IA41

Concepts fondamentaux en Intelligence Artificielle et langages dédiés

CM #2-#3

Architecture des systèmes à base de connaissances et introduction à PROLOG

Fabrice LAURI

utbm

Plan

1. Architecture générale des systèmes à base de connaissances

2. Historique et domaines d'applications de PROLOG

3. Eléments constitutifs d'un programme PROLOG

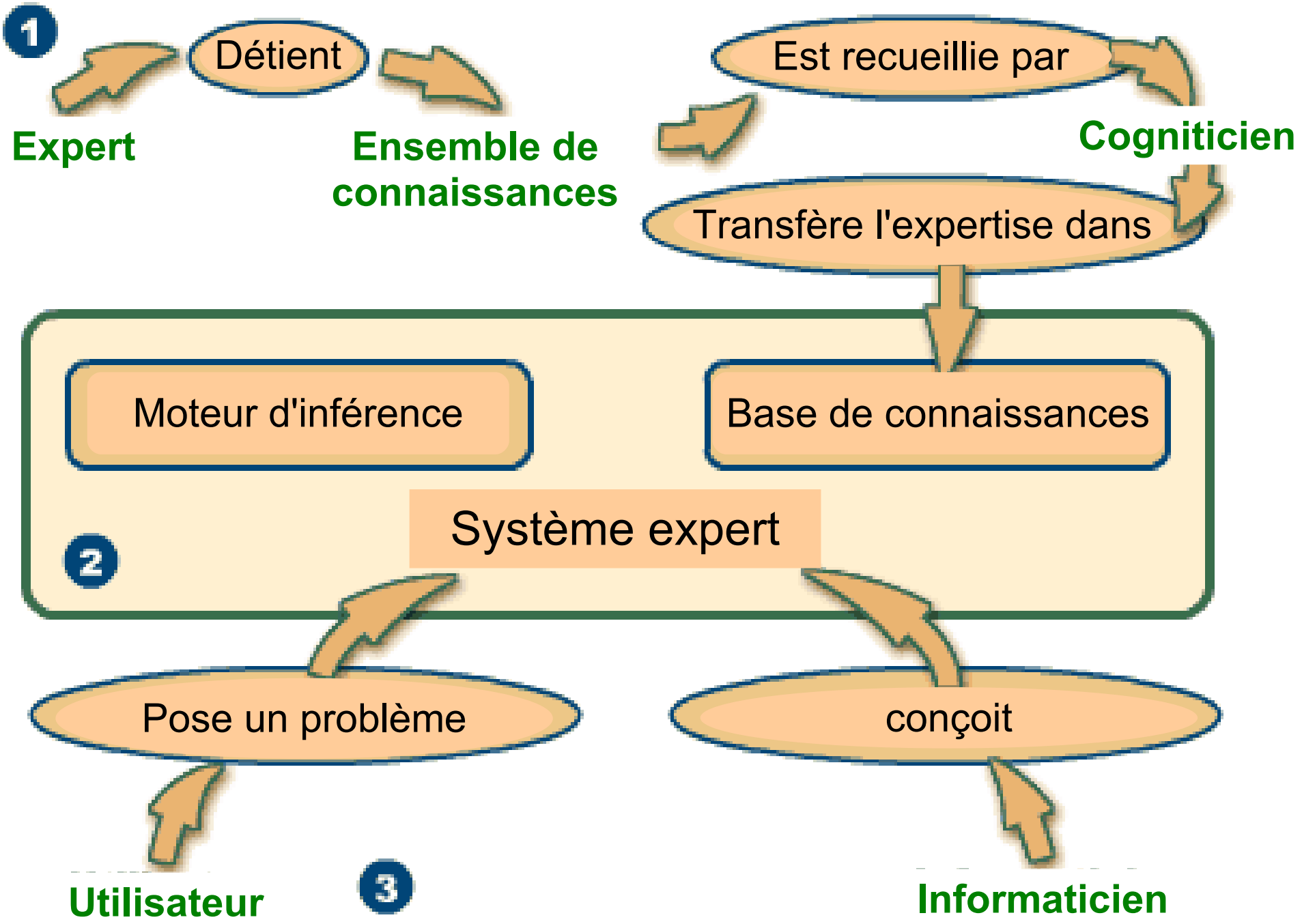
4. Fonctionnement du moteur d'inférences de PROLOG
   Contrôle sur la stratégie de résolution de PROLOG

5. Prédicats prédéfinis en PROLOG

1

Architecture générale des systèmes à base de connaissances

Architecture d'un système expert



2

Historique, introduction et domaines d'applications de PROLOG

Historique

- **1930** : Démonstration automatique par *J. Herbrand*
- **1965** : Algorithme d'unification par *A. Robinson* et *J. Pitrat*
- **1972** : création d'un interpréteur PROLOG par *A. Colmerauer* et *P. Roussel* au Groupe d'Intelligence Artificielle de Luminy à Marseille.
- **1976** : collaboration avec *R. Kowalski*
  - Syntaxe Edimbourg : Quintus, SWI-Prolog, Prolog IV...
  - Syntaxe Marseille : Prolog II, Prolog III, ...
- **1977** : compilateur PROLOG par *D. Warren* à Edimbourg
- **1981** : projets internationaux de développement de l'IA

- **Aujourd'hui** : PROLOG largement répandu dans les universités, les laboratoires de recherche et dans l'industrie.

Communication homme-machine en PROLOG (1972)

Enoncé :

- *Tout psychiatre est une personne.*
- *Chaque personne qu'il analyse est malade.*
- *Jacques est un psychiatre à Marseille.*
- *Est-ce que Jacques est une personne ?*
- *Où est Jacques ?*
- *Est-ce que Jacques est malade ?*

Réponses du système :

- *Oui.*
- *A Marseille.*
- *Je ne sais pas.*

Historique

- **1930** : Démonstration automatique par *J. Herbrand*
- **1965** : Algorithme d'unification par *A. Robinson* et *J. Pitrat*
- **1972** : création d'un interpréteur PROLOG par *A. Colmerauer* et *P. Roussel* au Groupe d'Intelligence Artificielle de Luminy à Marseille.
- **1976** : collaboration avec *R. Kowalski*
  - Syntaxe Edimbourg : Quintus, SWI-Prolog, Prolog IV...
  - Syntaxe Marseille : Prolog II, Prolog III, ...
- **1977** : compilateur PROLOG par *D. Warren* à Edimbourg
- **1981** : projets internationaux de développement de l'IA

- **Aujourd'hui** : PROLOG largement répandu dans les universités, les laboratoires de recherche et dans l'industrie.

Introduction (1/5)

PROLOG est un langage de **programmation déclaratif**.
Il permet de **représenter** et de **manipuler des connaissances** sur un **domaine** particulier.

- **Représentation** = logique des prédicats du premier ordre
- **Domaine** = ensemble d'objets
- **Connaissances** = ensemble de relations entre des objets, pour décrire leurs propriétés, leurs interactions.

Introduction (2/5)

Exemple :

« Jean est le père de Paul » signifie qu'une relation *« est père de »* lie les objets *Jean* et *Paul*.

En PROLOG, on écrirait : *est-pere-de('Jean', 'Paul').*

ou plus simplement : *pere(jean, paul).*

« Qui est le père de Paul ? » revient à chercher si la relation *« est père de »* lie *Paul* à un autre objet qui sera la réponse à la question.

Introduction (3/5)

Programmation déclarative :

- L'utilisateur définit une base de connaissances, c'est-à-dire :

  Faits = énoncés vrais au sujet d'une réalité dans le domaine d'étude.

  Règles = permettent d'établir de nouveaux faits.

- L'interpréteur PROLOG utilise cette base de connaissances pour répondre à des **questions** posées par l'utilisateur.

- **Questions** : permettent de déterminer si un fait est ou peut être établi.

Introduction (4/5)

Faits, règles et questions sont exprimés dans le langage logique des prédicats du premier ordre (PP).

Prédicat : énoncé qui est soit vrai, soit faux.

En tant que langage de programmation, PROLOG comporte des mécanismes de contrôle qui spécifie la **stratégie de résolution** à utiliser pour trouver une solution.

Le programmeur en PROLOG doit connaître ces mécanismes pour pouvoir trouver des solutions rapidement.

Introduction (5/5)

→ l'utilisateur ne spécifie que le « **quoi** » et pas le « **comment** ».

Avantages :

- souplesse et pouvoir d'expression du langage
- rapidité de développement d'un système
- simplification de la maintenance

Inconvénients :

- bonne maîtrise des stratégies de résolution de PROLOG
- expertise dans l'ordre des faits et règles à inoculer au système

Domaines d'applications

PROLOG : langage utilisé dans les domaines de l'**Intelligence Artificielle** et de la **Programmation Logique avec Contraintes**, comme :

- ordonnancement et planification
- systèmes d'aide à la décision
- simulation de processus physiques complexes
- optimisation sous contraintes
- ...

SICStus PROLOG utilisé par :

- NASA (reconnaissance de la parole)
- ERICSSON (conception et configuration de réseaux)

3

Eléments constitutifs d'un programme PROLOG :

- termes et prédicats,
- faits, règles et requêtes
- conjonction, disjonction et négation

Les termes en PROLOG (1/5)

Un **terme** en PROLOG désigne un **objet**, qui peut être :

- Une **constante** (appelée aussi **atome**)

- Une **variable**

- Une **fonction** (ou terme fonctionnel), constitué d'un nom de fonction et d'une liste d'arguments qui sont eux-même des termes.

Les conventions de syntaxe varient selon les implémentations de PROLOG. Nous considèrerons uniquement la syntaxe d'Edimbourg, valable sous *SWI-Prolog* par exemple.

Les termes en PROLOG (2/5)

Avec la syntaxe d'Edimbourg :

- un **atome** peut s'écrire :
  - comme un **identificateur** commençant par une **minuscule**
    Exemples : *jean*, jean*_paul_II*
  - comme un **nombre**
    Exemples : *123*, *1.23*
  - comme une **chaîne de caractères entre apostrophes**
    Exemples : *'René'*, *'Jean-Claude'*
  - comme une **liste**
    Exemples : *[ ], [a, b, 1, 5], ['andré',5,[paris,nancy]]*

Les termes en PROLOG (3/5)

- En PROLOG, la liste vide est notée [ ].
- La liste [e | L] représente une liste dont e est la tête et L le reste de la liste.

Exemples :

```
?- [A | B] = [1,2,3]
        A = 1
        B = [2,3]
?- [A,B | C] = [1,5,10,15]
        A = 1
        B = 5
        C = [10, 15]
```

Les termes en PROLOG (4/5)

Avec la syntaxe d'Edimbourg :

- une **variable** est une chaîne alphanumérique commençant par une **majuscule** ou par **_** :

  Exemples : *X*, *Nom*, *Roi_de_France*, *_N53*

Une variable peut prendre la valeur de n'importe quelle constante apparaissant dans le programme PROLOG.

Les termes en PROLOG (5/5)

- une **fonction** est représenté par une **chaîne alphanumérique** commençant par une **minuscule**, suivie d'une liste de termes entre parenthèses et séparés par une virgule.

Exemple : ***femme(jean)***

La fonction « femme de Jean » est un nouvel objet construit à partir de l'objet jean.

Remarque :

femme/1 signifie que la fonction *femme* est d'arité 1, c'est-à-dire qu'elle n'accepte qu'un seul argument.

Les prédicats (1/3)

Les prédicats s'interprètent soit par VRAI, soit par FAUX.

Ils permettent de représenter des faits, des règles ou des questions :

- Fait : *p(...).*
  Exemple : *pere(jean, paul).*
- Règle : *p(...) :- q(...), ..., r(...).*
  Exemple : *grand-pere(X,Y) :- pere(X,Z), pere(Z,Y).*
- Question : *s(...), ..., t(...).*
  Exemples : *pere(herve,X).*
  mere(X, annie).

Les faits et règles correspondent aux données d'un programme PROLOG, les questions correspondent aux buts.

Les prédicats (2/3)

Les faits correspondent aux données d'un programme PROLOG.

Exemple bibliographique :

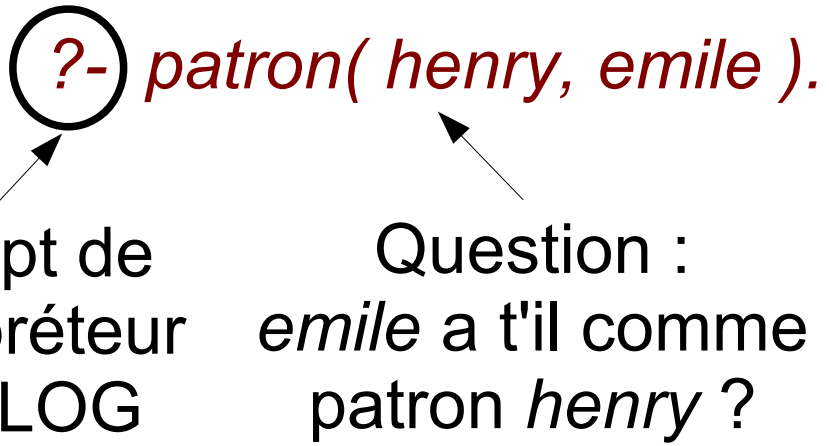
```
/* Commentaire : bibliothèque personnelle */
livre( 'Hugo', 'Hermani', edition( gallimard, 1975 ) ).
livre( 'Hugo', 'Les misérables', edition( poche, 1984 ) ).
livre( 'Giannesini', 'Prolog', edition( 'InterEditions', 1985 ) ).
bibliothecaire( emile ).
caissier( joseph ).
patron( henry, emile ).
patron( henry, joseph ).
```

Toute déclaration d'un fait doit se terminer par un point.

Les prédicats (3/3)

Les prédicats simples peuvent également servir à élaborer des questions.

Exemple :



Ici, on interroge le système pour savoir si ce fait existe dans la base. C'est une question fermée (aucune variable), qui a comme réponse VRAI ou FAUX.

D'où : *?- patron(henry, emile).*

yes

La conjonction

Le **ET logique** (« *Henry est le patron d'Emile **ET** de Joseph* ») est noté par une **virgule** en PROLOG.

Une conjonction est vraie si toutes ses composantes, évaluées de gauche à droite, sont vraies.

Exemple : *?- patron(henry, emile), patron(henry, joseph).*

Ici, on veut savoir si *henry* est le patron d'*emile* et de *joseph*.

Les variables (1/7)

L'usage des variables en PROLOG permet de transformer des questions fermées en questions ouvertes, c'est-à-dire en questions susceptibles d'avoir plusieurs réponses possibles.

Exemples :

?- *patron(X, emile)*.
?- *patron(henry, X)*.

Pour répondre à ces questions, l'interpréteur PROLOG doit remplacer la variable X par une des constantes apparaissant précédemment dans le programme, afin que le fait résultant existe dans la base de faits.

Les variables (2/7)

?- *patron(henry, X)*.

Dans ce cas, *emile* et *joseph* sont de telles constantes. Ainsi :

?- *patron(henry, X)*.

X = emile ;

X = joseph ;

no

L'interpréteur délivre une seule solution à la fois. Pour avoir une autre solution éventuelle, l'utilisateur doit taper ***;***.

no indique que toutes les possibilités ont été épuisées.

Les variables (3/7)

?- *patron(X, joseph)*.

X = henry ;

no

?- *patron(X, Y)*.

X = henry, Y = emile ;

X = henry, Y = joseph ;

no

Les variables peuvent aussi être utilisées pour décrire des faits. Par exemple, *patron(xavier, X)*. serait une transcription de la phrase : *« Xavier est le grand patron. »* et de la formule logique : ∀x patron(xavier, x).

Les variables (4/7)

Utilisée dans une conjonction, une variable établit une relation entre les prédicats qui la composent.

Exemple : *?- patron(henry,X), bibliothecaire(X).*

signifie : « De quel(s) bibliothécaire(s) henry est-t'il le patron ? »

La réponse est calculée en satisfaisant successivement les 2 parties de la conjonction.

1er but partiel : *patron(henry,X)* → { X = emile }
2ème but partiel : *bibliothecaire(emile)* → yes
D'où : X = emile
...

Les variables (5/7)

Notion de Retour arrière :

Si une deuxième réponse est demandée, le système cherche d'abord à satisfaire une seconde fois le second but partiel *bibliothecaire(emile)*. Dans notre cas, il n'y a pas d'autre moyen de le satisfaire.
Le système revient alors à *patron(henry,X)* : c'est ce que l'on nomme un retour arrière ou *backtracking*.
Il trouve X = joseph, puis bibliothecaire(joseph) → no
Le message *no* sera alors affiché.

Les variables (6/7)

Variables anonymes :

Elles se comportent comme des variables ordinaires dans la recherche des correspondances, mais elles ne prennent jamais de valeurs et n'apparaissent pas dans les réponses.
Notation : _ (*underscore*)

Exemples : *?- livre('Hugo', _, _).*
yes

La base de fait contient effectivement au moins un livre écrit par Hugo.

Les variables (7/7)

Variables anonymes (suite et fin) :

Alors que : *?- livre('Hugo', X, Y).*

X = 'Hermani', Y = edition(gallimard, 1975) ;

X = 'Les misérables', Y = edition(poche, 1984) ;

no

?- livre(A, _, _).

A = 'Hugo' ;

A = 'Giannesini' ;

no

Remarque : différentes occurrences du trait de soulignement _ désignent des variables anonymes différentes.

La disjonction (1/3)

PROLOG accepte l'usage de disjonctions et de négation dans le corps des règles et dans les questions, c'est-à-dire dans les buts à satisfaire.

Le **;** représente le connecteur OU.

Exemple :

Formule logique : $\exists x$ patron(henry,x) $\vee$ bibliothecaire(x)
En PROLOG : *?- patron(henry,X) ; bibliothecaire(X).*
En clair : *Existe-t'il quelqu'un dont Henry est le patron ou qui est bibliothécaire ?*

La disjonction (2/3)

Quand un but contient une disjonction, l'interpréteur cherche d'abord à satisfaire la partie gauche et s'il n'y réussit pas, il cherche ensuite à satisfaire la/les partie(s) droite(s).

Exemple :

?- patron(henry,X) ; bibliothecaire(X)

X = emile
X = joseph
} Premier prédicat satisfait 2 fois

X = emile → Deuxième prédicat satisfait 1 fois

no

La disjonction (3/3)

Lorsqu'un fait ou une règle contient une disjonction, il est d'usage d'écrire le fait ou la règle en deux lignes, le premier terme de la disjonction apparaissant sur la première ligne, le deuxième terme sur la deuxième ligne.

Exemple :

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

Les règles (1/10)

Une règle permet de généraliser la manière d'exprimer une connaissance. Elle permet à PROLOG de réaliser des déductions et d'établir ainsi de nouveaux faits.

Exemple : Base de données familiale

OU

mere('Marie', 'Stéphanie').
mere('Marie', 'Alyscia').
mere('Stéphanie', 'Téhane').

pere('Jean-Louis', 'Stéphanie').
pere('Daniel', 'Alyscia').
pere('Julien', 'Téhane').

enfant('Stéphanie', 'Marie').
enfant('Alyscia', 'Marie').
enfant('Téhane', 'Stéphanie').

enfant('Stéphanie', 'Jean-Louis').
enfant('Alyscia' ,'Daniel').
enfant('Téhane', 'Julien').

Les règles (2/10)

Pour introduire la notion de *grand-pere* (ou *grand-parent*), on peut :

- soit définir de nouveaux faits,
- soit définir une règle.

Première solution : nécessite une plus grande place mémoire ainsi que plus de temps d'écriture que la deuxième solution.

La relation grand-pere (ou grand-parent) peut se déduire des relations pere et mere (respectivement enfant).

Les règles (3/10)

« X est un grand-parent de Y s'il existe une personne Z dont Y est l'enfant et qui est elle-même enfant de X. » se traduit par :

grand-parent(X, Y) :- enfant(Y, Z), enfant(Z, X).

se lit : « si » ou « à condition que »

Les règles (4/10)

« X est le grand père de Y si ((X est le père de Z ET Z est le père de Y) OU (X est le père de Z ET Z est la mère de Y)) » se traduit par 2 règles en PROLOG :

grand-pere(X, Y) :- pere(X, Z), pere(Z, Y).
grand-pere(X, Y) :- pere(X, Z), mere(Z, Y).

Les règles (5/10)

Forme générale d'une règle : les clauses

$$C :- H_1, H_2, ..., H_n.$$

se lit : C si H_1 et H_2 et ... et H_n

avec C, H_1, H_2, ..., H_n des prédicats.

- C (partie gauche) représente la **tête de la règle**, ou **conclusion**. Elle est obligatoirement présente.
- H_1, H_2, ..., H_n (partie droite) représente le **corps de la règle**, c'est-à-dire les contraintes liées par l'opérateur de conjonction. Le corps de la règle comprend 0, 1, ou plusieurs termes. Ce sont les **prémisses** de la règle, qui doivent être vérifiées pour que la tête de la règle soit vraie.

Les règles (6/10)

Analogie avec les clauses de Horn en logique

C si $(H_1 \wedge H_2 \wedge ... \wedge H_n)$

équivaut à $(H_1 \wedge H_2 \wedge ... \wedge H_n) \Rightarrow C$

ou $\neg H_1 \vee \neg H_2 \vee ... \vee \neg H_n \vee C$: clause de Horn

Un programme PROLOG ressemble alors à une suite de clauses de Horn :

$\neg H_1 \vee \neg H_2 \vee ... \vee \neg H_n \vee C$

$\neg L_1 \vee \neg L_2 \vee ... \vee \neg L_m \vee D$

...

$\neg T_1 \vee \neg T_2 \vee ... \vee \neg T_o \vee Q$

Les règles (7/10)

Analogie avec une déclaration de procédure dans un langage de programmation

Une règle PROLOG peut également s'apparenter à une déclaration de procédure dans un langage de programmation.
A chaque fois que la tête d'une règle apparaît, elle est remplacée par le corps de la règle correspondant.

Exemple : Soit *grand-parent(X, Y) :- enfant(Y, Z), enfant(Z, X).*

?- grand-parent('Jean-Louis', 'Téhane').
sera transformé en :
?- enfant('Téhane', Z), enfant(Z, 'Jean-Louis').

Les règles (8/10)

Remarques

- Une règle dont la tête est vide est un but qui s'exécute automatiquement.
- Un fait est une clause sans corps. C'est une règle sans prémisses, c'est-à-dire qu'elle est vraie dans tous les cas puisqu'il n'y a pas de conditions préalables :
  - *C.* se lit donc : « C est vrai »
- Les règles sont les énoncés les plus généraux en PROLOG :
  - Règle : *C :- H_1, H_2, ..., H_n.* se lit « C est vrai si ($H_1 \wedge H_2 \wedge ... \wedge H_n$) est vraie »
  - Question : *?- H_1, H_2, ..., H_n.* se lit « ($H_1 \wedge H_2 \wedge ... \wedge H_n$) est-elle vraie ? »

Les règles (9/10)

Remarques (suite)

- On dit que toutes les variables associées à une règle sont **quantifiées universellement (∀) dans la tête de règle**.
- Par contre, toutes les variables apparaissant dans le **corps** d'une règle et n'apparaissant pas dans sa tête sont **quantifiées existentiellement (∃)**.

Ainsi, la règle :

grand-parent(X, Y) :- enfant(Y, Z), enfant(Z, X).

s'interprète comme :

$\forall X\ \forall Y\ \exists Z,\ enfant(Y, Z) \wedge enfant(Z, X) \Rightarrow grand\text{-}parent(X, Y)$

Les règles (10/10)

Remarques (suite et fin)

- Les **variables** apparaissant dans une **question** sont liées par un **quantificateur existentielle**.

Ainsi, la règle :

?- enfant('Téhane', X).

s'interprète comme :

∃x enfant('Téhane', x).

Autrement dit, existe-t'il un (ou plusieurs) X tel que 'Téhane' soit l'enfant de X ?
En clair : de qui Téhane est l'enfant ?

Les règles récursives

La plupart des problèmes sur les listes sont résolus par des règles mettant en jeu la récursivité.

Exemple : fonction d'appartenance d'un élément à une liste

element-de(X, [X | _]).
element-de(X, [_ | Y]) :- element-de(X, Y).

Tout prédicat défini par une règle récursive doit évidemment posséder au moins un cas trivial (appel terminal). Dans le cas contraire, le programme bouclerait.
Tout cas trivial d'une règle récursive doit apparaître avant le(s) appel(s) récursif(s).

La négation (1/3)

En PROLOG, la négation s'écrit comme un prédicat à un argument dont le nom est *not* et l'argument le prédicat à nier.
Par exemple, si f est une formule, sa négation est notée *not(f).*

Remarque :

PROLOG pratique la **négation par l'échec** : pour démontrer *not(f)*, PROLOG tente de démontrer f. Si la démonstration de f échoue, PROLOG considère que *not(f)* est démontrée.

Autrement dit, *not(f)* réussit lorsque :

- f est faux, ou
- f n'est pas déductible, c'est-à-dire **inexistant** dans la base.

Hypothèse du monde clos : tout ce qui n'est pas connu est faux.

La négation (2/3)

Des conclusions douteuses peuvent alors parfois être émises...

Exemple #1 : Soit la base de connaissance suivante :

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

A la question :

?- femme(adam).

l'interpréteur PROLOG retournera : **yes !?!**

La négation (3/3)

Exemple #2 : en considérant la base bibliographique :

?- not(patron(arsene, emile))

yes

?- not(patron(X, emile))

no

Un prédicat nié est considéré comme VRAI s'il n'y a pas moyen de démontrer le prédicat débarassé de la négation.
Ce point de vue pratique n'est pas celui de la logique classique.

En PROLOG, *not(f)* signifie donc que la formule f n'est pas démontrable (n'a pas pu être prouvée).

La portée des identificateurs

Les identificateurs employés en PROLOG pour les constantes, les variables, les prédicats ou les fonctions sont choisis librement par le programmeur en respectant la syntaxe qui diffère selon les implémentations de PROLOG (syntaxe Marseille vs. syntaxe d'Edimbourg).

- La portée d'une variable s'étend à l'énoncé (fait, règle ou question) dans lequel il figure.

- La portée d'une constante, fonction et prédicat s'étend à tout le programme.

Un exemple de programme PROLOG

Menus d'un restaurant (1/3)

« Un restaurant dispose :
- comme hors d'oeuvre : d'*Artichauts Mélanie*, de *Truffes sous le sel* et de *Crudités*.
- comme viande : des *Grillades de boeuf* et du *Poulet au tilleul*
- comme poisson : du *Bar aux algues* et du *Thon à la sauce aigre-douce*
- comme dessert : *Peche Melba*, *Poire Belle-Hélène* et *Ile flottante*

Un menu dans ce restaurant est composé d'un hors d'oeuvre, d'un plat à base de viande ou de poisson et d'un dessert. »

Comment représenter ces connaissances en PROLOG ?

Menus d'un restaurant (2/3)

hors_d_oeuvre('Artichauts Mélanie').
hors_d_oeuvre('Truffes sous le sel').
hors_d_oeuvre('Crudités').

viande('Grillade de boeuf').
viande('Poulet au tilleul').

poisson('Bar aux algues').
poisson('Thon à la sauce aigre-douce').

dessert('Peche Melba').
dessert('Poire Belle-Hélène').
dessert('Ile flottante').

Faits

Menus d'un restaurant (3/3)

```
plat( X ) :- viande( X ).
plat( X ) :- poisson( X ).

repas( X, Y, Z ) :- hors_d_oeuvre( X ),
                    plat( Y ),
                    dessert( Z ).
```

Règles

Exemples de requêtes :

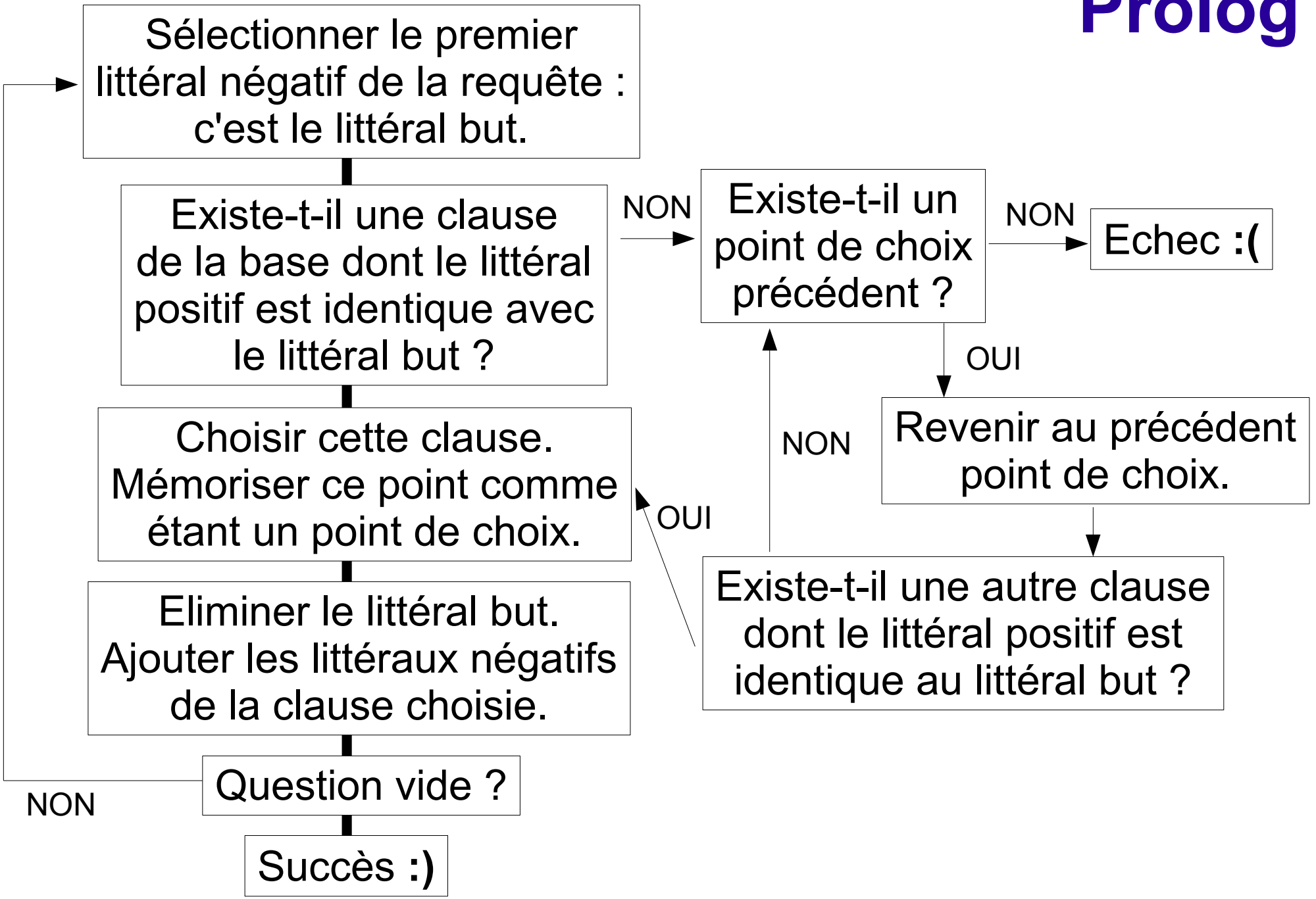
```
?- dessert( X ).
?- repas( X, Y, Z ).
?- repas( 'Truffes sous le sel', X, Y ).
?- repas( X, Y, Z ), viande( Y ).
```

4

Moteur d'inférences

- algorithme simplifié de résolution de Prolog
- exemple de démonstration de requêtes
- retour-arrière et points de choix
- résolution = parcours d'arbre ET-OU
- substitution et unification
- algorithme de résolution
- exemple avec variables

Algorithme simplifié de résolution de Prolog



Démonstration de requêtes

Soit le programme (base de faits) suivant :

titi :- toto.
tutu.
toto.

Supposons que la question soit :

?- titi, tutu.

Retour-arrière et points de choix

Soit le programme (base de faits) suivant :

titi :- toto.
titi.

Supposons que la question soit :

?- titi.

Dernier exemple sans variables

Soit le programme (base de faits) suivant :

titi :- toto, tutu.
titi :- tata, toto.
tata :- toto.
tata :- tutu.
toto.

Supposons que la question soit :

?- titi.

Cette requête peut être résolue par un parcours dans un arbre ET-OU...

Cas des requêtes avec variables

Répondre à une question comportant des variables consiste à déterminer les valeurs des variables telles que la question soit déductible des faits et règles du programme.

Pour démontrer une question :

- mécanisme de ***substitution***
- ***unification*** de chaque terme de la question avec l'une des clauses du programme

Substitution

Définition :

Une substitution σ est une fonction de l'ensemble des variables dans l'ensemble des termes. Mσ indique que σ est appliqué à M.

Exemples :

- σ = { X=Y ; Z=f(A,Y) } est la substitution qui remplace la variable X par Y et la variable Z par f(A,Y) et laisse inchangées toutes les autres variables.

- Si l'on définit la règle :

  soeur(X,Y) :- femme(X), enfant(X,Z), enfant(Y,Z), X \= Y.

  avec \= le prédicat de différence.

  Soit σ = { X='Téhane' ; Y='Alyscia' }.

  Alors soeur(X,Y)σ donne soeur('Téhane','Alyscia').

Unification (1/3)

Définition :

Procédé par lequel on essaie de rendre deux formules identiques en donnant des valeurs aux variables qu'elles contiennent.

En d'autres termes, on dit qu'une substitution σ unifie M et P ssi $M\sigma = P\sigma$.

Exemple :

– *Si l'on définit la règle :*

soeur(X,Y) :- femme(X), enfant(X,Z), enfant(Y,Z), X \= Y.

avec \= le prédicat de différence.

– Lorsque l'on pose la requête :

?- soeur('Téhane', Qui).

PROLOG tente d'unifier cette requête avec la clause *soeur(X,Y).*

Unification (2/3)

Le **résultat** d'une unification est un **unificateur** (ou substitution), c'est-à-dire un ensemble d'affectations de variables.

Exemple d'unification de *soeur('Téhane', Qui)* avec *soeur(X, Y)* :
{ X='Téhane', Y=Qui }

Remarques :

- Le résultat n'est pas forcément unique.
- L'unification peut échouer :

  Exemple : *e(X, X)* et *e(2, 3)* ne peuvent être unifiés.

Unification (3/3)

Le prédicat d'unification en PROLOG est le « = ».

Exemples :

- *?- a(B,C) = a(2,3).* donne pour résultat :
  { B=2, C=3 }

- *?- a(X,Y,L) = a(Y,2,carole).* donne pour résultat :
  { X=2, Y=2, L=carole }

- *?- a(X,X,Y) = a(Y,u,v).* donne pour résultat :
  no

Algorithme de résolution (1/2)

Soit le but G_1, G_2, ..., G_n à résoudre et un programme P.

Cinq étapes sont nécessaires pour obtenir une réponse :

1. Choix du sous-but G_i. En général, on choisit le plus à gauche.
2. Choix d'une clause C :- H_1, ..., H_m de P telle que C s'unifie avec G_i.

   En général, le programme P est parcouru de haut en bas pour rechercher la clause.

   Notons σ l'unificateur de C et G_i.
3. Réécriture de G_1,...,G_n en leur appliquant et en remplaçant G_i par H_1, ..., H_m.

   Le but G_i vient d'être effacé.

Algorithme de résolution (2/2)

4. Recommencer les étapes 1. à 3. de manière récursive avec les nouveaux sous-buts.

5. Si l'exécution de cette nouvelle liste de sous-buts termine avec succès, alors la réponse à la liste originale de buts est un succès.
   Sinon, abandonner la nouvelle liste de buts, revenir au dernier point de choix (*backtracking*) et recommencer à l'étape 1.

Au départ, il n'y a qu'un seul but : la question posée. On aboutit à un succès si la liste des buts devient vide. La réponse donnée par PROLOG est alors la composition des unificateurs successifs.

Résolution d'une requête avec variables

On considère le programme suivant :

p(X,Y) :- q(X), r(X,Y).
p(c,c1).
q(a).
q(b).
r(a,a1).
r(a,a2).
r(b,b1).
r(b,b2).
r(b,b3).

Quelles sont les solutions du but *p(Z,T)* ?
Construisons l'arbre ET-OU de résolution...

Notion de coupure : comment contrôler la stratégie de résolution de PROLOG ?

La coupure en PROLOG (1/5)

- La coupure (ou *cut*) est notée !.
- Elle s'efface (réussit) toujours.
- Permet au programmeur de contrôler l'exécution de ses programmes :
    - en élaguant certaines branches de l'arbre de résolution.
    - en rendant les programmes **déterministes**, plus simples et plus efficaces

Fonctionnement :

- Coupe toutes les branches **ET** qui sont à sa gauche et qui dépendent du même noeud que lui
- Coupe toutes les branches **OU** au même niveau que lui

La coupure en PROLOG (2/5)

Exemple :

```
titi(X) :- toto(X), !, tata(X).
titi(3).
toto(1).
toto(2).
tata(2).
```

Requête :

```
?- titi( Y )
```

La coupure en PROLOG (3/5)

Attention : ! peut modifier la sémantique d'un prédicat

- La coupure est **verte** si la sémantique du prédicat est inchangée

  Exemple :

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

- La coupure est **rouge** si la sémantique du prédicat est modifiée

  Exemple : prédicat *element* tel que :

  ```
  element( X,[X | _] ) :- !.
  element( X,[_ | R] ) :- element( X,R ).
  ```

  avec la requête : *element(X,[a,b,c]).*

La coupure en PROLOG (4/5)

Exemple : on considère encore le programme suivant :

```
p( X,Y ) :- q( X ), r( X,Y ).
p( c,c1 ).
q( a ).
q( b ).
r( a,a1 ).
r( a,a2 ).
r( b,b1 ).
r( b,b2 ).
r( b,b3 ).
```

Quelles sont les solutions du but *p(Z,T)* en fonction de l'endroit où l'on introduit une coupure dans le corps de la clause p(X,Y) ?

La coupure en PROLOG (5/5)

- Que répond PROLOG aux questions suivantes :
  - *member(L,[[a,b],[c,d,e]]), member(X,L).*
  - *member(L,[[a,b],[c,d,e]]), member(X,L), !.*
  - *member(L,[[a,b],[c,d,e]]), !, member(X,L).*
  - *!, member(L,[[a,b],[c,d,e]]), member(X,L).*

- Comment exprimer un **Si-Alors-Sinon** ?

- Comment s'exprime la négation ***not*** /1 à l'aide d'une coupure et du prédicat ***fail*** (qui est toujours évalué à faux) ?

5

Prédicats prédéfinis, définition d'opérateurs et opérations d'entrée/sortie

Prédicat d'unification

Prédicat d'unification : = */2*

X = Y réussit s'il est possible d'unifier le terme X avec le terme Y afin que X soit égal à Y. Echoue dans le cas contraire.

Exemples :

?- a = a.	***yes***
?- X = Y, X = abc.	***X = abc, Y = abc.***
?- f(X,Y) = f(1,Z), Y=5+3.	***X = 1 Z = 5+3 Y = 5+3***
?- f(X,Y) = f(Z,Z), X = 2, Y = 3.	***no***

Prédicat de non-unification

Prédicat de non-unification : \= /2

X \= Y réussit si la contrainte T1 = T2 est insoluble, autrement dit s'il n'y a pas moyen d'unifier T1 avec T2. Echoue dans le cas contraire.

Exemples :

?- a \= a.	***no***
?- a \= 1.	***yes***
?- f(X) \= f(1).	***no***
?- f(X, Y) \= f(Z, Z),X=2,Y=3.	***no***
?- X=2,Y=3, f(X, Y) \= f(Z, Z).	***X = 2, Y = 3.***

Prédicat d'échec

Prédicat d'échec : fail /1

Ce prédicat s'efface toujours et retourne No.

Exemples :

?- fail. ***No***

?- a \=1, fail. ***No***

Prédicat d'affectation numérique

Prédicat d'évaluation directionnelle : is /2

T is E pose l'équation T = v, où v est le résultat de l'évaluation du terme E.

Exemples :

?- X = 2+3.	***X = 2+3***
?- X is 2+3.	***X = 5***
?- 1 is 0.5+0.5.	***no***
?- 1.0 is 0.5+0.5.	***yes***
?- X is 0.5+0.5.	***X = 1.0***
?- Y=10, X is Y+2.	***Y = 10, X=12***

Prédicats de comparaisons numériques (1/2)

Comparaison arithmétique :

=:= /2, **=\=** /2, **<** /2, **=<** /2, **>** /2, **>=** /2

Après évaluation de T1 donnant V1 et de T2 donnant V2, on a les fonctionnalités :

T1 **=:=** T2	réussit ssi V1 égale V2
T1 **=\=** T2	réussit ssi V1 diffère de V2
T1 **<** T2	réussit ssi V1 inférieur strictement à V2
T1 **=<** T2	réussit ssi V1 inférieur ou égal à V2
T1 **>** T2	réussit ssi V1 supérieur strictement à V2
T1 **>=** T2	réussit ssi V1 supérieur ou égal à V2

Prédicats de comparaisons numériques (2/2)

Exemples :

?- 1 =:= 2 .	***no***
?- 1 =:= 1 .	***yes***
?- 1 =:= 1.0.	***yes***
*?- 3*2.0 =:= 12.0 / 2 .*	***yes***
?- 5.0 =< 24 .	***yes***

Quelques prédicats arithmétiques (1/2)

// /2	division entière
mod /2	modulo
abs /1	valeur absolue
sqrt /1	racine carré
sign /1	signe du terme
floor /1	plus grand des entiers inférieurs
truncate /1	partie entière
round /1	entier le plus proche
ceiling /1	plus petit des entiers supérieurs
log /1	logarithme (base *e*)
exp /1	exponentielle

Quelques prédicats arithmétiques (2/2)

float /1	réussit si le terme est un flottant
integer /1	réussit si le terme est un entier
number /1	réussit si le terme est un nombre
atomic /1	réussit si le terme est atomique
sin /1	
cos /1	
pi	constante prédéfinie
** /2	fonction puissance

Gestion des clauses d'un programme

listing /0	affichage de l'ensemble des clauses
dynamic /2	rend dynamique un bloc de clauses
assert /1	insertion d'une clause
asserta /1	insertion d'une clause en tête du programme
assertz /1	insertion d'une clause à la fin du programme
retract /1	effacement d'une clause
abolish /2	effacement d'un bloc de clauses
module /2	définition d'un module
use_module /1	utilisation d'un module

Gestion des réponses d'un prédicat (1/6)

Soit la base de connaissances suivante :

habite(jean, belfort).
habite(lucie, paris).
habite(christian, toulouse).
habite(adeline, paris).
habite(nicolas, paris).

On désire construire, à l'aide de requêtes, les listes contenant :
- toutes les personnes habitant à Paris,
- toutes les villes spécifiées dans la base.

Gestion des réponses d'un prédicat (2/6)

Pour constituer une liste à partir des réponses d'une requête, il existe trois prédicats :

- findall /3

- bagof /3

- setof /3

Gestion des réponses d'un prédicat (3/6)

- ***findall(+X, +Requete, -Liste)*** stocke dans *Liste* toutes les substitutions de X permettant de démontrer *Requete*.
  Si *Requete* n'a pas de solution, *findall* retourne [].
- ***bagof(+X, +Requete, -Liste)*** se comporte comme *findall*, à la différence que :
  - *bagof* échoue si *Requete* n'a pas de solution
  - a un comportement différent concernant les variables libres de *X* (voir plus loin)
- ***setof(+X, +Requete, -Liste)*** se comporte comme *bagof*, à la différence que :
  - il y a suppression des doublons
  - les résultats sont triés en utilisant le prédicat sort /2

Gestion des réponses d'un prédicat (4/6)

^ *spécifie dans bagof* /3 et *setof* /3 les variables qui doivent rester libres (ne pas être unifiées).

Var^But permet de spécifier à *bagof* et *setof* que la variable *Var* ne doit pas être unifiée avec les différentes valeurs retournées par *But*.

findall considère que toutes les variables de *But* sont spécifiées comme étant libres.

Gestion des réponses d'un prédicat (5/6)

Exemple :

habite(jean, belfort).
habite(lucie, paris).
habite(christian, toulouse).
habite(adeline, paris).
habite(nicolas, paris).

?- findall(X, habite(X,paris), L).
L = [lucie, adeline, nicolas]

?- setof(X, Y^habite(Y,X), L).
L = [belfort, paris, toulouse]

Gestion des réponses d'un prédicat (6/6)

Exemple :

test(a, b, c).
test(a, b, f).
test(b, c, e).
test(b, c, d).
test(c, c, g).

?- bagof(C, test(A,B,C), L).
?- bagof(C, A^test(A,B,C), L).
?- bagof(C, A^B^test(A,B,C), L).
?- bagof(C, A^B^C^test(A,B,C), L).
?- setof(C, A^B^C^test(A,B,C), L).
?- findall(C, test(A,B,C), L).

Les opérateurs (1/6)

- PROLOG dispose des opérateurs arithmétiques courants, tels que +, -, * et /, en **notation infixée** : 1 **+** 2, 8 ***** (5 **-** 3)...

- PROLOG permet aussi à l'utilisateur de définir de nouveaux opérateurs, et de modifier ainsi sa syntaxe.

Un opérateur est défini par :
- son nom
- sa priorité (entre 0 et 1200)
- son type (infixé | préfixé | postfixé) et son caractère associatif (à gauche | à droite | non associatif)

Les opérateurs (2/6)

Trois types d'opérateurs :

- Opérateur **binaire infixé** : il figure entre ses deux arguments et désigne un arbre binaire.

  Exemple : X **+** Y

- Opérateur **unaire préfixé** : il figure avant son argument et désigne un arbre unaire

  Exemple : **-**X

- Opérateur **unaire postfixé** : il figure après son argument et désigne un arbre unaire

  Exemple : X**^2**

Les opérateurs (3/6)

Associativité des opérateurs :

- A gauche :

  Exemple : *X op Y op Z* est lu comme *(X op Y) op Z*

- A droite :

  Exemple : *X op Y op Z* est lu comme *X op (Y op Z)*

- Non associatif : les parenthèses sont obligatoires

  Exemple : la syntaxe *X op Y op Z* est interdite

Les opérateurs (4/6)

Déclaration des opérateurs :

En enregistrant une nouvelle règle sans tête, comme :

:- op(Priorité, Spécification, Nom).

où :

- Priorité : comprise entre 0 (plus forte priorité) et 1200
- Spécification : type et caractère associatif de l'opérateur
- Nom : nom de l'opérateur

Les opérateurs (5/6)

Tableau des spécificateurs (types et associativité) :

Spécificateur	Type	Associativité
fx	préfixe	non associatif
fy	préfixe	associatif à droite
xfx	infixe	non associatif
xfy	infixe	associatif à droite
yfx	infixe	associatif à gauche
xf	postfixe	non associatif
yf	postfixe	associatif à gauche

Les opérateurs (6/6)

Exemple de déclaration d'un opérateur :

:- op(1000, xfx, aime) : définit un opérateur infixé non associatif *aime.*

Dans ce cas, Prolog traduira une expression du type *X aime Y* en le terme *aime(X, Y)*, et si Prolog doit afficher le terme *aime(X, Y)*, il affichera *X aime Y*.

Exercice :

Définissez les opérateurs *et* (conjonction), *ou* (disjonction), *non* (négation) afin de pouvoir écrire : *(fail ou non true et fail)*...

Les entrées/sorties (1/5)

- Ecriture dans un fichier ou affichage à l'écran
- Lecture à partir d'un fichier ou via le clavier

Affichage de termes à l'écran : write/1, nl/0, tab/1 et display/1

write(1+2)	→ 1+2
write(X)	→ _245 (_245 étant par exemple le nom interne de X)
X is 15,write(X)	→ 15
nl	→ retour charriot
tab(N)	→ affiche N espaces
display(3+4)	→ +(3, 4) (représentation sous forme d'arbre)

Les entrées/sorties (2/5)

Lecture d'un terme au clavier : read/1

read/1 lit un terme au clavier et l'unifie avec son argument. Le terme lu doit être obligatoirement suivi d'un point.
Certains interpréteurs PROLOG affichent un prompt lorsque *read/1* est utilisé.

Exemple :

?- read(X).
|: a(1, 2).

→ X = a(1, 2)

Les entrées/sorties (3/5)

Ouverture d'un fichier : open/3

Avant de pouvoir lire ou écrire des termes dans un fichier, celui-ci doit être ouvert en lecture ou en écriture à l'aide d'une instruction *open(<nom>, <mode>, <flux>)* où :

- <nom> : le nom du fichier à ouvrir
- <mode> : mode d'ouverture *write* | *append* | *read*
- <flux> (ou stream) : identificateur du fichier utilisé lors des opérations d'E/S ultérieures

Les entrées/sorties (4/5)

Lecture dans un fichier : read/2

read(Flux, X) permet de lire un terme écrit dans le fichier dont l'identifiant est *Flux*.

Ecriture dans un fichier : write/2

write(Flux, X) permet d'écrire le terme X dans le fichier dont l'identifiant est *Flux*.

nl(Flux) permet d'écrire un retour chariot dans le fichier *Flux*.

Fermeture d'un fichier : close/1

close(Flux) permet de fermer le fichier dont l'identifiant est *Flux*.

Les entrées/sorties (5/5)

Exemple d'utilisation :

Définition d'une règle ***ecrire*** */1* qui écrit un terme suivi d'un retour chariot dans un fichier nommé « test.txt » :

```
ecrire( T ) :-   open( 'test.txt', append, Flux ),    % ouverture
                 write( Flux, T ), nl( Flux ),        % écriture
                 close( Flux ).                       % fermeture
```