

Prolog HLIN602

10 avril 2019 – Ch. Retoré

Prolog - Applications

- Systèmes experts
- Bases de données déductives (datalog)
- Planning, vérification des circuits électroniques
- Traitement automatique du langage naturel

1

3

Prolog - Naissance

- ⌚ 1971 - Robert Kowalski (Edimbourg)
Résolution
- ⌚ 1972 - Premier implémentation d'un interpréteur de Prolog par Alain Colmerauer (Marseille)
- ⌚ 1977 - Premier compilateur grâce aux travaux de David H.D. Warren (Edimbourg).
- ⌚ 1987 - Constraint Logic Programming.

Prolog - Introduction

- Prolog - Programmation en logique,
- Prolog utilise les clauses de Horn (pour les programmes) et la résolution (pour l'exécution de ces programmes)
- Prolog utilise l'unification pour trouver la solution pour des quantificateurs universels
- La syntaxe est très proche de celle de la logique des prédicats.

2

4

Prolog - Introduction

• SWI Prolog (Prolog gratuit)
<http://www.swi-prolog.org/>

• Introduction à Prolog (en Anglais, mais le livre est disponible en Français)
<http://www.learnprolognow.org/>

5

7

Prolog - Bonnes habitudes

(et pour d'autres langages de programmation aussi)

- écrivez du commentaire avec votre programme (environ une ligne de commentaire pour une ligne de code).
- choisissez des noms clairs pour les prédictats (fonctions dans des autres langages) et des variables, écrivez clairement le sens du prédictat.

6

Prolog Syntaxe

• atomes: séquences commençant avec un minuscule, suivi des minuscules, chiffres et “_”

• variables: commencent avec un majuscule, suivie par une combinaison de minuscules, majuscules, chiffres et “_”.

Prolog Syntaxe

• nombres entiers: séquences des chiffres, préfixé “-” optionnel 23 -492 0

• nombres réels: 23.5 1.32E-21
10.0e100

• On appelle l'ensemble des atomes, nombres entiers et nombres réels les termes atomiques.

• les termes atomiques correspondent aux constantes dans la logique des prédictats.

8

Prolog Syntaxe - Termes

- des termes atomiques (atomes, nombres entiers et nombres réels) et des variables sont des termes
 - si a est un atome et t_1, \dots, t_n sont des termes, alors $a(t_1, \dots, t_n)$ est un terme. on dit que a est le symbole de fonction (Anglais **functor**), les termes t ses arguments et n son arité
 - Parallèle function $f(7,5)$ prédicat $p(7,5)$:
 - Un prédicat est une fonction à valeur dans $\{0,1\}$

\times $+(X,Y)$ $=(X,3)$ $=\!(X,+(X,1))$

livre(anna_karenina) X=X+1

contient(bibliotheque, journeaux, 171) 9

Prolog Syntaxe- Prédicats

- si a est un atome et t_1, \dots, t_n sont des termes, alors $a(t_1, \dots, t_n)$ est un prédicat.
 - alors, comme dans la logique des prédicats, les termes et les prédicats ont la même syntaxe

Terme = structure de données

Prédicat = formule logique atomique
= programme

Prolog Syntax - Clauses

- si p , q , r sont des prédictats, les expressions suivantes sont des clauses.
 - on a deux types des clauses: des faits, de la forme " $p.$ " et des règles du forme " $p :- q, \dots, r.$ "

on peut voir des clauses de la forme "p." comme ayant la forme "p :- true." (pourquoi?)

Prolog Syntax - Clauses

- si p, q, r sont des prédictats, les expressions suivantes sont des clauses.
 - on a deux types des clauses: des faits, de la forme “p.” et des règles du forme “p :- q,...,r.”

```

auteur(leo_tolstoy, anna_karenina).
auteur(X) :-          ∀x.∀y auteur(x,y) →
                  auteur(X,Y).      auteur(x)
livre(Y) :-          ∀x.∀y auteur(x,y) →
                  auteur(X,Y).      livre(y)

```

Prolog Syntaxe - Clauses

- si p, q, r sont des prédictats, les expressions suivantes sont des clauses.
- on a deux types des clauses: des faits, de forme “p.” et des règles de la forme
- “p :- q,...,r.”

13

Prolog Syntaxe - Programme

☞ Un programme en Prolog est un ensemble de clauses.

```
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

15

Prolog Syntaxe - Clauses

```
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).  
                                ↑  
                                ↓  
variable anonyme
```

14

Prolog Syntaxe - Programme

☞ Si p,q,r,... sont de prédictats, une question (Anglais: query) est de la forme

```
?- p,q,...,r.  
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

16

Prolog Syntaxe - Programme

```
?- auteur(X).  
  
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

17

Prolog Syntaxe - Programme

```
?- auteur(X).  
  
Interprétation: pour quel X peut-on démontrer que  
X est auteur?  
  
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

19

Prolog Syntaxe - Programme

Sémantique

?- auteur(X).

Remarque: c'est Prolog qui fournit le “?-”

```
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

18

- ➊ Un programme en Prolog a un sens déclaratif, qui est sa traduction directe en logique,
- ➋ “:-” correspond à “ \rightarrow ”,
- ➌ “,” (entre prédictats) correspond à “ \wedge ”
- ➍ et les variables de chaque clause sont (implicitement) quantifiées par \forall

20

Sémantique

- Le sens procédural provient de la résolution; à ce niveau l'ordre des clauses et l'ordre des prédictats dans une clause est important.
- Prolog commence toujours avec la première clause qui correspond au premier prédictat de la question.
- Prolog garde les autres clauses qui correspondent à la question : en cas d'échec, on essaie la clause suivante.

21

Prolog Syntaxe - Programme

?- auteur(X,_).

Seule

possibilité !

auteur(leo_tolstoy, anna_karenina).

→ auteur(X) :-
 auteur(X,_).

livre(Y) :-
 auteur(_,Y).

23

Prolog Syntaxe - Programme

?- auteur(X).

Seule

possibilité !

auteur(leo_tolstoy, anna_karenina).

→ auteur(X) :-
 auteur(X,_).
livre(Y) :-
 auteur(_,Y).

22

Prolog Syntaxe - Programme

?- auteur(X,_).

Seul

possibilité !

→ auteur(leo_tolstoy, anna_karenina).

auteur(X) :-
 auteur(X,_).
livre(Y) :-
 auteur(_,Y).

24

Prolog Syntaxe - Programme

```
?- auteur(X,_).      Réponse:  
Seule                  X = leo_tolstoy  
possibilité !  
→ auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

25

Prolog Syntaxe - Programme

```
?- auteur(X,anna_karenina).  
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

27

Prolog Syntaxe - Programme

```
?- auteur(X,Y).
```

```
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

26

Prolog Syntaxe - Programme

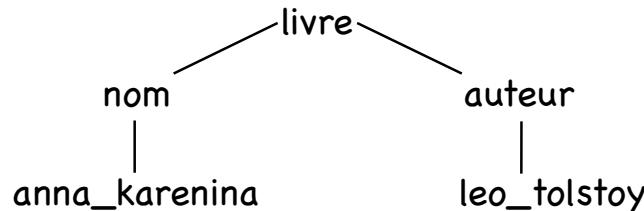
```
?- auteur(X),livre(Y).  
auteur(leo_tolstoy, anna_karenina).  
auteur(X) :-  
    auteur(X,_).  
livre(Y) :-  
    auteur(_,Y).
```

28

Unification

- Les termes vus comme des arbres.

livre(nom(anna_karenina),auteur(leo_tolstoy))

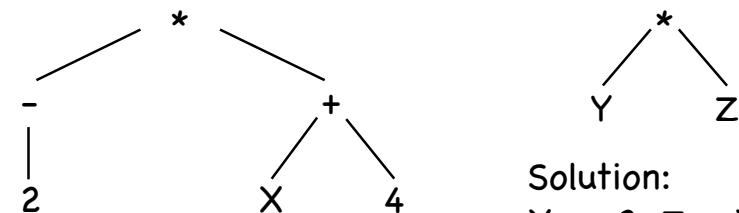


29

Unification

- L'unification est un algorithme simple pour déterminer si deux termes peuvent être égaux en donnant des valeurs aux variables.

$$\begin{aligned}-2^*(X+4) &= *(-2,+ (X,4)) \\ Y^*Z &= *(Y,Z)\end{aligned}$$



Solution:
 $X = -2, Z = X+4$

Unification

- L'unification est un algorithme simple pour déterminer si deux termes peuvent être égaux en donnant des valeurs aux variables.

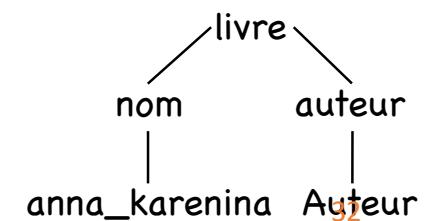
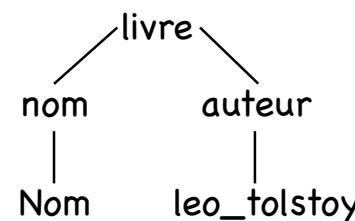
Cas fréquent: unification entre une variable et un autre terme.

$$X = 3$$

30

Unification

- L'unification est un algorithme simple pour déterminer si deux termes peuvent être égaux en donnant des valeurs aux variables.



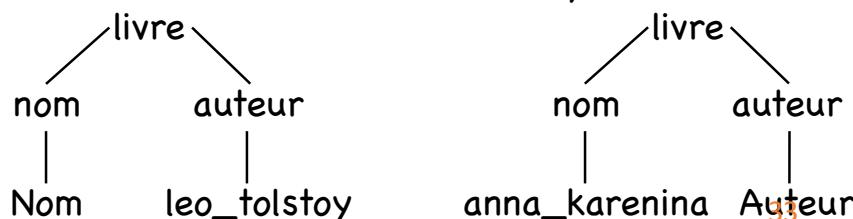
31

Unification

➊ L'unification est un algorithme simple pour déterminer si deux termes peuvent être égaux en donnant des valeurs aux variables.

Solution

Nom = anna_karenina
Auteur = leo_tolstoy



Unification

- Prolog utilise l'unification pour chaque résolution d'un atome de la question avec une clause.
- On peut utiliser l'unification explicitement, en utilisant $X = Y$ unifie les termes X et Y (à éviter)
- Un cas difficile est $X = f(X)$. La réponse correct serait un échec (pourquoi?), la réponse de Prolog est $f(f(f(f(\dots))))$

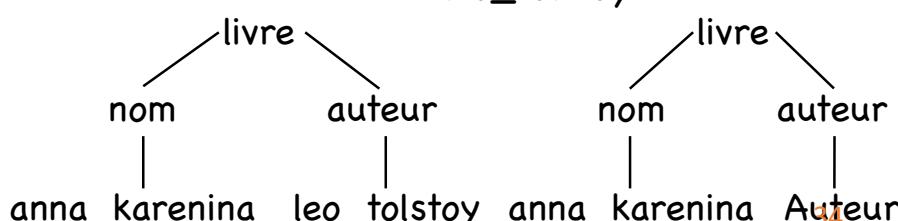
35

Unification

➋ L'unification est un algorithme simple pour déterminer si deux termes peuvent être égaux en donnant des valeurs aux variables.

Solution

Nom = anna_karenina
Auteur = leo_tolstoy



Unification

Exemples

$$\begin{aligned}f(X) &= f(a,b) \\f(X,g(b,Y)) &= g(b,a) \\f(X,h(a)) &= f(g(Z),h(Z)) \\f(g(b,a),c) &= f(g(X,X),Y)\end{aligned}$$

36

Unification

Exemples

$f(X) = f(a, b)$
 ~~$f(X, g(b, Y)) = g(b, a)$~~
 $f(X, h(a)) = f(g(Z), h(Z)) \leftarrow f(g(a), h(a))$
 ~~$f(g(b, a), c) = f(g(X, X), Y)$~~
 $X = g(a)$
 $Z = a$

37

Unification, Egalité et Inégalité

$X = Y$	unifie X et Y
$X == Y$	X et Y sont strictement égaux
$X \neq Y$	$X == Y$ est faux

39

Unification, Egalité et Inégalité

$X = Y$ unifie X et Y

$X == Y$ X et Y sont strictement égaux

$X \neq Y$ Alors $X == X$ et $a == a$ sont vrais
Mais $X == Y$ et $X == a$ sont faux

38

Unification, Egalité et Inégalité

$X = Y$ unifie X et Y

$X == Y$ X et Y sont strictement égaux

$X \neq Y$ $X == Y$ est faux
Alors $X \neq X$ et $a \neq a$ sont faux
Mais $X \neq Y$ et $X \neq a$ sont vrai

40

Sémantique Déclarative

1. étant donnée une question q, r, ... et un programme avec des clauses
 $t :- s, u, v.$
dont on appelle t la tête, et s, u, v le corps.
2. on prend la première clause dont la tête t s'unifie avec le premier prédicat atomique q de la question, en cas d'échec on continue avec la clause suivante.
3. on remplace q par le corps de la clause et on continue avec 2. jusqu'au moment où la question est vide.

41

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

43

Exemple avec des termes

- ➊ 0 dénote le nombre 0,
- ➋ si X dénote le nombre N, s(X) dénote le nombre N + 1,

0	0
1	s(0)
2	s(s(0))
3	s(s(s(0)))

Bien sûr que ceci n'est pas une façon efficace de coder les entiers !

42

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(s(0)),s(s(0)),Z).
```

44

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(s(0),s(s(0))),Z).
```

échec d'unification
 $O \neq s(s(0))$

45

Résolution Clauses de Horn

```
addition(0,X,X).  
addition(s(X),Y,s(V)) :-  
    addition(X,Y,V).
```

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

$X = s(0)$
 $Y = s(s(0))$
 $Z = s(V)$

On remplace la question par
 $\text{addition}(X,Y,V) = \text{addition}(s(0),s(s(0)),V)$

$Z = s(V)$

47

Résolution Clauses de Horn

```
addition(0,X,X).  
addition(s(X),Y,s(V)) :-  
    addition(X,Y,V).
```

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(s(0),s(s(0))),Z).
```

$X = s(0)$
 $Y = s(s(0))$
 $Z = s(V)$

$Z = s(V)$

46

Résolution Clauses de Horn

```
addition(0,X,X).  
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(0),s(s(0)),V).
```

$X = 0$
 $Y = s(s(0))$
 $V = s(W)$

$Z = s(V)$

48

Résolution Clauses de Horn

```
addition(0,X,X).
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(0),s(s(0)),V).
```

$X = 0$
 $Y = s(s(0))$
 $V = s(W)$

Unification est possible que avec la deuxième clause.
 $s(0) \neq 0$

$Z = s(s(W))$

49

Résolution Clauses de Horn

```
addition(0,X,X).
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

$X = 0$
 $Y = s(s(0))$
 $V = s(W)$

On remplace la question par
 $\text{addition}(X,Y,V) = \text{addition}(0,s(s(0)),W)$

$Z = s(s(W))$

51

Résolution Clauses de Horn

```
addition(0,X,X).
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(s(0),s(s(0)),V).
```

$X = 0$
 $Y = s(s(0))$
 $V = s(W)$

Remarque: la solution Z devient plus grande
 $Z = s(V) = s(s(W))$

$Z = s(s(W))$

50

Résolution Clauses de Horn

```
addition(0,X,X).
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(0,s(s(0)),W).
```

$W = X$
 $X = s(s(0))$

$Z = s(s(W))$

52

Résolution Clauses de Horn

```
addition(0,X,X).  
addition(s(X),Y,s(W)) :-  
    addition(X,Y,W).
```

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(0,s(s(0)),W).
```

$W = X$
 $X = s(s(0))$

Finalement, la première clause s'applique et on trouve une solution.

$Z = s(s(s(s(0))))$

53

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(X,s(0),s(s(0))).
```

$X = s(V)$
 $Y = s(0)$
 $Z = s(0)$

55

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(X,s(0),s(s(0))).
```

Un avantage de la formulation des propriétés en logique est qu'il n'y a pas vraiment d'entrée et de sortie: on peut poser des questions comme on veut.

54

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(V,s(0),s(0)).
```

$X = s(V)$
 $Y = s(0)$
 $Z = s(0)$

56

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(V,s(0),s(0)).
```

$V = 0$
 $W = s(0)$

$X = s(V)$

57

Résolution Clauses de Horn

```
addition(0,Z,Z).  
addition(s(V),Y,s(Z)) :-  
    addition(V,Y,Z).
```

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(X,Y,s(s(0))).
```

$X = s(V)$
 $Y = Z$
 $Z = s(s(0))$

59

Résolution Clauses de Horn

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

```
multiplication(0,X,0).  
multiplication(s(X),Y,Z) :-  
    multiplication(X,Y,V),  
    addition(V,Y,Z).
```

```
addition(V,s(0),s(0)).
```

$V = 0$
 $W = s(0)$

$X = s(0)$

58

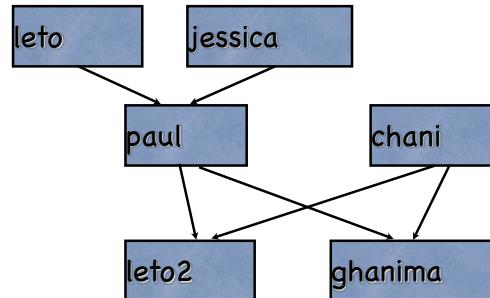
Familles

- ➊ Il est presque obligatoire de parler des familles dans un cours de Prolog.
- ➋ Donc, on en a vu en TP, et on en reparle ici.

60

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```



61

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```

```
homme(leto).  
homme(paul).  
homme(leto2).  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
pere(Parent, Enfant) :-  
    parent(Parent, Enfant),  
    homme(Parent).  
  
mere(Parent, Enfant) :-  
    parent(Parent, Enfant),  
    femme(Parent).
```

63

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```

```
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

Familles

```
pere(leto, paul).  
pere(paul, leto2).  
pere(paul, ghanima).  
  
mere(jessica, paul).  
mere(chani, leto2).  
mere(chani, ghanima).
```

On peut choisir les prédictats de base qui nous conviennent. Alors, on pourrait prendre *pere/2* et *mere/2* comme prédictats de base et donner des règles pour *parent/2*.

```
parent(Parent, Enfant) :-  
    pere(Parent, Enfant).
```

```
parent(Parent, Enfant) :-  
    mere(Parent, Enfant).
```

62

64

Familles

```
pere(leto, paul).  
pere(paul, leto2).  
pere(paul, ghanima).
```

```
mere(jessica, paul).  
mere(chani, leto2).  
mere(chani, ghanima).
```

```
parent(Parent, Enfant) :-  
    pere(Parent, Enfant).
```

Remarque: on a perdu de l'information ! homme/1 et femme/1 ne sont pas totalement définissable en termes de pere/2 et mere/2.
Pourquoi?

```
parent(Parent, Enfant) :-  
    mere(Parent, Enfant).
```

65

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```

```
homme(leto).  
homme(paul).  
homme(leto2).  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

grandpere(GrandParent, Enfant) :-

```
parent(GrandParent, Parent),  
parent(Parent, Enfant),  
homme(GrandParent).
```

GrandParent = leto,
PetitEnfant = Enfant

?- grandpere(leto, PetitEnfant).

67

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).
```

```
homme(leto).  
homme(paul).  
homme(leto2).  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

66

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).
```

```
homme(leto).  
homme(paul).  
homme(leto2).  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

grandpere(leto, PetitEnfant) :-

```
parent(leto, Parent),  
parent(Parent, PetitEnfant),  
homme(leto).
```

GrandParent = leto,
PetitEnfant = Enfant

?- parent(leto, Parent), parent(Parent, PetitEnfant),
homme(leto).

68

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

grandpere(leto, PetitEnfant) :-

```
parent(leto, Parent),
parent(Parent, PetitEnfant),
homme(leto).
```

GrandParent = leto,
PetitEnfant = Enfant

?- parent(leto, Parent), parent(Parent, PetitEnfant),
homme(leto).

69

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).

grandpere(GrandParent, Enfant) :-  
parent(GrandParent, Parent),
parent(Parent, Enfant),
homme(GrandParent).
```

?- parent(paul, PetitEnfant), homme(leto).

71

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

grandpere(GrandParent, Enfant) :-

```
parent(GrandParent, Parent),
parent(Parent, Enfant),
homme(GrandParent).
```

Parent = paul

?- parent(leto, paul), parent(paul, PetitEnfant), homme(leto).

70

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

grandpere(GrandParent, Enfant) :-

```
parent(GrandParent, Parent),
parent(Parent, Enfant),
homme(GrandParent).
```

PetitEnfant = leto2

?- parent(paul, leto2), homme(leto).

72

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).
```

Solution: PetitEnfant = leto2

73

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).
```

PetitEnfant = ghanima

?- parent(paul, PetitEnfant), homme(leto).

75

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).
```

“;” demande à Prolog
de trouve d’autres
solutions

Solution: PetitEnfant = leto2

74

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).
```

PetitEnfant = ghanima

?- parent(paul, ghanima), homme(leto).

76

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).           PetitEnfant = ghanima
```

```
?- homme(leto).
```

77

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).           ";" une deuxième fois  
                                    ne donne pas d'autres  
                                    solutions.
```

Solution: PetitEnfant = ghanima

79

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       femme(jessica).
parent(chani, leto2).        femme(chani).
parent(chani, ghanima).      femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).           PetitEnfant = ghanima
```

Solution: PetitEnfant = ghanima

78

Familles

```
parent(leto, paul).          homme(leto).
parent(jessica, paul).       homme(paul).
parent(paul, leto2).         homme(leto2).
parent(paul, ghanima).       parent(paul, ghanima).
parent(chani, leto2).        femme(jessica).
parent(chani, ghanima).      femme(chani).
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).           ";" une deuxième fois  
                                    ne donne pas d'autres  
                                    solutions.
```

?- parent(paul, PetitEnfant), homme(leto).

80

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
grandpere(GrandParent, Enfant) :-  
    parent(GrandParent, Parent),  
    parent(Parent, Enfant),  
    homme(GrandParent).  
  
    ";" une deuxième fois  
    ne donne pas d'autres  
    solutions.
```

```
?- parent(leto, Parent), parent(Parent, PetitEnfant),  
    homme(leto).  
                                         81
```

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
ancetre(Ancetre, Descendant) :-  
    parent(Ancetre, Descendant).  
ancetre(Ancetre, Descendant) :-  
    ancetre(Ancetre1, Descendant),  
    ancetre(Ancetre, Ancetre1).
```

83

Familles

```
parent(leto, paul).  
parent(jessica, paul).  
parent(paul, leto2).  
parent(paul, ghanima).  
parent(chani, leto2).  
parent(chani, ghanima).  
  
homme(leto).  
homme(paul).  
homme(leto2).  
  
femme(jessica).  
femme(chani).  
femme(ghanima).
```

```
ancetre(Ancetre, Descendant) :-  
    parent(Ancetre, Descendant).  
ancetre(Ancetre, Descendant) :-  
    parent(Parent, Descendant),  
    ancetre(Ancetre, Parent).
```

82

Familles

```
?- ancetre(A, jean).
```

```
ancetre(Ancetre, Descendant) :-  
    parent(Ancetre, Descendant).  
ancetre(Ancetre, Descendant) :-  
    ancetre(Ancetre1, Descendant),  
    ancetre(Ancetre, Ancetre1).
```

84

Familles

?- parent(A, jean).

échec (car on ne sait rien de jean).

ancetre(Ancetre, Descendant) :-

parent(Ancetre, Descendant).

ancetre(Ancetre, Descendant) :-

ancetre(Ancetre1, Descendant),

ancetre(Ancetre, Ancetre1).

85

Familles

?- ancetre(A2,A1), ancetre(A1,A),
ancetre(A, jean).

... et on peut continuer comme ça

ancetre(Ancetre, Descendant) :-

parent(Ancetre, Descendant).

ancetre(Ancetre, Descendant) :-

ancetre(Ancetre1, Descendant),

ancetre(Ancetre, Ancetre1).

87

Familles

?- ancetre(A1,A), ancetre(A, jean).

on revient sur ancêtre et on essaie la
deuxième (et dernière) clause

ancetre(Ancetre, Descendant) :-

parent(Ancetre, Descendant).

ancetre(Ancetre, Descendant) :-

ancetre(Ancetre1, Descendant),

ancetre(Ancetre, Ancetre1).

86

Familles

?- ancetre(A3,A2), ancetre(A2,A1),
ancetre(A1,A), ancetre(A, jean).

... et on peut continuer comme ça

Conclusion: une bonne définition en logique ne
donne pas nécessairement un bon programme

ancetre(Ancetre, Descendant) :-

parent(Ancetre, Descendant).

ancetre(Ancetre, Descendant) :-

parent(GrandAncetre,Ancetre),

ancetre(Ancetre,Descendant)

88

Arithmétique

- ➊ Bien qu'on puisse traiter l'arithmétique avec des termes de Prolog (et de façon purement logique), il est utile de pouvoir faire du calcul arithmétique directement.
- ➋ Prolog fournit le prédicat “is” pour le faire.

89

Arithmétique

- ➊ $X \text{ is } Y + Z$
- ➋ rappel que ceci est un façon d'écrire $\text{is}(X, +(Y, Z))$
- ➌ Etant donnée que des expressions avec “is” n'ont un sens que quand on connaît les valeurs des variables à droite de “is”, cette opération n'est pas purement logique.

91

Arithmétique

- ➊ $X \text{ is } Y + Z$
- ➋ X est une variable libre,
- ➌ dont on ne connaît pas encore la valeur.
- ➍ Y et Z sont des variables
- ➎ dont on connaît déjà les valeurs.
- ➏ Prolog donne une erreur
- ➐ si cette condition n'est pas respectée pendant la résolution du programme.

90

Arithmétique

- ➊ Opérations de comparaison entre expressions arithmétiques.
Ces opérations n'ont de sens que quand on connaît préalablement X et Y .
- ➋ $X < Y, X = < Y,$
- ➌ $X > Y, X = > Y,$
- ➍ $X =:= Y, X = \backslash= Y$

92

Arithmétique

% = max(X, Y, Z)
% vrai si Z est le maximum de X et Y.

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

93

95

Arithmétique

% = factorielle(X, F)
% vrai si F est le factoriel de X.

```
factorielle(0, 1).  
factorielle(NO, F) :-  
    NO > 0,  
    N is NO - 1,  
    factorielle(N, F0),  
    F is F0 * NO.
```

94

Listes

- ➊ Une structure de données utile, avec des abréviations syntaxiques spéciales, est la liste.
- ➋ Le liste vide: []
- ➌ Chaque liste non-vide est du forme [H|T], avec H le premier élément de la liste (Head) et T un liste contenant
 - ➍ les autres éléments (Tail).

Listes

- ➊ Liste
- ➋ []
- ➌ [1|[]]
- ➍ [1|[2|[]]]
- ➎ [1|[2|[3|[]]]]
- ➏ [1|[2|[3|[4|[]]]]]
- ➐ Version simple
- ➋ []
- ⌒ [1]
- ⌓ [1,2]
- ⌔ [1,2,3]
- ⌕ [1,2,3,4]

96

Listes

```
% = est_liste(Terme)          [2,3]
% vrai si Terme est un liste   [2|[]]
                           [1|2]
est_liste([]).                [1,2|3]
est_liste([_|L]) :-             [1,2|[3]]
                           est_liste(L).
```

97

Listes

```
% = membre(Element, Liste)
%
% vrai si Liste contient Element.
membre(X, [X|_]).           membre(X, [_|Ys]) :- 
                             membre(X, Ys).
```

99

Listes

```
% = est_liste(Terme)          [2,3] = [2|[3|[]]]
% vrai si Terme est un liste   [2|[]] = [2]
                           [1|2]
est_liste([]).                [1,2|3]
est_liste([_|L]) :-             [1,2|[3]] =
                           est_liste(L).
                           [1,2,3] =
                           [1|[2|[3|[]]]]
```

98

Listes

```
% = append(Liste1, Liste2, Liste3)
%
% vrai si Liste3 contient les éléments de Liste1
% suivi par les éléments de Liste2, c'est-à-dire
% Liste3 est la concatenation de Liste1 et
% Liste2
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- 
                           append(Xs, Ys, Zs).
```

100

Append Listes

```
% = append(Liste1, Liste2, Liste3)
%
% vrai si Liste3 contient les éléments de Liste1
% suivi par les éléments de Liste2, c'est-à-dire
% Liste3 est la concaténation de Liste1 et      %
Liste2

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

101

Reverse: Listes

```
reverse([], []).
reverse([X|Xs], Rs) :-
    reverse(Xs, [X], Rs).
```

```
reverse([], Rs, Rs).
reverse([X|Xs], Ys, Rs) :-
    reverse(Xs, [X|Ys], Rs).
```

103

Reverse Listes

```
% = reverse(Liste1, Liste2)
%
% vrai si Liste1 et Liste2 contiennent les mêmes
% éléments mais dans l'ordre inverse.

reverse([], []).
reverse([X|Xs], Rs) :-
    reverse(Xs, Rs0),
    append(Rs0, [X], Rs).
```

102

Reverse: Listes

```
?- reverse([a,b,c,d], Reverse).
```

→

```
reverse([], []).
reverse([X|Xs], Rs) :-
    reverse(Xs, [X], Rs).
```

```
reverse([], Rs, Rs).
reverse([X|Xs], Ys, Rs) :-
    reverse(Xs, [X|Ys], Rs).
```

X = a
Xs = [b,c,d]
Reverse = Rs

104

Reverse: Listes

```
?- reverse([a,b,c,d], Reverse).  
  
reverse([], []).  
→ reverse([a|[b,c,d]], Reverse) :-  
    reverse([b,c,d], [a], Reverse).  
  
reverse([], Rs, Rs).  
reverse([X|Xs], Ys, Rs) :-  
    reverse(Xs, [X|Ys], Rs).
```

105

Reverse: Listes

```
?- reverse([b,c,d], [a], Reverse).  
  
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).  
  
reverse([], Rs, Rs).  
→ reverse([X|Xs], Ys, Rs) :-  
    reverse(Xs, [X|Ys], Rs).
```

107

Rappel: Listes

```
?- reverse([b,c,d], [a], Reverse).  
  
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).  
  
reverse([], Rs, Rs).  
→ reverse([X|Xs], Ys, Rs) :-  
    reverse(Xs, [X|Ys], Rs).
```

106

Reverse: Listes

```
?- reverse([b,c,d], [a], Reverse).  
  
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).  
  
reverse([], Rs, Rs).  
→ reverse([b|[c,d]], [a], Reverse) :-  
    reverse([c,d], [b|[a]], Reverse).
```

108

Rappel: Listes

```
?- reverse([b,c,d], [a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

```
reverse([], Rs, Rs).
```

➡ **reverse([b,c,d], [a], Reverse) :-
 reverse([c,d], [b,a], Reverse).**

X = b
Xs = [c,d]
Ys = [a]
Rs = Reverse

109

Reverse: Listes

```
?- reverse([c,d], [b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

➡ **reverse([X|Xs], Ys, Rs) :-
 reverse(Xs, [X|Ys], Rs).**

X = c
Xs = [d]
Ys = [b,a]
Rs = Reverse

111

Reverse: Listes

```
?- reverse([c,d], [b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

```
reverse([], Rs, Rs).
```

➡ **reverse([X|Xs], Ys, Rs) :-
 reverse(Xs, [X|Ys], Rs).**

110

Reverse: Listes

```
?- reverse([c,d], [b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

```
reverse([], Rs, Rs).
```

➡ **reverse([c,d], [b,a], Reverse) :-
 reverse([d], [c,b,a], Reverse).**

X = c
Xs = [d]
Ys = [b,a]
Rs = Reverse

112

Reverse: Listes

```
?- reverse([d], [c,b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).  
  
reverse([], Rs, Rs).  
→ reverse([X|Xs], Ys, Rs) :-  
    reverse(Xs, [X|Ys], Rs).
```

X = d
Xs = []
Ys = [c,b,a]
Rs = Reverse

113

Reverse: Listes

```
?- reverse([], [d,c,b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

→ reverse([], Rs, Rs).
reverse([X|Xs], Ys, Rs) :-
 reverse(Xs, [X|Ys], Rs).

Rs = [d,c,b,a]
Rs = Reverse

115

Reverse: Listes

```
?- reverse([d], [c,b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).  
  
reverse([], Rs, Rs).  
→ reverse([d], [c,b,a], Reverse) :-  
    reverse([], [d,c,b,a], Reverse).
```

X = d
Xs = []
Ys = [c,b,a]
Rs = Reverse

114

Reverse: Listes

```
?- reverse([], [d,c,b,a], Reverse).
```

```
reverse([], []).  
reverse([X|Xs], Rs) :-  
    reverse(Xs, [X], Rs).
```

→ reverse([], Rs, Rs).
reverse([X|Xs], Ys, Rs) :-
 reverse(Xs, [X|Ys], Rs).

Reverse =
[d,c,b,a]

116

Prolog

➊ On a déjà vu des concessions à l'efficacité, par exemple le calcul des expressions arithmétique avec "is" qui ne sont pas purement logique.

➋ On va voir des autres constructions qui ont qu'un sens procédural : si on ne fait pas attention, elles peuvent détruire le sens logique des prédictats où elles sont utilisées.

117

Prolog Constructions non-logiques

➊ Les constructions non-logiques qu'on a déjà vu sont:
— X is Expression
— == et \==

X is X0 + 1

"is" a un sens pour Prolog que quand on connaît la valeur de X0

119

Prolog Constructions non-logiques

➊ Les constructions non-logiques qu'on a déjà vu sont:
— X is Expression
— == et \==

118

Prolog Constructions non-logiques

➊ Constructions non-logiques déjà rencontrées:
— X is Expression
— == et \==

Contrairement à l'unification X == Y échoue quand X et Y sont deux variables qui peuvent dénoter des objets différents.

120

Prolog

Constructions non-logiques

- ❶ Les constructions non-logiques se divisent en plusieurs catégories:
 - entrée/sortie (interaction avec fichiers et l'utilisateur)
 - contrôle (de la recherche des preuves)
 - introspection (verification de types)
 - meta-programmation (changements de base de données, etc.)

121

Prolog

Entrée/Sortie

- ❶ Les entrées et sorties se font grâce à quelque prédicats (comme "read" et "write") qui sont faits pour.
- ❶ Les sens logique des ces prédicats est "true" (vrai) mais on s'intéresse plutôt aux effets de borne (side effects) qui sont généralement l'affichage ou la lecture des termes.

122

Prolog

Entrée/Sortie

- ❶ `read(Terme)` - Terme est une variable qui sera unifié avec un Terme lu du clavier; pour entrer ce Terme, l'utilisateur doit terminer par "." et Terme doit respecter le syntaxe des termes en Prolog
- ❶ `write(Terme)` - écrit Terme vers le terminal.
- ❶ `writeln(Terme)` - comme `write(Terme)`, mais finit sur une nouvelle ligne; équivalent à `write(Terme), nl`

123

Prolog

Entrée/Sortie

- ❶ `nl`, saut de ligne
- ❶ `writeln(Terme)`, équivalent à "`write(Terme), nl`"
- ❶ `tab(Int)`, imprime Int espaces.

124

Prolog Entrée/Sortie

```
% = write_liste(Liste)
% écrit Liste vers le terminal, avec
% chaque element sur une ligne.
```

```
write_liste([]).
write_liste([X|Xs]) :-
    writeln(X),
    write_liste(Xs).
```

125

Prolog Contrôle

- ➊ Attention : essayer d'éviter la coupure le plus possible !
- ➋ Un prédicat défini avec une coupure n'a pas forcément d'interprétation logique (contrairement aux clauses en Prolog "pure") : c'est-à-dire que le sens de la coupure est procédural

127

Prolog Contrôle

- ➊ Prolog permet plusieurs constructions qui ont pour but de changer la façon dont Prolog cherche des preuves.
- ➋ Le plus important est la coupure "!" (en Anglais cut)
- ➌ Beaucoup d'autres constructions (comme le "if...then...else", la négation et "once") se définissent grâce à la coupure.

126

Prolog Contrôle

- ➊ Il y a plusieurs raisons pour utiliser une coupure. Le plus important est l'efficacité : la coupure peut aider Prolog à éviter de faire des calculs inutiles.
- ➋ Mais, cherchez toujours d'abord une solution sans coupure.
- ➌ Et écrivez clairement dans le commentaire ce que la coupure est censée de faire.

128

Prolog Contrôle

➊ Le sens de la coupure “!”

- réussit toujours (comme “true”)
- toutes les solutions alternatives pour le prédicat (les clauses du même prédicat après celle-ci) sont éliminées.
- toutes les solutions alternatives pour des prédicats qui apparaissent dans la même clause avant la coupure sont éliminés.

129

Prolog Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-  
    auteur(X, _),  
!
```

131

Prolog Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

```
auteur(X) :-  
    auteur(X, _).
```

130

Prolog Contrôle

```
% = auteur(Auteur, Livre)
auteur(leo_tolstoy, anna_karenina).
auteur(leo_tolstoy, war_and_peace).
```

```
% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre
```

→

```
auteur(X) :-  
    auteur(X, _),  
!
```

132

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
    !.
```

133

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
    !.
```

134

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
    !.
```

135

Prolog Contrôle

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
    !.
```

136

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
→ auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
!.
```

137

139

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
!.
```

138

Prolog Contrôle

```
% = auteur(Auteur, Livre)  
auteur(leo_tolstoy, anna_karenina).  
auteur(leo_tolstoy, war_and_peace).
```

% = auteur(Auteur)
% vrai si Auteur est l'écrivain d'au moins un livre

```
auteur(X) :-  
    auteur(X, _),  
!.
```

139

139

Prolog Contrôle

?- max(34, 24, Max).

% = max(X, Y, Z)
% vrai si Z est le maximum de X et Y.

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

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

140

Prolog Contrôle

```
?- max(34, 24, Max).
```

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

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

141

Prolog Contrôle

```
?- max(34, 24, 34).
```

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
→ max(34,24,34) :-      X = 34  
    34 >= 24,              Y = 24  
    !.                      Max = X = 34  
→ max(X,Y,Y) :-  
    Y > X.
```

143

Prolog Contrôle

```
?- max(34, 24, Max).
```

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
→ max(X,Y,X) :-      X = 34  
    X >= Y,              Y = 24  
    !.                      Max = X = 34  
→ max(X,Y,Y) :-  
    Y > X.
```

142

Prolog Contrôle

```
?- max(34, 24, 34).
```

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
max(34,24,34) :-      X = 34  
    34 >= 24,              Y = 24  
    !.                      Max = X = 34  
→ max(X,Y,Y) :-  
    Y > X.
```

144

Prolog Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
max(34,24,34) :-      X = 34
                  34 >= 24, →
                  !.
→ max(X,Y,Y) :-      Max = X = 34
                  Y > X.
```

145

Prolog Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
max(34,24,34) :-      X = 34
                  34 >= 24,
                  !. →
→ max(X,Y,Y) :-      Max = X = 34
                  Y > X.
```

147

Prolog Contrôle

?- max(34, 24, 34).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
max(34,24,34) :-      X = 34
                  34 >= 24,
                  !. →
→ max(X,Y,Y) :-      Max = X = 34
                  Y > X.
```

146

Prolog Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

```
→ max(X,Y,X) :-      X = 0
                  X >= Y,
                  !.
→ max(X,Y,Y) :-      Max = X = 0
                  Y > X.
```

148

Prolog Contrôle

?- max(0, 24, 0).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

→ max(0,24,0) :-
 0 >= 24,
 !.
→ max(X,Y,Y) :-
 Y > X.

X = 0
Y = 24
Max = X = 0

149

Prolog Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

X = 0
Y = 24
Max = Y = 24

151

Prolog Contrôle

?- max(0, 24, 0).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

→ max(0,24,0) :-
 0 >= 24,
 !.
→ max(X,Y,Y) :-
 Y > X.

X = 0
Y = 24
Max = X = 0

150

Prolog Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

→ max(X,Y,X) :-
 X >= Y,
 !.
→ max(0,24,24) :-
 24 > 0.

X = 0
Y = 24
Max = Y = 24

152

Prolog Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

```
max(0,24,24) :-  
    → 24 > 0.
```

X = 0
Y = 24
Max = Y = 24

153

Prolog Contrôle

?- max(0, 24, 24). →

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

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

X = 0
Y = 24
Max = Y = 24

155

Prolog Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

```
max(0,24,24) :-  
    24 > 0. →
```

X = 0
Y = 24
Max = Y = 24

154

Prolog Contrôle

?- max(0, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

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

Si le test dans le premier clause échoue on peut être sûr que le test dans la deuxième clause réussit.

156

Prolog Contrôle

?- max(0, 24, Max).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

Alors, on peut être tenté de supprimer le deuxième test.
Quel est le problème avec ce programme ?

157

Prolog Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

Réussit, car il n'y a plus de vérification que Y > X.

159

Prolog Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

Echec d'unification, car on ne peut pas unifier X à la fois avec 34 et avec 24.

158

Prolog Contrôle

?- max(34, 24, 24).

% = max(X, Y, Z)

% vrai si Z est le maximum de X et Y.

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

Version correcte :
unification explicite après la coupure.

160

Prolog Contrôle

```
% = membre(Element, Liste)
%
% vrai si Liste contient Element.
% s'utilise pour chercher un Element, mais aussi
% pour énumérer les différents Elements
```

```
membre(X, [X|_]).
```

```
membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

161

163

Prolog Contrôle

```
% = verifie_member(Element, Liste)
%
% vrai si Element est strictement égal à un
% des membres de Liste.
```

```
verifie_membre(X, [Y|_]) :-  
    X == Y,  
    !.
```

```
verifie_membre(X, [_|Ys]) :-  
    verifie_membre(X, Ys).
```

162

Prolog Contrôle

- Alors, quelle est la différence entre
 - membre(a, [a,b,c]). ("member" est prédéfini et marche pareil)
 - verifie_membre(a, [a,b,c]). ("memberchk" est prédéfini et marche pareil)

Prolog Contrôle

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

→ membre(X, [X|_]).

X = a
_ = [b,c]

→ membre(X, [_|Ys]) :-
 membre(X, Ys).

164

Prolog Contrôle

```
?- membre(a, [a,b,c]).  
  
membre(X, [X|_]). → X = a  
                                _ = [b,c]  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).      Réussite directe,  
                        mais ...
```

165

Prolog Contrôle

```
?- membre(a, [a,b,c]). → X = a  
                                _ = [b,c]  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).      C'est-à-dire, si on  
                        demande une autre  
                        solution ...
```

167

Prolog Contrôle

```
?- membre(a, [a,b,c]). →  
  
membre(X, [X|_]).          X = a  
                                _ = [b,c]  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).      Prolog a encore le  
                        choix !
```

166

Prolog Contrôle

```
?- membre(a, [a,b,c]). →  
  
membre(X, [X|_]).          X = a  
                                _ = [b,c]  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).      Prolog va essayer  
                        d'en trouver.
```

168

Prolog Contrôle

```
?- membre(a, [a,b,c]).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
_ = a
Ys = [b,c]

Prolog va essayer
d'en trouver.

169

Prolog Contrôle

```
?- membre(a, [a,b,c]).  
  
membre(X, [X|_]).  
  
membre(a, [_|[b,c]]) :-  
    → membre(a, [b,c]).
```

X = a
_ = a
Ys = [b,c]

Prolog va essayer
d'en trouver.

171

Prolog Contrôle

```
?- membre(a, [a,b,c]).  
  
membre(X, [X|_]).  
  
→ membre(a, [_|[b,c]]) :-  
    membre(a, [b,c]).
```

X = a
_ = a
Ys = [b,c]

Prolog va essayer
d'en trouver.

170

Prolog Contrôle

```
?- membre(a, [b,c]).  
  
→ membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
X = b
échec!

Prolog va essayer
d'en trouver.

172

Prolog Contrôle

```
?- membre(a, [b,c]).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
Ys = [c]

Prolog va essayer
d'en trouver.

173

Prolog Contrôle

```
?- membre(a, [c]).  
  
→ membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
X = c
échec!

Prolog va essayer
d'en trouver.

175

Prolog Contrôle

```
?- membre(a, [b,c]).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
Ys = [c]

Prolog va essayer
d'en trouver.

174

Prolog Contrôle

```
?- membre(a, [c]).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
Ys = []

Prolog va essayer
d'en trouver.

176

Prolog Contrôle

```
?- membre(a, []).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
Ys = []

Prolog va essayer
d'en trouver.

177

Prolog Contrôle

```
?- membre(a, []).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
[_|Ys] \= []
échec

Prolog va essayer
d'en trouver.

179

Prolog Contrôle

```
?- membre(a, []).  
  
→ membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
[] \= [X|_]
échec!

Prolog va essayer
d'en trouver.

178

Prolog Contrôle

```
?- membre(a, []).  
  
membre(X, [X|_]).  
  
→ membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

X = a
[_|Ys] \= []
échec

Prolog va essayer
d'en trouver.
Et ça demande
des efforts! 180

Prolog Contrôle

```
?- membre(a, [a,a,a]).  
membre(X, [X|_]).  
  
membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

Deuxième problème potentiel: qu'est-ce qui ce passe avec la question en haut ?

181

Prolog Contrôle

```
?- membre(a, [a,a,a]).  
membre(X, [X|_]).  
  
membre(X, [_|Ys]) :-  
    membre(X, Ys).
```

Deuxième problème potentiel: qu'est-ce qui ce passe avec la question en haut ?

Prolog dit "oui", comme il faut, mais il le fait trois fois

182

Prolog Contrôle

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

→ `verifie_membre(X, [Y|_]) :-
 X == Y,
 !.`

X = a
Y = a
_ = [b,c]

→ `verifie_membre(X, [_|Ys]) :-
 verifie_membre(X, Ys).`

183

Prolog Contrôle

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

→ `verifie_membre(X, [Y|_]) :-
 X == Y,
 !.`

X = a
Y = a
_ = [b,c]

→ `verifie_membre(X, [_|Ys]) :-
 verifie_membre(X, Ys).`

184

Prolog Contrôle

```
?- verifie_membre(a, [a,b,c]).  
  
verifie_membre(X, [Y|_]) :-  
    X == Y,  
    !.  
  
→ verifie_membre(X, [_|Ys]) :-  
    verifie_membre(X, Ys).
```

X = a
Y = a
_ = [b,c]

185

Prolog Contrôle

- une coupure verte est une coupure qui élimine des démonstrations, mais n'élimine pas de solutions : c'est à dire qu'il aide Prolog à éviter des démonstrations qui ne peuvent pas réussir.
- une coupure rouge est un coupure qui élimine des démonstrations mais aussi des solutions.

187

Prolog Contrôle

```
?- verifie_membre(a, [a,b,c]).  
  
verifie_membre(X, [Y|_]) :-  
    X == Y,  
    !.  
  
verifie_membre(X, [_|Ys]) :-  
    verifie_membre(X, Ys).
```

X = a
Y = a
_ = [b,c]

186

Prolog Contrôle - négation

- On a vu que les clauses de Horn sont une restriction sur des formules en forme normale disjonctive tel que il y a exactement un littéral positif par clause.
- La négation en Prolog est un remède partiel qui permet d'avoir plusieurs littéraux positifs dans un clause.

188

Prolog Contrôle - négation

- ➊ La négation présente dans Prolog s'appelle "négation as failure" : un littéral est faux quand il n'y a pas de preuve démontrant qu'il est vrai.
- ➋ C'est pour ça que l'inverse de "true" (vrai) ne s'appelle pas "false" (faux) mais "fail" (échec).

189

Prolog Contrôle - négation

```
non_membre(Element, Liste) :-  
    membre(Element, Liste),  
    !,  
    fail.  
non_membre(_Element, _Liste).
```

191

Prolog Contrôle - négation

- ➊ On peut définir la négation avec la coupure et "fail" (l'inverse de "true", l'atome dont la preuve échoue directement).
- ➋ `not(p):-p,!;fail.`
- ➌ `not(p).`
- ➍ Si p est une expression complexe, il faut utiliser "call" qui déclenche l'évaluation de p.
- ➋ `not(X):-call(X),!,fail.`
- ➌ `not(X).`

190

Prolog Contrôle - négation

- ➊ Faites attention à la négation s'il y a un terme avec des variables libres. Le but `\+ auteur(leo_tolstoy, Livre)`
- ➋ ne veut pas dire "quels sont les livres qui ne sont pas écrit par leo_tolstoy" mais "il n'y a pas de livre dont leo_Tolstoy soit l'écrivain."

192

Prolog Contrôle - négation

```
non_auteur(Ecrivain, Livre) :-  
    auteur(Ecrivain, Livre),  
    !,  
    fail.  
non_auteur(Ecrivain, Livre).
```

193

Prolog Contrôle - if then else

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

195

Prolog Contrôle - if then else

```
% = max(X, Y, Z)  
% vrai si Z est le maximum de X et Y.  
  
max(X,Y,Z) :-  
    X >= Y,  
    !,  
    Z = X.  
max(X,Y,Y).
```

194

Prolog Introspection

- ➊ Il y a plusieurs prédicts en Prolog qui permettent de déterminer le type d'un terme.
 - var(X), vrai si X est une variable libre
 - atomic(X), vrai si X est un terme atomique (constante, nombre entier ou réel)
 - compound(X), vrai si X est un terme complexe (de forme f(A,B))

196

Prolog Introspection

- ➊ Un liste L est dit un liste propre si et seulement si L ne contient pas de sous-listes qui sont des variables libres.
- ➋ Alors [], [X,a] et [X,Y,Z] sont des listes propres.
- ➌ Mais X, [a|Y] et [a,b,c|Z] ne sont pas propres.

197

Prolog Introspection

- ➊ On a déjà vu que le syntaxe pour les termes (nos structures de données) et les prédictats (nos éléments de programmes) sont similaires.
- ➋ Le prédictat call(Terme) prend son argument Terme et l'appelle comme un prédictat.
- ➌ call(Terme) se généralise à call(Terme, Arg), call(Terme, Arg1, Arg2) etc.

199

Prolog Introspection

```
% = est_liste_propre(Terme)
% vrai si Terme est un liste propre

est_liste_propre(Var) :-
    var(Var),
    !,
    fail.

est_liste_propre([]).

est_liste_propre([_|L]) :-
    est_liste_propre(L).
```

198

Prolog Introspection

- ➊ Qu'est-ce que ça veut dire ?
 - call append([a,b],[c,d],Ys) est équivalent à append([a,b],[c,d],Ys)
 - call([a,b], [c,d], Ys) est aussi équivalent à append([a,b],[c,d],Ys)
 - call(parent, X, Y) est équivalent à parent(X,Y)

200

Prolog Introspection

```
% = map(Liste1, Predicat, Liste2)
%
% applique Predicat(Element1, Element2)
% vérifie que les deux n-ièmes éléments de
% Liste1 et de Liste2 sont dans la relation P.

map([], _, []).
map([X|Xs], P, [Y|Ys]) :-
    call(P, X, Y),
    map(Xs, P, Ys).
```

201

Prolog Meta-programmes

- ➊ le prédicat
- ➋ **assert(parent(jean,marie))**
- ➌ ajoute au programme le prédicat
- ➍ **parent(jean,marie)**
- ➎ Si parent(jean,marie) était faux (ou ne pas démontrable) avant, il sera vrai après l'assert.
- ➏ On peut aussi ajouter une règle
Assert((pere(X,Y):-homme(X),parent(X,Y)))

203

Prolog Meta-programmes

- ➊ **call(Terme)** nous permet de traiter un terme comme un prédicat (partie du programme) et d'essayer de faire la preuve qui en correspond.
- ➋ les prédicats **assert(Terme)** et **retract(Terme)** vont plus loin en nous permettant de changer le programme.

202

Prolog Meta-programmes

- ➊ si l'ordre des clauses est important, on peut utiliser deux variants d'assert :
 - **asserta(Clause)** ajoute Clause au début des clauses pour le prédicat,
 - **assertz(Clause)** ajoute Clause à la fin.

204

Prolog Meta-programmes

- ❷ inversement, le prédicat
- ❷ **retractall(parent(.,.))**
- ❷ supprime tout les clauses de la forme parent(X,Y).
- ❷ après il n'y aura aucun fait de la forme parent(X,Y) dans la base de données.

205

Prolog: Meta-programmes

`:- dynamic parent/2.`

`parent(jean, marie).`

...

`:- dynamic homme/1.`

`homme(jean).`

...

Directive au compilateur ou interpréteur, sous forme d'un clause sans tête

207

Prolog Meta-programmes

- ❷ assert et retract sont utiles pour enregistrer des données de façon permanente.
- ❷ alors ils peuvent servir pour avoir une base de données dynamique (où les données peuvent changer)
- ❷ mais aussi pour faire la tabulation : pour enregistrer les résultats de calcul intermédiaire et ainsi d'éviter de recalculer de résultats.

206

Prolog Meta-programmes

- ❷ alors le prédicat **assert(parent(jean,marie))**

- ❷ ajoute le prédicat **parent(jean,marie)**
- ❷ à notre programme. Si parent(jean,marie) était faux (ou ne pas démontrable) avant, il sera vrai après le "assert".

208

Prolog Meta-programmes

- si l'ordre des clauses est important, on peut utiliser deux variants d'assert :
 - asserta(Clause) ajoute Clause au début des clauses pour le prédicat,
 - assertz(Clause) ajoute Clause à la fin.

209

Prolog Meta-programmes

```
?- asserta(parent(jean,marie)).
```

```
:- dynamic parent/2.
```

```
parent(leto, paul).  
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).
```

211

Prolog Meta-programmes

```
:- dynamic parent/2.
```

```
parent(leto, paul).  
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).
```

210

Prolog Meta-programmes

```
?- asserta(parent(jean,marie)).
```

```
:- dynamic parent/2.
```

```
parent(jean,marie).  
parent(leto, paul).  
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).
```

212

Prolog Meta-programmes

```
?- asserta(parent(jean,marie)).
```

```
:- dynamic parent/2.
```

```
parent(jean,marie).
```

```
parent(jean,marie).
```

```
parent(leto, paul).
```

```
parent(leto, alia).
```

```
parent(jessica, paul)
```

```
parent(jessica, alia).
```

Attention : Prolog ajoute
un clause même si elle
existe déjà !

213

Prolog - Meta-programmes

```
?- assertz(parent(jean,marie)).
```

```
:- dynamic parent/2.
```

```
parent(leto, paul).
```

```
parent(leto, alia).
```

```
parent(jessica, paul)
```

```
parent(jessica, alia).
```

215

Prolog - Meta-programmes

```
:- dynamic parent/2.
```

```
parent(leto, paul).
```

```
parent(leto, alia).
```

```
parent(jessica, paul)
```

```
parent(jessica, alia).
```

214

Prolog -Meta-programmes

```
?- assertz(parent(jean,marie)).
```

```
:- dynamic parent/2.
```

```
parent(leto, paul).
```

```
parent(leto, alia).
```

```
parent(jessica, paul)
```

```
parent(jessica, alia).
```

```
parent(jean,marie).
```

216

Prolog Meta-programmes

```
?- retract(parent(leto,paul)).
```

```
:- dynamic parent/2.
```

```
parent(leto, paul).  
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).  
parent(jean,marie).
```

217

Prolog - Meta-programmes

```
?- retract(parent(jessica,X)).
```

```
:- dynamic parent/2.
```

```
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).  
parent(jean,marie).
```

219

Prolog - Meta-programmes

```
?- retract(parent(leto,paul)).
```

```
:- dynamic parent/2.
```

```
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).  
parent(jean,marie).
```

218

Prolog - Meta-programmes

```
?- retract(parent(jessica,X)).      X = paul
```

```
:- dynamic parent/2.
```

```
parent(leto, alia).  
parent(jessica, alia).  
parent(jean,marie).
```

220

Prolog Meta-programmes

- ❷ inversement, le prédicat
retractall(parent(_____,_____)) supprime
tout les clauses de la forme parent(X,Y).
- ❸ après il n'y aura aucun fait de la forme
parent(X,Y) dans la base de données (le fait que
parent soit dynamique reste par contre).

Prolog Meta-programmes

```
?- retractall(parent(_____,_____)).  
:- dynamic parent/2.
```

221

223

Prolog Meta-programmes

```
?- retractall(parent(_____,_____)).
```

```
:- dynamic parent/2.
```

```
parent(leto, alia).  
parent(jessica, paul)  
parent(jessica, alia).  
parent(jean, marie).
```

222

Prolog Un mini Système Expert

- ❷ tout ceci nous donne les bons outils pour
programmer un système expert

- ❸ Qui évolue durant son utilisation
❹(changements des faits et des règles)

224

Prolog

Un mini Système Expert

l'architecture est la suivante :

- le système expert est dans un certain état,
- suivant la réponse de l'utilisateur le système expert change d'état et pose une nouvelle question ou donne une réponse.
- chaque état fournit une explication sur le pourquoi de la question ou de la réponse.

225

227

Petit système expert

```
cheval(X):-  
    oreilles(pointues,X),  
    peau(poils,X),  
    taille(U,X),  
    U>150,  
    U<300.  
  
chien(X):-  
    oreilles(pointues,X),  
    peau(poils,X),  
    taille(U,X),  
    U<100.  
  
chat(X):-  
    oreilles(pointues,X),  
    peau(poils,X),  
    taille(U,X),  
    U<30.  
  
oiseau(X):-  
    peau(plumes,X).  
  
crocodile(X):-  
    peau(ecailles,X),taille(U,X),U>20  
    0,oreilles(inexistantes,X).
```

Petit système expert

```
:dynamic(known/3).
```

```
liste_animaux([chat,chien,  
cheval,oiseau,crocodile]).
```

```
animal(X):-  
    liste_animaux(L),  
    member(P, L),  
    T=..[P,X], % T=P(X)  
    call(T), % T  
    write('je pense que '),  
    write(X),  
    write(' est un '),  
    write(P).
```

226

Petit système expert

```
ask(U,Y,X):-  
    known(U,Y0,X), !, Y0 = Y.  
  
oreilles(Y,X):-  
    ask(oreilles,Y,X).  
  
peau(Y,X):-  
    ask(peau,Y,X).  
  
taille(Y,X):-  
    ask(taille,Y,X).
```

228

Petit système expert

```
ask(oreilles,Y,X) :-  
    not(knows(oreilles,_X)),  
    write('comment sont les  
oreilles de '),  
    writeln(X),  
    /* menu_ask s'ajoute ici */  
    menu_ask([pointues,tombant  
es,inexistantes], Y0),  
    assert(knows(oreilles,Y0,X)),  
%      !,  
%      Y0 = Y.
```

```
ask(peau,Y,X):-  
    not(knows(peau,_X)),  
    write('de quoi est couverte la  
peau de '),  
    writeln(X),  
    /* menu_ask s'ajoute ici */  
    menu_ask([poils,plumes,ecaill  
es], Y0),  
    assert(knows(peau,Y0,X)),
```

229

Prolog / conclusion

- Mise en œuvre de la résolution
- Pas d'algorithmique: il y a un unique algorithme pour tout: la résolution)
- Prolog langage très lié à l'IA
 - Vérification de circuits (passé de mode)
 - Reconnaissance de formes (passé de mode)
 - Traitement automatique du langage naturel (actuel)
 - Systèmes experts (actuel pour le prototypage)
 - Bases de données déductives (DataLog, actuel)
 - Représentation des connaissances (actuel)
- Revient à la mode
- Permet d'illustrer le cours de logique

231

Menus numérotés

Nième membre

```
member_n(1, [X|_], X).  
member_n(N0, [_|Ys], X) :-  
    /* verifie qu'on ne dépasse  
pas */  
    N0 > 1,      % comparaison  
    /* diminue */  
    N is N0 - 1,  % calcul avec  
    "is"  
    /* Ys a un élément moins et N  
est plus petit de 1 */  
    member_n(N, Ys, X).
```

```
read_between(Min, Max, Term) :-  
    /* lire un terme */  
    read(Term0),  
    /* vérifier que le terme qu'on  
vient de lire nous convient */  
    read_between1(Min, Max,  
    Term0, Term).
```

ETC

230