

Le langage Prolog

Didier Buchs

Université de Genève

9 avril 2018

Les langages déclaratifs

- La programmation logique et langages relationnels
- Prolog : champs d'application
- Le langage : syntaxe, modèle d'exécution, preuves
- Quelques références
- Prolog, une présentation détaillée

Langages déclaratifs

(propre du problème et non de la solution)

- La plupart des langages de programmation sont conçus pour permettre la spécification de programmes en tant que séquences ordonnées d'instructions à exécuter :
 - Les langages procéduraux / impératifs
- Mais il existe des langages où la notion d'ordre d'exécution est exclue, et où ce souci est délégué à l'implantation du langage (p.ex. SQL, Trilogy) :
 - Les langages déclaratifs (relationnels, fonctionnels)
 - Langages dont l'exécution des instructions dépend des données (langages fonctionnels "purs")
 - Les langages de programmation purement déclaratifs sont rares.
- En général on distingue la sémantique abstraite du mode opératoire (la stratégie d'évaluation)

Exemple du langage SQL

SQL (Structured Query Language) : langage d'interrogation et de mise à jour des bases de données.

```
SELECT Numetu, Nometu  
FROM ETUDIANT  
WHERE Dnaiss>='01-01-1980' AND Dnaiss<='12-31-1980'  
ORDER BY Nometu, Dnaiss ASC;
```

La base de données cache entièrement le mécanisme opérationnel sous-jacent.

Exemple du langage Trilogy

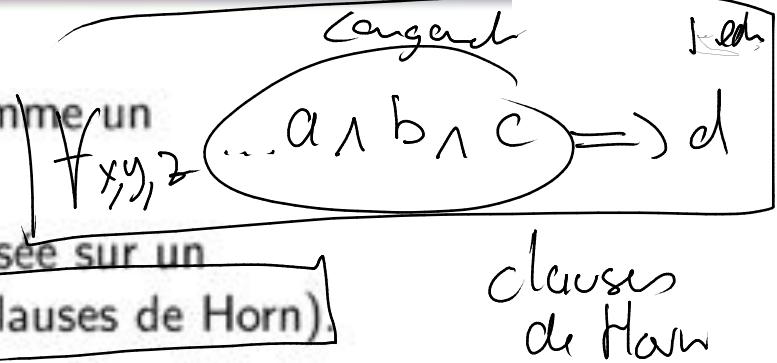
Langage de spécification de contraintes (CLP : Constraint Logic Programming) :

```
all  x::I & x>-100 & x<100 & x*x<=10
```

Il s'agit d'un langage de spécifications logiques pures (Prolog est hybride à ce point de vue).

La programmation logique

- La programmation logique peut être vue comme un sous-ensemble de l'approche relationnelle.
- La programmation logique est également basée sur un sous-ensemble de la logique des prédictats (clauses de Horn).
- Dans la programmation logique, formalisée par Kowalski dès 1972, le rôle de l'ordinateur consiste en une série contrôlée d'inférences logiques.
- Elle remonte au calcul des prédictats introduit par Frege en 1879, au principe d'unification dû à Herbrand en 1929 et au principe de résolution formulé par Robinson en 1963.
- Prolog (=PROgrammation en LOGique) en est le principal représentant

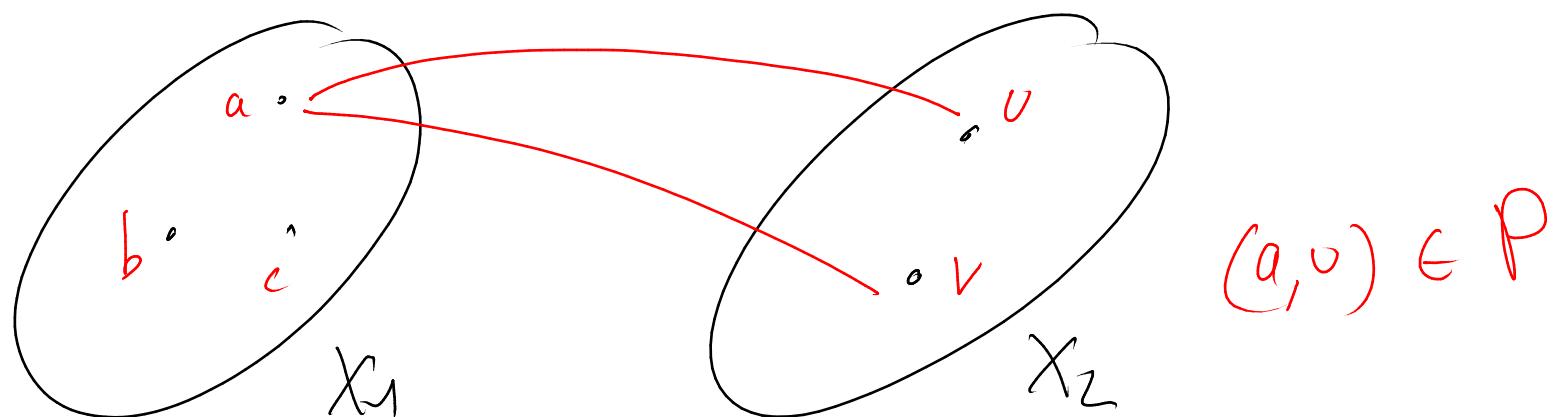


clauses
de Horn

La notion de “relation”

Définition :

une relation binaire sur les ensembles X_1 et X_2 est un ensemble de couples de la forme $\langle x_1, x_2 \rangle$,
où x_1 est élément de X_1 et x_2 est élément de X_2



Ce sont les éléments $\boxed{\langle x_1, x_2 \rangle} \in X_1 \times X_2$

$f: X_1 \rightarrow X_2 \subseteq X_1 \times X_2$ donc a est dénombrable à une unité 7/56

$$P \subseteq X_1 \times X_2$$

La notion de "relation"

- Généralisation :
 - une relation N-aire sur les ensembles $X_1 \dots X_N$ est un ensemble de tuples de la forme $\langle x_1, x_2, \dots, x_N \rangle$, où x_i est élément de X_i
- Définition :
 - une fonction de X_1 (domaine) vers X_2 (l'image) est une relation binaire qui met en relation au plus un élément de X_2 avec chaque élément de X_1
- Les langages fonctionnels manipulent des fonctions ; les langages relationnels, des relations N-aires

Haskell

Prolog , Datalog

Exemples de relations

- Exprimée comme fonction :
 - `genre(Personne)`
retourne des valeurs de Genres
 - P.ex :
`genre(julie)`
retourne
`femme`
- Exprimée comme relation :
 - `genre(Personne,Genre)`
est un prédicat
 - P.ex : `genre(jean,homme)` est vrai
 - Mais aussi : `genre(Qui,homme)` retourne vrai et `Qui=jean` ;
`Qui=luc`
- Exemple de relation ternaire : `concat(X1,X2,X3)`, la concaténation des chaînes de caractères `X1` et `X2` dans `X3`

Champs d'application de Prolog

- Prolog est conçu pour manipuler des symboles et des relations.
- Quelques domaines d'application :
 - analyse de langues informatiques ou naturelles ;
 - bases de données et systèmes experts ;
 - logique mathématique, preuve de théorèmes, résolution symbolique d'équations ;
 - travaux d'architecture, conception, plans de site, logistique ;
 - analyse biochimique et conception de médicaments.

Applications industrielles de Prolog

Quelques applications industrielles, rapportées par des vendeurs de compilateurs/interpréteurs Prolog :

www.visual-prolog.com/vipexamples/applications.htm

www.amzi.com/customers/index.htm www.amzi.com/customers/editors.htm

www.lpa.co.uk/ind_inf.htm

www.als.com/cust_stories.html

www.swi-prolog.org/

Syntaxe de Prolog (1)

(SWI Prolog)

- Programme = ensemble de relations + requête vérifiant qu'une relation existe
- 2 façons de spécifier des relations :
 - fait : indique un tuple particulier dans la relation
 - règle : indique comment déduire qu'un tuple fait partie d'une relation

Syntaxe de Prolog (2)

Syntaxe d'Edinburgh : quasi-standard

```
<variable> ::= identificateur débutant avec majuscule
<atome> ::= identificateur débutant avec minuscule
<terme> ::= <nombre>
<terme> ::= <variable>
<terme> ::= <atome>
<terme> ::= <atome> ( <liste_termes> )
<liste_termes> ::= <terme> { , <terme> }
<fait> ::= <terme> .
<règle> ::= <terme> :- <liste_termes> .
<requête> ::= <liste_termes> .
```

Les atomes et les faits

- Les < atome > désignent des symboles (donc sont des données) et sont aussi utilisés pour nommer les relations :
`<atome> (<liste termes>)`
- on dit que l'atome est ici le foncteur du terme et entre parenthèses on a ses arguments.

Exemple :

`genre(luc,homme)`

- Une relation peut s'exprimer en énumérant les tuples qu'elle contient comme des faits
- Exemple : soit $X_1 = \{luc, julie, jean\}$ et $X_2 = \{homme, femme\}$.

La relation genre s'exprime avec trois faits :

`genre(luc,homme).`

`genre(julie,femme).`

`genre(jean,homme).`

Les règles

- Les règles permettent de définir des relations à partir d'autres relations : < *terme* >: – < *listetermes* > .
- c'est une clause de Horn, qui est divisée en sa tête et ses conditions
 - Exemple 1 : *male(X)* : –*genre(X, homme)*.
Traduction logique : pour tout X tel que la relation *genre(X, homme)* existe, la relation *male(X)* existe.
 - Exemple 2 :
hermaphrodite(X) : –*genre(X, homme)*, *genre(X, femme)*.
Traduction logique : pour tout X tel que la relation *genre(X, homme)* existe et *genre(X, femme)* existe, la relation *hermaphrodite(X)* existe.

$$\forall x, \text{genre}(x, \text{hom}) \Rightarrow \text{male}(x)$$

Exercice

Définir la relation "mariage possible entre X et Y " telle que
 $mp(X, Y)$ si X et Y sont de sexe opposé.

→ écrire à l'intérieur d'une clause

```
mp(X, Y) :- genre(X, homme), genre(Y, femme).  
mp(X, Y) :- genre(X, femme), genre(Y, homme).
```

arbre
d'apparition
des clauses



on a ici un "ou" !

? $mp(X, Y)$.

X Y
luc julie
julie lucie
yann julie
jean

Modèle d'exécution de Prolog (1)

Selon Kowalski (1979) : algorithme = logique + contrôle

Où

- logique désigne les faits et règles (le quoi)
- contrôle est l'application de ces faits et règles selon un certain ordre (le comment)
- Un mécanisme de Résolution mécanique (dit de Robinson) est employé pour examiner successivement et systématiquement toutes les solutions possibles.
- Il y a un ET logique sous-entendu entre chaque condition d'une clause, et un OU inclusif entre chaque clause.

Modèle d'exécution de Prolog (2)

- Prolog tente de trouver, à l'aide de la procédure suivante, une preuve que la requête soumise est une relation qui existe :
prouver (requête) : *choix*
 - 1. Pour chaque fait et tête de règle qui **concorde** avec la requête:
Si c'est un fait: on a trouvé une preuve
Si c'est une tête de règle:
Pour chaque condition de cette règle: prouver(condition)
Si succès pour chaque condition: on a trouvé une preuve
 - 2. Si aucune concordance: Echec (=Retour arrière)
- Les faits et règles sont examinés dans leur ordre d'apparition dans le programme (stratégie linéaire définie, résolution SLD; d'autres stratégies sont envisageables)
- Si une sous-preuve échoue, la recherche revient sur ses pas (retour-arrière, backtracking) pour tenter la prochaine alternative de preuve

Arbres de preuves

À toute preuve complète correspond un arbre de preuve qui indique chaque sous-preuve effectuée pour prouver la requête.

Exemple : représentation du graphe de filiation

```
enfant(alice,jean).  
enfant(alice,luc).  
enfant(luc,julie).
```

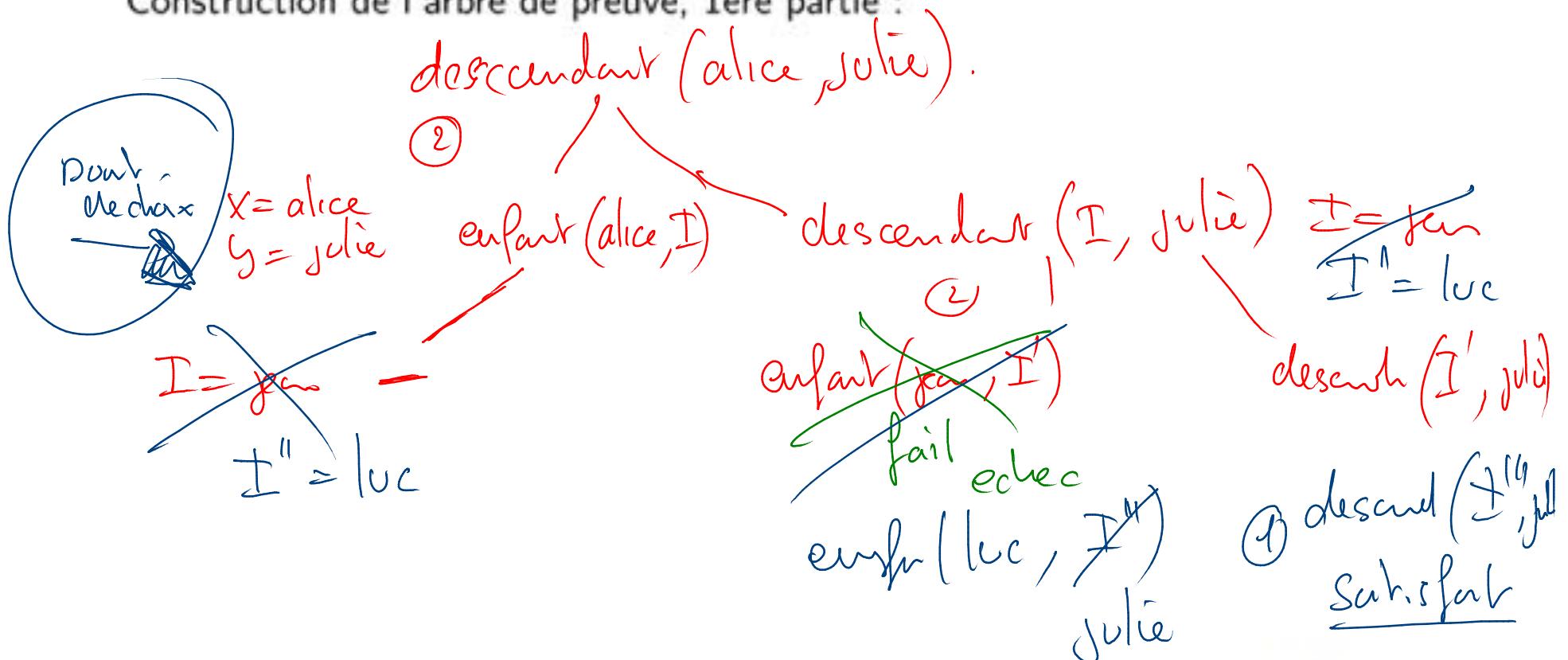
① descendant(P,P).
② descendant(X,Y) :-
 enfant(X,I),
 descendant(I,Y).

majuscule = variable
— P

Exemple d'arbre de preuves (1)

Requête : descendant(alice,julie).

Construction de l'arbre de preuve, 1ère partie :



Exemple d'arbre de preuves (2)

2ème partie de l'arbre de preuve, après retour-arrière :
La preuve a donc réussi, avec I=luc et I'=julie

- Le retour-arrière réalise le non-déterminisme de Prolog
- Définition :
 - Le non-déterminisme est la propriété qui permet à deux exécutions/évaluations successives de fournir des résultats différents
 - Le non-déterminisme se situe au niveau du choix de la tête de clause (pour les faits/règles ayant plusieurs clauses) et des liaisons de variables correspondantes.
 - Chaque emplacement où il y a non-déterminisme est considéré comme un point de choix vers lequel on reviendra en cas d'échec d'une sous-preuve.

- Les variables représentent des “inconnues” au sens mathématique
 - elles ne peuvent prendre qu'une seule valeur dans une branche d'exécution (en fait possiblement une succession d'unification)
 - une variable est soit libre, soit liée à un terme
 - Un terme est soit clos, soit libre (cf. termes)
 - une variable liée redevient libre par backtracking à son dernier point de choix
- Syntaxiquement une variable est soit nommée par un identificateur débutant avec une majuscule ou un 'underline' est une variable anonyme.
- Différent des langages impératifs : pas d'opération d'affectation donc écriture différente des algorithmes.

Structures de données en Prolog

Prolog possède une structure uniforme, formée de termes, depuis laquelle toutes les données et les programmes sont construits. Un terme est un arbre dont :

- les noeuds sont des constantes symboliques
- les feuilles sont soit des constantes, soit des variables P.ex : enfant(alice,X). Ou nbrEnfant(julie,2).

Prolog possède une opération fondamentale sur les termes : l'unification. Deux termes sont unifiables si on peut trouver des valeurs de variables (substitution) qui les rendent égaux.

- ?- $Y = julie$. la variable Y prend la valeur $julie$
- ?- $\text{enfant}(alice,X) = \text{enfant}(alice,jean)$. la variable X prend la valeur $jean$
- ?- $\text{enfant}(alice,paul) = \text{enfant}(alice,jean)$. il n'y a pas d'unification

Cf. algorithme de preuve : «concorder» = être unifiable

- C'est un langage qui dérange les habitudes mais qui est relativement simple.
- Le programmeur Prolog doit répondre aux questions suivantes par rapport au problème à résoudre :
 - Quels faits et quelles relations formelles existent-ils pour ce problème ?
 - Quelles relations doivent-elles être vérifiées pour qu'on ait une solution ?

Prolog fut choisi par les Japonais, en 1979, pour être le langage des ordinateurs dits de la cinquième génération (sans réel succès).

- Relativement facile si premier langage, sinon non !
- Avantages de Prolog
 - Séparation de la logique et du contrôle (focus sur la structure logique du problème, plutôt que sur le contrôle de l'exécution, d'où meilleure productivité du programmeur)
 - Avantageux pour le prototypage, la programmation exploratoire
 - Support transparent pour le parallélisme
- Désavantages
 - Implémentation opérationnelle pas fidèle à la sémantique déclarative (contrôle de l'ordre de résolution, modèle du monde clos, problèmes avec la négation)
 - Gestion difficile de projets importants, multi-langages
 - Efficacité réduite en-dehors des domaines d'application prévus

Quelques références

- Clocksin, W.E., and Mellish, C.S., Programming in Prolog, 2nd ed., Springer Verlag, New York, 1984.
- Colmerauer inventeur du langage
- Ivan Bratko, PROLOG Programming for Artificial Intelligence Addison-Wesley, August 2000.
- SWI-Prolog, un compilateur Prolog gratuit :
 - <http://www.swi-prolog.org>
- La FAQ du newsgroup comp.lang.prolog :
<http://www.faqs.org/faqs/prolog/>

Programmation logique en Swift

Prolog est difficile à combiner avec des langages 'classiques'

- Mécanisme opérationnel différent (backtracking).
- Structures de données incompatibles

Solution : Interface de type DSL=; structures de données différentes

LogicKit, adaptation de Swift a la programmation logique

Programme

```
let zero: Term = .lit("zero")
let x    : Term = .var("x")
let y    : Term = .var("y")
let z    : Term = .var("z")

let kb: KnowledgeBase = [
    .fact("add", zero, y, y),
    .rule("add", .fact("succ", x), y, z) {
        .fact("add", x, .fact("succ", y), z)
    }
]
```

Programme

```
add(zero, Y, Y).  
add(succ(X), Y, Z) :-  
    add(X, succ(Y), Z).  
  
?- add(X, Y, succ(succ(zero))).
```

Programme

```
let answers =  
    kb.ask(  
        .fact("add", x, y,  
              .fact("succ", .fact("succ", zero))))
```

Consulter la documentation en ligne pour plus de détails.

Présentation plus détaillée du langage Prolog

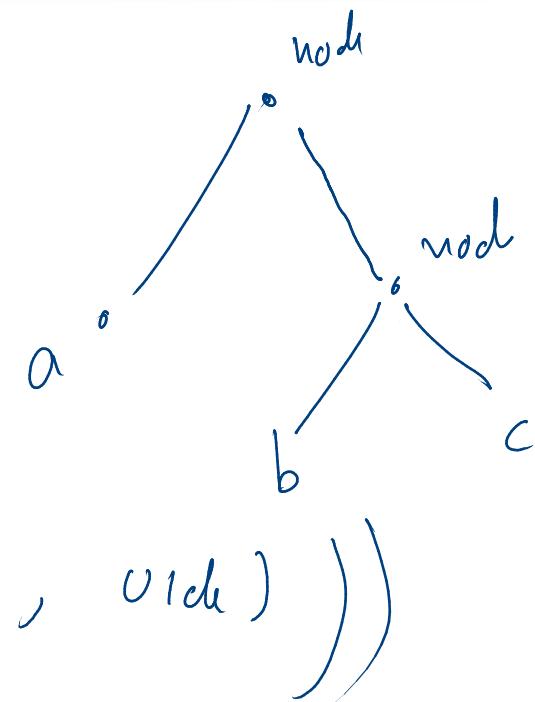
- Termes et structures de données
- Manipulation de termes
- Unification
- Base de données/faits
- Résolution et contrôle d'exécution
- Vue procédurale
- cut, failure
- Négation

- Tout peut être décrit par des termes !
 - Ensembles, tables
 - Listes, entiers
- Éventuellement il y a des relations d'équivalence entre termes !
- Mais, Prolog intègre des types de données ad-hoc :
 - Listes
 - Integer
 - Atoms
 - Chaines de caractères
- simplification de l'écriture, et augmentation de la performance, mais moins d'homogénéité
- Des prédictats prédéfinis existent pour ces structures de données

Termes

Exemple : arbre binaire node(a,node(b,c))
node(node(node(X,Y),3),node(node(a,b),c)).

[a, b, c]



```
birdie ( a, birdie ( b, birdie ( c, v1ck ) ) )
```

- Il n'y a pas de typage dans les termes donc n'importe quoi peut-être écrit sans restriction de compatibilité.
- Les nœuds de l'arbres sont appelé des foncteurs : *node(., .)*
- La taille est à priori non-limitée (sauf par les contraintes de mémoire)
- Le nombre de paramètre d'un terme est appelé son arité
 - *node/2* correspond aux foncteurs de taille 2 : *node(., .)*
 - *append/3* correspond aux foncteurs de taille 3 : *append(., ., .)*

Types de données ad-hoc : Listes

- La liste vide

`[]`

- Concaténation d'un élément H à une liste L

- `[H | L]`

- Exemples :

`[1,2,3]`

`[jean,paul,lea]`

`[node(X,Y),node(a,node(b,c)),Y]`

Unification de listes

```
?- [a,b]=[X,Y].  
X = a  
Y = b  
?- [H | L] = [1,2,3] .  
H = 1  
L = [2, 3]  
?- [node(X,Y),node(a,node(b,c)),Y]=[node(a,b)|L].  
X = a  
Y = b  
L = [node(a, node(b, c)), b]
```

Unification de listes

Quelques prédicts sur les listes

`member(X,L)`

réussi si X est dans L (prédefinis dans SWI)

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

`member(X,[Y|L]) :- member(X,L).`

Interprétation :

`member(a,[a,b,c,d]).`

Yes

Unification de listes

```
?- member(X, [a,b,c,d]).  
X = a ;  
X = b ;  
X = c ;  
X = d ;  
No
```

passe à la solution suivante

```
?- member(a, [a,b,c,a]).  
Yes
```

Unification et execution

```
?- member(a,[a,b,c,a]).
```

Yes

Question : est-ce que le prédicat est vraiment déterministe (une seule réponse !!) Question logicalement équivalente !!

```
?- member(Y,[a,b,c,a]),Y=a.
```

Y = a ;

Y = a ;

No

En fait opérationnellement NON !!

40/56

Entiers

Prolog définit des constantes telles que 1,2,3 ... Des opérateurs sont également définis +,-,*,//

Les entiers (et les flottants) sont employés d'une manière un peu différente que les autres structures de données car des expressions peuvent-être évaluées.

Actions des opérations : Vue naïve :

```
?- X = 1+ 2.
```

```
X = 1+2
```

= est le symbole de l'unification, pour activer l'application d'opérateurs il faut utiliser le prédicat is.

```
?- X is 1 + 2.
```

```
X = 3
```

Expressions vues comme des termes

En prolog un certain nombres d'opérateurs sont prédéfinis. Même les expressions de bases comme les clauses et les connecteurs logiques sont des opérateurs !

```
?- +(1,2) = X.  
X = 1+2
```

```
?- :- (a,b) = X.  
X = a:-b
```

```
?- ',,'(a,b) = X.  
X = a, b
```

Manipulation de termes

functor(A,f,3) : construit une structure de 3 éléments

arg(2,A,1) : place un élément '1' à la 2ème position des paramètres

```
?- functor(A,f,3),arg(2,A,1).
```

```
A = f(_G229, 1, _G231)
```

Prédicat réversibles :

```
?- functor(f(g(2),2,3),A,N).
```

```
A = f
```

```
N = 3
```

```
?- arg(1,f(g(2),3,4),X).
```

```
X = g(2) ;
```

Autre prédicat :

`f(g(2),3,4) =.. [f,g(2),3,4]`

Intérêt : méta-programmation, construction de tableaux à accès directs, ...

definir un predicado de concatenación
de listas;

concat / 3

concat ([1,2], [3,4,5], [1,2,3,4,5]).

concat ([], L, L).

concat ([H|X], L, [H|L]) :- concat (X, L, LL).

reverse ([1,2,3], [3,2,1]).

reverse ([], []).

reverse ([H|X], R) :- reverse (X, L), concat (L, [H], R).

Unification

Il s'agit de trouver une substitution aux variables pour rendre égaux deux termes (En prolog) :

```
node(x,b) = node(A,X).  
A = x  
X = b ;  
node(node(node(X,Y),3),node(node(a,b),c)) = T  
X = _G157  
Y = _G158  
T = node(node(node(_G157, _G158), 3), node(node(a, b), c))
```

Unification

Mais si on ajoute une précision sur X, donc aussi une solution !

```
node(node(node(X,Y),3),node(node(a,b),c))= T, X = d.  
X = d  
Y = _G158  
T = node(node(node(d, _G158), 3), node(node(a, b), c))
```

Naturellement la solution la plus générale est calculée et elle est unique !

mgu = most general unifier

Unification : particularités et problèmes

```
enfant(alice,X)= X.  
X = enfant(alice, enfant(alice, enfant(alice, enfant(alice
```

```
[X,Y,X] = [a,b,c]. Echec !
```

```
[X,Y,X] = [a,b,T].  
X = a  
Y = b  
T = a
```

Type d'un terme :

- `var(X)` X est une variable
- `nonvar(X)` X n'est pas une variable
- `atom(X)` X est un atome
- `integer(X)` X est un entier
- `float(X)` X est un flotant
- `atomic(X)` X est un atome ou un nombre
- `compound(X)` X est une structure

Comparaison de termes

X == Y X et Y sont identiques
X \== Y X et Y sont pas identiques
X =:= Y X et Y sont arithmétiquement identiques
X == Y X et Y sont non arithmétiquement identiques
X @< Y X précède Y
?- - X @< Y.
X = _G157
Y = _G158
?- a(X) @< Y.
No
?- a(X) @< a(Y).
X = _G157
Y = _G159
?- a(X) @< a(b(Y)).
X = _G157
Y = G159

L' unification en général

Il s'agit de trouver la solution au système d'équations

$U \equiv V \Leftrightarrow \exists s \text{ une substitution, t.q. } sU = sV$

Une substitution est une application des variables dans les termes :

$s = \{X = \text{node}(Z, 3); Y = b\}$

$X = \text{node}(Z, 3)$

$Y = b$

sU est l'application de la substitution au terme U fournissant un terme instancié

$s(\text{node}(\text{node}(\text{node}(X, Y), 3), \text{node}(\text{node}(a, b), c))) =$
 $\text{node}(\text{node}(\text{node}(\text{node}(Z, 3), b), 3), \text{node}(\text{node}(a, b), c))$

L' unification en général

- Combien de solution (les plus générales) ?
- Théorie unitaire = 1
 Unification de termes libres comme en Prolog
- Théorie finitaire = un nombre fini
 Unification d'ensembles :
 $\{X, Y\} = \{a, b\}$ solution ?
- Théorie infinitaire = une infinité
 Unification de fonctions surjectives !
 $\text{Odd}(X) = \text{true}$ solution ?

Unification, implémentations des différentes variantes

Unitaire et finitaire alors possibilités d'implémentation (sémantique opérationnelle)

Structure de données : chaînes de caractères

Atomes et chaînes de caractères 'alphaomega' est un atome

```
?- atom_concat('alpha', 'omega', X)
```

```
X='alphaomega'
```

```
?- append([a], [b], C).
```

```
C=[a,b];
```

```
?- append("a", "b", C).
```

```
C = [97, 98] ;
```

```
?- append("aa", "bb", C).
```

```
C = [97, 97, 98, 98] ;
```

Conclusions

- Prolog implémente les principes de la logique prédictive.
- Prolog possède une sémantique opérationnelle et une sémantique déclarative.
- Prolog est très utile pour le prototypage d'applications.
- Prolog est également très pratique pour prototyper des langages.
 - parsing en implémentant les règles syntaxiques
 - Interprétation en exécutant les règles de la sémantique opérationnelle
 - Compilation en générant du prolog