Programmation Logique

L3 Info

Céline Rouveirol

2007-2008

Plan

Objectifs et plan du cours

- Maîtrise des concepts de :
  - la Programmation Logique
  - PROLOG
- Introduction à la Programmation Logique Contrainte et aux algorithmes de résolution de problèmes de satisfaction de contraintes

Sources du cours - Bibliographie

- J.-M. Alliot, T. Schiex, P. Brisset, F. Garcia, Intelligence Artificielle et Informatique Théorique, 2ème ed., 2002
- Poly de cours de logique, S. Cerrito
  `http://www.lami.univ-evry.fr/~serena/`
- P. Flach, Simply Logical : Intelligent Reasoning by Example John Wiley 1994
- M. R. Genesereth et N. J. Nilsson, Logical Foundations of Artificial Intelligence, Morgan Kaufmann, 1987
- J. Lloyd, Foundations of Logic Programming, Springer, 1987
- L. Sterling et E. Shapiro, The Art of Prolog : Advanced Programming Techniques, 2nd ed, 1994

Syntaxe logique propositionnelle

atomes ou variable propositionnelle : des énoncés dont nous ne cherchons pas pas connaître la structure interne. Notés p, q, r, s, ...

connecteurs : les connecteurs $\rightarrow$, $\leftrightarrow$, $\neg$, $\vee$, $\wedge$, ...

formule

- un atome
- si A est une formule, $\neg A$ est une formule
- si A et B sont des formules $A \leftrightarrow B$, $A \rightarrow B$, $A \vee B$ et $A \wedge B$ sont des formules
- formule vide notée $\square$, et définie par $(a \wedge \neg a)$

littéral : positif = atome et littéral négatif = négation d'un atome

Sémantique

- Valuation : à chaque variable propositionnelle de la formule, on associe une valeur de vérité $\{t, f\}$.
- Interprétation d'une formule : une valuation pour les variables prop. de la formule. Par abus de language, on liste parfois pour décrire une interprétation I uniquement les variables prop. qui ont pour valeur t dans I. Notation $I(p)$ (resp. $I(F)$) : la valeur de vérité de la variable p (resp. de la formule F) dans l'interprétation I

Tables de vérité

On associe à chaque connecteur (unaire, binaire) une fonction booléenne qui permet de calculer la v. v. d'une formule dont ce connecteur est le connecteur principal, à partir de la v.v. de ses sous-formules A et B.

A	B	$\neg A$	$A \vee B$	$A \wedge B$	$A \rightarrow B$	$A \leftrightarrow B$
t	t	f	t	t	t	t
t	f	f	t	f	f	f
f	t	t	t	f	t	f
f	f	t	f	f	t	t

Sémantique

- Satisfiabilité ($\models$). Soit F une formule et I une interprétation de F. On définit la relation de satisfiabilité ($I \models F$) par :
  - F est une variable V : $I \models F$ ssi $i(V) = t$
  - F est une formule de la forme $\neg G$: $I \models F$ ssi $i(G) = f$
  - F est une formule de la forme $G \vee H$: $I \models F$ ssi $i(G) = t$ ou $i(H) = t$
  - F est une formule de la forme $G \wedge H$, $I \models F$ ssi $I(G) = t$ et $I(H) = t$

Sémantique

modèle I est un modèle pour une formule si $I \models F$. I est un modèle pour un ensemble de formules $F_i, 1 \leq i \leq n$ si pour tout i $I \models F_i$

conséquence logique La formule F est conséquence logique de la formule G (noté $G \models F$) si pour tout I tel que $I \models G$ alors $I \models F$

satisfiabilité

- Une formule est *satisfiable* si il existe au moins une interprétation I telle $I \models F$. Une formule est dite *insatisfiable* s'il n'existe aucun I telle que $I \models F$.
- Une formule qui est vraie pour toute interprétation I est une *tautologie*. On dit aussi qu'elle est *valide*. Notation : $\models F$.

Sémantique (suite)

- Théorème de déduction : $F_1, \ldots, F_n \models G$ ssi $(F_1 \wedge \ldots F_n \rightarrow G)$ est valide
- Soit G une formule quelconque. Si E est un ensemble insatisfiable de formules, alors G est conséquence logique de E
- Si G est valide, G est conséquence logique d'un ensemble quelconque de formules
- Soit E un ensemble de formules et G une formule, $E \models G$ ssi $E, \neg G$ est insatisfiable.

Equivalence logique

- Soit F et G deux formules quelconques. F est logiquement équivalente à G, noté $F \equiv G$ ssi $F \models G$ et $G \models F$.
- $F \equiv G$ ssi $\models F \leftrightarrow G$
- Quelques équivalences utiles
  1. $F \leftrightarrow G \equiv F \rightarrow G \wedge G \rightarrow F$
  2. $F \rightarrow G \equiv \neg F \vee G$
  3. $\neg(F \wedge G) \equiv \neg F \vee \neg G$ et $\neg(F \vee G) \equiv \neg F \wedge \neg G$ (loi de Morgan)
  4. $\neg\neg F \equiv F$ (loi de double négation)
  5. $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$ et $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$ (distributivité de $\vee$ sur $\wedge$ et vice versa)

Axiomatique du calcul propositionnel

- Méthodes algorithmiques pour décider automatiquement et aussi efficacement que possible de la satisfiabilité d'un ensemble de formules.
- Un système de preuve est défini par un ensemble de règles (dites *règles d'inférence*) à appliquer "mécaniquement", en ne tenant compte que de la syntaxe des formules traitées.
- Notation :

$$\frac{P_1 \qquad P_2}{C}$$

où P_1 et P_2 sont les prémisses du séquent et C sa conclusion.

- Règle du modus ponens :

$$\frac{A \rightarrow B \qquad A}{B}$$

Propriétés d'un système d'inférence

On note classiquement $\vdash$ la notion de dérivation par un système d'inférence.

Correction Un système d'inférence est correct si, pour toute règle du système, la formule conclusion est conséquence logique de sa formule prémisse (de la conjonction de ses formules prémisses) : $\vdash$ est correcte si .

Complétude Un système d'inférence est complet s'il suffit à prouver toute formule valide : pour toute formule F telle que $\models F$, alors $\vdash F$.

Formes normales

Littéral : formule atomique ou négation d'une formule atomique

Clause : formule de la forme $l_1 \vee \ldots \vee l_n$ où chaque l_i est un littéral

FNC : formule de la forme $D_1 \wedge \ldots \wedge D_k$ où $k > 0$ et chaque D_i est une clause

Conjonction élémentaire : formule de la forme $l_1 \wedge \ldots \wedge l_n$ où chaque l_i est un littéral

FND : formule de la forme $C_1 \vee \ldots \vee C_k$ où $k > 0$ et chaque C_i est une conjonction élémentaires

Toute formule admet une forme normale conjonctive et disjonctive qui lui est logiquement équivalente (voir TD)

Algorithme de mise sous FNC

Utilisation des équivalences vues au transparent 11.

1. Eliminer les $\rightarrow$ et les $\leftrightarrow$: application des équivalences 1 et 2
2. Limiter la portée des négations (faire "descendre" les négations) : application des équivalences 3 et 4
3. Mettre sous formpe clausale : application des équivalences 5

Notes :

- Pour une formule f donnée, il existe plusieurs f' telle que $f \equiv f'$ et f' en FNC.
- La taille de f' peut croître (exponentiellement)

Exemple de mise sous FNC

$$(p \rightarrow (q \leftrightarrow \neg r)) \wedge (r \leftrightarrow \neg s) \equiv$$

$$\neg p \vee ((q \rightarrow \neg r) \wedge (\neg r \rightarrow q)) \wedge (r \leftrightarrow \neg s) \equiv$$

$$\neg p \vee ((q \rightarrow \neg r) \wedge (\neg r \rightarrow q)) \wedge (r \rightarrow \neg s) \wedge (\neg s \rightarrow r) \equiv$$

$$\neg p \vee ((\neg q \vee \neg r) \wedge (r \vee q)) \wedge (r \rightarrow \neg s) \wedge (\neg s \rightarrow r) \equiv$$

$$\neg p \vee ((\neg q \wedge r) \vee (\neg r \wedge q)) \wedge (r \rightarrow \neg s) \wedge (\neg s \rightarrow r) \equiv$$

$$(\neg p \vee \neg q \vee \neg r) \wedge (\neg p \vee r \vee q) \wedge (\neg r \vee \neg s) \wedge (s \vee r)$$

Résolution en calcul propositionnel

Méthode de démonstration par réfutation.

- Ensemble de formules insatisfiable : un ensemble de formules $\{F_i\}, 1 \leq i \leq n$ est insatifiable ssi il n'existe aucun modèle M tel que pour tout i, $M \models F_i$.
- Principe de déduction : une formule G est conséquence logique d'un ensemble de formules $F = \{F_i, 1 \leq i \leq n\}$ ssi $F \cup \neg G$ est insatisfiable.

Résolution en calcul propositionnel

Deux règles d'inférence :

- Résolution. Soient C_1 et C_2 deux clauses, soit p un littéral.

$$\frac{C_1 \vee p \qquad C_2 \vee \neg p}{R = C_1 \vee C_2}$$

  On appelle R la résolvante de $C_1 \vee p$ et $C_2 \vee \neg p$

- Factorisation. Soient C une clause et p une littéral

$$\frac{C \vee p \vee p}{C \vee p}$$

Dérivation par résolution propositionnelle

Soient E un ensemble de clauses et c une clause. Une *dérivation* par résolution de c à partir des *hypothèses* E est une suite de clause $r_1, \ldots, r_n$ telles que pour tout i, $1 \leq i \leq n$:

- $r_i \in E$
- il existe un $j \leq i$ tel que $\dfrac{r_j}{r_i}$ par factorisation
- il existe $j, k \leq i$ tels que $\dfrac{r_j \qquad r_k}{r_i}$ par résolution (r_i est appelé résolvante de r_j et r_k
- Une *réfutation* de E est une dérivation par résolution de $\square$ à partir des hypothèses de E

Exemple de réfutation

$$S = \{p \vee q \vee r; \neg p \vee q \vee r; \neg q \vee r; \neg r\}$$

$$\dfrac{\dfrac{\dfrac{\neg q \vee r \qquad \neg r}{\neg q} \qquad \dfrac{\dfrac{p \vee q \vee r \qquad \neg p \vee q \vee r}{q \vee q \vee r}}{q \vee r}}{r} \qquad \neg r}{\square}$$

Propriétés de la résolution propositionnelle

Correction Soient A, B et C des clauses.
$\{A \vee C, B \vee \neg C\} \models A \vee B$

Non complétude Il existe une théorie clausale F et une clause C telle que $F \models C$, mais il n'existe pas de dérivation par résolution de C à partir de F.

Complétude pour la réfutation Un ensemble de formules F est insatisfiable ssi on peut dériver la clause vide $\square$ de F par résolution. Si $F \models C$, il existe une réfutation à partir de $F \cup \neg C$.

Résolution et clause de Horn

Clauses de Horn une clause comprenant au plus un littéral positif.

Résolution unitaire au moins un des deux parents est unitaire

$$(\ell) : \frac{\ell \qquad L \vee \neg\ell}{L}$$

Stratégie de résolution

1. Si $\square$ est dans F alors F est insatisfiable
2. Sinon, choisir une clause C et p une clause unitaire positive et C contient $\neg p$
3. Calculer la résolvante R de p et C
4. Remplacer dans F C par R

- La résolution unitaire n'est pas complète par réfutation pour les théories clausales quelconques. Exemple : $\{a \vee b; \neg a \vee b; a \vee \neg b; \neg a \vee \neg b\}$
- La résolution unitaire est complète par réfutation pour les ensembles de clauses de Horn. Exemple : $\{a; c; \neg a \vee b; \neg b \vee c\}$

PROLOG : Programmation Déclarative

- Un programme logique est un ensemble d'axiomes définissant les relations entre des objets.
- La signification d'un programme, c'est l'ensemble des conséquences logiques de ce programme.
- Un programme PROLOG est un ensemble de clauses définies, un requête PROLOG est un but défini.
- Mécanisme de base en PROLOG : étant donné un programme logique P et un but B, PROLOG construit une preuve (par résolution) de B étant donné P.
- Pour chercher à prouver que Q est conséquence logique d'un programme logique P, on cherche à prouver la contradiction de P et Q (déduction de la clause vide).

Syntaxe PROLOG

Alphabet

- constantes. Ex : `toto, 1, "bonjour", a,b,c,....`
- `variables. Ex : X, Y, Z, ...`
- `symboles de fonction.`
  `Ex : f,g,h, succ, sag, sag, ...`
- `symboles de prédicat. Ex : p, q, r, plus, entier, parcours_arbre, ...`
- `connecteurs : ,`

PROLOG : premier contact

Soit le programme logique (ensemble de faits) PROLOG suivant :

```
père(jean,jules).
plus(un,zéro,un)
```

- Requête unitaire close

```
| ?- pere(jean,jules).
yes
| ?- pere(jean,julie).
no
| ?-
```

PROLOG : premier contact (2)

- Requête existentielle ?- q(X)
- Interprétation intuitive de la requête `?- q(X)`, étant donné le programme P : étant donné P, existe-t-il une substitution de X telle que $q(X)$ soit vrai ?

```
 | ?- pere(jules,X).
no
| ?- pere(jean,X).
X = jules
yes
| ?- plus(un,X,un).
X = zero.
```

PROLOG : premier contact (3)

- Il peut y avoir plusieurs réponses à une requête existentielle (retour arrière ou backtrack).

Soit le programme logique (ensemble de faits clos) suivant :

```
pere(jean,jules).
pere(jean,julie).
plus(un,zero,un).
plus(un,un,deux).
 | ?- père(jean,X).
X = jules ;
X = julie ;
no
| ?-plus(un,X,Y).
X = zero, Y=un ;
X = un, Y=deux ;
```

PROLOG : premier contact (5)

- Requête conjonctive (composée)
  - Notation PROLOG `| ? q(X), r(X).`
  - Interprétation informelle de Q : existe-t-il un X tel que $q(X)$ et $r(X)$ soient vrais (étant donné un programme logique P) ?
  - Une requête conjonctive réussit s'il existe au moins une instance de la requête qui soit conséquence logique du programme.

PROLOG : premier contact (6)

- Soit le programme défini suivant :

```
 père(toto,titi).
sexe(titi, masculin).
père(titi,lulu).
 |? père(toto,X), sexe(X,masculin).
X = titi
|? père(toto,X), père(X,Y).
X = titi, Y = lulu
```

- X est une variable partagée. La portée d'une variable logique est l'expression logique dans laquelle elle apparaît et celle-ci seulement.

PROLOG : premier contact (7)

- Programme = { Clauses unitaires avec variables }
- Exemple `plus(X,0,X).`
- Interprétation informelle : les variables sont dans ce cas quantifiées universellement.
- Soit P un P.L. sous forme de clauses unitaires non closes et Q une requête close. La requête Q réussit s'il existe au moins un fait F de P et une substitution θ telle que $F\theta = Q$.
- Requête non close : La requête Q réussit si Q et au moins un fait non clos F de P ont une instance commune.

PROLOG : premier contact (8)

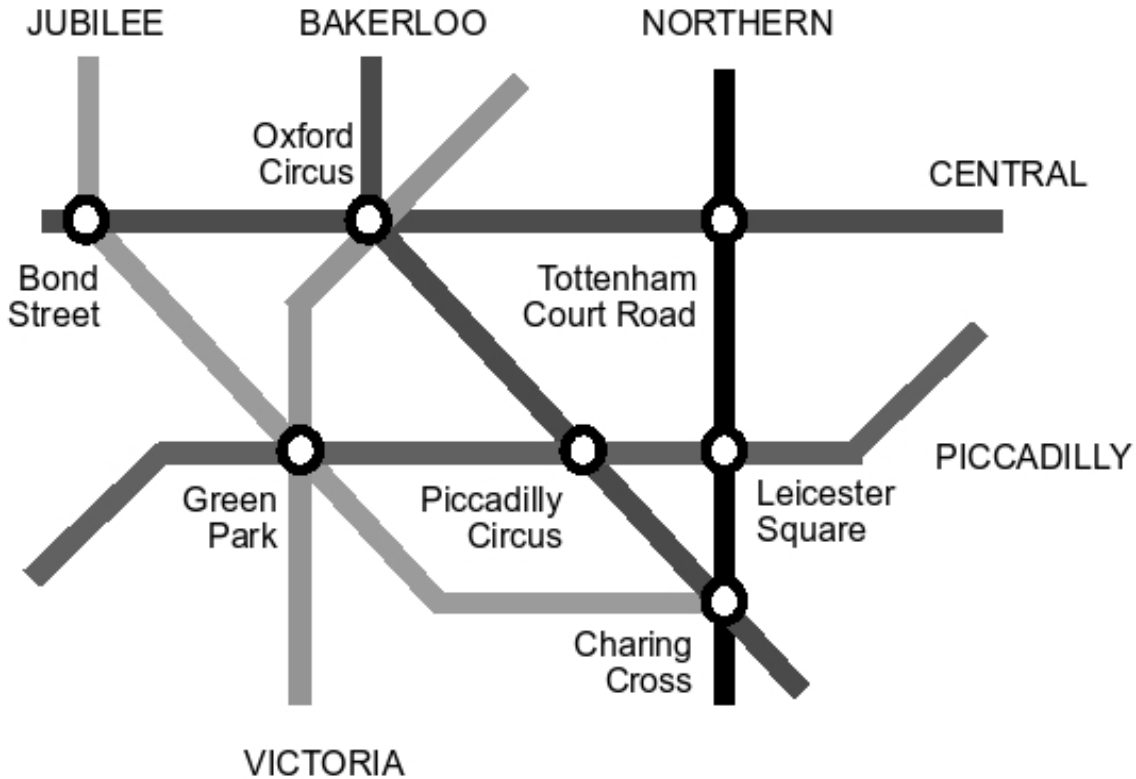
- Soit le P.L. `plus(X,0,X).` et la requête `plus(1,0,Z).`
- La réponse est `X = 1, Z = 1`
- Programme = { Clauses définies }
- Ex : `fils(X,Y) :- pere(Y,X),sexe_masculin(X).`
- Interprétation procédurale : Pour répondre à la requête : existe-t-il un X qui soit le fils de Y, il faut répondre à la requête conjonctive 'existe-t-il un Y qui soit le père de X et tel que X soit de sexe masculin' ?
- Interprétation déclarative : X est le fils de Y si Y est le père de X et X est de sexe masculin.
- Les variables qui apparaissent dans la tête d'une règle sont quantifiées universellement et celles qui apparaissent dans le corps sont quantifiées existentiellement

PROLOG : premier contact (9)

- Soit le programme défini suivant :

```
 grand-père(X,Y) :- père(X,Z), père(Z,Y).
père(toto,titi).
sexe(titi, masculin).
père(titi,lulu).
 | ? grand-pere(X,Y).
X = titi, Y = lulu
```

PROLOG : un exemple



LRT Registered User No. 94/1954

PROLOG : un exemple

```
connectée(bondstreet,oxfordcircus,central).
connectée(oxfordcircus,tottenhamcourtoad,central).
connectée(bondstreet,greenpark,jubilee).
connectée(greenpark,charingcross,jubilee).
connectée(greenpark,piccadillycircus,piccadilly).
connectée(piccadillycircus,leicestersquare,piccadilly).
connectée(greenpark,oxfordcircus,victoria).
connectée(oxfordcircus,piccadillycircus,bakerloo).
connectée(piccadillycircus,charingcross,bakerloo).
connectée(tottenhamcourtroad,leicestersquare,northern).
connectée(leicestersquare,charingcross,northern).
```

PROLOG : un exemple

Deux stations sont voisines sur la même ligne, avec au plus une station entre les deux :

```
station-voisine(bondstreet,oxfordcircus).
station-voisine(oxfordcircus,tottenhamcourtroad).
station-voisine(bondstreet,tottenhamcourtroad).
station-voisine(bondstreet,greenpark).
station-voisine(greenpark,charingcross).
station-voisine(bondstreet,charingcross).
station-voisine(greenpark,piccadillycircus).
.....
```

ou encore mieux `station-voisine(X,Y) :-connectée(X,Y,L).`

```
station-voisine(X,Y) :-connectée(X,Z,L), connectée(Z,Y,L).
```

PROLOG : un exemple

Comparez

```
station-voisine(X,Y) :-connectée(X,Y,L).
station-voisine(X,Y) :-connectée(X,Z,L),connectée(Z,Y,L).
```

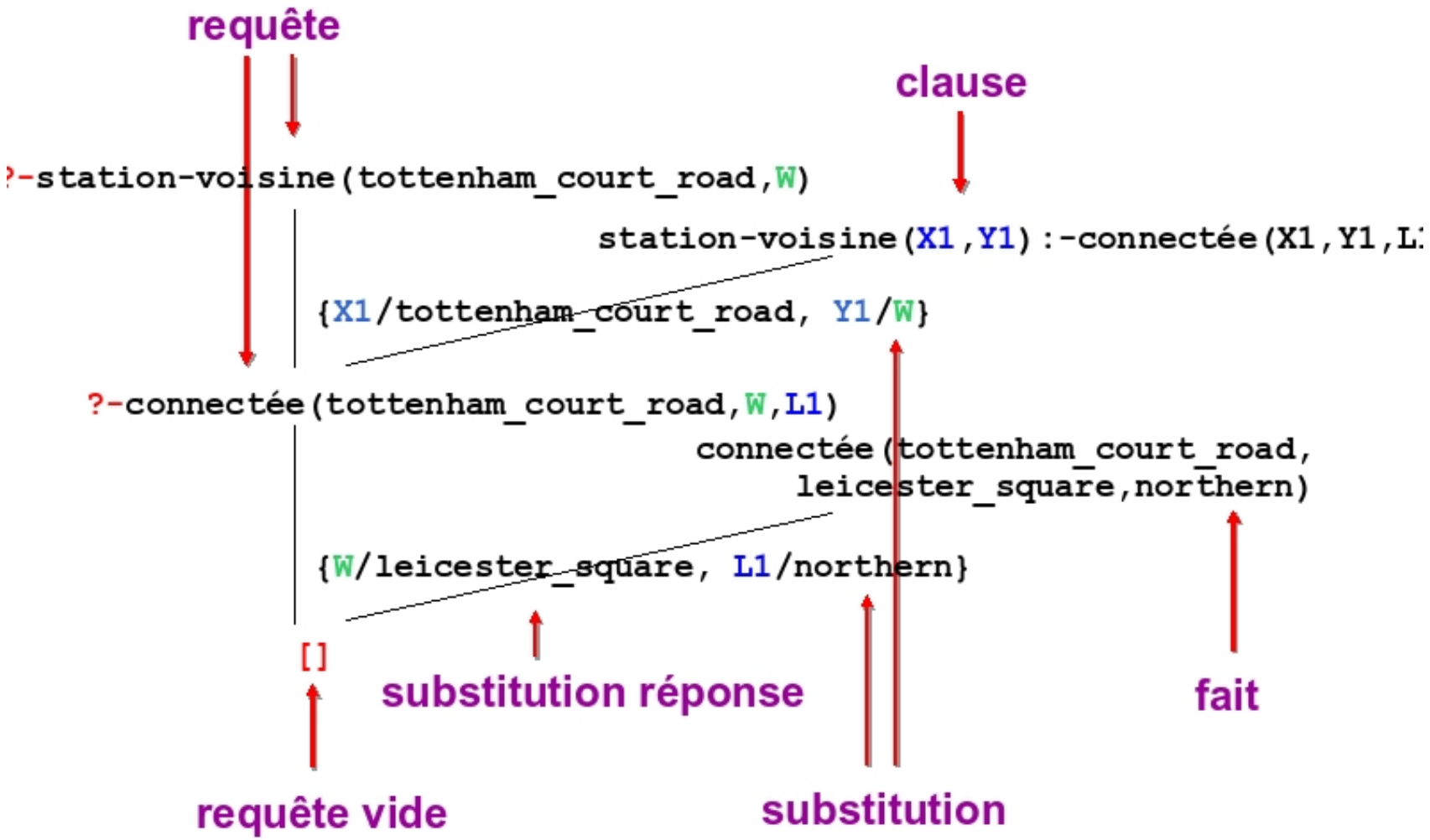
avec

```
pas-trop-loin(X,Y) :-connectée(X,Y,L).
pas trop-loin(X,Y) :-connectée(X,Z,L1),connectée(Z,Y,L2).
```

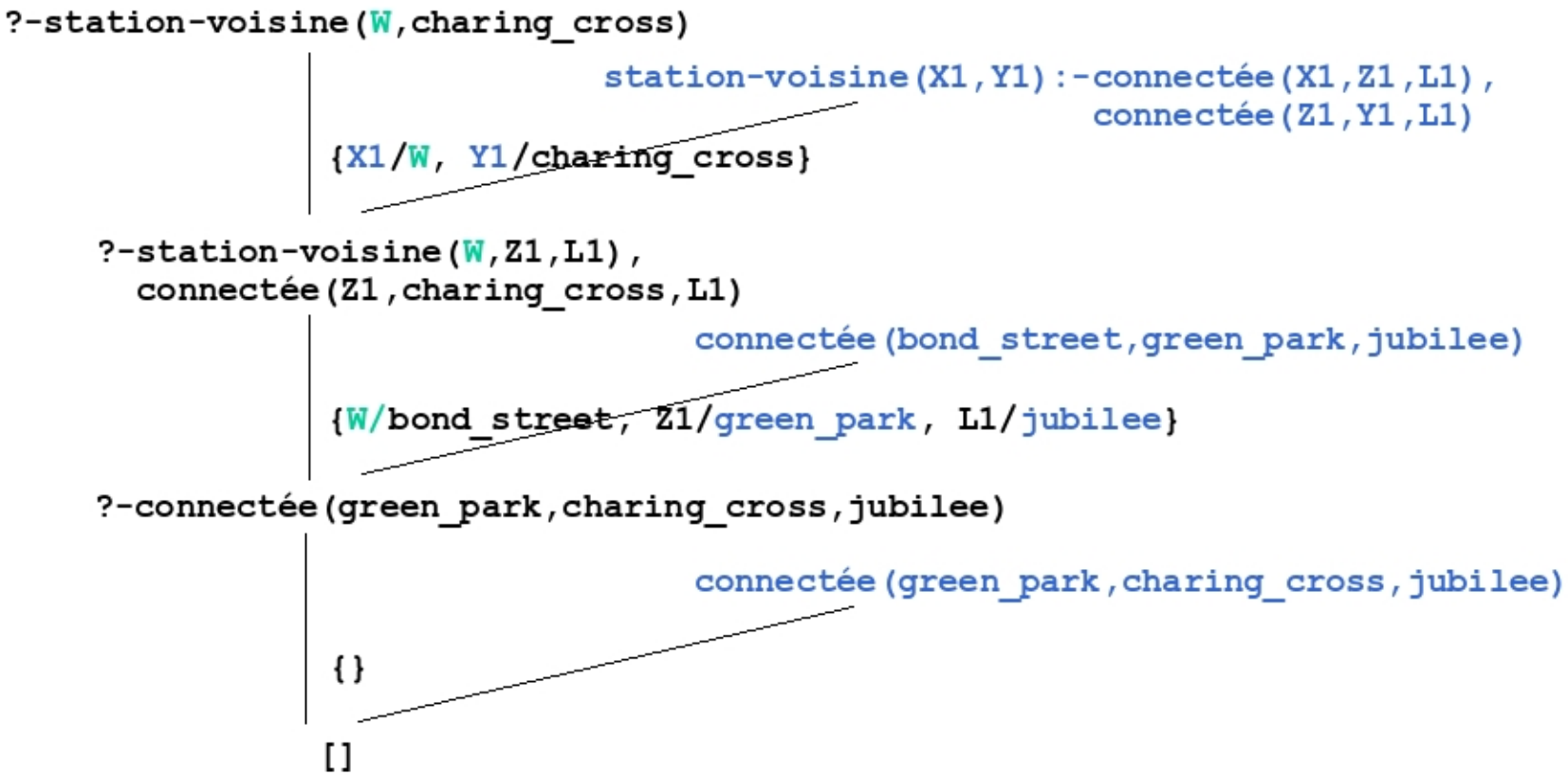
On peut réécrire la même chose avec des variables “anonymes” :

```
pas-trop-loin(X,Y) :-connectée(X,Y,_).
pas-trop-loin(X,Y) :-connectée(X,Z,_),connectée(Z,Y,_).
```

PROLOG : un exemple



PROLOG : un exemple



PROLOG : un exemple

On peut atteindre une station à partir d'une autre si elles sont sur la même ligne, ou avec un, deux, ... changements :

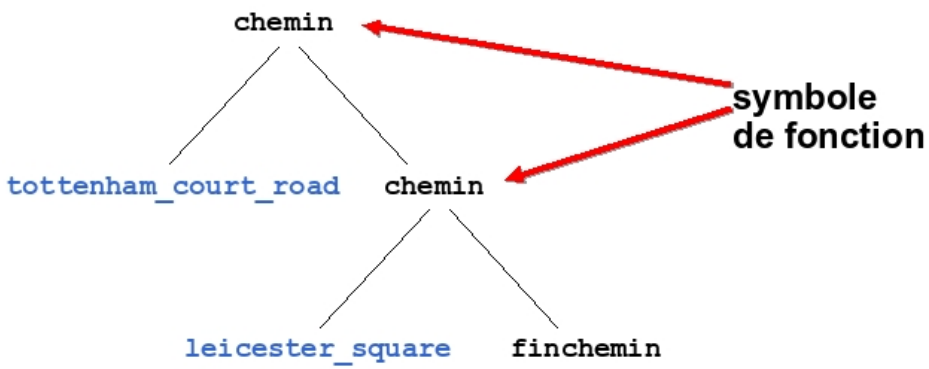
```
atteignable(X,Y) :-connectée(X,Y,L).
atteignable(X,Y) :-connectée(X,Z,L1),connectée(Z,Y,L2).
atteignable(X,Y) :-connectée(X,Z1,L1),connectée(Z1,Z2,L2),
connectée(Z2,Y,L3).
.....
```

ou mieux

```
atteignable(X,Y) :-connectée(X,Y,L).
atteignable(X,Y) :-connectée(X,Z,L),atteignable(Z,Y).
```

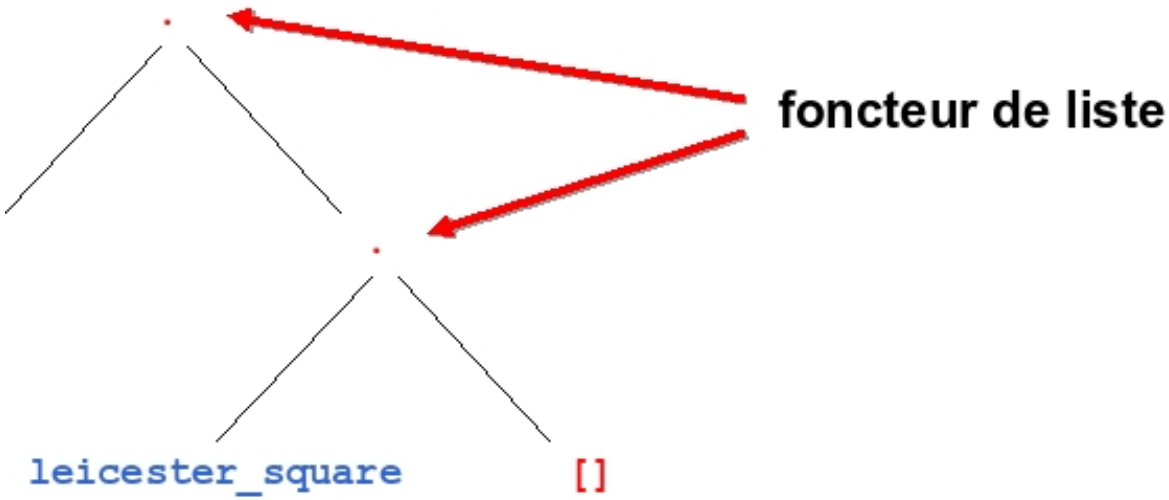
PROLOG : un exemple

```
atteignable(X,Y)(X,Y,finchemin) :-connectée(X,Y,L).
atteignable(X,Y)(X,Y,chemin(Z,R)) :-connectée(X,Z,L),
atteignable(Z,Y,R).
?-atteignable(oxfordcircus,charingcross,R).
R = chemin(tottenhamcourtroad,chemin(leicestersquare,finchemin)) ;
R = chemin(piccadillycircus,finchemin) ;
R = finchemin(picadillycircus,chemin(leicestersquare,finchemin)) ;
```



PROLOG : un exemple

```
atteignable(X,Y,[]) :-connectée(X,Y,L).
atteignable(X,Y,[Z|R]) :-connectée(X,Z,L),
atteignable(Z,Y,R).
?-atteignable(oxfordcircus,charingcross,R).
R = [tottenhamcourtroad,leicestersquare] ;
R = [piccadillycircus] ;
R = [picadillycircus,leicestersquare]
```



Substitution

- Substitution : Ensemble fini (éventuellement vide) de paires de la forme X_i/t_i (X_i étant une variable, t_i un terme), telles que $X_i \neq X_j$ pour $i \neq j$, et X_i n'apparaît pas dans t_i pour i quelconque. On appelle *domaine de la substitution* l'ensemble $\{X_i\}$.
  Ex : $\{X/a\}$, $\{X/0, Y/s(0)\}$ sont des substitutions valides.
  $\{X/f(X)\}$, $\{X/plus(1,2,3)\}$ ne sont pas des substitutions valides..
- Substitution à variables pures : chaque t_i est une variable.
  Substitution close : chaque t_i est un terme clos (sans variable).

Substitutions

- On applique une substitution $\theta = \{X_i/t_i\}_{i=1..n}$ à une formule F en remplaçant simultanément chaque occurrence *libre* de X_i par le terme t_i, pour $1 \leq i \leq n$.
  Soient $p(X, Y, f(a))$ et $\sigma = \{X/b, Y/X\}$,
  $p(X, Y, f(a))\sigma = p(b, X, f(a))$
- Si $\theta = \{X_1/s_1, \ldots, X_n/s_m\}$ et $\sigma = \{Y_1/t_1, \ldots, Y_n/t_n\}$ sont deux substitutions, $\theta\sigma$ est une substitution.
  $\theta\sigma = \{X_1/s_1\sigma, \ldots, X_n/s_m\sigma, Y_1/t_1, \ldots, Y_n/t_n\}$, où on supprime tous les élément de la forme X/X et Y_j/t_j tel que $Y_j \in dom(\theta)$.
  Soient $\theta = \{X/f(Y), Y/Z\}$ et $\sigma = \{X/a, Y/b, Z/Y\}$.
  $\theta\sigma = \{X/f(b), Z/Y\}$.

Substitutions

- Instance : Soient deux termes t_1 et t_2. S'il existe une substitution θ telle que $t_1.\theta = t_2$, alors t_2 est une instance de t_1 et t_1 est une généralisation de t_2. Ex : $t_1 = X$, $t_2 = s(0)$
- Soient E et F deux expressions. E et F sont des variantes s'il existe deux substitutions θ et σ telles que $E = F\theta$ et $F = E\sigma$. Ex : $E = p(X, Y)$, $F = p(A, B)$. Attention, $E = p(X, X)$ n'est pas une variante de $F = p(A, B)$.
- Soit E une expression et V l'ensemble des variables de E. Une substitution de renommage est une substitution à variables pures $\{X_i/Y_i\}$ telle que les $Y_i \neq Y_j$ pour $i \neq j$ et $(V - \{X_i\}) \cap \{Y_i\} = \emptyset$.

Unification

- Soit un ensemble fini de termes $P = \{t_i = s_i\}$. Un unificateur pour P est une substitution θ telle que pour tout $1 \leq i \leq n$ et avec $i \neq j$, $t_i\theta = s_i\theta$. Si une telle substitution existe, on dit que les termes t_i et s_i sont *unifiables*. On note $U(P)$ l'ensemble des unificateurs de P.
- Un unificateur pour deux atomes $p(s_1, \ldots, s_m)$ et $p(t_1, \ldots, t_m)$ est un unificateur pour $P = \{s_i = t_i\}$. Si $U(P) \neq \emptyset$, on dit que P est *unifiable*.
- L'unificateur principal ou plus général (upg) de n expressions E_i est une substitution σ telle que pour tout unificateur θ_j de E_i, il existe une substitution σ_j telle que $\sigma\sigma_j = \theta_j$.
  Ex : $t_1 = p(s(X))$, $t_2 = p(s(s(Y)))$, $\sigma = \{X/s(Y'), Y/Y'\}$, $\theta_1 = \{X/s(0), Y/0\}$

Algorithme d'unification

Algorithme de Montelli-Montanari

1: Procédure UNIFICATION(P) ▷ *Un ensemble d'équations $t_i = s_i$*
2: $X = X \cup E \rightarrow E$ ▷ *Tautologie*
3: $X = t \cup E$ (où X apparaît dans E mais pas dans t) $\rightarrow$
4: $\{X = t\} \cup E[X/t]$ ▷ *Application*
5: $t = X \cup E$ (t n'est pas une variable) $\rightarrow X = t \cup E$ ▷ *Orientation*
6: $f(s_1, \ldots, s_n) = f(t_1, \ldots, t_n) \cup E \rightarrow$
7: $\{s_1 = t_1, \ldots, s_n = t_n\} \cup E$ ▷ *Décomposition*
8: $g(s_1, \ldots, s_m) = f(t_1, \ldots, t_n) \cup E \rightarrow$ échec ▷ *Clash*
9: $X = t \cup E$ (où X apparaît dans t) $\rightarrow$ échec ▷ *Test d'occurrence*
10: Fin Procédure

Exemples

- $\{p(f(X), a), p(Y, f(W))\}$ n'est pas unifiable
- $\{p(f(X), Z), p(Y, a)\}$ est unifiable
- $\{p(f(X), h(Y), a), p(f(X), Z, a), p(f(X), h(Y), b)\}$ unifiable ?
- $\{p(f(a), g(X)), p(Y, Y)\}$
- $\{p(a, X, h(g(Z))), p(Z, h(Y), h(Y))\}$
- $\{p(X, X), p(Y, f(Y))\}$

Propriété de l'algorithme d'unification

- L'algorithme précédent termine
- Chaque application préserve l'ensemble des solutions
- Si P est unifiable, l'algorithme termine avec un upg pour P
- S'il existe, l'upg d'un ensemble d'expressions est unique à un renommage de variable près.

L' interprète PROLOG

- Mécanisme de base : Résolution SLD (réduction de buts).
- Soit A_i un littéral (négatif) dans une requête PROLOG. La réduction de A_i étant donné un programme P est le remplacement de A_i par le corps d'une instance d'une clause $A : -B_1, \ldots B_i, \ldots B_m$ de P telle que A et A_i aient une instance commune.
- Résolvante : Etat intermédiaire de la requête PROLOG.

L'interprète PROLOG (simplifié)

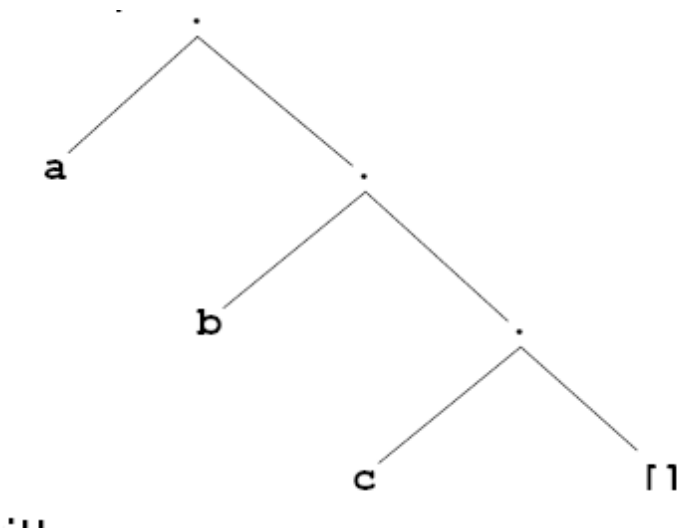
1: Fonction INTERPRETEPROLOG(P, Q) ▷ Q requête et P programme logique ▷ La réponse sera $Q\theta$ si Q est conséquence logique de P, echec sinon
2: $Resolvante \leftarrow Q$
3: Tant que $Resolvante : -A_1, ..A_i, ..A_n \neq \square$ et qu'il existe un but A_i et une clause de P (renommée) $A : -B_1, ..B_i, ..B_m$, telle que $A\theta = Ai\theta$ Faire
4: $Resolvante \leftarrow (: -(A_1, .., A_{i-1}, B_1, ..., B_m, A_{i+1}, .., A_n)\theta$)
5: $Q \leftarrow Q\theta$
6: Fin Tant que
7: Si $Resolvante = \square$ Alors
8: Retourner Q
9: Sinon
10: Retourner ECHEC
11: Fin Si
12: Fin Fonction

L' interprète PROLOG, remarques

- Trace du programme PROLOG : Suite des résolvantes pendant le déroulement du programme PROLOG.
- Arbre de preuve PROLOG : Arbre des buts réduits pendant le déroulement du programme PROLOG.
- Points de choix
  - Choix du sous but à résoudre A_i : le plus à gauche dans la résolvante
  - Choix de la clause de P à utiliser dans la réduction : la première par ordre d'apparition dans P satisfaisant $A\theta = Ai\theta$
- Mécanisme de retour arrière intégré dans PROLOG.
  Si échec, remise en cause du choix immédiatement précédent de la clause de P utilisée pour résoudre le sous-but courant.
  On peut également forcer le retour arrière (;) : PROLOG énumère alors toutes les solutions.

Représentation des listes

- Une liste est une structure de données définie comme une séquence d'élements de longueur n
  - Exemple : la liste contenant les constantes demain, ce, sont, les, vacances s'écrira en Prolog : `[demain, ce, sont, les, vacances]`
- Comment est réprésentée une liste en Prolog ?
  - comme tous les objets en Prolog, sous la forme d'un arbre. Par exemple, la liste `[a, b, c]` est représentée, à l'aide de l'opérateur . (cons) comme l'arbre



...

Représentation des listes

- Il est souvent nécessaire de manipuler la totalité de la queue d'une liste
  - Exemple : soit `L = [a,b,c]`.
    `?- L = [Head|Tail]` réussit avec `Head = a` et `Tail = [b,c]` et L = .(a, Tail)
- Il est possible de lister n'importe quel nombre d'éléments de la liste suivi de « — » et le reste de la liste :
  - [a,b,c] = [a—[b,c]] = [a,b—[c]] = [a,b,c—[]]

Opérations sur les listes

- Les principales opérations sur les listes sont les suivantes
  - verifier si un éléement est présent dans une liste
  - concaténer deux listes, i.e., obtenir une troisième liste qui correspond à l'union des deux listes
  - ajouter ou supprimer un élément à une liste
  - compter le nombre d'éléments d'une liste
  - ...
- En Prolog, la liste permet également de modéliser les ensembles

Opérations sur les listes

- Le programme permettant de déterminer si la relation `member(X,L)` est vrai s'appuie sur les observations suivantes : `X` est un membre de `L` si
  - `X` est la tête de `L`, ou
  - `X` est membre de la queue de `L`
- Le programme s'écrit en Prolog

```
member(X, [X| Tail]).
member(X, [_|Tail]) :- member(X, Tail).
```

Opérations sur les listes

- Soit la relation `member(X,L)` qui est vrai `X` est élément d'une liste `L`
  - `?- member(b,[a,b,c])` réussit
  - `?- member(b,[a,[b,c]])` échoue
  - `?- member([b,c],[a,[b,c]])` réussit
  - `?- member(X,[a,b,c])` ?
  - `?- member(a,L)` ?

Opérations sur les listes

- L'opérateur de concaténation : pour écrire le programme permettant de concaténer deux listes, deux cas doivent être considérés en fonction du premier argument `L1` :
  - Le premier argument est la liste vide, la seconde liste et le résultats sont identiques
  - Le premier argument est une liste non vide, qui peut donc être décomposé en une tête et une queue

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

Opérations sur les listes

- L'opérateur de concaténation peut servir à vérifier qu'une liste respecte une certaine forme
- Exemple : il est possible de trouver un élément qui précède un élément donné de la liste :

```
?- conc (Before , [may| After] , [jan, feb, mar,
apr, may, jun, jul, aug, sep, oct, nov, dec]).
Before = [jan, feb, mar , apr]
After = [jun, jul, aug, sep, oct, nov, dec]
```

- Autre exemple :

```
?- conc (_, [Month1,may,Month2|_] , [jan, feb,
mar, apr, may, jun, jul, aug, sep, oct, nov, dec])
.
Month1 = apr
Month2 = jun
```

Opérations sur les listes

- L'opérateur de concaténation peut être utile pour décomposer une liste en deux listes
- Utilisation inversée de l'opérateur `conc`

```
?- conc (L1, L2, [a, b, c]).
conc (L1, L2, [a, b, c]).
L1 = [ ]
L2 = [a b, c] ;
L1 = [a]
L2 = [b, c] ;
L1 = [a, b]
L2 = [c] ;
L1 = [a, b, c]
L2 = [ ] ;
```

Opérations sur les listes

- L'opérateur de concaténation peut permettre de supprimer les éléments d'une liste après un élément ou une une séquence d'éléments
- Exemple :

```
?- L1 = [a, b, z, z, c, z, z, z, d, e] , conc (L2,
[z, z, z|_] , L1).
L1 = [a , b, z, z, c, z, z, z, d, e]
L2 = [a , b, z, z, c]
```

- L'opérateur de concaténation peut permettre de tester si un élément est présent dans une liste ou non

```
member1(X,L) :- conc (_, [X|_ ],L)
```

Logique clausale propositionnelle

- Connecteurs
  - `:-` si
  - `;` ou
  - `,` et
- clause : `tete [ :- corps ].`
- tete : `[ litteral[ ; litteral ] ]`
- corps : `litteral [ , litteral ]`
- Exemple :

  ```
  marie ; celibataire :- homme, adulte.
  ```

  Un homme adulte est soit marié, soit célibataire

Logique clausale propositionnelle

- Un programme est un ensemble fini de clauses, toutes terminées par un point ; un ensemble de clauses doit s'interpréter comme une conjonction de clauses.
- En notation PROLOG

```
homme ;femme :- humain.
humain :- homme.
humain :- femme.
```

- En notation logique classique

$(humain \rightarrow (homme \vee femme)) \wedge$
$(homme \rightarrow humain) \wedge$
$((femme \rightarrow humain)$

- ou encore

$(\neg humain \vee homme \vee femme) \wedge$
$(\neg homme \vee humain) \wedge$
$(\neg femme \vee humain)$

Clause

Une clause $H_1; \ldots; H_n : -B_1, \ldots, B_m$ est équivalente à

$H_1 \vee \ldots H_n \vee \neg B_1 \vee \ldots \vee \neg B_m$

- Une tête vide se lit comme la constante `faux`, un corps vide se lit comme la constante `vrai`

  ```
  homme.
  :- impossible.
  ```

  est équivalent à $(vrai \rightarrow homme) \wedge (impossible \rightarrow faux)$
  i.e., $homme \wedge \neg impossible$

Sémantique

- La base de Herbrand BP d'un programme P est l'ensemble de tous les atomes qui apparaissent dans P.
- Une interprétation de Herbrand de P est une application $i : BP \mapsto \{true, false\}$
- On représente i par l'ensemble des atomes vrais.
- Une interprétation est un modèle pour une clause si la clause est vraie dans cette interprétation, i.e. si soit sa tête est vraie soit son corps est faux.
- Une interprétation est un modèle pour le programme si elle est modèle pour chaque clause du programme.

Sémantique, exemple

- Soit P :

  ```
  femme ;homme :- humain.

  humain :- homme.

  humain :- femme.
  ```

- alors, $BP = \{homme, femme, humain\}$, et
  $i = \{femme, humain\}$ est une interpretation possible
- L'interprétation $j = \{femme\}$ n'est pas un modèle de P.

Conséquence logique

- Une clause C est conséquence logique du programme P, noté $P \models C$, si tout modèle de P est aussi modèle de C.

```
femme.
```

```
femme ;homme :- humain.
```

```
humain :- homme.
```

```
humain :- femme.
```

- $P \models humain$. P a deux modèles : $M_1 = \{femme, humain\}$, $M_2 = \{femme, homme, humain\}$. Intuitivement, on préfère M_1 qui rend vrai le minimum de faits.

Modèle minimal

- On définit le modèle préféré comme le plus petit.
- Soit P_0 :

  ```
  femme ;homme :- humain.

  humain.
  ```

- P a 3 modèles

  $M_1 = \{femme, humain\}$

  $M_2 = \{homme, humain\}$

  $M_3 = \{femme, homme, humain\}$
- et M_1 et M_2 sont tous les deux minimaux !
- Si on se restreint (comme dans Prolog) aux clauses définies (1 seul littéral positif), un programme défini a un unique modèle minimal

Résolution

- ou, comment calculer les conséquences logiques sans énumérer tous les modèles ?
- Utiliser la résolution comme règles d'inférence.

  `marié ;célibataire :- homme,adulte.`

  `a_une_femme :- homme,marié.`
- En utilisant la résolution, on obtient `a_une_femme ;célibataire :- homme, adulte.` qui est une conséquence logique du programme.
- La résolution est correcte et réfutation complète.

Logique clausale relationnelle

- PROLOG sans autre symbole de fonction que des constantes
- Par rapport à la logique propositionnelle : on ajoute les constantes et les variables quantifiées
- `apprécie(peter,S) :- etudiant(S,peter)` $(\forall X\ (apprecie(peter, S) \vee \neg etudiant(S, peter)))$ signifie pour tout S, si S est un étudiant de peter, alors peter apprécie S
- L'arité d'un prédicat est le nombre de ses arguments. Par exemple, `apprécie/2`. En Prolog, `p/2` est différent de `p/3`.
- Un terme (atome) est clos s'il ne contient aucune variable.

Sémantique

- L'univers de Herbrand d'un programme P est l'ensemble de tous les termes clos qui apparaissent dans P.
- La base de Herbrand BP de P est l'ensemble de tous les atomes clos qui peuvent être construits à partir des prédicats de P et d'arguments pris dans l'univers de Herbrand de P.
- Une interprétation de Herbrand I de P est un sous-ensemble $I \subseteq BP$ d'atomes clos qui sont vrais, étant donné P.

Sémantique, exemple

- Soit P_3 :

  ```
  apprécie(peter,S) :- étudiant_de(S,peter).
  étudiant_de(maria,peter).
  ```

- $BP_3 =$
  $\{apprecie(peter, peter), apprecie(peter, maria),$
  $apprecie(maria, peter), apprecie(maria, maria),$
  $etudiant\_de(peter, peter), etudiant\_de(peter, maria),$
  $etudiant\_de(maria, peter), etudiant\_de(maria, maria)\}$
- Une interprétation de P_3 :
  $I_3 = \{apprecie(peter, maria), etudiant\_de(maria, peter)\}$

Sémantique

- Une instance de la clause C est obtenue en appliquant à C une substitution. Une instance close de C ne contient que des atomes clos (sans variable).
- Une interprétation I de C est une modèle de C si et seulement si elle est modèle de toutes les instances closes de C.
- Toutes les instances closes de clauses de P_3 sont :

```
apprécie(peter,peter) :- étudiant_de(peter,peter).
apprécie(peter,maria) :- étudiant_de(maria,peter).
étudiant_de(maria,peter).
```

- Donc,
  $M_3 = \{apprecie(peter, maria), etudiant\_de(maria, peter)\}$ est un modèle de P_3.

Théorie de la Preuve

- Résolution en logique clausale relationnelle : on peut faire de la résolution avec toutes les instances closes de clauses de P, ce qui est excessivement lourd ! !
- Afin d'être plus efficace, et de mener toutes les résolutions sur toutes les instances closes de clauses de P, on a introduit la notion d'unificateur le plus général
- Ceci va permettre de faire des résolutions avec de multiples instances de clauses closes "à la fois"

Résolution en Logique des Prédicats

C_1, C_2, C sont des clauses, l_1 et l_2 sont deux littéraux. On suppose que $C_1 \vee l_1$ et $C_2 \vee \neg l_2$ ne partagent aucune variable. Les littéraux l_1 et l_2 ont un upg θ.

- Résolution.

$$\frac{C_1 \vee l_1 \qquad C_2 \vee \neg l_2,}{R = (C_1 \vee C_2)\theta}$$

- Factorisation.

$$\frac{C \vee l_1 \vee l_2}{(C \vee l_1)\theta}$$

Exemples

- Soit P_4 :

  `apprécie(peter,S) :-étudiant_de(S,peter).`

  `étudiant_de(S,T) :- suit_cours(S,C), enseigne(T,C).`

  `enseigne(peter,plog).`

  `suit_cours(maria,plog).`

  `'y-a-t il quelqu'un que peter apprécie ?'` (requête)

  ajouter `'peter n'apprécie personne'`

  `:-apprécie(peter,N).`

- $P_4 \models apprecie(peter, maria)$

Correction, complétude

- La logique clausale relationnelle est correcte et réfutation complète

  $(P \vdash C) \Rightarrow (P \models C)$

  P, C incohérent $\Rightarrow P, \neg C \vdash \square$
- La question $P \models C$ est décidable pour la logique clausale relationnelle parce que la base de Herbrand est finie.

Logique clausale

- Par rapport à la logique clausale relationnelle, on ajoute les symboles de fonction d'arité non nulle.

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

- Les programmes exprimés en logique clausale peuvent avoir des modèles de Herbrand infinis.
- $BP_5 =$
  $\{0, s(0), s(s(0)), \ldots, s^n(0), \ldots\}$
- $M_5 =$
  $\{plus(0,0,0), plus(s(0),0,s(0), plus(s(s(0)),0,s(s(0)), \ldots,$
  $plus(0,s(0),s(0)), plus(s(0),s(0),s(s(0))), \ldots\}$

Exemples

- Soit $C : \neg p(X, a) \vee \neg p(f(Y), Y) \vee q(Y, Z)$
  - On applique l'algorithme d'unification à $\{X = f(Y), a = Y\}$, qui rend un unificateur le plus général $\theta = \{X/f(a), Y/a\}$.
  - On applique θ à C : un facteur de C est donc : $C\theta : \neg p(f(a), a) \vee q(a, Z)$
- Soit $C : \neg p(Y) \vee \neg p(f(Y))$. Cette clause a-t-elle un facteur ?

Exemples

- Soient les clauses $C_1 : p(X) \vee p(X')$ et $C_2 : \neg p(Y) \vee \neg p(Y')$

$$\dfrac{\dfrac{C_1 : p(X) \vee p(X')}{C_1' : p(X)} \qquad \dfrac{C_2 : \neg p(Y) \vee \neg p(Y')}{C_2' : p(Y)}}{\square}$$

- Soient les clauses $C_1 : p(X)$, $C_2 : \neg p(Y) \vee p(f(Y))$ et $C_3 : \neg p(f(f(Z)))$

$$\dfrac{\dfrac{\dfrac{C_1 : p(X) \qquad C_2 : \neg p(Y) \vee p(f(Y))}{p(f(Y))} \qquad C_2 : \neg p(Y) \vee p(f(Y))}{p(f(f(Y')))} \qquad C_3 : \neg p(f(f(Z)))}{\square}$$

Correction, complétude

- La logique clausale est correcte et réfutation complète
- La question $P \models C$ est seulement semi-decidable, i.e. il n'existe pas d'algorithme qui va toujours répondre à la question (par oui ou non) en temps fini ; par contre, il existe un algorithme qui, si $P \models C$, répond oui en temps fini, mais qui peut boucler si $P \not\models C$.

Formes normales

- En logique propositionnelle, toute formule a une formule logiquement équivalente sous forme normale conjonctive
- Est-ce le cas pour toute formule en logique des prédicats ? ... presque

Nouvelles équivalences

Dans la suite, Q et Q' dénote un quantificateur parmi $\{\forall, \exists\}$, $*$ dénote $\wedge$ ou $\vee$.

1. $\neg(\forall X\ A) \equiv \exists X(\neg A)$
2. $\neg(\exists X\ A) \equiv \forall X(\neg A)$
3. $\forall Y \forall X\ A \equiv \forall X\ \forall Y\ A$
4. $\exists Y \exists X\ A \equiv \exists X\ \exists Y\ A$
5. $\forall X\ (A \wedge B) \equiv \forall X\ A \wedge \forall X\ B$.
6. $\exists X\ (A \vee B) \equiv \exists X\ A \vee \exists X\ B$
7. Les équivalences du type : $QX\ (A * C) \equiv QX\ A * C$ sont vérifiées si C ne contient pas la variable X
8. Les équivalences du type :
   $QX\ A * Q'X\ B \equiv QX\ Q'Y\ (A * B[X/Y])$ où Y est une variables qui n'apparaît ni dans A ni dans B

Equivalences, suite

Attention, les formules suivantes ne sont que des conséquences logiques :

- $\exists X \forall Y A \models \forall Y \exists X A$
- $\forall X A \vee \forall X B \models \forall X (A \vee B)$
- $\exists X (A \wedge B) \models \exists X A \wedge \exists X B$

Une formule F est sous forme rectifiée si

- $VarLibres(F) \cap VarLiees(F) = \emptyset$
- si Q_i et Q_j sont deux occurrences distinctes de quantificateurs en F, Q_i et Q_j portent sur deux variables différentes.

Principe de renommage : si X est une variable apparaissant dans F et Q un quantificateur $\in \{\exists, \forall\}$. $Q\ X\ F \equiv Q\ Y\ F[X/Y]$

Mise sous forme normale

On peut associer à une formule close F un ensemble de clauses qui admet un modèle si et ssi F est satisfiable.

1. Mise sous forme prénexe (équivalente à F)
2. Skolémisation (préservation de la satisfiabilité de F), donc il existe une skolémisation de F insatisfiable si et ssi F est insatisfiable
3. Mise sous forme clausale (conjonction de disjonctions)

Forme prénexe

- Une fbf F est sous forme prénexe lorsqu'elle a la forme suivante $Q_1X_1 \ldots Q_nX_n\ G$ où les Q_i sont des quantificateurs et G ne contient aucun quantificateur.
- Toute formule de LPO a une formule équivalent en forme prénexe
- Le principe est le même que pour la mise sous forme clausale pour les formules de la logique des propositions + on "repousse" les quantificaticateurs à l'extérieur de la formule en utilisant les équivalences 5,6,7,8 du transparent 36
- $(\forall X p(X)) \rightarrow (\exists X q(X)) \equiv \exists X(\neg p(X) \vee q(X))$
- $(\exists X(p(X) \rightarrow q(X))) \rightarrow (\forall X P(X) \rightarrow \exists X Q(X)) \equiv$
  $\forall X \exists Y((P(X) \wedge \neg Q(X)) \vee (\neg P(Y) \vee Q(Y)))$

Formes clausale - de Skolem

- Une fbf est en forme clausale si elle est la fermeture universelle d'une conjonction de disjonction de littéraux
- Si F est une fbf quelconque, il n'est pas toujours possible de trouver une forme clausale équivalente à F
- Si F est est insatisfiable, c'est toujours possible
- La forme de skolem associée à une formule sous forme prénexe $Q_1X_1 \ldots Q_nX_n\ F$, avec F sous la forme d'une conjonction de disjonction de littéraux d'un langage du premier ordre s'obtient en supprimant les quantificateurs existentiels de gauche à droite de la façon suivante :

Formes clausale - de Skolem

Fonction SKOLÉMISATION($F : Q_1X_1 \dots Q_nX_n\ G$) ▷ F *forme prénexe*
  Tant que il reste un quantificateur existentiel dans F Faire
    i : rang du premier $\exists$ dans F
    X_i :la variable sur laquelle il porte
    Soit f un symbole de fonction d'arité $i-1$ t.q. $f \notin G$
    $F \leftarrow$
    $Q_1X_1 \dots Q_{i-1}X_{i-1}Q_{i+1}X_{i+1} \dots Q_nX_n\ G[X_i/f(X_1, \dots, X_{i-1})])$
  Fin Tant que
  Retourner F
Fin Fonction

- Soit $F : \exists X\ \exists Y\ \forall Z\ \forall T\ \exists V\ p(X, Y, Z, T, V)$. Une forme skolémisée de F est $\forall Z\ \forall T\ p(a, b, Z, T, f(Z, T))$.
- Théorème : Soit F une formule sous forme prénexe et soit F_s une forme skolémisée de F. F est satisfiable si et ssi F_s l'est.

Logique clausale

- Pour des raisons d'efficacité : restriction à des clauses définies (dont la tête ne contient qu'un litéral).
- $A : -B_1, \ldots, B_n$
- Interprétation : pour prouver A, il faut prouver chacun des $B_1, \ldots, B_n$.
- Comment représenter `marié(X) ;célibataire(X) :-homme(X), adulte(X).` dans un langage declauses définies ?
- Pour prouver `marié(X)`, il faut démontrer `homme(X)` puis `adulte(X)` et enfin que `not célibataire(X)` .

Clauses générales

- Une clause générale (pseudo-définie) peut contenir des négations dans son corps.

  `marié(X) :- homme(X), adulte(X), not(célibataire(X)).`

  Etant donnés `homme(jim).` `adulte(jim).`, cette clause aura pour modèle (minimal) { `marié(jim), adulte(jim), homme(jim)` }

  `célibataire(X) :- homme(X), adulte(X), not(marié(X)).` a, étant donné `homme(jim).` et `adulte(jim).`, pour modèle (minimal) {`célibataire(jim), adulte(jim), homme(jim)`

Prédicats non logiques

- Prédicats de types pour les termes : `var(X)`, `nonvar(X)`, `atom(X)`, `integer(X)`, `atomic(X)`, `compound(X)` ...
- Comparaison de termes : `X = Y`, `X \= Y`, `X == Y`, `X \== Y`, ...
- Accès aux structures `functor(Terme,Foncteur,Arité)`, `arg(Nieme,Terme,Arg)`, `Terme =.. Liste` où `Liste = [Foncteur|LArgs]`
- Entrées / Sorties : voir la doc en ligne ...
- Prédicats arithmétiques : `is`, +, -, >, <, =<, ...

Contrôle en PROLOG

Conjonction : `,`

Disjonction : `;`

Coupure : `!`

Appel méta de l'interprète : `call(X)` où `X` est un but PROLOG.

If-then-else : `(P -> Q ; R)`

If -then : `(P -> Q)`

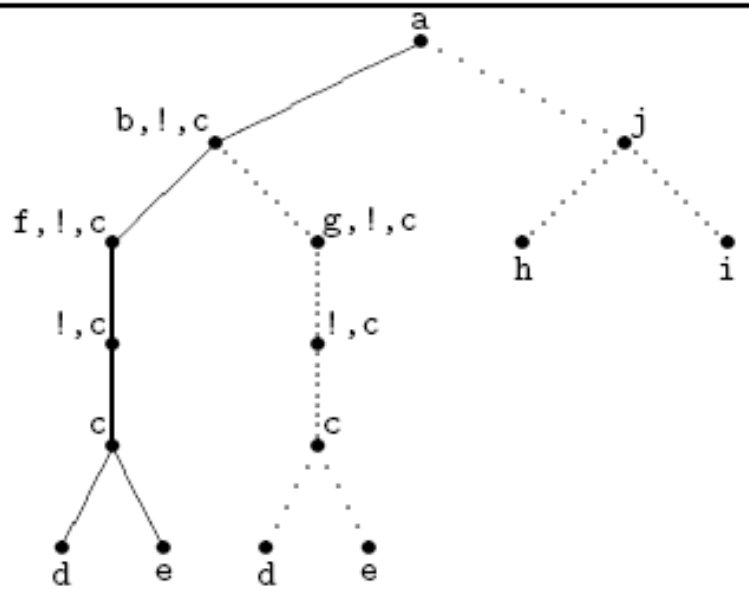
`true,fail`

Le cut : introduction

- Coupe choix, ! : contrôle dynamique du bactrack PROLOG.
- Affecte le comportement procédural des programmes PROLOG : élaguage dynamique de l'arbre de recherche de PROLOG.
- Avantage : efficacité accrue des programmes PROLOG (cuts verts). Inconvénient : pas d'interprétation déclarative.
- Le cut empêche de remettre en cause tous les choix qui ont été faits depuis le moment où le but parent a été unifié avec la tête de la clause où apparaît le cut.
- Le cut est un prédicat qui réussit toujours et qui ne réussit qu'une fois.

Le cut : exemple

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



Le cut en pratique

- empêche de considérer les autres clauses de la même procédure qui sont situées après celle qui contient le cut.
- empêche de remettre en cause les sous buts du corps de la clause contenant le cut placés avant le cut.
- n'affecte en rien la façon dont sont résolus les buts après le cut.
  ! ! ! Sensibilité accrue à l'ordre des clauses (non déclarativité).
- Son utilisation :
  - on veut élaguer des branches qui ne mènent pas à une solution.
  - on ne veut pas produire toutes les solutions

Différents types de cuts

- Dans tous les cas, l'interprétation procédurale du programme est modifiée : le programme de n'exécute pas de la même manière avec ou sans coupe-choix
- Dans certains cas, la signification déclarative du programme est conservée (coupe-choix "vert") : le programme a la même interprétation logique avec et sans coupe-choix
- Dans les autres cas, la signification déclarative du programme est modifiée (coupe-choix "rouge") : le programme n'a pas la même interprétation logique avec et sans coupe-choix.

Le cut “vert”

Rend un programme déterministe, élague une partie de l'arbre recherche qui mènerait forcément à un échec.
Pour exprimer la nature mutuellement exclusive de tests (test arithmétique, par exemple).

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

peut se réécrire en

```
max2(X,Y,X) :- X >= Y, !.
max2(X,Y,Y) :- X < Y.
```

Le cut “rouge”

Ommission de tests, modification de la signification du progamme.

```
max3(X,Y,X) :- X >= Y, !.
max3(X,Y,Y).
```

Attention ! max3(5,2,2) réussit !

```
max4(X,Y,Z) :- X >= Y, !, X=Z.
max4(X,Y,Y).
```

Autre exemple de cut rouge

- Soit le programme défini `s(1). s(2). s(3).`
- ```
  ps(X) :- s(X),!.
  ds(X) :- ps(Y),s(X), X \== Y,!.
  ```
- ```
  ?- ps(X).
  X = 1 ;
  No
  ?- ds(X).
  X = 2 ;
  No
  ```

La négation en PROLOG

- Le cut permet de distinguer les actions à effectuer suivant qu'une condition est ou n'est pas vérifiée. Il permet donc de représenter une forme restreinte de la négation logique, la négation par l'échec.

```
not X :- call(X), !, fail.
not X.
```

- Sémantique procédurale : si `X` réussit, `not X` échoue ; si `X` échoue, `not X` réussit.
- Attention : La négation par l'échec ne correspond pas à la négation logique, c'est-à-dire, si `not X` réussit, cela ne veut pas forcément dire que `X` est faux, mais qu' on n'a pas pu prouver que `X` était vrai.

Remarques sur le not

- La terminaison de `not X` dépend de la terminaison de `X`.

  ```
  p(s(X)) :- p(X).
  q(a).
  ?- not(p(X),q(X)). ne termine pas
  ```

- L'ordre dans lequel les clauses sont écrites est très important (à cause du cut).
- Le not peut conduire à des réponses incorrectes si le(s) but(s) sur lesquels porte le not ne sont pas instanciés.

  ```
  different(X,Y) :- not X = Y.
  ?- different(2,Y), Y=3.
  No.
  ```

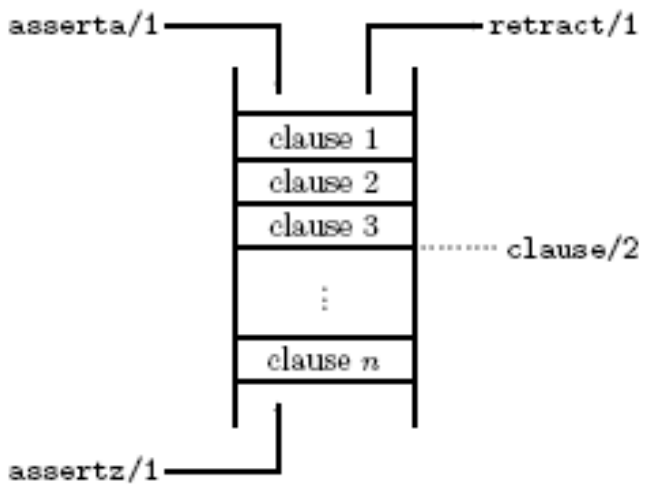
- `not X` n'instancie pas les variables de X en cas de succès.
- `not( not (X = Y))` teste si X et Y sont unifiables sans effet de bord d'instanciation.

If then else

- Symbole de prédicat ->
- ```
  If -> Then ; Else :-
  call(If), !,
  call(Then).
  ```
- ```
  If -> Then ; Else :-
  !,
  call(Else).
  ```
- ```
  If -> Then :-
  call(If), !,
  call(Then).
  ```
  ```

Modifications dynamiques d'un programme PROLOG

- Possibilité en Prolog de modifier dynamiquement la définition de certains prédicats, préalablement déclarés dynamique.
- `:- dynamic toto/2.`
- Prédicats prédéfinis `assert/1`, `asserta/1`, `retract/1`, permettent de modifier le programme



Modifications dynamiques d'un programme PROLOG

Utilisations possibles :

- ajout d'une clause dérivable par le programme (mémo-fonction)

```
fib(0,1) :- !.
fib(1,1) :- !.
fib(N,F) :-
NA is N - 1, NB is N - 2,
fib(NA,FNA), fib(NB,FNB),
F is FNA + FNB,
asserta((fib(N,F) :- !)).
```

- Complexité linéaire en fonction de N au lieu d'exponentielle.

Modifications dynamiques d'un programme PROLOG

Utilisations possibles :

- simulation d'une variable globale avec possibilité d'affectation destructive.

```
affecter(NomDrapeau,Valeur) :-
nonvar(NomDrapeau),
retract(tmpdrap(NomDrapeau,V)),!,
asserta(tmpdrap(NomDrapeau,Valeur)).

affecter(NomDrapeau, Valeur) :-
nonvar(NomDrapeau),
asserta(tmpdrap(NomDrapeau,Valeur)).

valeur(NomDrapeau, Valeur) :-
tmpdrap(NomDrapeau,Valeur).
```

Ensemble de solutions

```
findall(T,But,L) :- call(But), assertz(sol(T)),
fail.
findall(T,But,L) :- assertz(sol('fin')), fail.
findall(T,But ;L) :- recup(L).
recup([T|Q]) :- retract(sol(T)), T \== 'fin',
recup(Q).
recup([]).
```

Findall, example

```
s(b,1). s(a,1). s(c,1). s(d,2). s(a,1).
```

```
?- findall(T,s(T,1),L).
L = [b, a, c, a] ;
No
?- findall(T,s(T,X),L).
L = [b, a, c, a, d] ;
No
?- findall(T,s(T,3),L).
L = [] ;
No
```

```
?- bagof(T,s(T,1),L).
L = [b, a, c, a] ;
No
?- bagof(T,s(T,X),L).
X = 1
L = [b, a, c, a] ;
X = 2
L = [d] ;
No
?- bagof(T,s(T,3),L).
No
```