Plan du cours de “Programmation logique”

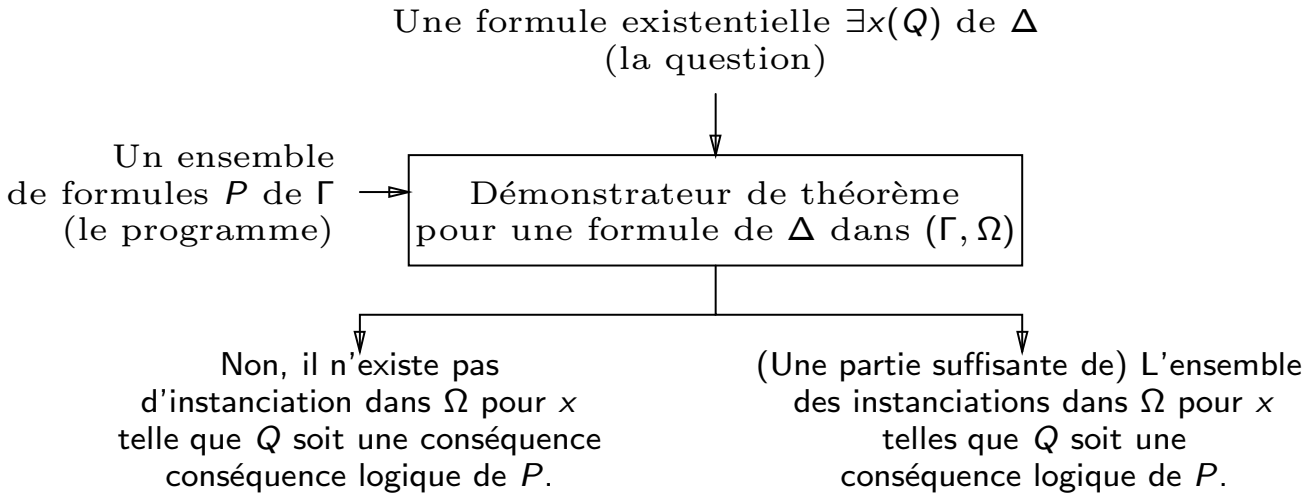
La logique comme langage de programmation

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

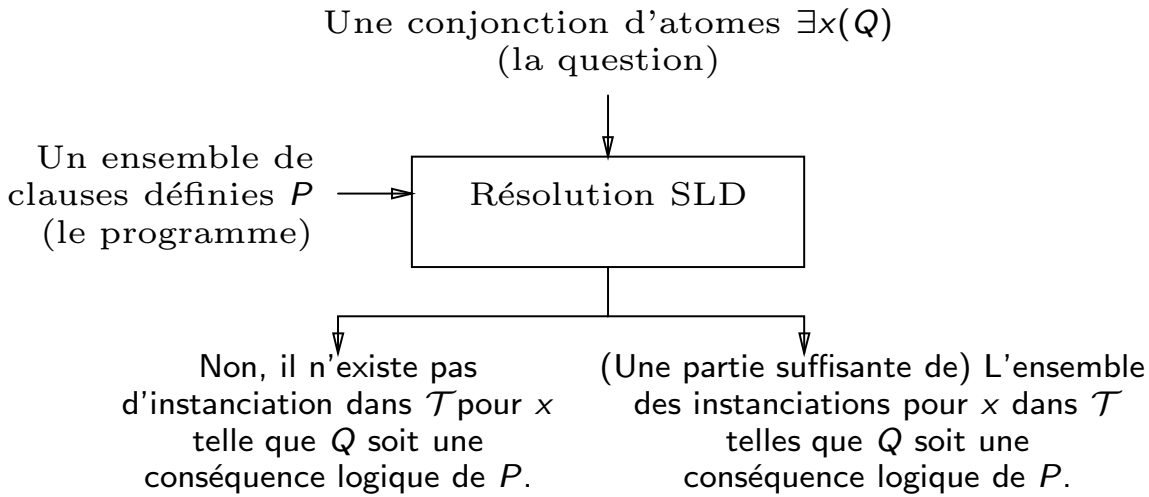
- Analyse de la "Langue naturelle" ;
- Intelligence artificielle et modélisation des raisonnements (les Systèmes experts, le Diagnostique, ...) ;
- Bases de données ;
- Prototypage ;
- Compilation ;
- Automates formels déterministes et non-déterministes ;
- Vérification de programmes ; ...

Plan du cours de “Programmation logique”

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."

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

  $boit(socrate, cigue), poison(cigue)$ 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).
```

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).

Notation :

$$\sigma(\{t_1 = t'_1, \ldots, t_n = t'_n\}) =$$
$$\{\sigma(t_1) = \sigma(t'_1), \ldots, \sigma(t_n) = \sigma(t'_n)\}$$

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.

Questions existentielles

- Dans une question, la variable logique exprime une interrogation sur l'existence d'une donnée qui vérifie le prédicat.
  "Existe-t-il une donnée 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 question peut être une conjonction d'atomes...

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

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

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

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.

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

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 $\sigma_0 = upg(meurt(X_1), meurt(X)) = [X \leftarrow X_1]$
$\sigma_0(\theta_0(mortel(X)), \theta_0(empoisonne(X))) = mortel(X_1), empoisonne(X_1).$

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

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 $\sigma_3 = upg(poison(cigue), poison(Y_2)) = [Y_2 \leftarrow cigue]$
$\sigma_3(animal(X_3), boit(X_3, Y_2)) = animal(X_3), boit(X_3, cigue).$

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

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

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

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

$$\overbrace{\underline{boit(socrate, cigue)}}^{1} \quad C_6,1 \in \{1\}$$

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

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

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

et

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

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

meurt(platon) ?

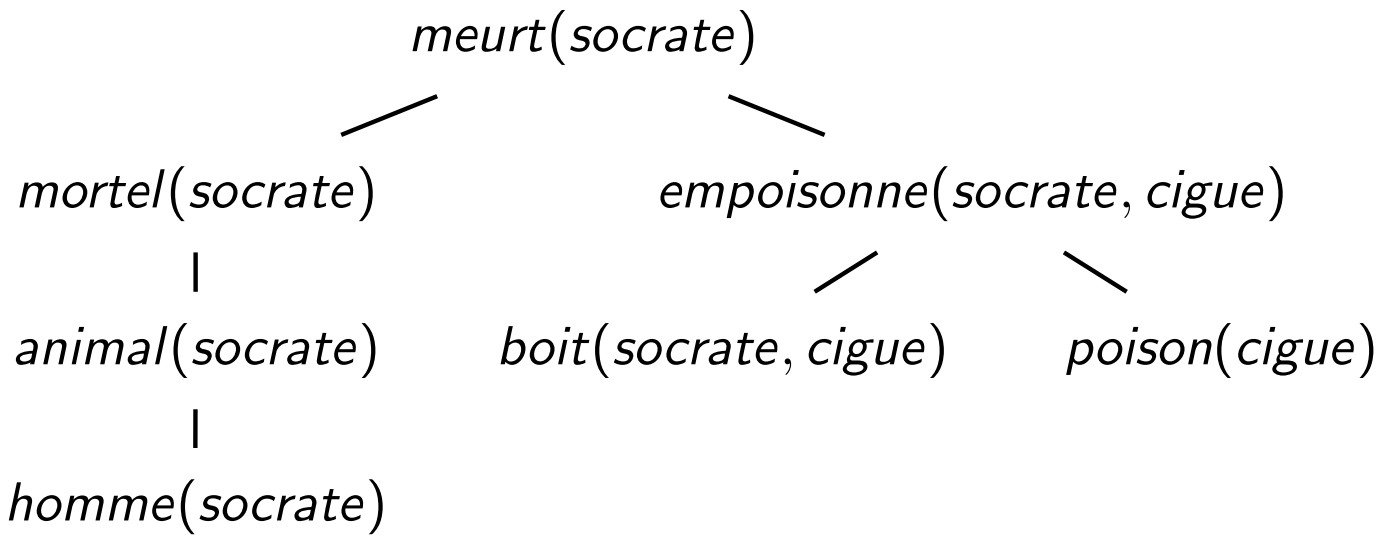
$$\frac{\overbrace{homme(X_5)}^{1}, \overbrace{boit(X_5, cigue)}^{2}}{boit(platon, cigue)} \quad C_5,1 \in \{1,2\}$$

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

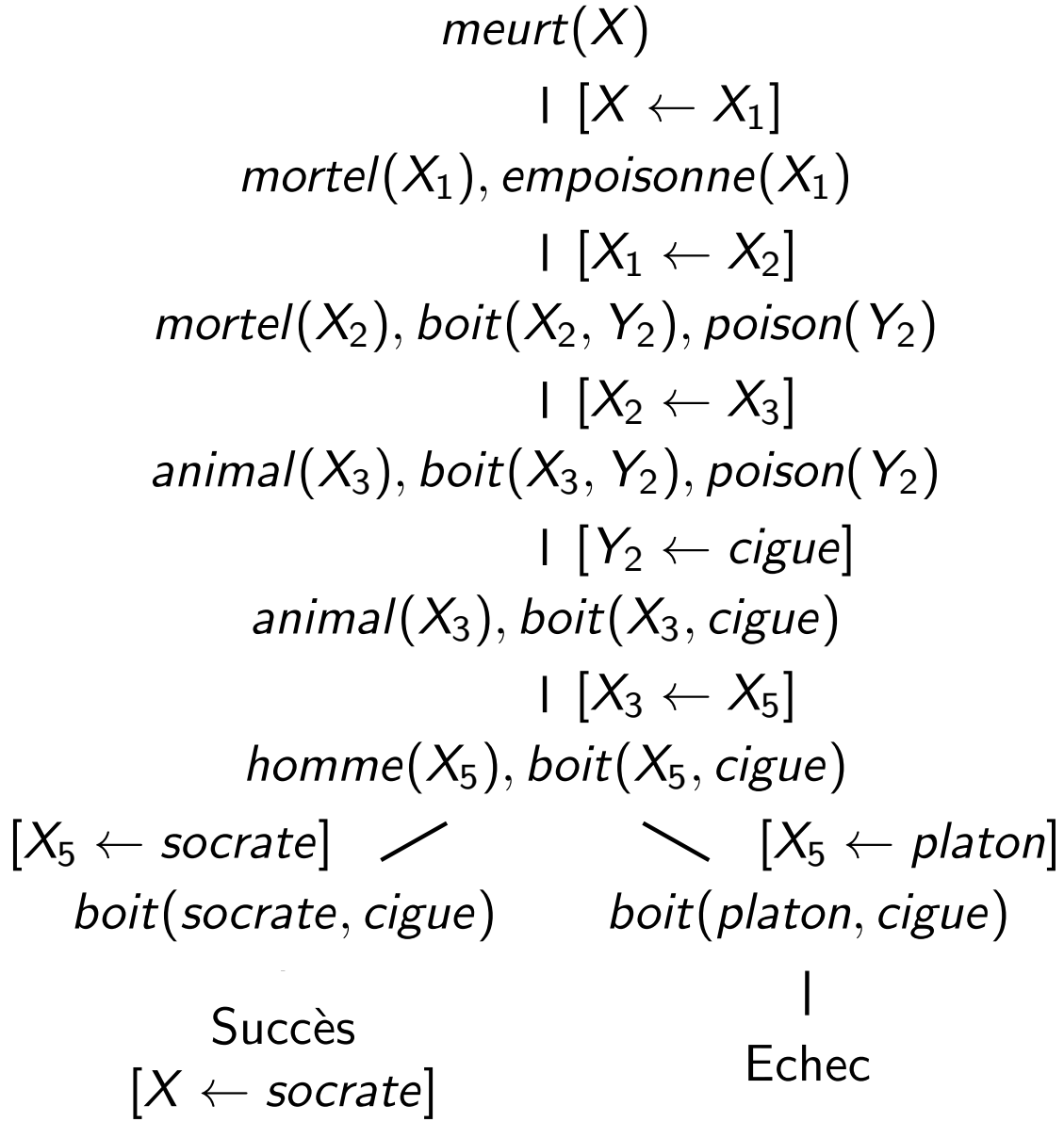
- Représentation d'une dérivation (instanciée) sous forme d'arbre de preuve.



- 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)
[X ← X1]
mortel(X1), empoisonne(X1)
[X1 ← X2]
mortel(X2), boit(X2, Y2), poison(Y2)
[X2 ← X3]
animal(X3), boit(X3, Y2), poison(Y2)
[Y2 ← cigue]
animal(X3), boit(X3, cigue)
[X3 ← X5]
homme(X5), boit(X5, cigue)
[X5 ← socrate]
[X5 ← platon]
boit(socrate, cigue)
boit(platon, cigue)
Succès
[X ← 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 êatre quelconque.

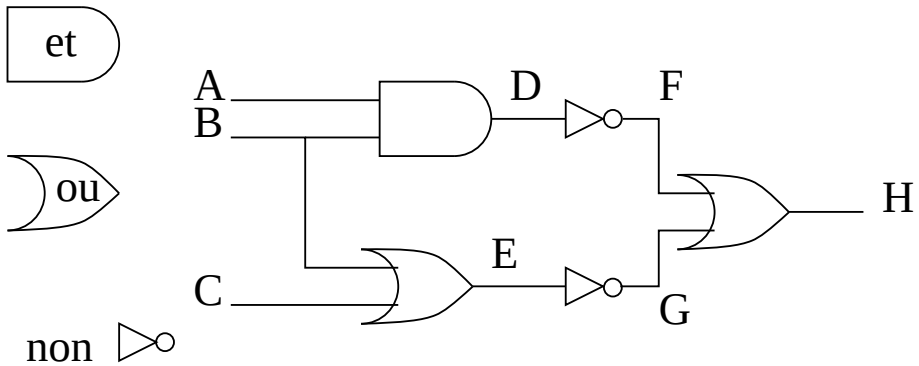
De la réversibilité

- 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 :

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

Calcul de la relation complète :

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

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.

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

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

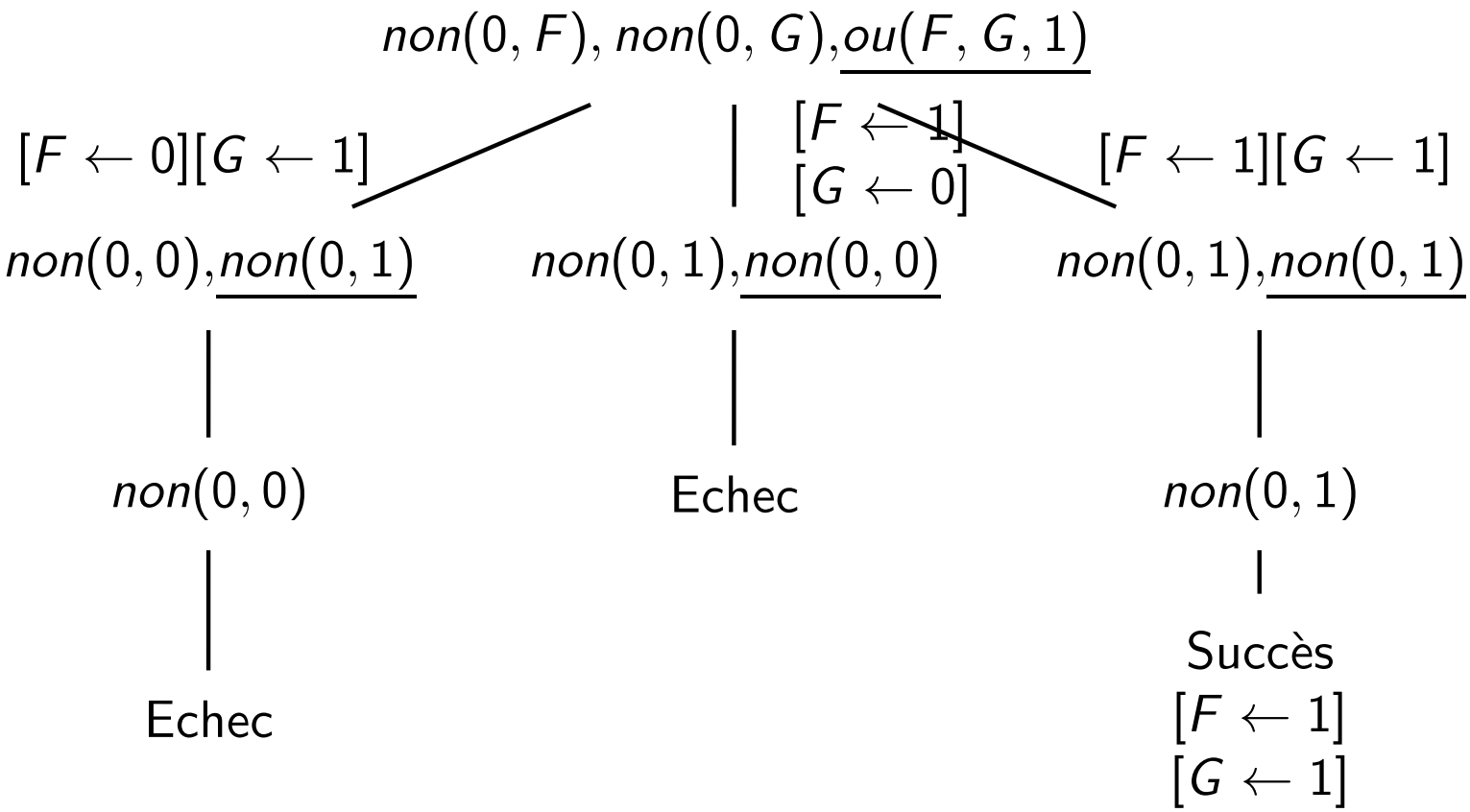
$$\underline{non(0, G)}, ou(1, G, 1)$$

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

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

$$|$$

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


non(0, F), non(0, G), ou(F, G, 1)
[F ← 0][G ← 1]
[F ← 1]
[G ← 0]
[F ← 1][G ← 1]
non(0, 0), non(0, 1)
non(0, 1), non(0, 0)
non(0, 1), non(0, 1)
non(0, 0)
Echec
non(0, 1)
Echec
Succès
[F ← 1]
[G ← 1]

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 :
  $disjointe(Xs, Ys) : -\+(membre(Z, Xs), membre(Z, Ys))$.

Plan du cours de “Programmation logique”

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...

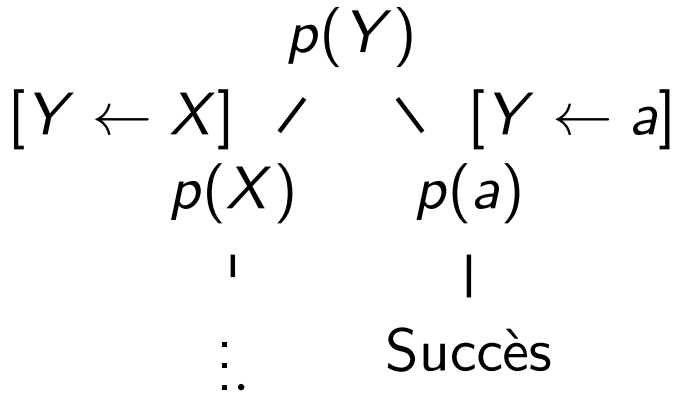
Prolog est un sous-ensemble de la programmation logique en clauses de Horn

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

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

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

? $p(Y)$.
∞



? $p(Y)$
$[Y \leftarrow a]$
∞

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

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.

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

Etendre une spécification

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).
```

Le prédicat *longueur*

Le prédicat *longueur* est tel que *longueur*(L, N) est vrai si l'entier N est la longueur de la liste L.

```
longueur([],zero).
longueur([X|L],s(N)) :-
   longueur(L,N).
```

Le prédicat *prefixe*

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*

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*

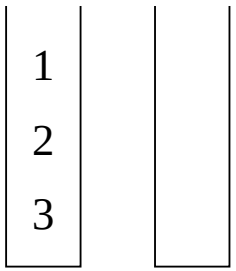
Le prédicat *renverse* est tel que *renverse*(L, R) est vrai si la liste R est la liste L dont l'ordre des éléments est inversé.

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

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

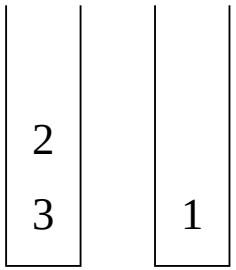
$$\frac{renverse([1,2,3], Renv)}{renverse\_([1,2,3], [], Renv)}$$

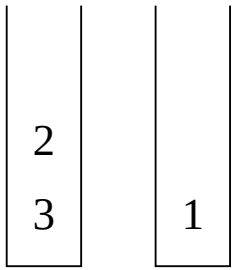
$$\theta_0 = [L \leftarrow L_1][R \leftarrow R_0]$$
$$\sigma_0 = upg(\ renverse(L_1, R_0),\ renverse([1,2,3], Renv))$$
$$= [L1 \leftarrow [1,2,3]][R_0 \leftarrow Renv]$$



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

$$\theta_1 = [X \leftarrow X_1][L \leftarrow L_1][Acc \leftarrow Acc_1][R \leftarrow R_1]$$
$$\sigma_1 = upg(\ renverse\_([X_1|L_1], Acc_1, R_1),\ renverse\_([1,2,3], [], Renv))$$
$$= \ [X_1 \leftarrow 1][L_1 \leftarrow [2,3]][Acc_1 \leftarrow []][R_1 \leftarrow Renv]$$
$$\sigma_1(\theta_1(renverse\_(L, [X|Acc], R)))$$
$$= renverse\_([2,3], [1|[]], Renv)$$





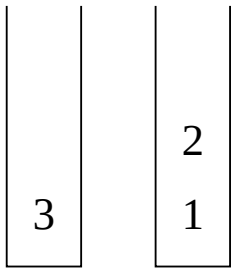
$$\frac{renverse\_([2,3],[1],Renv)}{renverse\_([3],[2,1],Renv)}$$

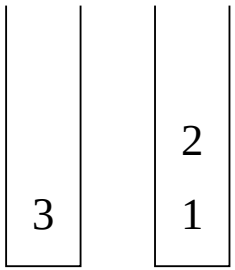
$\theta_2 = [X \leftarrow X_2][L \leftarrow L_2][Acc \leftarrow Acc_2][R \leftarrow R_2]$

$\sigma_2 = upg(\ renverse\_([X_2|L_2], Acc_2, R_2),$
$renverse\_([2,3],[1],Renv))$

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

$\sigma_2(\theta_2(renverse\_(L,[X|Acc],R)))$
$= renverse\_([3],[2|[1]],Renv)$





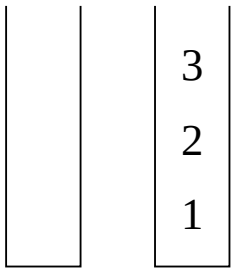
$$\frac{renverse\_([3],[2,1],Renv)}{renverse\_([],[3,2,1],Renv)}$$

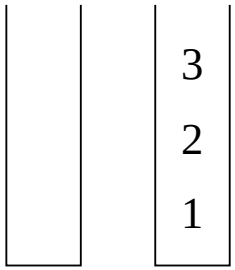
$\theta_3 = [X \leftarrow X_3][L \leftarrow L_3][Acc \leftarrow Acc_3][R \leftarrow R_3]$

$\sigma_3 = upg(\ renverse\_([X_3|L_3], Acc_3, R_3),$
$renverse\_([3],[2,1],Renv))$

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

$\sigma_3(\theta_3(renverse\_(L,[X|Acc],R)))$
$= renverse\_([],[3|[2,1]],Renv)$





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

$\theta_4 = [Acc \leftarrow Acc_4]$

$\sigma_4 = upg(\ renverse\_([], Acc_4, Acc_4),$
$\qquad\qquad renverse\_([], [3,2,1], Renv))$

$= \quad [Renv \leftarrow Acc_4][Acc_4 \leftarrow [3,2,1]]$

La réponse calculée pour la question $renverse([1,2,3], Renv)$ est
$[Renv \leftarrow Acc_4][Acc_4 \leftarrow [3,2,1]](renverse([1,2,3], Renv))$
$= \quad renverse([1,2,3], [3,2,1])$

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).
```

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,['<','>=','>','=<',':=','=\=']).
```