **a:- b1, b2, b3.** on utilise deux opérateurs prédéfinis. Une clause est donc un terme.

Un opérateur prédéfini très utile est l'opérateur ';' (lisez ou).

la clause f:- b,(c;d), e.

représente la connaissance si **b et (d ou c) et e alors f**.
Sans l'opérateur ou, il faudrait utiliser deux clauses :
f :- b, c, e.
f :-b, d, e.

3. Arithmétique :

Une expression arithmétique est un terme Prolog infixé utilisant les opérateurs prédéfinis de l'arithmétique.

Ainsi l'expression arithm. 2.4 + 5.7 * 4 est un terme Prolog, sa notation préfixée est +(2.4, *(5.7, 4)). La représentation graphique :

Il faut comprendre qu'une expression arithm. est un terme et qu'elle n'a pas de valeur.

Le seul moyen d'évaluer une expression arithm. est d'utiliser le prédicat prédéfini **is**.

Lors de l'appel du but "X is Y", Y doit être instancié à une expression arithm. evaluable.

?- X is 2+5*8
X=42

?- 3 is 4 – 1.
yes

?- 2+3 is 1+4
non

Lors de la réduction d'un but **X is Y** Y est évalué et unifié à X. L'évaluation de Y (1 + 4) donne 5 ; mais l'atome 5 n'est pas unifiable à la structure 2+3. Il ne faut pas oublier que l'expression arithm., telle que 2+3, n'est pas une valeur mais un arbre fini et que le seul moyen de l'évaluer est de l'utiliser comme deuxième argument du prédicat is.

?- X is Y+2.
error
L'expression Y+2 n'est pas évaluable.

Prolog dispose de relations de comparaisons : >, <, =< et >=.

4. Exemples :

?- 2 < 3
yes

?- 2 < 3 + 5
erreur

5. Exemples de programmes :

- **progr.1**

```
fact(0,1).
fact(N,M) :- N>0, N1 is N-1, fact(N1,M1), M is M1*N.
```

Le mode d'utilisation envisagé pour la relation fact(X,Y) est : X est un entier, Y est une variable ou un entier (on calcule la factorielle d'un entier ou on teste sa valeur).

```
?- fact(4,X).
X=24
yes
?-fact(3,8).
no
```

Pour ce mode d'utilisation de la relation fact, l'ordre des buts dans la deuxième clause fact est l'ordre naturel d'expression de la connaissance : on regarde si l'entier N est strictement positif, si oui, on calcule N-1 et sa factorielle, puis on multiplie par N. Cet ordre des buts assure la terminaison de la résolution du but fact(N,M), lorsque N est instancié à un entier.

```
?- fact(X,6).
**error : X>0 is not evaluable **
```

- **Progr.2**

```
longueur([],0).
longueur([X|Xs],N) :- longueur(Xs,N1), N is N1 + 1.
```

Ce progr. est correcte si on cherche à calculer (ou tester) la longueur d'une vraie liste, mais il ne se termine pas si on cherche à construire une liste de longueur définie.

- **Progr.3**

```
de_long([],0).
de_long([X,Xs],N) :- N>0, N1 is N-1, de_long(Xs,N1).
```

Le mode d'utilisation est : N est instancié à un entier.

6. Egalité et inégalité :

Il existe deux opérateurs prédéfinis = et \=, dits égalité et inégalité.

X=Y : réussit si les termes X et Y sont unifiables et échoue autrement.

?- ete=ete.
yes

?- pere(jean,paul)=pere(jean,marie).
no

?-pere(jean,X)=pere(Y,marie).
X=marie Y=jean
yes

?-2+3=5.
no

En réalité le prédicat = définit le concept d'identité et non celui de l'égalité. Si le prédicat = n'était pas prédéfini, on pourrait le définir trivialement par la clause :
X=X.

X\=Y : réussit si X et Y ne sont pas unifiables, échoue si X et Y sont unifiables.

?-jean\=paul.
yes

?-pere(jean) \= pere(marie).
yes

?- X \= 3.
no

X est unifiable à 3, donc X\=3 échoue.

Une utilisation correcte de ce prédicat impose de s'assurer qu'au moment de l'appel du but X \= Y, X et Y sont instanciés à des termes clos, c.à.d, sans occurrences de variables. Si X et Y ne sont pas clos, alors ce prédicat n'a aucune signification logique.

Par ex., la réponse à la question X\=3? est non, alors que la réponse correcte à cette question est : il existe des valeurs de X différentes de 3 et il existe des valeurs de X égales à 3.

- **Progr. 4**

sœur(X,Y) :- X\=Y, femme(X), parent(Z,X), parent(Z,Y).

Si on pose la question sœur(X,Y) Prolog répond non. En effet le but X\=Y échoue puisque X et Y sont des variables.

Changeons l'ordre des buts dans la clause sœur :

sœur(X,Y):-parent(Z,X), parent(Z,Y), X\=Y.

Cette fois ci Prolog répond correctement en fournissant l'ensemble des couples X et Y telles que X est sœur de Y. En effet, cette fois-ci, X et Y sont instanciés au moment de l'appel de X\=Y et on a alors le concept d'inégalité.

Chap.5 Contrôler le retour-arrière : la coupure.

La coupure est un prédicat prédéfini permettant d'agir sur le comportement de l'interprète Prolog lors du retour-arrière.

La coupure (appelée aussi cut ou coupe-choix) permet d'écrire des programmes plus efficaces en temps calcul et un enrichissement du pouvoir d'expression du Prolog.

1. Introduction

Plusieurs clauses d'un programme peuvent être candidates à la résolution d'un but : c'est le non-déterminisme de Prolog.

Pour résoudre un but, l'interprète Prolog essaye toutes les clauses unifiables avec ce but, suivant leur ordre d'apparition dans le programme, en utilisant le mécanisme de retour-arrière. Or, il n'est pas toujours nécessaire d'essayer toutes ces clauses.

- **Ex :**

(1) inserer(X,[],[X]).
(2) inserer(X,[Y|Ys],[Y|Zs]) :- X>Y, inserer(X,Ys,Zs).
(3) inserer(X,[Y|Ys],[X,Y|Ys]) :- X =< Y.

Pour résoudre un but inserer(X,Xs,Ys), Prolog va essayer ces trois clauses suivant leur ordre d'apparition dans le programme.

Mais la clause (1) est réservée au cas où Xs est la liste vide, donc si un but inserer s'unifie avec la tête de cette clause, il est inutile d'essayer les deux autres clauses lors du retour-arrière.

Si Xs est une liste non vide, une seule des clauses (2) ou (3) permettra de résoudre le but suivant que X>Y ou non. Si le but X>Y, il faudrait dire à Prolog qu'il n'est pas nécessaire d'utiliser la clause (3) lors du retour-arrière.

- **En résumé :**

a) Si un but inserer est unifiable à la clause (1), il est inutile d'essayer les clauses (2) et (3) lors du retour-arrière. On reformule : Si un but inserer est unifiable à la clause (1), il faut supprimer les points de choix restants.
b) Si un but est unifiable à la clause (2) et que le but X>Y réussit, il est inutile d'essayer la clause (3) lors du retour-arrière (le point de choix restant peut être supprimé).

Ce type d'information peut être fourni par le programmeur, en utilisant le prédicat prédéfini, la coupure.

La coupure est un but Prolog noté !. Lorsque l'interprète doit réduire la coupure, la coupure est réduite à la clause vide (elle est donc toujours vraie), mais la satisfaction de la coupure a des effets de bord consistant dans la suppression de certains points de choix.
Ce prédicat permet de supprimer des points de choix, donc de couper une partie de l'arbre Prolog d'un but, jugée inintéressante par le programmeur.

La coupure permet de programmer certaines relations de manière plus efficace.

Le mécanisme de la coupure permet aussi d'enrichir le langage : elle permet de définir la négation par l'échec et d'exprimer des connaissances du type "si alors sinon".

2. Sémantique de la coupure

(1) inserer(X,[],[X]) :- !.
(2) inserer(X,[Y|Ys],[Y|Zs]) :- X>Y, !, inserer(X,Ys,Zs).
(3) inserer(X,[Y|Ys],[X,Y|Ys]) :- X=<Y.

La coupure de la clause (1) indique que dans la résolution d'un but inserer, si la liste Xs est vide, il est inutile d'essayer les clauses (2) et (3). Si la clause (2) réussit, il est inutile d'essayer la clause (3).

Afin de comprendre comment la présence des coupures dans le programme, induit le comportement, nous allons définir la sémantique de la coupure.

- **Exemple de programme :**

(1) a :- u, v.
(2) a :- b1, b2, !, c1, c2.
(3) a :- h, i.
(4) a :- t.

Chacune de ces clauses permet de réduire le but "a". Sans la coupure, les 4 clauses seront essayées, suivant leur ordre d'apparition, grâce au mécanisme du retour-arrière.

La présence de la coupure dans la clause (2) va modifier le comportement de l'interprète Prolog. Pour réduire le but "a", Prolog essaiera d'abord la clause (1) ; lors du retour-arrière, il essaiera la clause (2).

La réduction du but "a" avec la clause (2) entraîne que Prolog va essayer de résoudre b1, puis b2.

Si le but b1, b2 échoue, la coupure ne sera atteinte et il y aura un retour arrière afin de réduire le but "a" avec la clause (3), puis la clause (4).

Si la coupure n'est pas atteinte, le comportement de l'interprète Prolog n'est pas modifié.

Mais si b1 et b2 réussissent, le prochain but à réduire afin de résoudre "a" est la coupure.

La coupure, !, est immédiatement réduite à la clause vide, mais sa réduction entraîne que lors du retour arrière, Prolog n'essaiera pas de réduire le but "a" avec les clauses (3) et (4) : les points de choix encore disponibles sur "a" sont supprimés.

Il est peut-être possible de réduire b1 et b20 avec d'autres clauses que celles qui ont été essayées.

Mais la encore les points de choix sur les buts précédents la coupure (b1 et b2) sont supprimés.

La satisfaction de la coupure dans la clause a:-b1, b2, !, c1, c2. entraîne les effets suivants :

a) les clauses non essayées et susceptibles de réduire le but "a" ne seront pas essayées : on dit qu'on supprime les points de choix sur le prédicat de la clause appelant la coupure (le prédicat "a" dans le cas d'exemple).
b) les clauses non encore essayées et susceptibles de réduire les buts (b1 et b2) de la clause situés avant la coupure ! , ne le seront pas.
c) cependant, toutes les clauses susceptibles de réduire les buts situés après la coupure (c1 et c2), seront essayées.

De manière plus précise, la satisfaction de la coupure dans la clause
a :- b1, …, bm, !, c1, c2, …, cn.
supprime tous les choix accumulés entre l'appel de la clause et l'appel de la coupure.

- **Ex :**

G1=inserer(5,[2,3,6],U). {2}
{U = [2|U1]}

G2= 5 >2, !, inserer(5,[3,6],U1} {}

G3=!, inserer(5,[3,6],U1) {}

G4 = inserer(5,[3,6],U1) {2}
{U1=3|U2}

G5= 5 >3 , !, inserer (5, [6], U2) {}

G6= !, inserer (5, [6], U2) {}

G7=inserer(5, [6], U2) {2}
U2={[6|U3]}

G8= 5 > 6, !, inserer (5,[],U3).

Dans G8 le but 5>6 échoue ; Prolog fait un retour-arrière et retourne au but le plus récent ayant des points de choix, soit le but G7 et essaye la clause (3).

G7=inserer(5,[6],U2) {3}
{U2=[5,6]}

G8=5=<6 {}

G9={}
Le but initial a été réduit à la clause vide, une solution a été trouvée U=[2,3,5,6]

3. Déterminisme et modes d'utilisation d'une relation

```
homme(paul).
homme(jean).
homme(pierre).

singe(chita).

chat(felix).
chat(minou).

age(paul,39).
age(jean,13).
age(pierre,17).
age(chita,6).
age(felix,1).
age(minou,4).
```
(version 1)
```
jeune(X) :- homme(X), age(X,Y), Y=<21.
jeune(X) :-singe(X), age(X,Y), Y=<7.
jeune(X) :- chat(X), age(X,Y), Y=<2.
```

Supposons qu'on cherche à rendre plus déterministe la définition du prédicat jeune.

On peut raisonner comme suit : si je cherche à déterminer si un individu donné X est jeune, il n'est pas nécessaire d'essayer les trois clauses jeune.

Si X est un homme, seule la première clause est susceptible de démontrer qu'il est jeune.

de même si X est un singe, seule la deuxième clause le permettra.

La nouvelle version du programme :
(version 2)
```
jeune(X) :- homme(X), !, age(X,Y), Y=<21.
jeune(X) :- singe(X), !, age(X,Y), Y=<7.
jeune(X) :- chat(X), !, age(X,Y), Y=<2.
```

Si X est instancié, les solutions du but jeune(X) obtenues par ces deux programmes sont identiques ; mais, si X est une variable, la version 2 répond non à la question jeune(X).

Voici la trace :

G1 = jeune(X)
G2 = homme(X), !, age(X,Y), Y=<21
G3= !, age(paul,Y), Y=<21.

La satisfaction de la coupure dans G3 a pour effet de bord d'empêcher de réduire jeune(X) et homme(X) avec les autres clauses possibles ; le but age(paul, Y), Y=<21 échouant, le but jeune(X) échoue.

Si on utilise le prédicat jeune pour trouver tous les individus jeunes, alors il est nécessaire de pouvoir utiliser les trois clauses jeune afin de déterminer tous les hommes jeunes, tous les singes jeunes et tous les chats jeunes et la présence de coupures dans la version 2 rend cela impossible.

Lorsque le programmeur utilise la coupure pour rendre plus déterministe un programme, il suppose que le programme va être utilisé dans un mode bien précis.

- **Première solution d'un but.**

non_disjoints(Xs,Ys) :- element(X,Xs), element(X,Ys).

La définition de cette relation est du type generate and test, le but element(X,Xs) générant les éléments X de la liste Xs et le but element(X,Ys) testant si l'élément généré appartient à la liste Ys.

Dans le mode d'utilisation envisagé pour la relation non_disjoints(Xs,Ys) (Xs et Ys instanciées à des vraies listes), un but non_disjoints(Xs,Ys) a une seule solution : oui ou non.

Mais l'interprète Prolog après avoir trouvé un élément X commun aux deux listes Xs et Ys, utilise le retour-arrière pour en chercher d'autres et répond autant de fois oui, que les listes Xs et Ys ont des points communs.

non_disjoints([a,b,c],[d,a,b]).
yes
yes

Lorsqu'un but a une seule solution, on aimerait que Prolog s'arrête dans la construction de l'arbre Prolog dès qu'il a trouvé une solution.
?- non_disjoints(Xs,Ys) :-element(X,Xs), element(X,Ys), !.

Cette usage de la coupure, correspond à la recherche de la première solution d'un but.

Il y a aussi des situation quand Prolog boucle après avoir trouvé la première solution.

```
frere(jean,paul).
frere(X,Y):-frere(Y,X).
```

G1= frere(paul,jean).

Ce programme va boucler après avoir trouvé la solution.

Pour remédier à ce problème :

```
frere(jean,paul) :-!.
frere(X,Y):-frere(Y,X).
```

- **Pièges possibles :**

Programme correct :
version 1

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

Programme incorrect :
version 2

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

Le raisonnement est incorrect, comme le prouve la réponse de Prolog au but suivant :

```
?- max(10,5,5)
yes.
```

L'erreur de raisonnement est de supposer que l'utilisation de la deuxième clause max implique que Prolog a déjà essayé de résoudre le but X >= Y de la première clause max.

En effet, un but max(t1,t2,t3) peut ne pas être unifiable à la tête de la première clause et l'être avec celle de la seconde.

- **Rectification intéressante :**

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

Maintenant tout but unifiable à la tête de la deuxième clause max soit unifiable à la tête de la première clause max.

Dans la première clause max, on retarde l'évaluation de Z ; cet exemple est typique de l'utilisation du prédicat =.

Cependant, il est clair que le premier programme est plus lisible que le troisième, et un gain d'efficacité est négligeable.

4. La négation par l'échec

- **Le prédicat prédéfinit « not »**

Si P est un but Prolog, le but not(P) réussit si le but P échoue, et échoue si le but P a au moins une solution.

- **Ex :**

Posons quelques questions à l'ancien programme :

```
pere(eric,jeanne).
pere(paul,jacques).
pere(paul,catherine).

mere(françoise, catherine).
mere(catherine,pierrette).
mere(catherine,jeanne).
mere(pierrette,nathalie).

femme(catherine).
femme(nathalie).
femme(jeanne).
femme(pierrette).
femme(françoise).

homme(paul).
homme(jacques).
homme(eric).
```

?- not(pere(eric,jeanne).

Comme pere(eric,jeanne) est un fait du programme, le but pere(eric,jeanne) réussit, donc not(pere(eric,jeanne)) échoue.

?-not(pere(eric,paul)).
yes

Comme le but pere(eric,paul) échoue, le but not(pere(eric,paul)) réussit.

Prolog conclut qu'Eric n'est pas le père de Paul, parce que rien dans le programme ne permet de démontrer qu'Eric est le père de Paul.

Prolog utilise l'hypothèse que tout ce qui n'est pas démontrable à partir de la connaissance du programme, est faux.

On dit que la négation Prolog est la négation par l'échec.

On peut aussi utiliser la négation dans la définition d'un prédicat. Cependant, l'utilisation de la négation n'est possible que dans la queue d'une clause, le prédicat not étant un prédicat prédéfini ; on ne peut pas écrire la clause :

not(aime(jeanne,marie)).

Rajoutons un prédicat sans_enfant à notre programme :

sans_enfant(X) :femme(X), not mere(X,Y).

?-sans_enfant(pierrette).
no

?- sans_enfant(nathalie).
yes
?- sans_enfant(X).

X=nathalie
yes
X=jeanne
yes

- **Clause erronée :**

sans_enfant(X) :- not mere(X,Y), femme(X).

Pourquoi?

?- sans_enfant(X).
no

Cette réponse erronée est due au non respect de la condition d'utilisation du prédicat not.

Il y a deux sortes de variables dans le but not(mere(X,Y)) de la clause sans_enfant : la variable globale X et la variable locale Y ; X est une variable globale, car elle a des occurrences dans d'autres buts de la clause (sans_enfant(X)), contrairement à Y.

La contrainte d'utilisation du prédicat not est : lors de la résolution d'un but not(P), les variables globales doivent être instanciées à des termes clos.

Pour résoudre le but sans_enfant(X), Prolog commence par résoudre le but not(mere(X,Y)) et la variable globale X n'est instanciée à un terme clos, d'où la réponse erronée.

Par contraste, dans la clause

sans_enfants(X) :- femme(X), not mere(X,Y).

la variable globale X est toujours instanciée à un terme clos, car la résolution du but femme(X), qui instancie X, précède celle de not mere(X,Y).

Chap.6 Les prédicats extra-logiques

1. Prédicats d'écriture

write(X) : écriture sur le flot de sortie du terme X.

```
?- write(pere(jean,paul)).
pere(jean,paul)
yes

?- write(X).
_123
X=_123
yes
```

Prolog écrit le nom interne _123 de la variable X.

```
?- write('hello').
```

display(X) : écriture sur le flot de sortie du terme X en notation préfixée.

put(C) : à l'appel de ce but, C doit être instancié à un entier ; put(C) écrit sur le flot de sortie le caractère de code ASCII C.
Par exemple,

```
?- put(106), put(101), put(97), put(110).
jean
yes
```

nl : Le prédicat nl (new line) force l'impression de toutes les sorties suivantes sur la prochaine ligne du flot de sortie.

tab(X) : Le prédicat tab(X) provoque un déplacement de curseur sur l'écran vers la droite, de X caractères espaces. Ce prédicat peut se définir à l'aide du prédicat put. Pour écrire un blanc, on résout le but put(32), 32 étant le code ASCII du blanc.

```
tab(N) :- N>0, put(32), N1 is N-1, tab(N1).
tab(0).
```

Ces prédicats permettent, entre autres choses, de faire des sorties formatées.

2. Prédicats de lecture

La lecture de caractères se fait à l'aide des prédicats get0(X) et get(X). L'utilisation de ces prédicats force l'ordinateur à attendre la frappe d'un caractère.

get0(X) : Si X est une variable, get0(X) instancie X au code ASCII du caractère tapé ; si X n'est pas une variable, ce but échoe ou réussit suivant que X est ou n'est pas le code ASCII du premier caractère disponible sur le flot d'entrée ; ce caractère n'est plus disponible en lecture.

?- get0(X).
a c
X = 97
yes

get0(X) unifie X au code ASCII du premier caractère du flot d'entrée, 'a'. Ce prédicat étant déterministe seul le caractère 'a' est lu ; les caractères blanc et 'c' n'ont pas été lus.

?- get0(99).
a c
no

99 est le code ASCII du caractère 'c'. get0(99) essaye d'unifier 99 au code ASCII du premier caractère disponible et échoue. Ce prédicat étant déterministe, il n'y a pas de lecture du caractère 'c'.

get(X) : La différence avec get0(X) est que ce prédicat saute tous les caractères non-imprimables jusqu'au premier caractère imprimable. L'utilisateur est obligé de taper au moins un caractère imprimable.

read(X) : L'utilisation de ce prédicat force l'ordinateur à attendre la frappe d'un terme suivi par un point et un caractère non imprimable tel qu'espace ou retour chariot.

read(X) unifie X au prochain terme du flot d'entrée qui est consommé pour la lecture.

?-read(X), read(Y).
jean. marie.
X=jean Y=marie
yes
?- read(jean).
pierre. jean.
no

read(jean) unifie jean au premier terme du flot d'entrée, pierre, et échoue.

- **Ex :**

Version 1

```
go :- read(fin).
go :- read(X), X\=fin, fact(X,Y), write(Y), nl, go.
```

On lit un terme X. Si le terme X est fin, le prédicat go réussit et il y a "arrêt" du programme. Si le terme X n'est pas l'atome fin, on calcule sa factorielle, on l'imprime et on recommence le processus go.
Quoique sa logique soit correcte, ce programme est incorrect. En effet, le prédicat read est extra-logique : la résolution d'un but read(X), consomme définitivement un terme en lecture ; que le but read(X) échoue ou réussisse, une fois qu'un terme est lu, il est lu définitivement. Si le premier terme tapé par l'utilisateur n'est pas "fin", la première clause go est utilisée, elle échoue car le but read(fin) échoue, mais la résolution de ce but a consommé le terme en entrée. Quand Prolog utilise la deuxième clause go et appelle le prédicat read(X), le terme qui sera lu ne sera pas le premier terme tapé.

Version 2

```
go :- read(X), go(X).
go(fin).
go(X) :- X\=fin, fact(X,F), write(F), nl, go.
```

Le prédicat go lit un terme X et appelle go(X). Si le terme X est l'atome fin, on s'arrête, sinon on calcule la factorielle et on recommence le processus go.

3. Les prédicats prédéfinis « repeat » et « fail »

repeat
Bien que ce prédicat soit prédéfini, on pourrait en donner la définition suivante :

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

Ce but réussit une infinité de fois. L'intérêt de ce prédicat est de permettre de générer des solutions multiples dans le cas des prédicats prédéfinis extra-logiques et déterministes.

fail
Le prédicat fail est un prédicat qui n'est jamais démontrable. Il provoque donc un échec de la démonstration ou il figure. Il pourra être défini par fail :- 0=1.

L'utilisation conjointe de repeat, fail et du coupe-choix permet de réaliser des boucles.

- **Ex :**

```
affiche :- repeat, read(X), afficher(X), !.
afficher(fin) :- !.
```

```
afficher(X) :- display(X), fail.


go : repeat, read(X), traiter(X), !.

traiter(X) :- X= =fin, !.
traiter(X) :- fact(X,Y), write(Y), nl, fail.
```

Le prédicat = = réussit si, au moment de sa résolution, les variables X et Y sont déjà instanciés à des termes identiques.

Le prédicat = = n'est pas de nature logique , mais métalogique : dans une question telle que X= = Y, on ne cherche pas les instanciations des variables X et Y, telles que X et Y deviennent identiques, mais on regarde si l'objet X et l'objet Y sont déjà identiques.

Le prédicat X \= = Y réussit si X = = Y échoue.

4. Construction et accès aux arguments d'une structure

Les prédicats prédéfinis functor, arg et .. (cet opérateur se lit univ) permettent d'avoir accès aux arguments des structures et de construire des structures.

functor(Terme,F,N) : Terme est une structure ou un atome, F est le nom de la structure, l'entier N est son arité.

On peut utiliser le prédicat functor pour avoir accès au nom et à l'arité d'une structure :

```
?-functor(pp(jean,X,marie),F,N).
F=pp N=3
yes

?-functor([a,X,b],F,N).
F='.' N=2
yes
```

Une liste non vide est une structure de nom interne le point et ayant deux arguments, la tête et la queue de la liste.

```
?-functor(jean,F,N).
F=jean N=0
yes
```

Un atome est assimilé à une structure d'arité 0.

Une deuxième utilisation de functor permet de créer des structures de nom et d'arité donnés :

?-functor(Term,toto,3).
Term=toto(X,Y,Z)
yes

Les arguments de la structure créée sont des variables.

arg(Term,N,Arg) : Term doit être instancié à une structure et N à un entier ; ce but réussit si on peut unifier Arg au N-ième argument de la structure Term.

Le prédicat arg permet d'avoir accès aux arguments d'une structure ou d'instancier les variables des arguments :

?- arg(pere(jean,paul),2,Arg).
Arg=paul
yes
?- arg(pere(X,paul),1,jean).
X=jean
yes

X=..L : Le prédicat univ, X=..L, permet de passer d'une structure X à une liste L dont la tête est le nom de la structure et la queue est la liste des arguments de la structure :

?- foo(a,b,c) = ..L.
L=[foo,a,b,c]
yes

5. Le prédicat « call »

Le prédicat call/1 accepte un argument et le considère comme une question qu'il démontre. Si Prolog doit démontrer call(P) où P est un terme quelconque, il démontre P. Tout se passe comme si P était écrit à la place de la formule call(P).

L'intérêt de call est que son argument peut être calculé.

6. Exemple sur la base de données

Nous voulons faire un programme qui demandera à son utilisateur un nom et qui affichera les caractéristiques de la personne.

Il faudra donc lire le nom de la personne qui sera unifié à une variable Nom, puis construire la question Nom(X,Y) afin de rechercher toutes les propriétés et les afficher.

anne(age,24).
anne(profession,étudiante).

```
anne(adresse,paris).
interrogation:-write('Quel est le nom de la personne ?'),nl,read(Nom),
Question=..[Nom,Propriete,Valeur],
call(Question),
write(Propriete),write(' : '),write(Valeur),nl,fail.


interrogation :- write('Toutes les information ont été affichées'),nl.


?-interrogation.
Quel est le nom de la personne ?
:anne.
age : 24
profession : étudiante
adresse : paris
Toutes les  informations ont été affichées
```

bagof(T,Q,L) : Prolog considère Q comme une formule qu'il démontre puis il fabrique la liste L des termes T correspondant à chacune des démonstrations de Q.

setof : Prédicat similaire qui élimine les duplicata dans la liste fabriquée.

- **Ex :**

```
?- bagof(X,member([X,X],[[1,1],[2,2],4,[1,1]]),L).

X=_388
L=[1,2,1]

?- setof(X,member([X,X],[[1,1],[2,2],4,[1,1]]),L).

X=_388
L=[1,2]
```

Il y a d'autres prédicats métalogiques.

Chap.7 Les fichiers

1. Introduction

Un fichier peut être ouvert en lecture ou en écriture; Si un fichier est ouvert en écriture, il peut être en mode *write* ou *append*. (Signification habituelle)

L'ouverture d'un fichier est réalisée par le prédicat open/3.

?-open('toto.pl',write,F).

Le premier argument est le nom du fichier.

Le deuxième est le mode d'ouverture qui peut être write, append ou read.

Le troisième argument est une variable qui va recevoir un identificateur de fichier appelé un **flux** ou un **stream**.

Les prédicats utilisables avec le fichier ouvert en écriture :

write(Flux,X), nl(Flux), tab(Flux,X), put(Flux,N)

Si Flux est un identificateur de fichier ouvert en lecture, on peut utiliser les prédicats :

read(Flux,X), get(Flux,N), get0(Flux,N)

Lorsque les opérations d'entrée-sortie ont été effectuées, le fichier doit être fermé.

- **Ex :**

```
ecrire(T) :- open('toto.pl',append,Flux),
             write(Flux,T), nl(Flux),
             close(Flux).
```

2. Re-direction temporaire

Une autre manière de lire ou d'écrire dans le fichier est de rediriger les flux d'entrée-sortie standards que sont l'écran et le clavier.

Le prédicat **tell** prend en argument un nom de fichier et redirige la sortie standard vers ce fichier. Ainsi, toutes les écritures qui s'affichaient sur l'écran seront faites dans le fichier.

L'effet tell prend fin lorsque le prédicat told est utilisé.

```
ecrire(T) :-tell('toto.pl'),
            write(T), nl,
            told.
```

Le prédicat **see** prend en argument un nom de fichier et redirige le flux d'entrée de Prolog.

Toutes les lectures qui étaient faites au clavier le seront maintenant dans le fichier.

L'effet de see se termine lorsque la fin de fichier est atteinte ou lorsque le prédicat seen est utilisé.

Chap.8 Applications

1. Analyser un langage rationnel

Soit la grammaire suivante du start symbol S :

S →ε
S →l
S →l S l

l représente une lettre et ε un mot vide.
Une telle grammaire décrit les palindromes.
motS prend en argument une liste d'atomes et réussit si cette liste est un palindrome.

A chaque règle de la grammaire correspond une règle Prolog :

```
motS([]) :- !.
motS(L) :- atom(L),!.
motS(S) :- append([L1|Cs],[L1],S),atom(L1),motS(Cs).

?- motS([l,a,v,a,l]).
```

Des systèmes Prolog proposent un formalisme appelé DCF, acronyme de "Definite Clause Grammars", qui permet de générer automatiquement des analyseurs syntaxiques à partir d'une grammaire.

2. Les problèmes à transition d'états

On dispose d'un système pouvant être caractérisé par un état.

On considère deux états : un état initial et un état final.

Nous disposons de règles décrivant les changements d'états du système qui sont admissibles.

La solution du problème est la liste des changements d'états qui permettent de passer de l'état initial à l'état final.

- **Problème typique :**

celui du fermier, du loup, de la chèvre et du chou.

Tous les 4 sont sur la rive gauche d'une rivière (état initial) et doivent parvenir sur la rive droite (état final). Le fermier dispose d'une barque dans laquelle il ne peut emmener qu'un seul animal ou un seul légume à la fois tant elle est petite. Si le loup et la chèvre se retrouvent ensemble sans le fermier, le loup mange la chèvre. Si la chèvre et le chou se retrouvent ensemble sans le fermier, la chèvre mange le chou.

On représente un état par un terme etat(Fermier,Loup,Chevre,Chou) ou Fermier,Loup,Chevre,Chou sont la position g pour gauche ou la position d pour droite. L'état initial est donc etat(g,g,g,g) et l'état final etat(d,d,d,d).

```
etat_initial(etat(g,g,g,g)).
etat_final(etat(d,d,d,d)).
```

Un état est admissible si le fermier est en compagnie de la chèvre, ou si fermier est en compagnie du loup et du chou.

```
etat_possible(etat(X,_,X,_)):-!.
etat_possible(etat(X,X,_,X)).
```

Une transition est le passage de la barque d'une rive à la rive opposée. la barque contient le fermier et l'un des autres éléments. Lors d'un passage, le fermier et l'un des éléments doivent changer de rive.

```
rive_opposée(g,d) :-!.
rive_opposée(d,g).

transition(etat(P1,P2,P3,P4),etat(Q1,Q2,P3,P4)):-
rive_opposee(P1,Q1),
rive_opposee(P2,Q2).
transition(etat(P1,P2,P3,P4),etat(Q1,P2,Q3,P4)):-
rive_opposee(P1,Q1),
rive_opposee(P3,Q3).

transition(etat(P1,P2,P3,P4),etat(Q1,P2,P3,Q4)):-
rive_opposee(P1,Q1),
rive_opposee(P4,Q4).
```

Une transition possible est une transition qui conduit à un état admissible :

```
transition_possible(Etat1,Etat2) :- transition(Etat1,Etat2),etat_possible(Etat2).
```

Pour résoudre le problème il faut définir un prédicat solution/3 qui prend 3 arguments :

a) état courant
b) une liste d'états qui contient la suite des états ayant conduit de l'état initial à l'état courant
c) une variable qui sera unifiée avec la solution, c.à.d. la liste des états depuis l'état initial jusqu'à l'état final.

Il y a un cas particulier ou l'état courant est l'état final, le troisième argument doit être unifié avec la solution qui se trouve dans le deuxième argument :

```
solution(E,C,S) :- etat_final(E), !, S=C.
```

La deuxième règle décrit le cas général. E est l'état courant et C est le chemin depuis l'état initial jusqu'à état courant.

On génére tous les états accessibles depuis E à l'aide d'une formule transition_possible(E,A).

Pour chacun de ces états on appelle récursivement le prédicat solution/3 avec A comme état courant et [A|C] comme chemin courant.

Le programme partira de l'état initial puis essaiera tous les états accessibles à partir de cet état et recommençera ainsi à partir de chacun de ces états jusqu'à tomber sur l'état final si celui-ci est accessible.

Pour éviter de boucler, il faut vérifier lorsque l'on passe à un nouvel état que celui-ci n'a pas déjà rencontré. Donc, on vérifie que le nouvel état n'est pas élément du chemin courant.

```
solution(E,C,S):-transition_possible(E,A),
not member(A,C),
solution(A,[A|C],S).
```

Finalement, pour trouver la solution de notre problème, il faut partir de l'état initial.

```
resoudre(S):-etat_initial(E), solution(E,[E],S).
```