Plan du cours de “Programmation logique”

Socrate meurt-il ?

Connaissance générale :

(1) « Tout homme est un animal » ;

(2) « Tout animal est mortel » ;

(3) « Si un entité mortel est empoisonné alors il meurt » ;

(4) « Si un entité boit du poison alors il est empoisonné ».

Connaissance particulière sur le monde de Socrate :

(5) « Socrate est un homme » ;

(6) « Platon est un homme » ;

(7) « Platon et Socrate sont amis » ;

(8) « Socrate boit la ciguë » ;

(9) « La ciguë est un poison ».

Que peut-on « déduire » de ces connaissances ?

Socrate meurt !

- « Socrate est un homme »(5) et « Tout homme est un animal »(1) donc « Socrate est un animal »(10) ;
- « Tout animal est mortel »(2) donc « Si Socrate est un animal alors Socrate est mortel »(11) ;
- « Si Socrate est un animal alors Socrate est mortel »(11) et « Socrate est un animal »(10) donc « Socrate est mortel »(12) ;
- « Si un entité boit du poison alors il est empoisonné »(4) donc « Si Socrate boit du poison alors il est empoisonné »(13) ;
- « Si Socrate boit du poison alors il est empoisonné »(13) et « La ciguë est un poison »(9) donc « Si Socrate boit de la ciguë alors il est empoisonné »(14) or « Socrate boit la ciguë »(8) donc « Socrate est empoisonné »(15) ;
- « Si un entité mortel est empoisonné alors il meurt »(3) et « Socrate est empoisonné »(15) et « Socrate est mortel »(12) donc « Socrate meurt »...

$meurt(Socrate)$?

Connaissance générale (ontologie) :

- $\forall x(homme(x) \rightarrow animal(x))$
- $\forall x(animal(x) \rightarrow mortel(x))$
- $\forall x((mortel(x) \wedge empoisonne(x)) \rightarrow meurt(x))$
- $\forall x(\forall y((boit(x, y) \wedge poison(y)) \rightarrow empoisonne(x)))$

Connaissance particulière sur le monde de Socrate :

- $homme(Socrate)$
- $homme(Platon)$
- $(ami(Socrate, Platon) \wedge ami(Platon, Socrate))$
- $boit(Socrate, cigue)$
- $poison(cigue)$

$\mathcal{C}$ connaissances générale et particulière.

... mais pas seulement

- $\mathcal{C} \models animal(Socrate)$
- $\mathcal{C} \models animal(Socrate) \rightarrow mortel(Socrate)$
- $\mathcal{C} \models mortel(Socrate)$
- $\mathcal{C} \models \forall y((boit(Socrate, y) \wedge poison(y)) \rightarrow empoisonne(Socrate))$
- $\mathcal{C} \models empoisonne(Socrate)$
- $\mathcal{C} \models \exists x(meurt(x))$ avec $[x \leftarrow Socrate]$
- $\mathcal{C} \models \exists x(\exists y(ami(x, y)))$
  avec $[x \leftarrow Platon][y \leftarrow Socrate]$
  et $[x \leftarrow Socrate][y \leftarrow Platon]$
- $\mathcal{C}, \neg mortel(Zeus) \models \neg homme(Zeus)$

Une formule peut être démontrée comme conséquence logique d'un ensemble de formules (les instances pour les variables existentielles étant calculées).

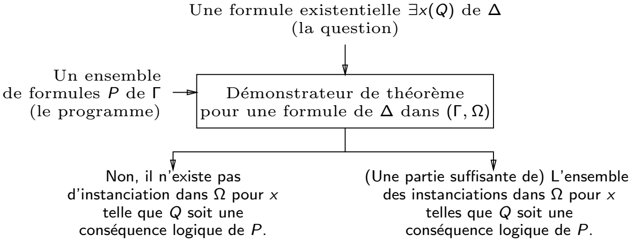
La logique comme langage de programmation

Un langage de programmation logique est défini par :

- un langage des données Ω ;

et deux sous-ensembles de la logique :

- un langage des programmes Γ ;
- un langage des questions Δ .



Programmation = Logique + contrôle

Programmation logique	Programmation impérative
formule	procédure
ensemble (conjonction) de formules	programme
question	appel de procédure
preuve	exécution
substitution (unification)	passage de paramètres

Contrôle : orienter et modifier le déroulement de la preuve.

Les principales caractéristiques des langages de la Programmation Logique

- Logique : le langage est un sous-ensemble de la logique et une exécution est une preuve.
- Symbolique : les données manipulées sont principalement des symboles.
- Déclaratif : le “que faire” plutôt que le “comment faire”.
- Relationnel : un programme logique décrit un état du “monde” en termes de données et de prédicats (relations) entre ces données.
- Indéterministe : le résultat est l’ensemble des données qui vérifient une question dans une description donnée du “monde”.
- Haut niveau : aucune gestion de la mémoire et masquage du caractère impératif de la machine.

Les domaines d'application de la Programmation Logique

- Analyse de la “Langue naturelle” ;
- Intelligence artificielle et modélisation des raisonnements (les Systèmes experts, le Diagnostique, ...) ;
- Bases de données ;
- Prototypage ;
- Compilation ;
- Automates formels déterministes et non-déterministes ;
- Vérification de programmes ; ...

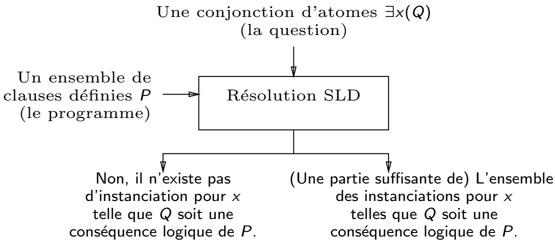
Plan du cours de “Programmation logique”

1. Introduction
2. Programmation Logique
3. Prolog, le langage
4. Applications

Syntaxe de la programmation logique en clauses de Horn

La programmation logique en clauses de Horn est définie par :

- langage des données : le langage des termes,
- langage des programmes : les clauses définies,
- langage des questions : les conjonctions d'atomes.



Termes et Atomes

Un terme est défini sur un ensemble de symboles de fonctions et de constantes :

- si X est une variable alors X est un terme ;
- si c est une constante alors c est un terme ;
- si f est un symbole de fonction et t1, . . ., tn sont des termes alors f(t1,. . .,tn) est un terme.

Un atome est défini sur un ensemble de symboles de prédicats :

- si p est un symbole de prédicats et t1, . . ., tn sont des termes alors p(t1,. . .,tn) est un atome.

Programme en clauses de Horn

Le langage des programmes de la programmation logique en clauses de Horn : l'ensemble des clauses définies.
Une clause définie est constituée dans cet ordre :

- d'une tête de clause (un atome)
- du symbole " :-" ("si")
- d'un corps de clauses, conjonction d'atomes séparés par le symbole ","
- le symbole ".".

- ≪ Socrate est empoisonné si Socrate boit la ciguë et la ciguë est un poison. ≫

$$empoisonne(socrate) :- \\ boit(socrate, cigue), poison(cigue).$$

  $boit(socrate, cigue), poison(cigue)$ est le corps et $empoisonne(socrate)$ est la tête de cette clause définie.
- Si le corps est absent, une clause définie est un fait et est constituée :
  - d'une tête de clause (un atome)
  - le symbole ".".

  ≪ Socrate est un homme. ≫
  $homme(socrate).$
- Un ensemble fini de clauses définies (une conjonction) ayant le même nom de prédicat dans la tête en est sa définition.
- Un programme logique est un ensemble (une conjonction) fini de définitions de prédicats.

Variables logiques

- Une variable logique représente une donnée quelconque mais unique.
  homme(X) signifie que l' "entité X est un homme".
- Une substitution est une fonction (un ensemble de couples $X_i \leftarrow t_i$) des variables dans les termes notée $[X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n]$ (X_i distincte de t_i et X_i toutes distinctes entre-elles).
- La substitution vide (l'identité) est notée ϵ.
- La substitution σ est étendue aux termes (et aux atomes) :
  - si $(X \leftarrow t) \in \sigma$ alors $\sigma(X) = t$ sinon $\sigma(X) = X$;
  - $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.

- Le résultat de l'application d'une substitution σ (constituée de couples $X_i \leftarrow t_i$) à un terme (ou un atome) t, nommée instance de t et notée $\sigma(t)$, est le terme t dont toutes les occurrences des X_i ont été remplacées simultanément par les t_i.
- La substitution est associative mais non commutative.
- La variable logique n'est pas un emplacement mémoire et la substitution n'est pas une affectation.
- Pour $\sigma = [X \leftarrow socrate, Y \leftarrow platon]$ et $t = ami(X, Y)$ alors $\sigma(t) = ami(socrate, platon)$.

Clauses Universelles et variables du programme

- Dans une clause (ou un fait), la variable logique exprime que la clause (ou le fait) est vrai pour n'importe quelle donnée.
  - "Pour toute donnée X, si X est un homme alors X est mortel."
    mortel(X) :- homme(X).
  - "Pour toute donnée X, si X est mortelle et X est empoisonnée alors X meurt."
    meurt(X) :- mortel(X), empoisonne(X).
- Les variables sont locales et génériques (elles sont renommées à chaque utilisation).
- Les variables peuvent aussi bien être utilisée en entrée qu'en sortie.

Le programme de la mort de Socrate

```
animal(X) :- homme(X).
mortel(X) :- animal(X).
meurt(X) :-
  mortel(X), empoisonne(X).
empoisonne(X) :-
  boit(X,Y), poison(Y).
homme(socrate).
homme(platon).
ami(socrate, platon).
ami(platon, socrate).
boit(socrate, cigue).
poison(cigue).
```

Variables logiques

- Une variable logique représente une entité quelconque mais unique.
  homme(X) signifie que l'« entité X est un homme ».
- Une substitution est une fonction des variables dans les termes notée $[X_1 \leftarrow t_1, \ldots, X_n \leftarrow t_n]$
  (X_i distincte de t_i et X_i toutes distinctes entre-elles).
- La variable logique n'est pas un emplacement mémoire et la substitution n'est pas une affectation.
- Le résultat de l'application d'une substitution σ à un terme (ou un atome) t, nommée instance de t et notée $\sigma(t)$, est le terme t dont toutes les occurrences des X_i ont été remplacées simultanément par les t_i.

L'unification

- L'unification calcule une substitution qui rend des termes égaux.
- Soient $E = \{t_1, \ldots, t_n\}$ un ensemble de termes et σ une substitution.
  σ unifie E si par définition $\sigma(t_1) = \ldots = \sigma(t_n)$
  (σ est un unificateur de E et E est unifiable).
- L'unificateur σ d'un ensemble de termes E est l'unificateur le plus général (upg) de E si quelque soit σ' un unificateur de E il existe une substitution η telle que $\sigma' = \sigma\eta$.
- Deux termes unifiables admettent un unique unificateur le plus général (au renommage des variables près).

Clauses Universelles et variables du programme

- Dans une clause (ou un fait), la variable logique exprime que la clause (ou le fait) est vrai pour n'importe quelle entité.
  - "Pour toute entité X, si X est un homme alors X est mortel."
    mortel(X) :- homme(X).
  - "Pour toute entité X, si X est mortelle et X est empoisonnée alors X meurt."
    meurt(X) :- mortel(X), empoisonne(X).
- Les variables sont locales et génériques (elles sont renommées à chaque utilisation).
- Les variables peuvent aussi bien être utilisée en entrée qu'en sortie.
- L'unification est l'unique mécanisme de passage de paramètres.

Questions existentielles

- Dans une question, la variable logique exprime une interrogation sur l'existence d'une entité qui vérifie le prédicat.
  ≪ Existe-t-il une entité X telle que X soit mortelle ? ≫
  $? \ mortel(X)$
- La réponse à une question existentielle est constructive (non uniquement existentielle) et collecte les instanciations pour les variables qui sont des solutions.
  $? \ mortel(X)$
  $[X \leftarrow socrate]$
- La réponse est une conséquence logique du programme et donc vraie pour le monde (correctement) modélisé.

- La question peut être une conjonction d'atomes...

  ```
  homme(socrate).
  homme(platon).
  ami(socrate,platon).
  ami(platon,socrate).
  ```

  ≪ Existe-t-il une entité X et une entité Y telles que X soit ami de Y et X soit un homme ? ≫
  ? $ami(X, Y), homme(X)$

- et la réponse multiple.
  $[X \leftarrow socrate][Y \leftarrow platon]$
  $[X \leftarrow platon][Y \leftarrow socrate]$

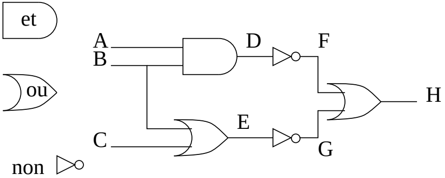
De la réversibilité

- Prolog est un langage relationnel : chaque prédicat décrit une relation.
- Les variables apparaissant dans la tête peuvent être en entrée ou en sortie.

Les circuits logiques

Un circuit logique est décrit par le comportement de ses composants

```
et(0,0,0). et(0,1,0). et(1,0,0). et(1,1,1).
ou(0,0,0). ou(0,1,1). ou(1,0,1). ou(1,1,1).
non(1,0). non(0,1).
```




```
circuit(A,B,C,H) :-
  et(A,B,D), ou(C,B,E), non(D,F), non(E,G), ou(F,G,H).
```

Vérification qu'une instanciation des entrées et du résultat appartient à la relation :

$$? \; circuit(1, 0, 0, 1).$$
oui
$$? \; circuit(1, 0, 0, 0).$$
non

Calcul du résultat du circuit pour une instanciation des entrées :

$$? \; circuit(1, 0, 1, H).$$
$$[H \leftarrow 1];$$
non

Calcul du résultat du circuit pour une instanciation partielle des entrées :

$$? \; circuit(A, 1, 0, H).$$
$$[A \leftarrow 0, H \leftarrow 1];$$
$$[A \leftarrow 1, H \leftarrow 0];$$
non

Calcul d'un ensemble d'entrées pour une instanciation du reste des entrées et du résultat :

$$? \; circuit(1, 1, C, 0).$$
$$[C \leftarrow 0];$$
$$[C \leftarrow 1];$$
non

Calcul des entrées pour une instanciation du résultat :

$$
\begin{array}{l}
?\ \mathit{circuit}(A, B, C, 0). \\
[A \leftarrow 1, B \leftarrow 1, C \leftarrow 0]; \\
[A \leftarrow 1, B \leftarrow 1, C \leftarrow 1]; \\
\mathit{non}
\end{array}
$$

Calcul de la relation complète :

$$
\begin{array}{l}
?\ \mathit{circuit}(A, B, C, H). \\
[A \leftarrow 0, B \leftarrow 0, C \leftarrow 0, H \leftarrow 1]; \\
\ldots \\
[A \leftarrow 1, B \leftarrow 1, C \leftarrow 1, H \leftarrow 0]; \\
\mathit{non}
\end{array}
$$

Algorithme d'unification (Martelli-Montanari)

Initialisation : $\theta_0 = \epsilon$ et $E_0 = \{t_j = t'_j\}_{1 \leq j \leq n}$.
Résultat : l'upg de $\{t_j = t'_j\}_{1 \leq j \leq n}$ s'il existe
Tant que E_i n'est pas l'ensemble vide,

- (1) si $E_i = \{f(s_1, \ldots, s_p) = f(s'_1, \ldots, s'_p)\} \cup E'_i$ alors $E_{i+1} = \{s_j = s'_j\}_{1 \leq j \leq p} \cup E'_i$ et $\theta_{i+1} = \theta_i$;
- (2) si $f(s_1, \ldots, s_p) = g(s'_1, \ldots, s'_m) \in E_i$ avec $f \neq g$ ou $p \neq m$ alors arrêt de la boucle avec échec ;
- (3) si $E_i = \{X = X\} \cup E'_i$ alors $E_{i+1} = E'_i$ et $\theta_{i+1} = \theta_i$;
- (4) si ($E_i = \{t = X\} \cup E'_i$ ou $E_i = \{X = t\} \cup E'_i$), $t \neq X$ et $X \notin V(t)$ alors $E_{i+1} = [X \leftarrow t](E'_i)$ et $\theta_{i+1} = \theta_i[X \leftarrow t]$;
- (5) si ($E_i = \{t = X\} \cup E_i$ ou $E_i = \{X = t\} \cup E_i$), $t \neq X$ et $X \in V(t)$ alors arrêt de la boucle avec échec.

$$E_0 = \{f(X, g(X, b, V), Z) = f(X, Z, g(h(a), U, W))\}, \theta_0 = \epsilon$$

$$\overset{(1)}{\leadsto} \quad E_1 = \{X = X, g(X, b, V) = Z, Z = g(h(a), U, W)\}, \theta_1 = \theta_0$$

$$\overset{(4)}{\leadsto} \quad \left\{ \begin{array}{ll} E_2 & = [Z \leftarrow g(X, b, V)](\{X = X, Z = g(h(a), U, W)\}) \\ & = \{X = X, g(X, b, V) = g(h(a), U, W)\}, \\ \theta_2 & = \theta_1[Z \leftarrow g(X, b, V)] \end{array} \right.$$

$$\overset{(3)}{\leadsto} \quad E_3 = \{g(X, b, V) = g(h(a), U, W)\}, \theta_3 = \theta_2$$

$$\overset{(1)}{\leadsto} \quad E_4 = \{X = h(a), b = U, V = W\}, \theta_4 = \theta_3$$

$$\overset{(4)}{\leadsto} \quad \left\{ \begin{array}{ll} E_5 & = [U \leftarrow b](\{X = h(a), V = W\}) \\ & = \{X = h(a), V = W\}, \\ \theta_5 & = \theta_4[U \leftarrow b] \end{array} \right.$$

$$\overset{(4)}{\leadsto} \quad \left\{ \begin{array}{ll} E_6 & = [V \leftarrow W](\{X = h(a)\}) \\ & = \{X = h(a)\}, \\ \theta_6 & = \theta_5[V \leftarrow W] \end{array} \right.$$

$$\stackrel{(4)}{\rightsquigarrow} \left\{ \begin{array}{ll} E_7 & = [X \leftarrow h(a)](\{\}) \\ & = \{\} \\ \theta_7 & = \theta_6[X \leftarrow h(a)] \end{array} \right.$$

$$\begin{array}{ll} \multicolumn{2}{l}{[Z \leftarrow g(X,b,V)][U \leftarrow b][V \leftarrow W][X \leftarrow h(a)](f(X, g(X,b,V), Z))} \\ = & [U \leftarrow b][V \leftarrow W][X \leftarrow h(a)](f(X, g(X,b,V), g(X,b,V))) \\ = & [V \leftarrow W][X \leftarrow h(a)](f(X, g(X,b,V), g(X,b,V))) \\ = & [X \leftarrow h(a)](f(X, g(X,b,W), g(X,b,W))) \\ = & f(h(a), g(h(a),b,W), g(h(a),b,W)) \end{array}$$

$$\begin{array}{ll} \multicolumn{2}{l}{[Z \leftarrow g(X,b,V)][U \leftarrow b][V \leftarrow W][X \leftarrow h(a)](f(X, Z, g(h(a),U,W)))} \\ = & [U \leftarrow b][V \leftarrow W][X \leftarrow h(a)](f(X, g(X,b,V), g(h(a),U,W))) \\ = & [V \leftarrow W][X \leftarrow h(a)](f(X, g(X,b,V), g(h(a),b,W))) \\ = & [X \leftarrow h(a)](f(X, g(X,b,W), g(h(a),b,W))) \\ = & f(h(a), g(h(a),b,W), g(h(a),b,W)) \end{array}$$

Sémantique opérationnelle

La règle de dérivation SLD :

$$\frac{B_1, \ldots, B_{i-1}, B_i, B_{i+1}, \ldots, B_p}{\sigma(B_1, \ldots, B_{i-1}, \theta(A_1), \ldots, \theta(A_n), B_{i+1}, \ldots, B_p)}\ C, i$$

si $C = (A : -A_1, \ldots, A_n) \in P$,
$V(\theta(A : -A_1, \ldots, A_n)) \cap V(B_1, \ldots, B_p) = \emptyset$,
$\sigma \in upg(\theta(A), B_i)$.
$B_1, \ldots, B_p$ est une résolvante et θ une substitution de renommage.
Deux degrés de liberté :
- le choix de l'atome à réduire ;
- le choix de la clause pour réduire.

L'unification est l'unique mécanisme de passage de paramètres.

- Une dérivation SLD de résolvante initiale R_0, la question, est une suite finie ou infinie $(R_j)_{j \geq 0}$ de résolvantes telle que

$$\frac{R_j}{R_{j+1}}\ C_j, i_j$$

- Une succès ou réfutation SLD d'une résolvante R_0 est une dérivation finie $(R_j)_{0 \leq j \leq r}$ telle que la dernière résolvante est vide.
- Une dérivation telle que la dernière résolvante ne peut plus inférer de nouvelle résolvante est une dérivation échec.

$$\frac{\overbrace{meurt(X)}^{1}}{mortel(X_1), empoisonne(X_1)}\; C_0, 1 \in \{1\}$$

avec $C_0 = meurt(X) : -mortel(X), empoisonne(X).$
et $\theta_0 = [X \leftarrow X_1]$ et $\theta_0(meurt(X)) = meurt(X_1)$
et donc $\sigma_0 = upg(meurt(X_1), meurt(X)) = [X \leftarrow X_1]$
$\sigma_0(\theta_0(mortel(X)), \theta_0(empoisonne(X))) = mortel(X_1), empoisonne(X_1).$

$$\frac{\overbrace{mortel(X_1)}^{1}, \overbrace{empoisonne(X_1)}^{2}}{mortel(X_2), boit(X_2, Y_2), poison(Y_2)}\; C_1, 2 \in \{1,2\}$$

avec $C_1 = empoisonne(X) : -boit(X, Y), poison(Y).$
$\theta_1 = [X \leftarrow X_2][Y \leftarrow Y_2]$ et $\theta_1(empoisonne(X)) = empoisonne(X_2)$
et donc $\sigma_1 = upg(empoisonne(X_2), empoisonne(X_1)) = [X_1 \leftarrow X_2]$
$\sigma_1(mortel(X_1), \theta_1(boit(X, Y)), \theta_1(poison(Y))) =$
$mortel(X_2), boit(X_2, Y_2), poison(Y_2).$

$$\frac{\overbrace{mortel(X_2)}^{1}, \overbrace{boit(X_2, Y_2)}^{2}, \overbrace{poison(Y_2)}^{3}}{animal(X_3), boit(X_3, Y_2), poison(Y_2)} \quad C_2,1 \in \{1,2,3\}$$

avec $C_2 = mortel(X) : -animal(X)$.
et $\theta_2 = [X \leftarrow X_3]$ et $\theta_2(mortel(X)) = mortel(X_3)$
et donc $\sigma_2 = upg(mortel(X_3), mortel(X_2)) = [X_2 \leftarrow X_3]$
$\sigma_2(\theta_2(animal(X)), boit(X_2, Y_2), poison(Y_2)) =$
$animal(X_3), boit(X_3, Y_2), poison(Y_2)$.

$$\frac{\overbrace{animal(X_3)}^{1}, \overbrace{boit(X_3, Y_2)}^{2}, \overbrace{poison(Y_2)}^{3}}{animal(X_3), boit(X_3, cigue)} \quad C_3,3 \in \{1,2,3\}$$

avec $C_3 = poison(cigue)$.
et $\theta_3 = \epsilon$
et donc $\sigma_3 = upg(poison(cigue), poison(Y_2)) = [Y_2 \leftarrow cigue]$
$\sigma_3(animal(X_3), boit(X_3, Y_2)) = animal(X_3), boit(X_3, cigue)$.

$$\frac{\overbrace{animal(X_3)}^{1}, \overbrace{boit(X_3, cigue)}^{2}}{homme(X_5), boit(X_5, cigue)} \quad {\scriptstyle C_4, 1 \in \{1,2\}}$$

avec $C_4 = animal(X) : -homme(X)$.
et $\theta_4 = [X \leftarrow X_5]$ et $\theta_4(animal(X)) = animal(X_5)$
et donc $\sigma_4 = upg(animal(X_5), animal(X_3)) = [X_3 \leftarrow X_5]$
$\sigma_4(\theta_4(homme(X)), boit(X_3, cigue)) = homme(X_5), boit(X_5, cigue)$.

$$\frac{\overbrace{homme(X_5)}^{1}, \overbrace{boit(X_5, cigue)}^{2}}{boit(socrate, cigue)} \quad {\scriptstyle C_5, 1 \in \{1,2\}}$$

avec $C_5 = homme(socrate)$.
dans $\{homme(platon)., homme(socrate).\}$
et $\theta_5 = \epsilon$
et donc $\sigma_5 = upg(homme(socrate), homme(X_5)) = [X_5 \leftarrow socrate]$
$\sigma_5(boit(X_5, cigue)) = boit(socrate, cigue)$.

$$\frac{\overbrace{boit(socrate, cigue)}^{1}}{} \; C_6, 1 \in \{1\}$$

avec $C_6 = boit(socrate, cigue)$.
et $\theta_6 = \epsilon$
et donc $\sigma_6 = upg(boit(socrate, cigue), boit(socrate, cigue)) = \epsilon$.
La substitution $\sigma_1 \ldots \sigma_r|_{V(R_0)}$ est une substitution calculée d'une réfutation SLD.
$\sigma_1 \ldots \sigma_r|_{V(R_0)}(R_0)$ est la réponse calculée de la réfutation SLD.

$$\sigma_0 \ldots \sigma_6 = [X \leftarrow socrate][Y_2 \leftarrow cigue],$$

$$\begin{aligned}&(\sigma_0 \ldots \sigma_6)|_{V(meurt(X))}\\ &\quad = ([X \leftarrow socrate][Y_2 \leftarrow cigue])|_{\{X\}}\\ &\qquad = [X \leftarrow socrate]\end{aligned}$$

et

$$\begin{aligned}& (\sigma_0 \ldots \sigma_6)|_{V(meurt(X))}(meurt(X)) \\ = \;& [X \leftarrow socrate](meurt(X)) \\ = \;& meurt(socrate)\end{aligned}$$

$meurt(socrate)$ ainsi que $meurt(X)$ sont conséquences logiques du programme.

meurt(platon) ?

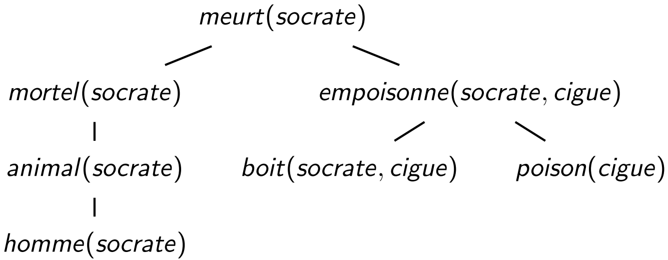
$$\frac{\overbrace{homme(X_5)}^{1}, \overbrace{boit(X_5, cigue)}^{2}}{boit(platon, cigue)} \quad C_5{,}1{\in}\{1{,}2\}$$

avec $C_5 = homme(platon)$.
dans $\{homme(platon)., homme(socrate).\}$
et $\theta_5 = \epsilon$
et donc $\sigma_5 = upg(homme(platon), homme(X_5)) = [X_5 \leftarrow platon]$.
Plus rien ne peut être inférée de boit(platon,cigue) donc cette dérivation mène à un échec.

- Les résolvantes qui ne mènent à aucune dérivation succès ne sont pas conséquences logiques du programme.
- *meurt*(*platon*) n'est pas conséquence logique du programme.

Stratégie de sélection

- Représentation d'une dérivation (instanciée) sous forme d'arbre de preuve.



- La stratégie de sélection choisit, dans une dérivation, l'atome à réduire.
- La stratégie de sélection correspond à un parcours de l'arbre de preuve.

Stratégie de recherche

- La stratégie de recherche choisit la clause pour réduire l'atome considéré.
- Pour une stratégie de sélection donnée, l'arbre SLD explicite toutes les dérivations possibles pour un but donné.

$$meurt(X)$$

$$\mid [X \leftarrow X_1]$$

$$mortel(X_1), empoisonne(X_1)$$

$$\mid [X_1 \leftarrow X_2]$$

$$mortel(X_2), boit(X_2, Y_2), poison(Y_2)$$

$$\mid [X_2 \leftarrow X_3]$$

$$animal(X_3), boit(X_3, Y_2), poison(Y_2)$$

$$\mid [Y_2 \leftarrow cigue]$$

$$animal(X_3), boit(X_3, cigue)$$

$$\mid [X_3 \leftarrow X_5]$$

$$homme(X_5), boit(X_5, cigue)$$

$$[X_5 \leftarrow socrate] \; / \qquad \backslash \; [X_5 \leftarrow platon]$$

$$boit(socrate, cigue) \qquad boit(platon, cigue)$$

Succès $[X \leftarrow socrate]$ — Echec

- La stratégie de recherche correspond au parcours d'une branche de l'arbre SLD.
- La stratégie de sélection et la stratégie de recherche sont des mécanismes de contrôle de la dérivation.
- Une stratégie de recherche est complète si tous les choix de clauses pour résoudre un but sont explorés après un nombre finis d'étapes.
- Lemme d'indépendance : Si la stratégie de recherche est complète alors la stratégie de sélection peut être quelconque.

Stratégie de sélection dite “le plus à gauche”

$$\frac{meurt(X)}{mortel(X_1), empoisonne(X_1)} \quad C_0,1$$

avec $C_0 = meurt(X) :- mortel(X), empoisonne(X).$
et $\theta_0 = [X \leftarrow X_1]$ et $\theta_0(meurt(X)) = meurt(X_1)$
et donc $\sigma_0 = upg(meurt(X_1), meurt(X)) = [X \leftarrow X_1]$
$\sigma_0(\theta_0(mortel(X)), \theta_0(empoisonne(X))) = mortel(X_1), empoisonne(X_1).$

$$\frac{mortel(X_1), empoisonne(X_1)}{animal(X_2), empoisonne(X_2)} \quad C_1,1$$

avec $C_1 = mortel(X) :- animal(X).$
et $\theta_1 = [X \leftarrow X_2]$ et $\theta_1(mortel(X)) = mortel(X_2)$
et donc $\sigma_1 = upg(mortel(X_2), mortel(X_1)) = [X_1 \leftarrow X_2]$
$\sigma_1(\theta_1(animal(X)), empoisonne(X_1)) =$
$animal(X_2), empoisonne(X_2).$

$$\frac{animal(X_2), empoisonne(X_2)}{homme(X_3), empoisonne(X_3)} \quad C_{2},1$$

avec $C_2 = animal(X) : -homme(X)$.
et $\theta_2 = [X \leftarrow X_3]$ et $\theta_2(animal(X)) = animal(X_3)$
et donc $\sigma_2 = upg(animal(X_3), animal(X_2)) = [X_2 \leftarrow X_3]$
$\sigma_2(\theta_2(homme(X)), empoisonne(X_2)) = homme(X_3), empoisonne(X_3)$.

$$\frac{homme(X_3), empoisonne(X_3)}{empoisonne(socrate)} \quad C_{3},1$$

avec $C_3 = homme(socrate)$.
dans $\{homme(platon)., homme(socrate).\}$
et $\theta_3 = \epsilon$
et $\sigma_3 = upg(homme(socrate), homme(X_3)) = [X_3 \leftarrow socrate]$
$\sigma_3(empoisonne(X_3)) = empoisonne(socrate)$.

$$\frac{empoisonne(socrate)}{boit(socrate, Y_4), poison(Y_4)} \quad C_4,1$$

avec $C_4 = empoisonne(X) : -boit(X, Y), poison(Y)$.
$\theta_4 = [X \leftarrow X_4][Y \leftarrow Y_4]$ et
$\theta_4(empoisonne(X)) = empoisonne(X_4)$
$\sigma_4 = upg(empoisonne(X_4), empoisonne(socrate)) = [X_4 \leftarrow socrate]$
$\sigma_4(\theta_4(boit(X, Y)), \theta_4(poison(Y))) =$
$boit(socrate, Y_4), poison(Y_4)$.

$$\frac{boit(socrate, Y_4), poison(Y_4)}{poison(cigue)} \quad C_5,1$$

avec $C_5 = boit(socrate, cigue)$. et $\theta_5 = \epsilon$
$\sigma_5 = upg(boit(socrate, Y_4), boit(socrate, cigue)) = [Y_4 \leftarrow cigue]$
$\sigma_5(poison(Y_4)) = poison(cigue)$.

$$\frac{poison(cigue)}{} \; C_6,1$$

avec $C_6 = poison(cigue)$.
et $\theta_6 = \epsilon$
et donc $\sigma_6 = upg(poison(cigue), poison(cigue)) = \epsilon$.

- L'arbre de preuve est identique au précédent.

Si *homme*(*platon*) est choisi

$$\frac{homme(X_3), empoisonne(X_3)}{empoisonne(platon)} \quad C_3,1$$

avec $C_3 = homme(platon).$
dans $\{homme(platon)., homme(socrate).\}$
et $\theta_3 = \epsilon$
et $\sigma_3 = upg(homme(platon), homme(X_3)) = [X_3 \leftarrow platon]$
$\sigma_3(empoisonne(X_3)) = empoisonne(platon).$

$$\frac{empoisonne(platon)}{boit(platon, Y_4), poison(Y_4)} \quad C_4,1$$

avec $C_4 = empoisonne(X) : -boit(X, Y), poison(Y).$
$\theta_4 = [X \leftarrow X_4][Y \leftarrow Y_4]$ et
$\theta_4(empoisonne(X)) = empoisonne(X_4)$
$\sigma_4 = upg(empoisonne(X_4), empoisonne(platon)) = [X_4 \leftarrow platon]$

- Plus rien ne peut être déduit de $boit(platon, Y_4)$ d'où un échec.
- Même si la stratégie de sélection "le plus à gauche" est abandonnée. . .

$$\frac{boit(platon, Y_4), poison(Y_4)}{boit(platon, cigue)} \quad C_5, 2$$

avec $C_5 = poison(cigue)$. et $\theta_5 = \epsilon$
et $\sigma_5 = upg(poison(cigue), poison(Y_4)) = [Y_4 \leftarrow cigue]$
$\sigma_5(boit(platon, Y_4)) = boit(platon, cigue)$.

- A nouveau, rien ne peut plus être déduit d'où un échec.

Un nouvel arbre SLD

$$meurt(X)$$

$$[X \leftarrow X_1]$$

$$mortel(X_1), empoisonne(X_1)$$

$$[X_1 \leftarrow X_2]$$

$$animal(X_2), empoisonne(X_2)$$

$$[X_2 \leftarrow X_3]$$

$$homme(X_3), empoisonne(X_3)$$

$[X_3 \leftarrow socrate]$	$[X_3 \leftarrow platon]$
$empoisonne(socrate)$	$empoisonne(platon)$
$boit(socrate, Y_4), poison(Y_4)$	$boit(platon, Y_4), poison(Y_4)$
$[Y_4 \leftarrow cigue]$	
$poison(cigue)$	Echec
Succès	
$[X \leftarrow socrate]$	

Pour une stratégie de recherche donnée, la stratégie de sélection (et l'ordre dans le corps des clauses) est capitale pour la taille de l'arbre SLD.

$$\underline{non(0,F)}, non(0,G), ou(F,G,1)$$

$$\Big| \; [F \leftarrow 1]$$

$$\underline{non(0,G)}, ou(1,G,1)$$

$$\Big| \; [G \leftarrow 1]$$

$$\underline{ou(1,1,1)}$$

$$|$$

Succès
$[F \leftarrow 1]$
$[G \leftarrow 1]$

$$non(0,F), non(0,G), \underline{ou(F,G,1)}$$

$[F \leftarrow 0][G \leftarrow 1]$ $\quad$ $[F \leftarrow 1]$ $[G \leftarrow 0]$ $\quad$ $[F \leftarrow 1][G \leftarrow 1]$

$non(0,0), \underline{non(0,1)}$ $\quad$ $non(0,1), \underline{non(0,0)}$ $\quad$ $non(0,1), \underline{non(0,1)}$

$non(0,0)$ $\quad$ Echec $\quad$ $non(0,1)$

Echec $\quad$ Succès $[F \leftarrow 1]$ $[G \leftarrow 1]$

Les listes

- La plus fameuse des structures séquentielles : la liste.
- La liste est une structure binaire récursive dont les symboles sont "." et "[]".
- La liste $.(X, L)$ sera notée $[X|L]$.
- La liste $[X1| \ldots |[Xn|[]]]$ sera notée en abrégé $[X1, \cdots, Xn]$.

$.(1, .(2, .(3, [])))$	$[1\|[2\|[3[]]]]$	$[1, 2, 3]$
$.(X, [])$	$[X\|[]]$	$[X]$
$.(1, .(2, X))$	$[1\|[2\|X]]$	$[1, 2\|X]$

- Le prédicat *liste* est tel que *liste*(L) est vrai si le terme L est une liste :
  - La constante [] est une liste.
  - Le terme $[X|L]$ est une liste si le terme L est une liste.

```
liste([]).
liste([_X|L]) :-
   liste(L).
```

- Le prédicat *liste_de_meme_longueur* est tel que *liste_de_meme_longueur*($L, L\_$) est si les listes L et $L\_$ sont d'une même longueur.

```
listes_de_meme_longueur([],[]).
listes_de_meme_longueur([_|L],[_|L_]) :-
   listes_de_meme_longueur(L,L_).
```

Parcours des listes

- Le parcours des listes est récursif.
- La sélection de la tête ou de la fin de liste se fait directement en exprimant la structure du terme attendu.
- Le prédicat *membre* est tel que *membre*(X, L) est vrai si X est un élément de la liste L.

```
membre(X,[X|_L]).
membre(X,[_Y|L]) :-
   membre(X,L).
```

Les prédicats *prefixe* et *suffixe*

Le prédicat *prefixe* est tel que *prefixe*$(L1, L2)$ est vrai si la liste $L1$ est un préfixe de la liste $L2$.

```
prefixe([],L).
prefixe([X|L1],[X|L2]) :-
   prefixe(L1,L2).
```

Le prédicat *suffixe* est tel que *suffixe*$(L1, L2)$ est vrai si la liste $L1$ est un suffixe de la liste $L2$.

```
suffixe(L,L).
suffixe(L1,[X|L2]) :-
   suffixe(L1,L2).
```

Le prédicat *sous_liste*

Le prédicat *sous_liste* est tel que *sous_liste*($L1, L2$) est vrai si la liste $L1$ est une sous liste de la liste $L2$.

```
sous_liste(L1,L2) :-
   prefixe(L1,L2).
sous_liste(L1,[X|L2]) :-
   sous_liste(L1,L2).
```

Construction de listes à la remontée

- Une liste peut être construite soit dans la tête de clause soit dans le corps de clause.
- Une liste construite dans la tête de clause est construite à la remontée : le résultat de l'appel récursif est modifié (ou augmenté).
- L'ordre des éléments dans la liste construite est le même que l'ordre des éléments dans la liste source.

Le prédicat *melange*

Le prédicat *melange* est tel que *melange*(L, L_-, LL_-) est vrai si la liste LL_- est constituée des éléments en alternance des listes (de tailles égales) L et L_-.

```
melange([],[],[]).
melange([X|L],[Y|L_],[X,Y|LL_]) :-
   melange(L,L_,LL_).
```

Construction de listes à la descente

- Une liste construite dans le corps de clause est construite à la descente : le résultat du cas d'arrêt de la récursion est le résultat du prédicat.
- La technique utilise un accumulateur et une variable de retour pour le résultat.
- L'ordre des éléments dans la liste construite est l'inverse de l'ordre des éléments dans la liste source.

Le prédicat *renverse*

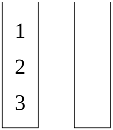
Le prédicat *renverse* est tel que *renverse*(L, R) est vrai si la liste R est la liste L dont l'ordre des éléments est inversé.

```
renverse(L,R) :- renverse_(L,[],R).

renverse_([],Acc,Acc).
renverse_([X|L],Acc,R) :-
   renverse_(L,[X|Acc],R).
```

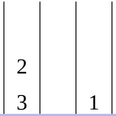
$$\frac{renverse([1,2,3], Renv)}{renverse\_([1,2,3],[], Renv)}$$

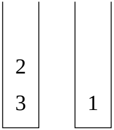
$$\begin{aligned}
&\theta_0 = [L \leftarrow L_1][R \leftarrow R_0]\\
&\sigma_0 = upg(\ \ renverse(L_1, R_0),\\
&\qquad\qquad\quad renverse([1,2,3], Renv))\\
&= [L1 \leftarrow [1,2,3]][R_0 \leftarrow Renv]
\end{aligned}$$



$$\frac{renverse\_([1,2,3],[], Renv)}{renverse\_([2,3],[1], Renv)}$$

$$\begin{aligned}
&\theta_1 = [X \leftarrow X_1][L \leftarrow L_1][Acc \leftarrow Acc_1][R \leftarrow R_1]\\
&\sigma_1 = upg(\ \ renverse_([X_1|L_1], Acc_1, R_1),\\
&\qquad\qquad\quad renverse_([1,2,3],[], Renv))\\
&= \ \ [X_1 \leftarrow 1][L_1 \leftarrow [2,3]][Acc_1 \leftarrow []][R_1 \leftarrow Renv]\\
&\sigma_1(\theta_1(renverse_(L,[X|Acc], R)))\\
&\quad = renverse_([2,3],[1|[]], Renv)
\end{aligned}$$





$$\frac{renverse\_([2,3],[1],Renv)}{renverse\_([3],[2,1],Renv)}$$

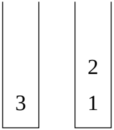
$$\theta_2 = [X \leftarrow X_2][L \leftarrow L_2][Acc \leftarrow Acc_2][R \leftarrow R_2]$$

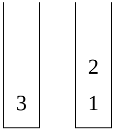
$$\begin{aligned}\sigma_2 = upg(\ & renverse\_([X_2|L_2], Acc_2, R_2),\\ & renverse\_([2,3],[1],Renv))\end{aligned}$$

$$= \ [X_2 \leftarrow 2][L_2 \leftarrow [3]][Acc_2 \leftarrow [1]][R_2 \leftarrow Renv]$$

$$\sigma_2(\theta_2(renverse\_(L,[X|Acc],R)))$$

$$= renverse\_([3],[2|[1]],Renv)$$





$$\frac{renverse\_([3],[2,1],Renv)}{renverse\_([],[3,2,1],Renv)}$$

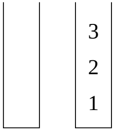
$$\theta_3 = [X \leftarrow X_3][L \leftarrow L_3][Acc \leftarrow Acc_3][R \leftarrow R_3]$$

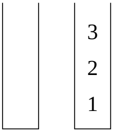
$$\begin{aligned}\sigma_3 = upg(\ & renverse\_([X_3|L_3], Acc_3, R_3),\\ & renverse\_([3],[2,1],Renv))\end{aligned}$$

$$= \quad [X_3 \leftarrow 3][L_3 \leftarrow []][Acc_3 \leftarrow [2,1]][R_3 \leftarrow Renv]$$

$$\sigma_3(\theta_3(renverse\_(L,[X|Acc],R)))$$

$$= renverse\_([],[3|[2,1]],Renv)$$





$$\underline{\textit{renverse\_}([], [3,2,1], \textit{Renv})}$$

$$\theta_4 = [\textit{Acc} \leftarrow \textit{Acc}_4]$$
$$\begin{aligned}\sigma_4 = \textit{upg}(\ & \textit{renverse\_}([], \textit{Acc}_4, \textit{Acc}_4), \\ & \textit{renverse\_}([], [3,2,1], \textit{Renv})) \\ = \quad & [\textit{Renv} \leftarrow \textit{Acc}_4][\textit{Acc}_4 \leftarrow [3,2,1]]\end{aligned}$$

La réponse calculée pour la question $\textit{renverse}([1,2,3], \textit{Renv})$ est
$$\begin{aligned}& [\textit{Renv} \leftarrow \textit{Acc}_4][\textit{Acc}_4 \leftarrow [3,2,1]](\textit{renverse}([1,2,3], \textit{Renv})) \\ = \quad & \textit{renverse}([1,2,3], [3,2,1])\end{aligned}$$

La concaténation

- “La concaténation met bout-à-bout deux listes pour en obtenir une troisième”.
- La liste vide est l’élément neutre de la concaténation (notée $\oplus$).
- $l \oplus [] = l$
  $[a, b] \oplus [c, d, e] = [a, b, c, d, e]$

```
conc([],L2,L2).
conc([X|L1],L2,[X|L3]) :-
   conc(L1,L2,L3).
```

Une liste l_1 est un préfixe d'une liste l_2 si il existe une liste L telle que $l_2 = l_1 \oplus l$.

```
prefixe(L1,L2) :- conc(L1,L,L2).
```

Une liste l_1 est un suffixe d'une liste l_2 si il existe une liste L telle que $l_2 = L \oplus l_1$.

```
suffixe(L1,L2) :- conc(L,L1,L2).
```

Une liste l_1 est une sous liste d'une liste l_2 si il existe deux listes l, l' et l'' telles que $l' = l \oplus l_1$ et $l_2 = l' \oplus l''$.

```
sous_liste(L1,L2) :-
   conc(L,L1,L_),
   conc(L_,L__,L2).
```

Structures incomplètes

- Un terme est une structure incomplète lorsque la structure (et non les éléments) n'est pas totalement instanciée.
- La variable représentant l'incomplétude de la structure est la sentinelle.
- Le terme $[1, 2|L]$ est une structure incomplète (de sentinelle L) mais $[X]$ n'en est pas une.

Exemple de structure incomplète : le dictionnaire

- Un dictionnaire est une liste incomplète qui s'utilise de la même manière en interrogation et en adjonction.
- Un dictionnaire vide est une variable.

```
mise_a_jour(Cle, Valeur, [asso(Cle, Valeur) | _NS]).
mise_a_jour(Cle, Valeur, [asso(Cle_, _) | Dico]) :-
   cles_differentes(Cle, Cle_),
   mise_a_jour(Cle, Valeur, Dico).
```

$$\cfrac{\cfrac{\cfrac{Dico = [asso(p,12)|\_S], mise\_a\_jour(j,18,Dico)}{mise\_a\_jour(j,18,[asso(p,12)|\_S])}}{cles\_differentes(j,p), mise\_a\_jour(j,18,\_S)}}{mise\_a\_jour(j,18,\_S)}$$

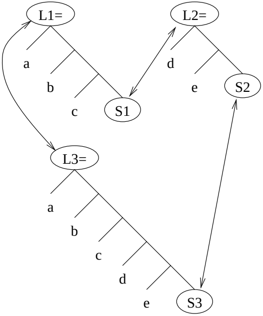
$$[\_S \leftarrow [asso(j,18)|\_NS1], Dico \leftarrow [asso(p,12), asso(j,18)|\_NS1]]$$

Structures en différence

- Une structure incomplète peut croître plutot que d'être augmentée par recopie.
- Une structure en différence est une structure incomplète à laquelle sont associées ses sentinelles.
- Une liste en différence est un couple : *ld*(Liste incomplete,Sentinelle).
- L'accès à une sentinelle est directe et la recopie du terme à la remontée est inutile.
- Le calcul n'est plus linéaire en nombre d'applications de la règle SLD par rapport au nombre de constructeurs de la structure mais constant.
- La manipulation de structures en différence est justifiée lors d'insertions récursives à leurs extrémités uniquement accessibles via la récursion.

Listes en différence

- Le couple $ld(S, S)$ est la liste en différence vide.
- La concaténation est l'opération élémentaire sur les listes.




```
conc_ld(ld(L1,S1),ld(L2,S2),ld(L3,S3)) :-
   L3=L1, S1=L2, S3=S2.
```

- Les listes en différence réalise la concaténation en un nombre d'applications de la règle SLD constant alors que la concaténation sur liste est linéaire en son premier argument.

```
conc_ld(ld(L1,L2),ld(L2,S2),ld(L1,S2)).
```

- la sentinelle de la première liste en différence est unifiée lors de la concaténation (la première liste est donc détruite) ;
- la sentinelle de la seconde liste est partagée avec la troisième liste.

Le *renverse_naif* dopé

- Les prédicats "naifs" sur les listes utilisant la concaténation sont rendus efficaces par les listes en différence.
- Le prédicat *renverse_naif* défini par :

```
renverse_naif([],[]).
renverse_naif([X|L1],L3) :-
   renverse_naif(L1,L2),
   conc(L2,[X],L3).
```

est modifié en :

```
renverse_naif(L,RL) :-
   renverse_naif_ld(L,ld(RL,[])).
renverse_naif_ld([],ld(S,S)).
renverse_naif_ld([X|L1],ld(L3,S3)) :-
   renverse_naif_ld(L1, ld(L2,S2)),
   conc_ld(ld(L2,S2), ld([X|S],S), ld(L3,S3)).
```

soit encore :

```
renverse_naif_ld([X|L1],ld(L3,S3)) :-
   renverse_naif_ld(L1, ld(L2,S2)),
   L2=L3, S2=[X|S], S=S3.
```

soit encore :

```
renverse_naif_ld([X|L1],ld(L2,S)) :-
   renverse_naif_ld(L1, ld(L2,[X|S])).
```

Le résultat est très proche de la construction d'un accumulateur.

Les matrices

- Plusieurs possibilités pour la représentation de matrices de taille n dont :
  - $(n+1)$-uplets ;
  - listes de listes sur n niveaux.
- Premier choix général et aisé à réaliser mais gourmand en espace et très inefficace.
- Deuxième choix efficace, peu gourmand en espace et en apparence simple.

Les listes de listes comme matrice $N \times N$

- Disymétrie entre le traitement des lignes et des colonnes.
- La manipulation de la première ligne et de la première colonne :

```
premiere_ligne(L,M,[L|M]).
premiere_colonne([],[],[]).
premiere_colonne([X|C],[L|M],[[X|L]|CM]) :-
   premiere_colonne(C,M,CM).
```

- La colonne extraite de la matrice est devenue une liste.
- Mais $premiere\_colonne(C, M, [[a11], [a21], [a31]])$ aura pour substitution solution

$$[M \leftarrow [[\,], [\,], [\,]]][C \leftarrow [a11, a21, a31]].$$

- Résultat de la confusion entre la liste vide et la matrice vide.

Un nouveau symbole pour la matrice vide

```
premiere_ligne([X|L], matrice_vide, matrice([[X|L]])).
premiere_ligne([X|L], matrice([L_|M]),
               matrice([[X|L],L_|M])) :-
   liste_de_meme_longueur([X|L], L_).

premiere_colonne([X|C], matrice_vide, matrice([[X]|MC])) :-
   en_liste_de_singletons([X|C], [[X]|MC]).

premiere_colonne([X|C], matrice([[Y|L]|LL]),
                 matrice([[X,Y|L]|CLL])) :-
   premiere_colonne_(C,LL,CLL).

premiere_colonne_([],[],[]).
premiere_colonne_([X|C],[L|M],[[X|L]|CM]) :-
   L= [_|_],
   premiere_colonne_(C,M,CM).
```

Négation par l'échec

- La négation par l'échec est une forme faible de négation logique.
- Hypothèse du monde clos : tout ce qui n'est pas conséquence logique est faux (nié par l'échec).
- Un but B échoue finiment si son arbre de recherche ne comprend ni succès ni branches infinies.
- La négation par l'échec \+(B) appartient à la sémantique d'un programme si B échoue finiment.
- Tester que deux termes t_1 et t_2 ne s'unifie pas s'écrit \+($t_1 = t_2$).
- Le prédicat *disjoint* qui est tel que $disjoint(Xs, Ys)$ est vrai si les deux listes Xs et Ys sont disjointes : $disjointe(Xs, Ys) :- \backslash+(membre(Z, Xs), membre(Z, Ys))$.

Les types en Programmation Logique

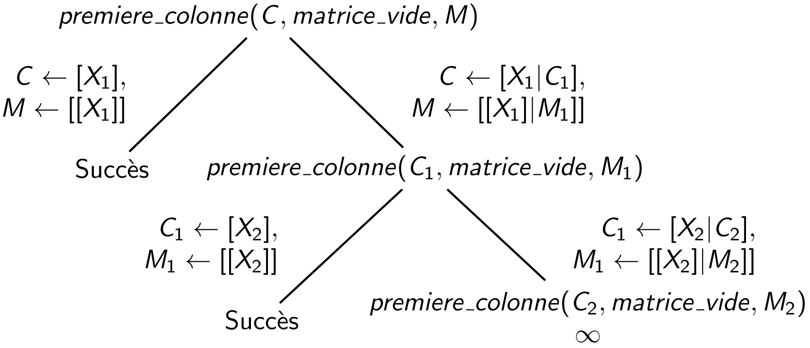
- Les langages de la Programmation Logique sont en général non typés.
- La Programmation Logique en clauses définies est intrinsèquement polymorphe (un type de donnée : le terme).
- Deux sortes de types :
  - structurels polymorphes (différenciés par l'unification),
  - sémantiques (par exemple le prédicat *liste*).
- Le profile détermine le type des constantes.
- Le profil détermine les types des arguments d'un symbole de fonction et le type du terme résultat.
- Le profil détermine les types des arguments d'un symbole de prédicat.

Les modes

- Un mode précise le dégré d'instanciation d'une variable d'une clause.
- 4 modes principaux :
  - $++$: le terme doit être complètement instancié ;
  - $+$: le terme est partiellement instancié (le foncteur principal est nécessairement connu) ;
  - ? : le terme est quelconque (c'est le mode par défaut) ;
  - $-$: le terme est une variable non-instanciée.
- $livre(zola, germinal)$ est compatible avec le mode $++$ mais aussi $+$ (et ?) ;
- $livre(proust, X)$, X non instanciée, est compatible avec le mode $+$ (et ?).

- Une directive de mode pour une règle en indique les contraintes sur les arguments.
- Si la stratégie de recherche est complète alors le mode est inutile.
- Si la stratégie de recherche est incomplète et si, pour la stratégie de sélection choisie, le prédicat est susceptible de générer récursivement une infinité d'existentielle, un mode peut être prescrit pour éviter une telle utilisation.

$premiere\_colonne(-,+,-)$



Utilisé en mode $premiere\_colonne(-,+,-)$, il y a une infinité de succès.

Plan du cours de “Programmation logique”

1. Introduction
2. Programmation Logique
3. Prolog, le langage
4. Applications

Prolog

Prolog : programmation logique en clauses de Horn avec :

- une stratégie de recherche en profondeur d'abord (gestion de l'indéterminisme par pile de retour-arrière) ;
- une stratégie de sélection "le plus à gauche" ;
- une unification sans test d'occurrence ;
- un mécanisme de contrôle de l'indéterminisme : la coupure ;
- une négation par l'échec (sous hypothèse du monde clos) ;
- des prédicats arithmétiques prédéfinis ;
- des prédicats métalogiques sur les types ;
- des prédicats ensemblistes et d'ordre supérieur ;
- des outils syntaxiques : opérateurs, modes, DCG...

Prolog est un sous-ensemble de la programmation logique en clauses de Horn

- efficace
- mais incomplet (stratégie de recherche en profondeur d'abord) et incorrect (absence de test d'occurrence) !

```
p(X) :- p(X).
p(a).
```

$? \ p(Y).$

∞

$$p(Y)$$

$$[Y \leftarrow X] \quad / \qquad \backslash \quad [Y \leftarrow a]$$

$$p(X) \qquad p(a)$$

$$\vdots \qquad \text{Succès}$$

```
p(a).
p(X) :- p(X).
```

$? \ p(Y)$

$[Y \leftarrow a]$

∞

Stratégie de recherche en profondeur d'abord et pile de retour-arrière

- Un *point de choix* est noté par un couple :
  - *Numéro Résolvante*
  - *Ordre dans la définition de la prochaine clause*.
- Simulation du non-déterminisme par mémorisation des points de choix dans une pile de retour-arrière.
- Lors d'un échec, le point de choix le plus récent est dépilé et la règle de dérivation SLD s'applique sur la clause suivante (dans l'ordre du programme) et la résolvante restaurée.
- *Numéro Résolvante* : [*Liste de points de choix*]
  ? *Résolvante*

Trace de la question $mortel(X), empoisonne(X)$:

0 : []
? $mortel(X), empoisonne(X)$

1 : []
? $mortel(X), boit(X, Y_1), poison(Y_1)$

2 : []
? $homme(X), boit(X, Y_1), poison(Y_1)$

3 : [2.2]
? $boit(socrate, Y_1), poison(Y_1)$

4 : [2.2]
? $poison(cigue)$

5 : [2.2]
?
$[X \leftarrow socrate]$

(6) : []
? $homme(X), boit(X, Y_1), poison(Y_1)$

7 : []
? $boit(platon, Y_1), poison(Y_1)$
Echec

La coupure

La coupure est un mécanisme extra-logique de contrôle du non-déterminisme.

- La coupure ote les points de choix produits entre
  - l'effacement du but par la tête de la clause qui introduit cette coupure et
  - dans le corps de la clause avant la coupure.
- Le point de choix, qui est généré par l'application d'une clause qui introduit la coupure, est inclus dans l'ensemble des points de choix supprimés.
- La coupure ne peut être interprétée que procéduralement.

Soit l'unique clause de la définition de q :

$q :- b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l.$

$i : [p_1, \ldots, p_m]$

$? \quad q, q_1, \ldots, q_n$

$i + 1 : [p_1, \ldots, p_m, !]$

$? \quad b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

$\vdots$

$i + i' : [p_1, \ldots, p_m, !, p_{m+1}, \ldots, p_{m+m'}]$

$? \quad !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

$i + i' + 1 : [p_1, \ldots, p_m]$

$? \quad b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

Soit la $1^{ère}$ (et non unique) clause de la définition de q :

$q :- b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l.$

$i : [p_1, \ldots, p_m]$

$? \quad q, q_1, \ldots, q_n$

$i + 1 : [p_1, \ldots, p_m, !, i.2]$

$? \quad b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

$\vdots$

$i + i' : [p_1, \ldots, p_m, !, i.2, p_{m+1}, \ldots, p_{m+m'}]$

$? \quad !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

$i + i' + 1 : [p_1, \ldots, p_m]$

$? \quad b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

Soit la $r^{ème} > 1$ ultime clause de la définition de q :
$q :- b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l.$
Soit la résolvante $j : q, q_1, \ldots, q_n.$

$i-1 : [p_1, \ldots, p_m, j.r]$
   ? *une résolvante*
$[p_1, \ldots, p_m, j.r]$
   Echec
$(i) : [p_1, \ldots, p_m]$
   ? $q, q_1, \ldots, q_n$
$i+1 : [p_1, \ldots, p_m, !]$
   ? $b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$
$\vdots$
$i+i' : [p_1, \ldots, p_m, !, p_{m+1}, \ldots, p_{m+m'}]$
   ? $!, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$
$i+i'+1 : [p_1, \ldots, p_m]$
   ? $b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

Soit la $r^{ème} > 1$ (non ultime) clause de la définition de q :
$q :- b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l.$
Soit la résolvante j : $q, q_1, \ldots, q_n$.

$i - 1 : [p_1, \ldots, p_m, j.r]$
  ? *une résolvante*
$[p_1, \ldots, p_m, j.r]$
  Echec
$(i) : [p_1, \ldots, p_m]$
  ? $q, q_1, \ldots, q_n$
$i + 1 : [p_1, \ldots, p_m, !, i.(r + 1)]$
  ? $b_1, \ldots, b_k, !, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$
$\vdots$
$i + i' : [p_1, \ldots, p_m, !, i.(r + 1), p_{m+1}, \ldots, p_{m+m'}]$
  ? $!, b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$
$i + i' + 1 : [p_1, \ldots, p_m]$
  ? $b_{k+1}, \ldots, b_l, q_1, \ldots, q_n$

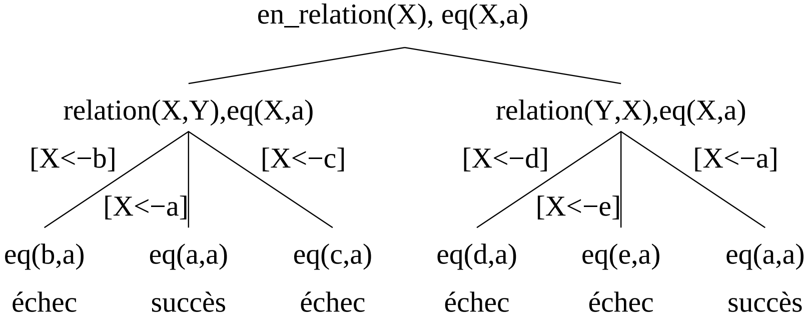
Coupure et sémantique

- La coupure modifie la sémantique de Prolog en éliminant des branches de l'arbre de recherche qui mènent à un succès.
- Soit le programme P suivant

```
relation(b,d).
relation(a,e).
relation(c,a).

en_relation(X) :- relation(X,Y).
en_relation(X) :- relation(Y,X).
eq(X,X).
```

$$
\begin{array}{ll}
? & en\_relation(X), eq(X, a) \\
 & [X \leftarrow a]; \\
 & [X \leftarrow a]; \\
 & non
\end{array}
$$



$?\ en\_relation(X), !, eq(X, a)$

$0 : [!]$

$?\quad en\_relation(X), !, eq(X, a)$

$1 : [!, 0.2]$

$?\quad relation(X, Y), !, eq(X, a)$

$2 : [!, 0.2, 1.2]$

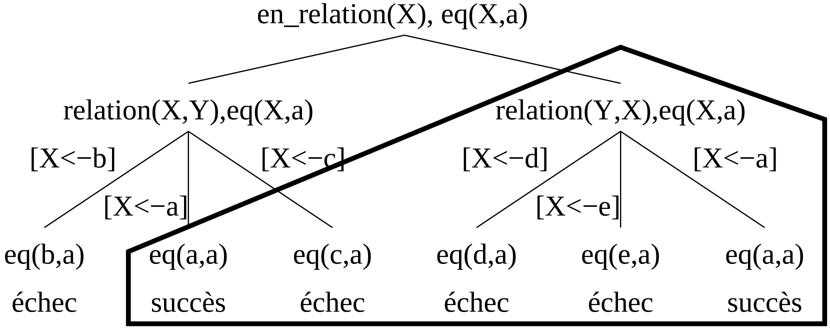
$?\quad !, eq(b, a)$

$3 : [\ ]$

$?\quad eq(b, a)$

$[\ ]$

Echec


en_relation(X), eq(X,a)
relation(X,Y),eq(X,a)
relation(Y,X),eq(X,a)
[X<−b]
[X<−a]
[X<−c]
[X<−d]
[X<−e]
[X<−a]
eq(b,a)
eq(a,a)
eq(c,a)
eq(d,a)
eq(e,a)
eq(a,a)
échec
succès
échec
échec
échec
succès

$?\ en\_relation(X), eq(X, a), !$

$0 : [!]$

$?\quad en\_relation(X), eq(X, a), !$

$1 : [!, 0.2]$

$?\quad relation(X, Y), eq(X, a), !$

$2 : [!, 0.2, 1.2]$

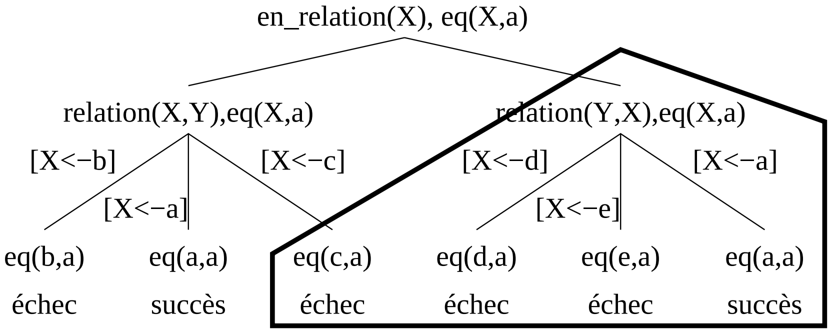
$?\quad eq(b, a), !$

$[!, 0.2, 1.2]$

$?\quad eq(b, a), !$

Echec

$[!, 0.2, 1.2]$
$\quad ? \quad eq(b, a), !$
$\qquad Echec$
$(3) : [!, 0.2]$
$\quad ? \quad relation(X, Y), eq(X, a), !$
$4 : [!, 0.2, 3.3]$
$\quad ? \quad eq(a, a), !$
$5 : [!, 0.2, 3.3]$
$\quad ? \quad !$
$6 : [\ ]$
$\quad ?$
$\qquad [X \leftarrow a]$

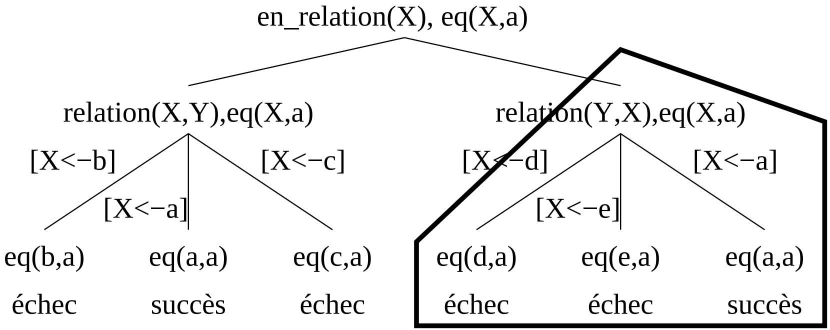

en_relation(X), eq(X,a)
relation(X,Y),eq(X,a)
relation(Y,X),eq(X,a)
[X<−b]
[X<−a]
[X<−c]
[X<−d]
[X<−e]
[X<−a]
eq(b,a)
eq(a,a)
eq(c,a)
eq(d,a)
eq(e,a)
eq(a,a)
échec
succès
échec
échec
échec
succès

? $en\_relation(X), eq(X, a)$

- Modification du programme :
  $en\_relation(X) :- !, relation(X, Y).$

$0 : [\,]$
  ? $en\_relation(X), eq(X, a)$
$1 : [!, 0.2]$
  ? $!, relation(X, Y), eq(X, a)$
$2 : [\,]$
  ? $relation(X, Y), eq(X, a)$
$3 : [2.2]$
  ? $eq(b, a)$
$[2.2]$
  Echec

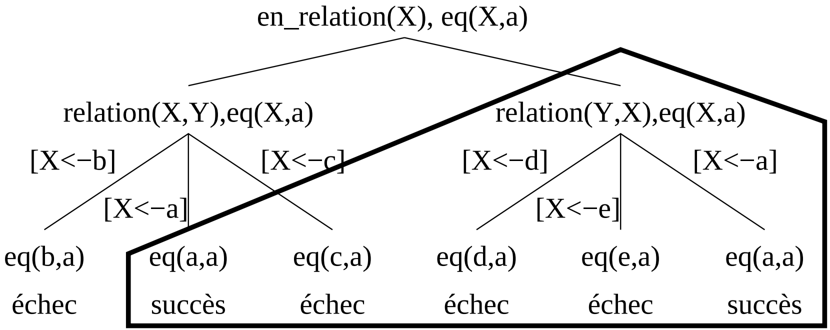
(4) : [ ]
  ? $relation(X, Y), eq(X, a)$
5 : [4.3]
  ? $eq(a, a)$
6 : [4.3]
  ?
    $[X \leftarrow a]$
(7) : [ ]
  ? $relation(X, Y), eq(X, a)$
8 : [ ]
  ? $eq(c, a)$
[ ]
    Echec


en_relation(X), eq(X,a)
relation(X,Y),eq(X,a)
relation(Y,X),eq(X,a)
[X<−b]
[X<−a]
[X<−c]
[X<−d]
[X<−e]
[X<−a]
eq(b,a)
eq(a,a)
eq(c,a)
eq(d,a)
eq(e,a)
eq(a,a)
échec
succès
échec
échec
échec
succès

$?\ en\_relation(X), eq(X, a)$

- Modification du programme :
  $en\_relation(X) :- relation(X, Y), !.$

$0 : [\ ]$
$\quad ?\quad en\_relation(X), eq(X, a)$
$1 : [!, 0.2]$
$\quad ?\quad relation(X, Y), !, eq(X, a)$
$2 : [!, 0.2, 1.2]$
$\quad ?\quad !, eq(b, a)$
$3 : [\ ]$
$\quad ?\quad eq(b, a)$
$[\ ]$
$\quad\quad Echec$

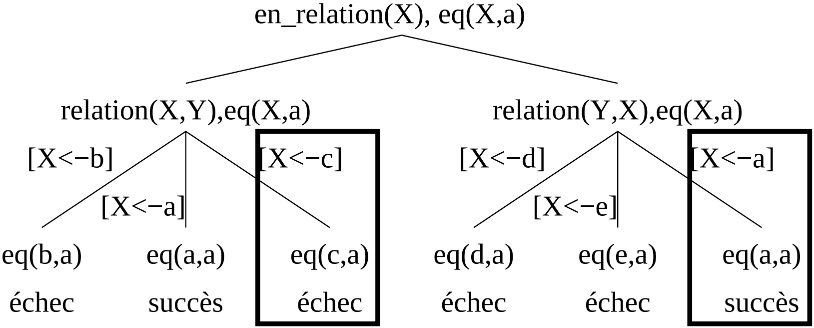

en_relation(X), eq(X,a)
relation(X,Y),eq(X,a)
relation(Y,X),eq(X,a)
[X<−b]
[X<−a]
[X<−c]
[X<−d]
[X<−e]
[X<−a]
eq(b,a)
eq(a,a)
eq(c,a)
eq(d,a)
eq(e,a)
eq(a,a)
échec
succès
échec
échec
échec
succès

? $en\_relation(X), eq(X, a)$

- Modification du programme : $relation(a, e) :- !.$

$0 : [\,]$
  ? $en\_relation(X), eq(X, a)$
$1 : [0.2]$
  ? $relation(X, Y), eq(X, a)$
$2 : [0.2, 1.2]$
  ? $eq(b, a)$
$[0.2, 1.2]$
  Echec
$(3) : [0.2]$
  ? $relation(X, Y), eq(X, a)$
$4 : [0.2, !, 3.3]$
  ? $!, eq(a, a)$
$5 : [0.2]$
  ? $eq(a, a)$

6 : [0.2]
  ?
    $[X \leftarrow a]$
(7) : [ ]
  ? $en\_relation(X), eq(X, a)$
8 : [ ]
  ? $relation(Y, X), eq(X, a)$
9 : [8.2]
  ? $eq(d, a)$
[8.2]
    Echec
(10) : [ ]
  ? $relation(Y, X), eq(X, a)$
11 : [!, 10.3]
  ? $!, eq(e, a)$
12 : [ ]
  ? $eq(e, a)$
[ ]
    Echec


en_relation(X), eq(X,a)
relation(X,Y),eq(X,a)
relation(Y,X),eq(X,a)
[X<−b]
[X<−a]
[X<−c]
[X<−d]
[X<−e]
[X<−a]
eq(b,a)
eq(a,a)
eq(c,a)
eq(d,a)
eq(e,a)
eq(a,a)
échec
succès
échec
échec
échec
succès

Déterminisme et coupure

- Une coupure qui ne modifie pas la sémantique d'un programme est dite verte.
- Une coupure verte explicite le déterminisme implicite.
- La coupure est placée après le test de déterminisation.
- Le minimum de deux nombres :

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

- Si l'unification dans la tête de clause réalise le test de déterminisation alors la coupure est le premier but du corps.
- Vérification de type pour des arbres binaires d'entiers :

```
arbre_binaire_entier(ab(X,G,D)) :-
    !,
    entier(X),
    arbre_binaire_entier(G),
    arbre_binaire_entier(D).
arbre_binaire_entier(X) :-
    entier(X).
```

- Une coupure qui modifie la sémantique est dite rouge.
- Lorsqu'une coupure est utilisée pour omettre une condition elle est rouge car elle modifie la sémantique du programme.

Coupure rouge

- Soit le prédicat *minimum_cut* :

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

  alors $minimum\_cut(2, 5, 5)$ appartient à la sémantique de *minimum_cut* !
- De tels prédicats ne fonctionnent opérationnellement que pour des modes précis.
- Soit le prédicat *membre_cut* :

```
membre_cut(X,[X|_]) :- !.
membre_cut(X,[_|L]) :- membre_cut(X,L).
```

  alors $[X \leftarrow 2]$ n'appartient pas à la sémantique de $membre\_cut(X, [1, 2, 3])$.

Le prédicat \+

- L'opérateur \+ est une mise-en-œuvre incomplète de la négation par l'échec.
- La négation par l'échec est définie (opérationnellement) par :
  - la négation par l'échec d'un but B est un succès si l'effacement de B provoque un échec et
  - la négation par l'échec d'un but B provoque un échec si l'effacement de B provoque un succès.
- La définition de l'opérateur \+ utilise une coupure rouge :

```
\+(B) :- call(B), !, fail.
\+(B).
```

- Le prédicat *fail* échoue en toute circonstance.
- Le prédicat \+ ne peut avoir qu'une lecture opérationnelle.
- La négation par l'échec sur un but non clos n'en instancie pas les variables.

- Dans le cas d'un but B dont la dérivation est infinie, deux comportements sont possibles :
  - la dérivation de \+(B) est infinie si la dérivation de B est infinie sans aucun succès ;
  - la dérivation de \+(B) est un succès si la dérivation de B est infinie après un (ou plusieurs) succès.
- Soit les clauses :

$$\begin{array}{ll} (1) & voisin(X, Y) : -voisin(Y, X). \\ (2) & voisin(pascal, stephane). \end{array}$$

- Le but \+($voisin(pascal, stephane)$). provoque un échec pour la suite de clauses (2)(1) mais sa dérivation est infinie pour la suite de clauses (1)(2).
- La correction du résultat de \+(B) n'est garantie que si B est un but complètement instancié : *mode* \+($++$).
- ? \+($X = 1$)$, X = 2$ échoue alors que $[X \leftarrow 2]$ est solution.

Règle par défaut

- Cas de coupure rouge : un ensemble de cas (coupés) et une règle par défaut.
- Le prédicat $Si->Alors;Sinon$ insere dans la résolvante :
  - soit *Alors* si la condition *Si* est un succès,
  - soit *Sinon* si la condition \+(*Si*) est un succès.
-
```
% mode '_->_;_'(++,+,+).
Si -> Alors ; _Sinon :- Si, !, Alors.
Si -> _Alors ; Sinon :- not(Si), Sinon.
```
- mais de façon incorrecte mais plus efficace :

```
Si -> Alors ; Sinon :- Si, !, Alors.
Si -> Alors ; Sinon :- Sinon.
```

Déclaration d'opérateurs

- La directive *op* permet la déclaration d'opérateurs unaires et binaires.
  $: -op(Précédence, Associativité, Prédicat)$
  - *Prédicat* est le nom du prédicat.
  - *Précédence* déclare la précédence de l'opérateur : plus la précédence est faible et plus l'opérateur est prioritaire.
- Par extension, la précédence d'un terme
  - non-structuré ou entouré de parenthèses est de 0.
  - structuré est la précédence de son foncteur principal.

- *Associativité* déclare l'associativité (binaire infixe) :
  - *xfx* n'est pas associatif ;
  - *xfy* est associatif à droite ;
  - *yfx* est associatif à gauche.
- ou l'auto-application (unaire) :
  - *fx* est préfixé et non auto-applicant ;
  - *fy* est préfixé et auto-applicant ;
  - *xf* est postfixé et non auto-applicant ;
  - *yf* est postfixé et auto-applicant.
- Techniquement :
  - *x* est un argument dont la priorité doit être strictement inférieure à celle de l'opérateur ;
  - *y* est un argument dont la priorité doit être inférieure ou égale à celle de l'opérateur.

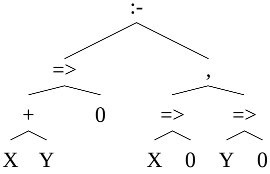
- Exemple de déclarations d'opérateurs :

```
:- op(1200,xfx,':-').
:- op(1000,xfy,',').
:- op(300,fx,'-').
:- op(500,yfx,'+').
```

- Soit la déclaration d'opérateur :

```
% '=>' : Int x Int.
:- op(700,xfx,'=>').
```

Alors le terme : $X + Y => 0 : -X => 0, Y => 0$
est équivalent au terme :
`':-'('=>'('+'(X,Y),0),','('=>'(X,0),'=>'(Y,0)))`



Arithmétique prédéfinie

- L'arithmétique préfinie est fonctionnelle.
- Le prédicat is permet l'évaluation des expressions arithmétiques : `Terme is Exp`

  ```
  :- op(700,xfx,is).
  % mode is(?,++).
  ```
- L'expression `Exp` est évaluée et le résultat est unifié avec le terme `Terme`.
- Si l'expression `Exp` n'est pas complètement instanciée alors une erreur est invoquée.
- Le prédicat `is` n'est pas une affectation.
  Par exemple `N is N+1` n'a pas de sens :
  - si N est instancié alors l'évaluation de `N+1` et `N` ne sont pas unifiables ;
  - si `N` n'est pas instancié alors l'évaluation de `N+1` provoque une erreur.

- L'arithmétique prédéfinie oriente les prédicats selon un mode précis.
- Le prédicat `longueur` de mode `longueur(+,++)` vérifie qu'une liste `L` est de taille `N` :

```
longueur([_|L],N) :-
   N > 0,
   N1 is N - 1,
   longueur(L,N1).
longueur([],0).
```

- C'est incorrecte en mode `longueur(+,--)`.
- La récursivité structurelle des entiers est absente de l'arithmétique prédéfinie.
- Prolog dispose de prédicats de test arithmétique de mode `(++,++)` :

```
:- op(700,xfx,['<','>=','>','=<','=:=','=\=']).
```

Méta-variables et appel dynamique de buts

- Les buts sont des termes.
- Une variable qui manipule des buts est une méta-variable.
- Le passage du but sous forme de terme en un appel de ce but est réalisé par le prédicat `call`.
- La disjonction dans les buts en prolog est définie par l'opérateur “;” ainsi :

```
X ; Y :- call(X).
X ; Y :- call(Y).
```

Prédicats méta-logiques et inspection de termes

- Termes du premier ordre : impossible de manipuler les foncteurs ni d'obtenir leurs arguments d'un terme dont la structure est inconnue a priori.
- Trois prédicats méta-logiques pour obtenir
  - le foncteur et son arité (`functor`),
  - un argument (`arg`) ou
  - le foncteur et sa liste d'arguments (`=..`).

- Deux modes d'utilisation pour `functor` :
  - `mode functor(+,?,?)`, pour obtenir le foncteur et/ou son arité à partir d'un terme ;
  - `mode functor(?,+,+)`, pour construire un terme à partir d'un foncteur et de son arité.
- Exemples :

$$
\begin{array}{ll}
? & \mathit{functor}(\mathit{ab}(1, \mathit{abv}, \mathit{ab}(2, \mathit{abv}, \mathit{abv})), \mathit{Fonc}, \mathit{Arite}). \\
 & [\mathit{Fonc} \leftarrow \mathit{ab}], \\
 & [\mathit{Arite} \leftarrow 3] \\
? & \mathit{functor}(\mathit{Terme}, \mathit{ab}, 3). \\
 & [\mathit{Terme} \leftarrow \mathit{ab}(\_, \_, \_)]
\end{array}
$$

- Un seul mode d'utilisation pour `arg` : pour obtenir le n^{ème} argument d'un terme,
  `mode  arg(+,+,?)`
- Exemples :

$$
\begin{array}{ll}
? & arg(3, ab(1, abv, ab(2, abv, abv)), Argument). \\
 & [Argument \leftarrow ab(2, abv, abv)] \\
? & arg(2, ab(1, X, abv), ab(2, abv, abv)). \\
 & [X \leftarrow ab(2, abv, abv)]
\end{array}
$$

- Deux modes d'utilisations pour `=..` :
  - `mode '=..'(+,?)`, pour obtenir le foncteur et la liste d'arguments à partir d'un terme ;
  - `mode '=..'(?,[+|?])`, pour construire un terme à partir d'un foncteur et de la liste des arguments.
- Exemples :

$$
\begin{array}{ll}
? & ab(1, abv, ab(2, abv, abv)) = ..[Fonc|Args]. \\
 & [Fonc \leftarrow ab, Args \leftarrow [1, abv, ab(2, abv, abv)]] \\
? & Terme = ..[ab, X, abv, ab(2, abv, abv)]. \\
 & [Terme \leftarrow ab(X, abv, ab(2, abv, abv))] \\
? & constante = ..[Fonc|Args]. \\
 & [Fonc \leftarrow constante] \\
 & [Args \leftarrow []]
\end{array}
$$

Construction de but et ordre supérieur

- Les prédicats méta-logiques permettent de construire dynamiquement du but et l'appel dynamique `call` permet de le réduire.
- Le prédicat méta-logique `map` applique à chaque élément d'une liste un prédicat passé en paramètre.

```
map(_,[],[]).
map(Predicat, [Arg1|Arg1S], [Res1|Res1S]) :-
   But=..[Predicat, Arg1, Res1],
   call(But),
   map(Predicat,Arg1S,Res1S).
```

Prédicats métalogiques sur les types prédéfinis

Un ensemble de prédicats métalogiques permettent de déterminer si un terme appartient à un type prédéfini :

- `var` vérifie que son argument est une variable ;

`? var(X).`	`? var(5).`	`? var([X|Y]).`
Succès	Echec	Echec

- `nonvar` vérifie que son argument n'est pas une variable ;

`nonvar(X).`	`nonvar(5).`	`nonvar([X|Y]).`
Echec	Succès	Succès

- `compound` vérifie qu'un terme n'est ni une constante ni une variable ;

`compound(X).`	`compound(5).`	`compound([X|Y]).`
Echec	Echec	Succès

- `atomic` vérifie qu'un terme est une constante.

`atomic(X).`	`atomic(5).`	`atomic([X|Y]).`
Echec	Succès	Echec

Exemple d'inspection de termes : l'unification explicite

```
unifie(Constante1,Constante2) :-
   atomic(Constante1), atomic(Constante2),
   Constante1=Constante2.
unifie(Construit1,Construit2) :-
   compound(Construit1), compound(Construit2),
   Construit1=..[Foncteur|Args1],
   Construit2=..[Foncteur|Args2],
   unifie_args(Args1,Args2).
unifie(Var1,Var2) :- var(Var1), var(Var2), Var1=Var2.
unifie(Var,PasVar) :- var(Var), nonvar(PasVar), Var=PasVar.
unifie(PasVar,Var) :- unifie(Var,PasVar).
unifie_args([],[]).
unifie_args([Arg1|Args1],[Arg2|Args2]) :-
   unifie(Arg1,Arg2), unifie_args(Args1,Args2).
```

Manipulation des variables logiques

- L'unification ne permet pas de distinguer une variable d'une autre.
- Impossible de distinguer deux variantes (identique au nommage des variables près) de deux termes.
- Le prédicat `var_non_presente` de mode `(-,?)` doit réussir si la variable n'est pas présente dans le terme.

```
var_non_presente(X,Constante) :-  atomic(Constante).
var_non_presente(X,Construit) :-
    compound(Construit), Construit =.. [_|Args],
    var_non_presente_args(X,Args).
var_non_presente(X,Var) :-
    var(Var), \+(X=Var).% pas suffisant...
```

car une variable est toujours en Prolog unifiable avec un autre terme.

- Le prédicat métalogique == réussit si les deux arguments sont identiques.
- Le prédicat métalogique =\= réussit si les deux arguments ne sont pas indentiques.

`? X == X.`	`? X =\= X.`
Succès	Echec
`? X == Y.`	`? X =\= Y.`
Echec	Succès
`? [1|L] == [1|L].`	`? [1|L] =\= [1|L].`
Succès	Echec
`? [1|L] == [1|S].`	`? [1|L] =\= [1|S].`
Echec	Succès

- Le prédicat `var_non_presente` de mode `(-,?)` réussit si la variable n'est pas présente dans le terme.

```
var_non_presente(X,Var) :-
    var(Var), X =\= Var.
```

Prédicats ensemblistes

- Prolog est un langage logique du 1^{er} ordre : calcul des termes vérifiant une propriété.
- Prédicats ensemblistes : extension du second ordre calculant l'ensemble des entités vérifiant une propriété.
- `bagof(Terme?, But+, Instances?)` est vrai si `Instances` est la liste de toutes les instances de `Terme` telles que `But` soit vrai.
- Les variables existentielles de `But` absentes de `Terme` subissent le retour-arrière.
- Deux interprétations logiques possibles.

Interprétation habituelle

- Si `X1, ..., Xn` sont les variables du terme `Terme` et `Y1,...,Ym` les variables de `But` absentes de `Terme` alors la signification logique est une approximation de la formule :
  $\exists X_{[1..n]} \exists Y_{[1..m]} (\forall y_{j \in [1..m]} ((y_1 \in Y_1 \wedge \ldots \wedge y_m \in Y_m)$
  $\Rightarrow But(X_{[1..n]}, y_{[1..m]})))$
- Exemple : $\exists EnfantS \exists Pere(\forall x(x \in EnfantS \rightarrow pere(Pere, x)))$
  `pere(paul,jeanne). pere(paul,denis).`

  `? bagof(Enfant, pere(Pere,Enfant), EnfantS).`
  $[EnfantS \leftarrow [jeanne, denis]]$, $[Pere \leftarrow paul]$
- En ajoutant au programme `pere(pierre,henri).`

  `? bagof(Enfant,pere(Pere, Enfant), EnfantS).`
  $[EnfantS \leftarrow [jeanne, denis]]$, $[Pere \leftarrow paul]$
  $[EnfantS \leftarrow [henri]]$, $[Pere \leftarrow pierre]$

Une interprétation différente

- Une autre interprétation est possible (I et $\overline{I}$ est une partition de $[1..n]$) :
  $$\exists X_I \exists Y_{[1..m]}(\forall y_{[1..m]}((y_1 \in Y_1 \wedge \ldots \wedge y_m \in Y_m) \Rightarrow (\exists x_{\overline{I}} But(X_I, x_{\overline{I}}, y_{[1..m]}))))$$
- Même exemple que précédemment :
  $\exists EnfantS(\forall x(x \in EnfantS \Rightarrow (\exists y(pere(y, x)))))$

  `? bagof(Enfant,∃(Pere,pere(Pere,Enfant)),EnfantS).`

  $[EnfantS \leftarrow [jeanne, denis, henri]]$

Autres prédicats ensemblistes

- `setof(Terme?, But+, Instances?)` est vrai si *Instances* est la liste triée sans doubles des instances de terme *Terme* telles que *But* soit vrai.
- `for_all(But+, Condition+)` est vrai si pour toute instance des variables de `But`, `Condition` est vrai.

```
for_all(But,Condition) :-
   set_of(Condition,But, CasS),
   verifie(CasS).
verifie([]).
verifie([Cas|CasS]) :-
   call(Cas),
   verifie(CasS).
```

Les prédicats dynamiques

- La directive de compilation :
  `:- dynamic Nom_de_prédicat/Arité.` permet de déclarer un prédicat dynamique.
- Le prédicat `asserta` (resp. `assertz`) est tel que `asserta(C)` (resp. `assertz(C)`) insère la clause `C` (terme suffisamment instancié) en tête (resp. à la fin) de la définition du prédicat constituant la tête de la clause `C`.
- Les variables non-instanciées des clauses insérées sont quantifiées universellement.
- Le prédicat `retracta` (resp. `retractz`) est tel que `retracta(C)` (resp. `retractz(C)`) ote du programme une clause qui s'unifie avec `C`.
- Les insertions ou rétractions ne subissent pas le retour-arrière.

Des prédicats dynamiques pour une plus grande efficacité

- Les prédicats dynamiques sont utilisés pour sauvegarder des résultats intermédaires.
- Le prédicat `fibonacci` :

```
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,FibN) :-
   N > 1,
   N1 is N-1, N2 is N-2,
   fibonacci(N1,FibN1),
   fibonacci(N2,FibN2),
   FibN is FibN1 + FibN2.
```

La complexité du prédicat `fibonacci` suivant passe de l'exponentiel au linéaire !

```
% mode fibonacci(++,?).
:- dynamic fibonacci/2.
fibonacci(0,1).
fibonacci(1,1).
fibonacci(N,FibN) :-
   N > 1,
   N1 is N-1, N2 is N-2,
   fibonacci(N1,FibN1),fibonacci(N2,FibN2),
   FibN is FibN1 + FibN2,
   asserta((fibonacci(N,FibN):-!)).
```

DCG

- Definite Clause Grammar : Grammaire à clause définie.
- Formalisme d'expression des grammaires hors-contexte integré à Prolog.
- Une grammaire des nombres en anglais :

```
digit --> [one].
...
digit --> [nine].
teen --> [ten].
...
teen --> [nineteen].
tens --> [twenty].
...
tens --> [ninety].
```

- ```
  nb_anglais --> [zero].
  nb_anglais --> digit, [hundred].
  nb_anglais -->
     digit, [hundred,and],moinscent.
  nb_anglais --> moinscent.
  moinscent --> digit.
  moinscent --> teen.
  moinscent --> tens, unite.
  unite --> [].
  unite --> digit.
  ```
- L'appel d'un but faisant référence à un non-terminal d'une DCG est réalisé par le prédicat `phrase` :
  ? $phrase(nb\_anglais, [two, hundred, and, twenty])$.
  Succes
  ```

- Les non-terminaux peuvent avoir des arguments pour synthétiser des résultats pendant l'analyse syntaxique.
- ```
  digit(1) --> [one].
  ...
  digit(9) --> [nine].
  teen(10) --> [ten].
  ...
  teen(19) --> [nineteen].
  tens(20) --> [twenty].
  ...
  tens(90) --> [ninety].
  ```
- $$? \; phrase(tens(N), [ninety]).$$
  $$[N \leftarrow 90]$$
  ```

- Des buts peuvent modifier les arguments synthétisés.
-
```
nb_anglais(0) --> [zero].
nb_anglais(N) --> digit(D),[hundred],{N is D*100}.
nb_anglais(N) -->
   digit(D),[hundred,and],moinscent(M),
   {N is D*100+M}.
nb_anglais(N) --> moinscent(N).
moinscent(N) --> digit(N).
moinscent(N) --> teen(N).
moinscent(N) --> tens(T), unite(U), {N is T+U}.
unite(0) --> [].
unite(N) --> digit(N).
```
-

  $$? \; phrase(nb\_anglais(N), [two, hundred, and, twenty, three]).$$
  $$[N \leftarrow 223]$$

Traduction automatique de grammaires hors-contexte

```
:- op(1200,xfx,'-->').
traduire_regle((Mg-->Md),(Tete :- Corps)) :-
   traduire_NT(Mg,Tete,E,S),
   traduire_NT(Md,Corps,E,S).
traduire_NT((A,B), (A1,B1), E,S) :-
   traduire_NT(A,A1,E,I),
   traduire_NT(B,B1,I,S).
traduire_NT(A,true,E,S) :-
   A=[_|_],
   conc(A,S,E).
traduire_NT(A,A1,E,S) :-
   atomic(A),
   A1=..[A,E,S].
traduire_NT([],true,E,E).
```

Plan du cours de “Programmation logique”

Automates finis non-déterministes

- Soit $M = (Q, \Sigma, \delta, s, F)$ un automate fini non-déterministe défini par :
  - Q est un ensemble fini d'états,
  - Σ est un alphabet,
  - $\Delta \subset (Q \times \Sigma \times Q)$ la relation de transition,
  - $s \in Q$ l'état initial,
  - $F \subseteq Q$ l'ensemble des états accepteurs.
- Le graphe de la relation de transition Δ est décrite en Prolog sous forme de faits.

- La configuration (q', w') est dérivable en une étape de la configuration (q, w) par l'AFND M (noté $(q, w) \vdash_M (q', w')$) si
  - $w = \sigma w', \sigma \in \Sigma$ et
  - $(q, \sigma, q') \in \Delta$.
- Un mot est accepté par un AFD M si $(s, w) \vdash_M^* (q, \varepsilon)$ avec $q \in F$.

```
derivable(Q1,[Sigma|W2],Q2) :-
   delta(Q1,Sigma,Qi),
   derivable(Qi,W2,Q2).
derivable(Q,[],Q).

accepte(W) :-
   initial(S),
   derivable(S,W,F),
   final(F).
```

L'ADNF :

$$
\begin{aligned}
(\ & \{q_0, q_1, q_2\}, \{a, b\}, \\
& \{(q_0, a, q_0), (q_0, b, q_0), (q_0, a, q_1), (q_1, b, q_2)\}, \\
& q_0, \{q_2\})
\end{aligned}
$$

accepte le langage $\{w \mid w \text{ se termine par } ab\}$.

```
delta(q0,a,q0).
delta(q0,b,q0).
delta(q0,a,q1).
delta(q1,b,q2).

initial(q0).

final(q2).
```

Un AFND spécialisé est obtenu par dépliage de la fonction de transition `delta` dans `derivable` et `accepte` :

```
se_termine_par_ab(W) :-
   se_termine_par_ab_(q0,W).

se_termine_par_ab_(q0,[a|W]) :-
   se_termine_par_ab_(q0, W).
se_termine_par_ab_(q0,[b|W]) :-
   se_termine_par_ab_(q0, W).
se_termine_par_ab_(q0,[a|W]) :-
   se_termine_par_ab_(q1, W).
se_termine_par_ab_(q1,[b|W]) :-
   se_termine_par_ab_(q2, W).
se_termine_par_ab_(q2,[]).
```

Automates à pile

- Soit $M = (Q, \Sigma, \Gamma, \Delta, Z, s, F)$ un automate à pile défini par :
  - Q est un ensemble fini d'états,
  - Σ est un alphabet d'entrée,
  - Γ est un alphabet de pile,
  - $\Delta \subset ((Q \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*))$ la relation de transition,
  - $Z \in \Gamma$ est le symbole initial de la pile,
  - $s \in Q$ l'état initial,
  - $F \subseteq Q$ l'ensemble des états accepteurs.
- La configuration (q', w', α') est *dérivable en une étape* de la configuration (q, w, α) par l'automate à pile M (noté $(q, w, \alpha) \vdash_M (q', w', \alpha')$) si
  - $w = uw', u \in \Sigma^*$,
  - $\alpha = \beta\delta, \beta, \delta \in \Gamma^*$,
  - $\alpha' = \gamma\delta, \gamma \in \Gamma^*$, et
  - $((q, u, \beta), (q', \gamma)) \in \Delta$.

Un mot est accepté par un automate à pile M si $(s, w, Z) \vdash_M^* (q, \varepsilon, Z)$ avec $q \in F$.

```
derivable(CF,Q) :-
   delta(CF,CF_),
   derivable(CF_,Q).
derivable(cf(Q,[],_),Q).


accepte(W) :-
   initial(Q),
   derivable(cf(Q,W,[]),F),
   final(F).
```

L'automate à pile :

$(\{s, p, q\}, \{a, b\}, \{aa, bb, zz\},$
$\{((s, a, \varepsilon), (s, aa)), ((s, b, \varepsilon), (s, bb)), ((s, \varepsilon, \varepsilon), (p, \varepsilon)), ((p, a, aa), (p, \varepsilon)),$
$((p, b, bb), (p, \varepsilon)), ((p, \varepsilon, zz), (q, \varepsilon))\}, zz, s, \{q\})$

accepte le langage $\{ww^R / w \in \{a, b\}^*\}$.

```
delta(cf(s,[a|W],Pile),cf(s,W,[aa|Pile])).
delta(cf(s,[b|W],Pile),cf(s,W,[bb|Pile])).
delta(cf(s,W,Pile),cf(p,W,Pile)).
delta(cf(p,[a|W],[aa|Pile]),cf(p,W,Pile)).
delta(cf(p,[b|W],[bb|Pile]),cf(p,W,Pile)).
delta(cf(p,W,[],),cf(q,W,[])).

initial(s).

final(q).
```

Un automate à pile spécialisé est obtenu par dépliage de la fonction de transition `delta` dans `derivable` et `accepte`.

```
miroir(W) :-  miroir_(s,W,[]).
miroir_(s,[a|W],Pile) :-
   miroir_(s,W,[aa|Pile]).
miroir_(s,[b|W],Pile) :-
   miroir_(s,W,[bb|Pile]).
miroir_(s,W,Pile) :-
   miroir_(p,W,Pile).
miroir_(p,[a|W],[aa|Pile]) :-
   miroir_(p,W,Pile).
miroir_(p,[b|W],[bb|Pile]) :-
   miroir_(p,W,Pile).
miroir_(p,W,[]) :-
   miroir_(q,W,[]).
miroir_(q,[],_).
```

Grammaires hors-contexte

- Soit la grammaire
  $S \to \varepsilon$
  $S \to aSbSc$.
- La traduction directe est :

```
s([]).
s(W) :-
    conc([a],S1,W2),
    conc(W2,[b],W3),
    conc(W3,S2,W4),
    conc(W4,[c],W),
    s(S1),
    s(S2).
```

- En introduisant les listes en différence et en simplifiant :

```
s(S) :-
   s_ld(S,[]).
s_ld(S,S).
s_ld([a|S1],SW) :-
   s_ld(S1,[b|S2]),
   s_ld(S2,[c|SW]).
```

- La traduction des grammaires hors-contexte peut être faite entièrement automatiquement.
- La plus part des implantations de la programmation logique en clauses de Horn intègrent cette traduction ainsi que les mécanismes d'héritage et de synthèse.

La méta-interprétation

- Méta-interprétation : définir un interpréteur en lui-même.
- Objectifs de la méta-interprétation en Prolog :
  - modifier la stratégie de recherche ;
  - modifier la stratégie de sélection ;
  - enrichir le langage de nouvelles primitives.

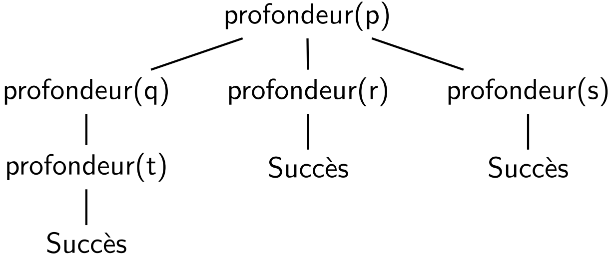
Méta-interpréteur Prolog sans résolvante

- la gestion des variables logiques, de l'unification, de l'indéterminisme et du méta-programme est laissée à Prolog ;
- une clause définie $p :- q, r$ est codée sous la forme : `cl(p,[q,r])` ;
- un fait p est codé sous la forme `cl(p,[])` ;
- absence de résolvante : parallèlisme de la conjonction sur les sous-buts (branches de l'arbre de preuve).

```
profondeur(But) :-
   cl(But,Corps),
   map(profondeur,Corps).
```

```
cl(p,[q,r,s]).
cl(q,[t]).
cl(r,[]).
cl(s,[]).
cl(t,[]).
```

Arbre de preuve de la méta-interprétation :



Méta-interpréteur Prolog avec résolvante

- la gestion des variables logiques, de l'unification et de l'indéterminisme est laissée à Prolog ;
- une clause définie $p :- q, r$ est codée sous la forme : `cl(p,[q,r])` ;
- un fait p est codé sous la forme `cl(p,[])` ;
- la gestion de la résolvante est assurée par une pile ;
- le méta-programme est géré sous la forme d'une liste.

```
profondeur([],_Programme).
profondeur([But|Resolvante],Programme) :-
   recherche(But,Programme,Corps),
   conc(Corps,Resolvante,NouvResolvante),
   profondeur(NouvResolvante,Programme).

recherche(But, [cl(Tete,Corps)|_], Buts) :-
   copy_term(cl(Tete,Corps),cl(But,Buts)).
recherche(But, [_|Programme], Buts) :-
   recherche(But, Programme, Buts).
```

```
? P = [cl(p,[q,r,s]),cl(q,[t]),
cl(r,[]),cl(s,[]),cl(t,[])],
profondeur([p],P).
```

Arbre de preuve de la méta-interprétation :

profondeur([p],P)
|
profondeur([q,r,s],P)
|
profondeur([t,r,s],P)
|
profondeur([r,s],P)
|
profondeur([s],P)
|
Succès

Méta-interpréteur Prolog en largeur d'abord

- le méta-interprète propositionnel ;
- une clause définie $p :- q, r$ est codée sous la forme : `cl(p,[q,r])` ;
- un fait p est codé sous la forme `cl(p,[])` ;
- la gestion de la résolvante est assurée par une pile ;
- le méta-programme est géré sous la forme d'une liste ;
- la gestion de l'indéterminisme est assuré par inférence simultanée sur une liste de résolvante.

```
largeur_d_abord(ResS,_P) :- membre([],ResS).
largeur_d_abord(ResS,P) :-
   mapinfere(ResS,P,NouvellesResS),
   largeur_d_abord(NouvellesResS,P).

mapinfere([],_P,[]).
mapinfere([[X|Resolvante]|ResS],P,NouvellesResS) :-
   infere(X,P,CorpsS),
   map(conc(_,CorpsS,_),Resolvante,DesResS),
   mapinfere(ResS,P,NouvellesResS_),
   conc(DesResS,NouvellesResS_,NouvellesResS).

infere(_X,[],[]).
infere(X,[cl(X,Corps)|P],[Corps|CorpsS]) :-
   infere(X,P,CorpsS).
infere(X,[cl(Y,_Corps)|P],CorpsS) :-
   \+(X=Y),
   infere(X,P,CorpsS).
```

```
? P = [cl(p,[q]),cl(q,[p]),cl(q,[])],
largeur_d_abord([p],P).
```

Arbre de preuve de la méta-interprétation en largeur d'abord :

largeur_d_abord([[p]],P)

|

largeur_d_abord([[q]],P)

|

largeur_d_abord([[p],[]],P)

|

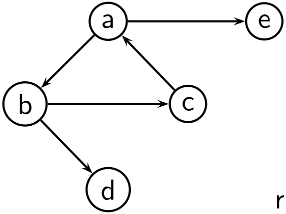
Succès

Recherche en profondeur d'abord dans un graphe

- Prérequis : le prédicat `successeur` est tel que `successeur(X,G,L)` est vrai si `L` est la liste des successeurs de `X` dans le graphe `G`.
- Extension de l'algorithme de recherche en profondeur dans un arbre.
- Risque de boucle dans les graphes cycliques.
  - Une liste `Ouverts` conserve les sommets non encore traversés.
  - Une liste `Fermes` conserve les sommets déjà traversés.

```
recherche_profondeur(Depart,Objectif,Graphe):-
   rp([Depart],Objectif,Graphe,[]).

rp([Objectif|_],Objectif,_Graphe,_Fermes).
rp([Sommet|Ouverts],Objectif,Graphe,Fermes):-
   (membre(Sommet,Fermes) ->
      (Ouverts_ = Ouverts,
       Fermes_ = Fermes) ;
      (successeurs(Sommet,Graphe,Successeurs),
       conc(Successeurs,Ouverts,Ouverts_),
       Fermes_ = [Sommet|Fermes])),
   rp(Ouverts_,Objectif,Graphe,Fermes_).
```



Arbre de preuve de la recherche en profondeur d'abord :

rp([a],d,Graphe,[])

rp([b,e],d,Graphe,[a])

rp([c,d,e],d,Graphe,[b,a])

rp([a,d,e],d,Graphe,[c,b,a])

rp([d,e],d,Graphe,[c,b,a])

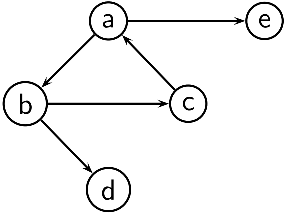
Succès

La liste des "fermés" est indispensable à la terminaison.

Recherche en largeur d'abord dans un graphe

```
recherche_largeur(Depart,Objectif,Graphe):-
   rl([Depart],Objectif,Graphe,[]).

rl([Objectif|_],Objectif,_Graphe,_Fermes).
rl([Sommet|Ouverts],Objectif,Graphe,Fermes):-
   (membre(Sommet,Fermes) ->
      (Ouverts_ = Ouverts,
       Fermes_ = Fermes) ;
      (successeurs(Sommet,Graphe,Successeurs),
       conc(Ouverts,Successeurs,Ouverts_),
       Fermes_ = [Sommet|Fermes])),
   rl(Ouverts_,Objectif,Graphe,Fermes_).
```



Arbre de preuve de la recherche en largeur d'abord :

rl([a],d,Graphe,[])

rl([b,e],d,Graphe,[a])

rl([e,c,d],d,Graphe,[b,a])

rl([c,d],d,Graphe,[e,b,a])

rl([d,a],d,Graphe,[c,e,b,a])

Succès

La liste des "fermés" est indispensable à la terminaison sur un échec (mais une optimisation si la recherche est assurée de réussir).