Le langage Prolog

Elise Bonzon
http://www.math-info.univ-paris5.fr/~bonzon/Cours/prolog.htm
`elise.bonzon@parisdescartes.fr`

Master 1 Informatique
UFR Mathématiques et Informatique
Université Paris Descartes

PROgrammation LOGique

- Origines
  - 1972 : Marseille, A. Colmerauer
  - 1980 : reconnaissance de Prolog comme le langage de développement en Intelligence Artificielle
  - Dans les années 90 : une version de Prolog tournée vers la programmation par contraintes
- Bibliographie
  - *L'art de Prolog*, L. Sterling, E. Shapiro, *Masson*
  - *Programmer en Prolog*, Clocksin, Mellosh, *Eyrolles*

Le langage PROLOG

- Langages de style impératif : C, ADA, PASCAL, JAVA (objet)...
- Langages de style fonctionnel : Lisp, Scheme, CAML, ML...
- Langage de style déclaratif : Prolog
  - $\rightarrow$ On “raconte” le problème à résoudre, mais *pas* comment on le résoud
  - $\rightarrow$ Souvent utilisé en Intelligence Artificielle

Le langage PROLOG

- Langage d'expression des connaissances fondé sur la logique des prédicats du 1er ordre
- Implémentation du **principe de résolution**
- Programmation **déclarative**
  - L'utilisateur définit une base de connaissances (des faits, des règles)
    - Pas d'instruction, pas d'affectation, pas de boucles explicites
    - Uniquement des **clauses** exprimant des faits, des règles, des questions
  - L'interpréteur PROLOG utilise cette base de connaissances pour répondre à des questions

Exemples de faits

```
pere(alain, jeanne).
pere(alain, michel).
pere(michel, robert).
mere(sylvie, robert).
mere(sylvie, luc).
pere(georges, sylvie).
```

Exemples de questions

Questions	Réponses	
`:- pere(alain, jeanne).`	oui	
`:- pere(alain, robert).`	non	
`:- pere(alain, X).`	oui	`X=michel`
		`X=jeanne`
`:- mere(X, robert).`	oui	`X=sylvie`
`:- mere(X, michel).`	non	
`:- pere(alain, X), pere(X, robert).`	oui	`X=michel`

Exemple de règle

- Relation *grand-père* en logique du premier ordre :

$$\forall X \forall Y (\exists Z \; pere(X,Z) \wedge (pere(Z,Y) \vee mere(Z,Y))) \Rightarrow papy(X,Y)$$

$$\begin{aligned} \forall X \forall Y \forall Z \quad & (((pere(X,Z) \wedge pere(Z,Y)) \Rightarrow papy(X,Y)) \\ & \wedge ((pere(X,Z) \wedge mere(Z,Y)) \Rightarrow papy(X,Y))) \end{aligned}$$

- Formulation prolog :

```
papy(X,Y) :- pere(X,Z), pere(Z,Y).
papy(X,Y) :- pere(X,Z), mere(Z,Y).
```

Interpréteurs PROLOG

- Il existe de nombreux interpréteurs Prolog
- Présentation de la syntaxe et des conventions les plus couramment utilisées à l'heure actuelle

  $\rightarrow$ SWI-Prolog

Logique propositionelle - syntaxe

- Logique propositionnelle = logique très simple
- Un **énoncé** est un énoncé atomique ou un énoncé complexe
- Un **symbole propositionnel** est une proposition qui peut être vraie ou fausse P, Q, R...
- Enoncés atomiques : un seul symbole propositionnel, *vrai* ou *faux*
- Enoncés complexes :
  - Si E est un énoncé, $\neg E$ est un énoncé (négation)
  - Si E_1 et E_2 sont des énoncés, $E_1 \wedge E_2$ est un énoncé (conjonction)
  - Si E_1 et E_2 sont des énoncés, $E_1 \vee E_2$ est un énoncé (disjonction)
  - Si E_1 et E_2 sont des énoncés, $E_1 \Rightarrow E_2$ est un énoncé (implication)
  - Si E_1 et E_2 sont des énoncés, $E_1 \Leftrightarrow E_2$ est un énoncé (équivalence)

Logique propositionelle - sémantique

- Un modèle : une valeur de vérité (*vrai* ou *faux*) pour chaque symbole propositionnel
  - 3 symboles propositionnels $P_{1,1}$, $P_{2,2}$ et $P_{3,1}$
  - $m_1 = \{P_{1,1} = Faux,\ P_{2,2} = Faux,\ P_{3,1} = Vrai\}$
- n symboles propositionnels = 2^n modèles possibles
- Règles pour évaluer un énoncé en fonction d'un modèle m :

$\neg E$	est vrai ssi	E est faux
$E_1 \wedge E_2$	est vrai ssi	E_1 est vrai **et** E_2 est vrai
$E_1 \vee E_2$	est vrai ssi	E_1 est vrai **ou** E_2 est vrai
$E_1 \Rightarrow E_2$	est vrai ssi	E_1 est faux **ou** E_2 est vrai
$E_1 \Rightarrow E_2$	est faux ssi	E_1 est vrai **et** E_2 est faux
$E_1 \Leftrightarrow E_2$	est vrai ssi	$E_1 \Rightarrow E_2$ est vrai **et** $E_2 \Rightarrow E_1$ est vrai

- Un processus récursif simple permet d'évaluer un énoncé. Par exemple :

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = Vrai \wedge (Faux \vee Vrai) = Vrai \wedge Vrai = Vrai$$

Table de vérité des connecteurs logiques

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
vrai	*vrai*	*faux*	*vrai*	*vrai*	*vrai*	*vrai*
vrai	*faux*	*faux*	*faux*	*vrai*	*faux*	*faux*
faux	*vrai*	*vrai*	*faux*	*vrai*	*vrai*	*faux*
faux	*faux*	*vrai*	*faux*	*faux*	*vrai*	*vrai*

Logique du premier ordre

- Logique propositionnelle : le monde contient des faits
- Logique du 1er ordre : comme dans le langage naturel, le monde contient des :
  - Objets : personnes, maisons, nombres, couleurs, match de foot, guerres...
  - Relations :
    - **relations unaires** ou **propriétés** : rouge, arrondi, faux, premier...
    - **relations *n*-aires** : frère-de, plus-grand-que, est-de-couleur, possède...
  - Fonctions : une seule “valeur” pour une “entrée” donnée : père de, meilleur ami, un de plus que...

Syntaxe de la logique du 1er ordre : élément de base

- Constantes : 2, $Jean$, X_1, ...
- Prédicats : $Frere$, $>$, $Avant$, ...
- Fonctions : $RacineCarre$, $JambeGauche$, ...
- Variables : x, y, a, b, ...
- Connecteurs : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- Egalité : $=$
- Quantificateurs : $\forall$, $\exists$

Enoncés atomiques

- Terme : Constante, Variable ou Fonction($t_1, \ldots, t_n$)
  - Exemple : *Jean*, *Pere*(*Richard*)
- Enoncé atomique : Prédicat($t_1, \ldots, t_n$) ou $t_1 = t_2$
  - Exemple : *Frere*(*Richard*, *Jean*); *Marie*(*Pere*(*Richard*), *Mere*(*Jean*))

Enoncés composés

- Les énoncés composés sont construits à partir des énoncés atomiques et des connecteurs
- $\neg E$, $E_1 \wedge E_2$, $E_1 \vee E_2$, $E_1 \Rightarrow E_2$, $E_1 \Leftrightarrow E_2$
- $Frere(John, Richard) \wedge Frere(Richard, John)$
- $> (1,2) \vee \leq (1,2)$

Modèles de la logique du 1er ordre

- La vérité d'un énoncé est déterminée par un modèle et une interprétation des symboles de l'énoncé
- Un modèle contient des **objets** (appelés éléments du domaine) qui sont liés entre eux par des relations
- Une interprétation spécifie à quoi réfèrent les symboles de l'énoncé :
  - Symboles de constantes $\rightarrow$ objets
  - Symboles de prédicats $\rightarrow$ relations
  - Symboles de fonctions $\rightarrow$ fonctions
- Un énoncé atomique est **vrai** dans un modèle donné, compte tenu d'une interprétation donnée, si la **relation** à laquelle renvoit le symbole du prédicat s'applique aux objets en arguments.

Quantification universelle

- $\forall \langle variables \rangle \langle enonce \rangle$
- Tous les étudiants sont intelligents :

$$\forall x \; Etudiant(x) \Rightarrow \; Intelligent(x)$$

- $\forall xP$ est vrai dans un modèle si et seulement si P est vrai **pour tous** les objets x
- $\forall xP$ est équivalent à la **conjonction** de toutes les **instanciations** de P :

$$\begin{array}{rc} & Etudiant(Paul) \Rightarrow Intelligent(Paul) \\ \wedge & Etudiant(Pierre) \Rightarrow Intelligent(Pierre) \\ \wedge & Etudiant(Sophie) \Rightarrow Intelligent(Sophie) \\ \wedge & Etudiant(Julie) \Rightarrow Intelligent(Julie) \\ \wedge & \vdots \end{array}$$

Quantification universelle : erreur fréquente à éviter

- Le connecteur principal à utiliser avec $\forall$ est l'implication $\Rightarrow$
- Erreur fréquente : utiliser la conjonction $\wedge$ comme connecteur principal avec $\forall$
- $\forall x \; Etudiant(x) \wedge Intelligent(x)$ signifie "tout le monde est étudiant et est intelligent"

Quantification existentielle

- $\exists \langle variables \rangle \langle enonce \rangle$
- Un étudiant est intelligent :

$$\exists x \; Etudiant(x) \wedge \; Intelligent(x)$$

- $\exists xP$ est vrai dans un modèle si et seulement si P est vrai **pour un** objet x
- $\exists xP$ est équivalent à la **disjonction** de toutes les **instanciations** de P :

$$\begin{array}{rc} & Etudiant(Paul) \wedge Intelligent(Paul) \\ \vee & Etudiant(Pierre) \wedge Intelligent(Pierre) \\ \vee & Etudiant(Sophie) \wedge Intelligent(Sophie) \\ \vee & Etudiant(Julie) \wedge Intelligent(Julie) \\ \vee & \vdots \end{array}$$

Quantification existentielle : erreur fréquente à éviter

- Le connecteur principal à utiliser avec $\exists$ est la conjonction $\wedge$
- Erreur fréquente : utiliser l'implication $\Rightarrow$ comme connecteur principal avec $\exists$
- $\exists x \; Etudiant(x) \Rightarrow \; Intelligent(x)$ est vrai s'il existe quelqu'un qui n'est pas étudiant!

Propriétés des quantificateurs

- $\forall x \forall y$ est équivalent à $\forall y \forall x$
- $\exists x \exists y$ est équivalent à $\exists y \exists x$
- $\exists x \forall y$ n'est pas équivalent à $\forall y \exists x$
  - $\exists y \forall x\ Aime(x, y)$ "Il existe une personne qui est aimé par tout le monde"
  - $\forall x \exists y\ Aime(x, y)$ "Tout le monde aime quelqu'un" (pour toute personne, il existe quelqu'un qu'il aime)
- Liens entre $\forall$ et $\exists$: Les deux quantifieurs sont liés par le biais de la négation :
  - $\forall x\ Aime(x, Glace)$ est équivalent à $\neg \exists x \neg Aime(x, Glace)$
  - $\exists x\ Aime(x, Brocoli)$ est équivalent à $\neg \forall x \neg Aime(x, Brocoli)$

Clauses de Horn

- Clauses de Horn : disjonction de littéraux dont **un au maximum est positif**
  - $(\neg P(x) \vee \neg B \vee C)$ est une clause de Horn
  - $(\neg C \vee P_{1,2} \vee P_{2,1})$ n'est pas une clause de Horn
- Toute clause de Horn peut s'écrire sous la forme d'une implication avec
  - Prémisse = conjonction de littéraux positifs
  - Conclusion = littéral positif unique
  - $(\neg P(x) \vee \neg B \vee C) = ((P(x) \wedge B) \Rightarrow C)$
- Clauses définies : clauses de Horn ayant **exactement** un littéral positif
- Littéral positif = tête; littéraux négatifs = corps de la clause
- Fait = clause sans littéraux négatifs

Constantes et variables

- Constantes
  - Nombres : 128, 4.5
  - Chaîne de caractères commençant par une minuscule
  - Chaîne de caractères entre “”
- Variables
  - Chaîne de caractères commençant par une majuscule
  - Variable **anonyme** (ou indéterminée) : _
  - Chaîne de caractères commençant par _

Connaissances : faits, règles

- Programme Prolog : ensemble de faits et de règles qui décrivent un problème
- Faits
  - $P(...)$. avec P un prédicat
  - Clauses de Horn, réduites à un littéral positif

    ```
    pere(jean,paul).
    mere(jean,marie).
    ```
- Règles
  - Clauses de Horn complètes
  - $P : -Q_1, \ldots, Q_n$.
  - Correspond en logique à $Q_1 \wedge \ldots \wedge Q_n \rightarrow P$
  - P : **tête** de clause, $Q_1, \ldots, Q_n$ **queue** de clause

    ```
    papy(X,Y) :- pere(X,Z), pere(Z,Y).
    ```

Connaissances : questions

- Questions (ou but, conclusion) :
  - Clauses de Horn sans littéral positif
  - $Q_1, \ldots, Q_n$.
  - `:- pere(jean, X), papy(jean,Y).`

→ Exécuter un programme = résoudre un but

  - Trouver toutes les valeurs des variables qui apparaissent dans la question et qui amènent à une contradiction

  → Faire une preuve

Unification

Unification : définition

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

Unification

Unification : définition

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

- Le résultat est un unificateur
  - Exemple : $\{Jean/X, Paul/Y\}$
- Résultat pas forcément unique, on cherche **l'unificateur le plus général**
- L'unification peut réussir ou échouer
  - $e(X,X)$ et $e(2,3)$ ne peuvent pas être unifiés

Déclaratif vs procédural

Soit la clause `p :- q,r`

- Aspect déclaratif : *p est vrai si q et r sont vrais*
  - → L'ordre des clauses n'a pas d'importance
- Aspect procédural : *Pour résoudre p, résoudre d'abord le sous-problème q, puis le sous-problème r*
  - → L'interpréteur considère les clauses les unes après les autres, dans l'ordre dans lesquelles elles se trouvent, celui-ci a de l'importance.

Littéraux ordonnés

- $b :- a_1, a_2, \ldots, a_n.$
  - pour résoudre b, il faut résoudre dans l'ordre a_1 puis a_2 puis... puis a_n
- $b.$
  - b est toujours vrai
- $:- a_1, a_2, \ldots, a_n.$
  - pour résoudre la question, il faut résoudre dans l'ordre a_1 puis a_2 puis... puis a_n

Résolution par réfutation : exemple

- Programme P
  - P_1 : `pere(charlie,david).`
  - P_2 : `pere(henri,charlie).`
  - P_3 : `papy(X,Y) :- pere(X,Z), pere(Z,Y).`
- Question
  - Q : `papy(X,Y).`

Graphe de résolution

$Q : \neg papy(X, Y)$

Graphe de résolution

$Q : \neg papy(X, Y)$ $\qquad$ $P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

Graphe de résolution

$Q : \neg papy(X, Y)$

$P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

Graphe de résolution

$Q : \neg papy(X, Y)$ $\quad\quad$ $P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$ $\quad$ $\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

Graphe de résolution

$Q : \neg papy(X, Y)$ $\qquad$ $P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$ $\qquad$ $\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

$\{charlie/X$
$david/Z_1\}$

$\neg pere(david, Y)$

Graphe de résolution

$Q : \neg papy(X, Y)$

$P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$

$\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

$\{charlie/X$
$david/Z_1\}$

$\neg pere(david, Y)$

Echec - Retour arrière

Graphe de résolution

$Q : \neg papy(X, Y)$

$P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$

$\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

$P_2 : pere(henri, charlie)$

$\{charlie/X$
$david/Z_1\}$

$\neg pere(david, Y)$

Echec - Retour arrière

Graphe de résolution

$Q : \neg papy(X, Y)$

$P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$

$\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

$P_2 : pere(henri, charlie)$

$\{charlie/X$
$david/Z_1\}$

$\{henri/X, charlie/Z_1\}$

$\neg pere(david, Y)$

$\neg pere(charlie, Y)$

Echec - Retour arrière

Graphe de résolution

$Q : \neg papy(X, Y)$

$P_3 : papy(X_1, Y_1) \vee \neg pere(X_1, Z_1) \vee \neg pere(Z_1, Y_1)$

$\{X/X_1, Y/Y_1\}$

$P_1 : pere(charlie, david)$

$\neg pere(X, Z_1) \vee \neg pere(Z_1, Y)$

$P_2 : pere(henri, charlie)$

$\{charlie/X$
$david/Z_1\}$

$\{henri/X, charlie/Z_1\}$

$\neg pere(david, Y)$

Echec - Retour arrière

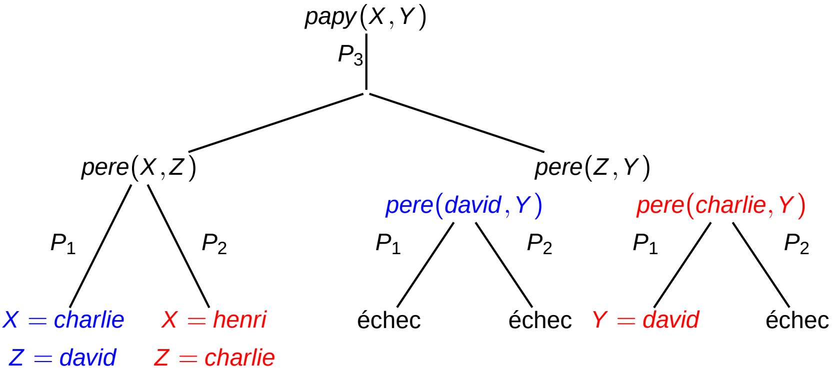
$\neg pere(charlie, Y)$

$P_1 : pere(charlie, david)$

$\{david/Y\}$

succès

Arbre de recherche : arbre ET/OU



Les listes

- Structure très importante en Prolog
- Peut contenir tout type de termes
- Entre crochets, éléments séparés par des virgules
- Exemple :

```
[a,b,c]
[]
[Y,b,[1,2,3]]
[Y,b,Z]
```

Unification des listes

- Une liste entière :

```
?- X = [a,b,c]
X=[a,b,c]
yes
```

- Certains éléments de la liste :

```
?- [a,b,c] = [X,Y,Z]
X = a
Y = b
Z = c
yes
```

```
?- [a,b,c] = [X,Y,c]
X = a
Y = b
yes
```

```
?- [a,b,c] = [X,Y,b]
no
```

Unification des listes

- Avec des listes imbriquées :

```
?- [a,[2,2],c] = [a,Y,c]
Y = [2,2]
yes
```

```
?- [X,b,c] = [a,Y,c]
X = a
Y = b
yes
```

```
?- [a,b,[c,d]] = [a,b,[c,X]]
X = d
yes
```

Les listes : notations

- **Notation énumérée**
  - $[x_1, x_2, \ldots, x_n]$
- **Notation structurée**
  - Une liste non vide = une tête + une queue
  - Notation : `[Tete|Queue]`

```
?- [a,b,c,d] = [a|[b,c,d]]
yes
```

```
?- [a|[b,c,d]] = [a|[b|[c,d]]]
yes
```

```
?- [a|[b|[]]] = [a,b]
yes
```

Les listes : notations

- **Notation structurée**
  - Toute liste non vide peut être représentée
  - La liste vide :
    - terme spécifique : à traiter comme un atome
    - ne peut être représentée pas la notation structurée

```
?- [] = [[]|[]]
no
```

Prédicat append

- `append` est le prédicat prédéfini pour la concaténation de listes

```
?- append([a,b,c],[d,e],L).
L = [a, b, c, d, e]
```

- Il est complètement symétrique et peut donc être utilisé pour
  - Trouver le dernier élément d'une liste :

```
?- append(_,[X],[a,b,c,d]).
X = d
```

  - Couper une liste en sous-listes :

```
?- append(L2,L3,[b,c,a,d,e]),append(L1,[a],L2).
L2 = [b, c, a]
L3 = [d, e]
L1 = [b, c]
```

Prédicats +, -, *, /

- Prolog intègre les prédicats +, -, *, /

```
?- X = 4+5.
X = 4+5
yes

?- X = 4+5, Y = 5+4, X=Y.
no
```

- Les résultats sont différents
- En fait Prolog n'a pas calculé le résultat mais a juste unifié X avec le terme '5+4' et Y avec '4+5'

Prédicats `is`

- Le prédicat `is` permet de forcer Prolog à évaluer les opérations arithmétiques

```
?- X is 4+5.
X = 9
yes

?- X is 4+5, Y is 5+4, X=Y.
yes
```

- Les résultats sont identiques
- Prolog a bien évalué l'opération arithmétique

Opérations

Arithmétique	Prolog
$4+1=5$	`5 is 4 + 1.`
$5-2=3$	`3 is 5 - 2.`
$2-5=-3$	`-3 is 2 - 5.`
$4/2=2$	`2 is 4 / 2.`
$2*6=12$	`12 is 2 * 6.`
5 modulo $2=1$	`1 is mod(5,2).`

Comparaisons

Arithmétique	Prolog
X $<$ Y	`X < Y.`
X $\leq$ Y	`X =< Y.`
X $=$ Y	`X =:= Y.`
X $\neq$ Y	`X =\= Y.`
X $\geq$ Y	`X >= Y.`
X $>$ Y	`X > Y.`

Influence de l'ordre des clauses : exemple

- 4 versions du programme *ancetre*

```
parent(michelle, bernard).
parent(thomas, bernard).
parent(thomas, lise).
parent(bernard, anne).
parent(bernard, pierre).
parent(pierre, jean).
ancetre(X, Z) :- parent(X,Z).
ancetre(X, Z) :- parent(X,Y), ancetre(Y,Z).
ancetre2(X, Z) :- parent(X,Y), ancetre2(Y,Z).
ancetre2(X, Z) :- parent(X,Z).
ancetre3(X, Z) :- parent(X,Z).
ancetre3(X, Z) :- ancetre3(X,Y), parent(Y,Z).
ancetre4(X, Z) :- ancetre4(X,Y), parent(Y,Z).
ancetre4(X, Z) :- parent(X,Z).
```

Influence de l'ordre des clauses : *ancetre*

`:- ancetre(thomas, pierre).`

1. `parent(thomas, pierre)` $\rightarrow$ échec
2. `parent(thomas, Y), ancetre(Y,pierre)`
   - 2.1. $Y = bernard$; but = `ancetre(bernard, pierre)`
     - 2.1.1. `parent(bernard, pierre)` $\rightarrow$ **succès** - Stop ou Continue?
     - 2.1.2. `parent(bernard, Y′), ancetre(Y′,pierre)`
       - 2.1.2.1. $Y' = anne$; but = `ancetre(anne, pierre)`
         - 2.1.2.1.1. `parent(anne, pierre)` $\rightarrow$ échec
         - 2.1.2.1.2. `parent(anne, Y″), ancetre(Y″, pierre)` $\rightarrow$ échec
       - 2.1.2.2. $Y' = pierre$; but = `ancetre(pierre, pierre)`
         - 2.1.2.2.1. `parent(pierre, pierre)` $\rightarrow$ échec
         - 2.1.2.2.2. `parent(pierre, Y″), ancetre(Y″, pierre)`
       - 2.1.2.2.2.1. $Y'' = jean$; but = `ancetre(jean, pierre)`
         - 2.1.2.2.2.1.1. `parent(jean, pierre)` $\rightarrow$ échec
         - 2.1.2.2.2.1.2. `parent(jean, Y″′), ancetre(Y″′, pierre)` $\rightarrow$ échec
   - 2.2 $Y = lise$; but = `ancetre(lise, pierre)`
     - 2.2.1. `parent(lise, pierre)` $\rightarrow$ échec
     - 2.2.2. `parent(lise, Y′), ancetre(Y′,pierre)` $\rightarrow$ échec

Influence de l'ordre des clauses : *ancetre2*

`:- ancetre2(thomas, pierre).`

1. `parent(thomas, Y), ancetre2(Y,pierre)`
   - 1.1. $Y = bernard$; but = `ancetre2(bernard, pierre)`
     - 1.1.1. `parent(bernard, Y'), ancetre2(Y',pierre)`
       - 1.1.1.1. $Y' = anne$; but = `ancetre2(anne, pierre)`
         - 1.1.1.1.1. `parent(anne, Y"), ancetre2(Y", pierre)` → échec
         - 1.1.1.1.2. `parent(anne, pierre)` → échec
       - 1.1.1.2. $Y' = pierre$; but = `ancetre2(pierre, pierre)`
         - 1.1.1.2.1. `parent(pierre, Y"), ancetre2(Y", pierre)`
       - 1.1.1.2.1.1. $Y" = jean$; but = `ancetre2(jean, pierre)`
           - 1.1.1.2.1.1.1. `parent(jean, Y"'), ancetre2(Y"', pierre)` → échec
           - 1.1.1.2.1.1.2. `parent(jean, pierre)` → échec
         - 1.1.1.2.2. `parent(pierre, pierre)` → échec
     - 1.1.2. `parent(bernard, pierre)` → **succès** - Stop ou Continue?
   - 1.2 $Y = lise$; but = `ancetre2(lise, pierre)`
     - 1.2.1. `parent(lise, Y'), ancetre(Y',pierre)` → échec
     - 1.2.2. `parent(lise, pierre)` → échec
2. `parent(thomas, pierre)` → échec

Influence de l'ordre des clauses : *ancetre3*

`:- ancetre3(thomas, pierre).`

1. `parent(thomas, pierre)` $\rightarrow$ échec
2. `ancetre3(thomas, Y), parent(Y, pierre)`
   - 2.1. `parent(thomas Y′), parent(Y′, pierre)`
     - 2.1.1. $Y' = bernard$; but = `parent(bernard, pierre)` $\rightarrow$ **succès**
     - 2.1.2. $Y' = lise$; but = `parent(lise, pierre)` $\rightarrow$ échec
   - 2.2 `ancetre3(thomas, Y″), parent(Y″, Y′), parent(Y′, pierre)`
     - 2.2.1. `parent(thomas Y″), parent(Y″, Y′), parent(Y′, pierre)`
       - 2.2.1.1. $Y" = bernard$; but = `parent(bernard, Y′), parent(Y′, pierre)`
         - 2.2.1.1.1. $Y' = anne$; but = `parent(anne, pierre)` $\rightarrow$ échec
         - 2.2.1.1.2. $Y' = pierre$; but = `parent(pierre, pierre)` $\rightarrow$ échec
       - 2.2.1.2. $Y" = lise$; but = `parent(lise, Y′), parent(Y′, pierre)` $\rightarrow$ échec
     - 2.2.2. `ancetre3(thomas, Y″′), parent(Y″′, Y″), parent(Y″, Y′), parent(Y′, pierre)`
       - 2.2.2.1. `parent(thomas, Y″′), parent(Y″′, Y″), parent(Y″, Y′), parent(Y′, pierre)` ... **branche infinie**
       - 2.2.2.2. `ancetre3(thomas, Y″″), parent(Y″″, Y″′), parent(Y″′, Y″), parent(Y″, Y′), parent(Y′, pierre)` ... **branche infinie**

Influence de l'ordre des clauses : *ancetre4*

```
:- ancetre4(thomas, pierre).
```

1. `ancetre4(thomas, Y), parent(Y, pierre)`
   1.1. `ancetre4(thomas, Y′), parent(Y′, Y), parent(Y, pierre)`
      1.1.1. `ancetre4(thomas, Y″), parent(Y″, Y′), parent(Y′, Y), parent(Y, pierre)`
         1.1.1.1. `ancetre4(thomas, Y″′), parent(Y″′, Y″), parent(Y″, Y′), parent(Y′, Y), parent(Y, pierre)`

         . . . **branche infinie**

Influence de l'ordre des clauses : conclusion

- Plan *déclaratif* : les quatre versions sont équivalentes
- Plan *procédural* : seules `ancetre` et `ancetre2` sont correctes
- Pourquoi?
  - → Il vaut mieux essayer d'abord ce qui est simple : `parent` plutôt que `ancetre`
  - → Donner la priorité à `parent` pour ordonner les clauses et les littéraux

Pourquoi un prédicat de coupe?

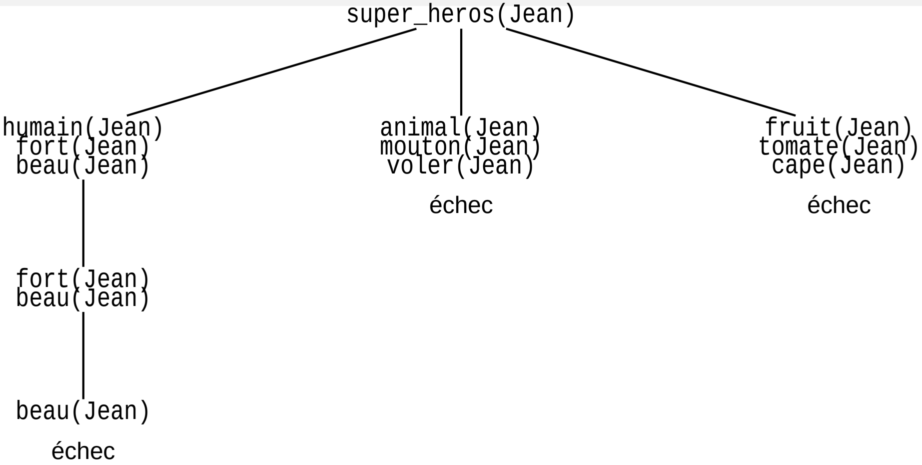
- Différents types de super héros
  - humain $\rightarrow$ un humain fort et beau
  - animal $\rightarrow$ un mouton volant
  - fruit $\rightarrow$ une tomate avec une cape

```
(1) super_heros(X) :- humain(X), fort(X), beau(X).
(2) super_heros(X) :- animal(X), mouton(X), voler(X).
(3) super_heros(X) :- fruit(X), tomate(X), cape(X).
```

- Une base de faits

```
humain(jean).
fort(jean).
animal(bobby).
animal(peggy).
cochon(peggy).
marche_au_plafond(peggy).
```

Pourquoi un prédicat de coupe?



Pourquoi un prédicat de coupe?

```
super_heros(Jean)
```

```
humain(Jean)
fort(Jean)
beau(Jean)
```

```
fort(Jean)
beau(Jean)
```

```
beau(Jean)
```

échec

```
animal(Jean)
mouton(Jean)
voler(Jean)
```

échec

```
fruit(Jean)
tomate(Jean)
cape(Jean)
```

échec

Comment éviter ces appels inutiles?

Pourquoi un prédicat de coupe?

super_heros(Jean)

humain(Jean)
fort(Jean)
beau(Jean)

animal(Jean)
mouton(Jean)
voler(Jean)

échec

fruit(Jean)
tomate(Jean)
cape(Jean)

échec

fort(Jean)
beau(Jean)

beau(Jean)

échec

Comment éviter ces appels inutiles?

- Attention: celà suppose que si Jean est un humain, il n'y a pas d'animal ou de fruit nommés Jean
- Sémantique du *Si, Alors, Sinon*

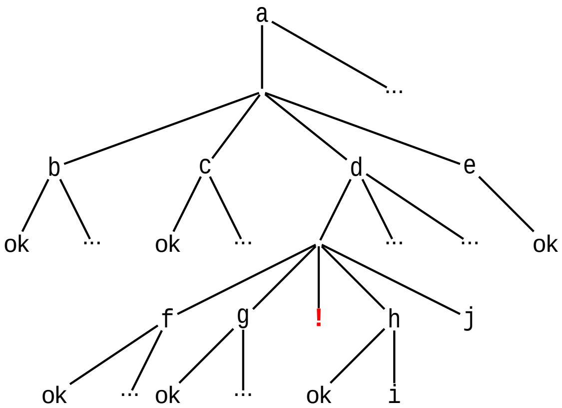
Prédicat CUT

- Prédicat prédéfini : !
- Sert à
  - interdire l'exploration de certaines branches
  - améliorer l'efficacité d'un programme
  - confirmer un choix que l'on sait être le seul ou plus pertinent
  - écrire un "si alors sinon"
- `p :- a, b, ..., !, ..., s`

  → Tous les choix mémorisés depuis l'appel de la tête de clause jusqu'à l'exécution du ! sont supprimés

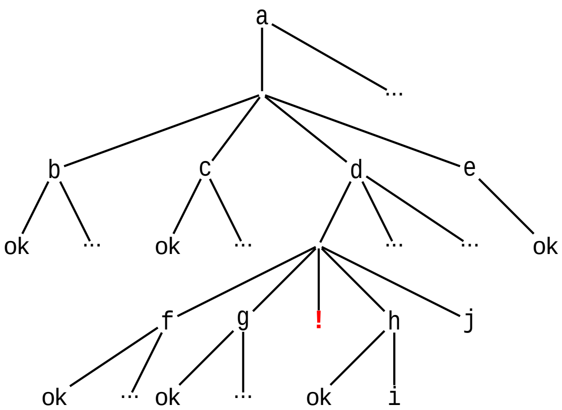
Coupure : quels points de choix supprimés?

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
d :- ...
d :- ...
f.
f :- ...
g.
g :- ...
h.
h :- i.
e.
```



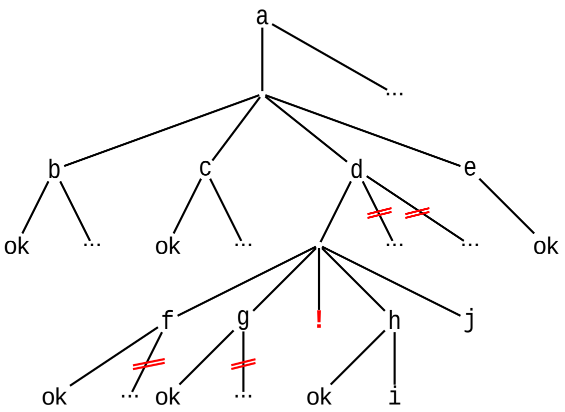
Coupure : quels points de choix supprimés?

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
d :- ...
d :- ...
f.
f :- ...
g.
g :- ...
h.
h :- i.
e.
```



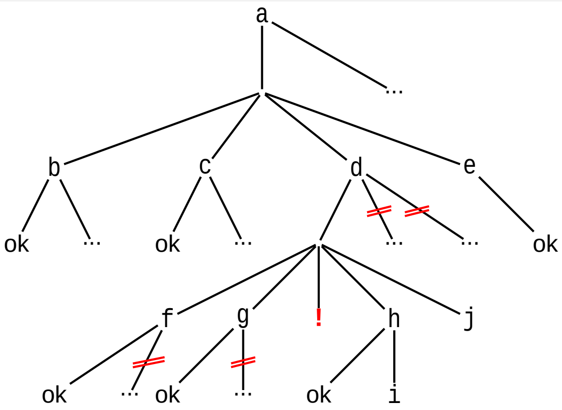
Coupure : quels points de choix supprimés?

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
(d :- ...)
(d :- ...)
f.
(f :- ...)
g.
(g :- ...)
h.
h :- i.
e.
```



Coupure : quels points de choix supprimés?

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
(d :- ...)
(d :- ...)
f.
(f :- ...)
g.
(g :- ...)
h.
h :- i.
e.
```



La coupure supprime les points de choix sur les sommets aînés et sur le sommet père

Exemple : super héros

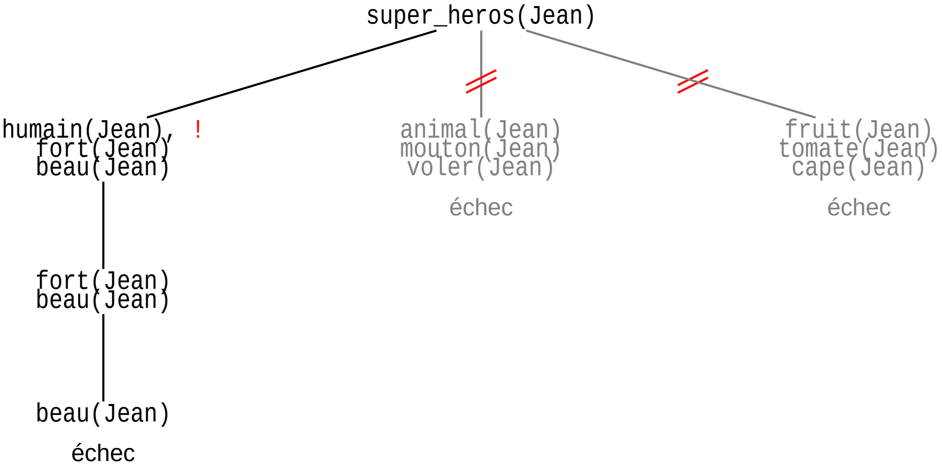
- On modifie les règles

```
(1) super_heros(X) :- humain(X), !, fort(X), beau(X).
(2) super_heros(X) :- animal(X), !, mouton(X), voler(X).
(3) super_heros(X) :- fruit(X), !, tomate(X), cape(X).
```

- La même base de faits

```
humain(jean).
fort(jean).
animal(bobby).
animal(peggy).
cochon(peggy).
marche_au_plafond(peggy).
```

Exemple : super héros



Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

p vérifie : $p \Leftrightarrow (a \wedge b) \vee c$

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

  p vérifie : $p \Leftrightarrow (a \wedge b) \vee c$

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

p vérifie : $p \Leftrightarrow (a \wedge b) \vee c$

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

p vérifie : $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

  p vérifie : $p \Leftrightarrow (a \wedge b) \vee c$

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

  p vérifie : $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

Prédicat CUT : exemple

- Soient les clauses suivantes :

```
p :- a, b.
p :- c.
```

p vérifie : $p \Leftrightarrow (a \wedge b) \vee c$

- Soient les clauses suivantes :

```
p :- a, !, b.
p :- c.
```

p vérifie : $p \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c)$

- Soient les clauses suivantes :

```
p :- c.
p :- a, !, b.
```

p vérifie : $p \Leftrightarrow c \vee (a \wedge b)$

La négation : prédicat `fail`

- `p(X)` est vrai si `X` vérifie la propriété `p`
- Comment écrire `non p(X)`, qui est vrai si `p(X)` est faux?

- Si `p(X)` est vrai, retourne Faux et ne pas déclencher la règle 2.; sinon retourner Vrai.

La négation : prédicat `fail`

- `p(X)` est vrai si `X` vérifie la propriété `p`
- Comment écrire `non p(X)`, qui est vrai si `p(X)` est faux?

```
1.  non p(X) :- p(X), !, fail.
2.  non p(X).
```

- Si `p(X)` est vrai, retourne Faux et ne pas déclencher la règle 2.; sinon retourner Vrai.

La négation : prédicat `fail`

- `p(X)` est vrai si `X` vérifie la propriété `p`
- Comment écrire `non p(X)`, qui est vrai si `p(X)` est faux?

```
1.  non p(X) :- p(X), !, fail.
2.  non p(X).
```

- Si `p(X)` est vrai, retourne Faux et ne pas déclencher la règle 2.; sinon retourner Vrai.