CC81IART - Intelligence Artificielle

Cours 03 : le langage Prolog

Pierre-Alexandre FAVIER

Ecole Nationale Supérieure de Cognitique

Plan

1. **Résolution de problèmes**
2. **Prolog**

Démontrer ?

- montrer que quelque chose est vrai
- dans le cas d'un système formelle :
  prouver un théorème $=$ vérifier sa vérité *formelle*

Démontrer systématiquement

- choisir un sous-ensemble des axiomes
- choisir un sous-ensemble des règles
- combiner ces axiomes, ces règles et les théorèmes déjà prouvés pour prouver un nouveau théorème
- réitérer jusqu'à obtention du théorème souhaité

Démonstration automatique ?

Pour la logique propositionnelle :

- les tables de vérité (`Wittgenstein`, 1922)
- les arbres sémantiques
- algorithme de `Quine`
- algorithme de `Davis & Putman`
- principe de résolution de `Robinson` (1965)
- **clauses de `Horn`**

La démarche

1. exprimer la proposition logique sous forme d'une expression unique
2. réduire cette expression à sa *forme normale conjonctive*
3. résoudre par réfutation

Résolution par réfutation

- démonstration par l'absurde
- construction d'une conjonction entre les hypothèses et la négation de la conclusion, l'expression formée est fausse si la proposition est vraie
  Exemple : pour prouver que
  $\{(p \rightarrow q), (q \rightarrow r)\} \Rightarrow (p \vee q \rightarrow r)$
  on prouve que $\{(p \rightarrow q), (q \rightarrow r), \neg(p \vee q \rightarrow)r\}$ est faux

Forme normale conjonctive

- que des disjonctions
- les négations ne portent que sur des atomes

Obtention d'une FNC

$$\begin{array}{rll} A \leftrightarrow B & \textit{devient} & (A \rightarrow B) \wedge (B \rightarrow A) \\ A \rightarrow B & \textit{devient} & \neg A \vee B \\ \neg(A \wedge B) & \textit{devient} & \neg A \vee \neg B \\ \neg(A \vee B) & \textit{devient} & \neg A \wedge \neg B \\ \neg\neg A & \textit{devient} & A \\ A \vee (B \wedge C) & \textit{devient} & (A \vee B) \wedge (A \vee C) \end{array}$$

La règle d'inférence

- une forme de modus ponens :
  si $A \vee C$
  et $C \rightarrow B$
  alors $A \vee B$
- sous forme normale conjonctive :
  si $\{(A \vee C), (\neg C \vee B)\} \Rightarrow A \vee B$

Application de la règle d'inférence

Sélectionner 2 clauses telles qu'on puisse :

1. trouver une proposition positive dans une de ces clauses
2. trouver la même proposition négative dans l'autre clause
3. fusionner ces 2 clauses en une nouvelle

Exemple

- si $\textit{jour} \vee \textit{nuit}$
- et $\neg \textit{nuit} \vee \textit{dormir}$
- alors $\textit{jour} \vee \textit{dormir}$

Généralisation

Pour prouver que des prémisses impliquent une conclusion :

1. mettre sous forme normale conjonctive la négation de la conclusion (pour réfutation)
2. mettre sous forme normale conjonctive les prémisses
3. appliquer la règle d'inférence jusqu'à obtention d'une contradiction (réfutation) :
    1. sélectionner 2 clauses
    2. les résoudre mutuellement
        - si on obtient une contradiction la démonstration est terminée (réfutation)
        - sinon on ajoute la nouvelle clause obtenue au prémisses

Exemple – 1/2

Pour démontrer $\{p \rightarrow r, q \rightarrow r\} \Rightarrow (p \vee q \rightarrow r)$
il faut infirmer $\{p \rightarrow r, q \rightarrow r, \neg(p \vee q \rightarrow r)\}$
Conversion en FNC :

$$
\begin{aligned}
p \rightarrow r &\quad \textit{devient} \quad \neg p \vee r && (1)\\
q \rightarrow r &\quad \textit{devient} \quad \neg q \vee r && (2)\\
\neg(p \vee r \rightarrow r) &\quad \textit{devient} \quad p \vee q && (3)\\
&\quad \textit{et} \quad \neg r && (4)
\end{aligned}
$$

Exemple – 2/2

Les FNC :

1. $\neg p \vee r$
2. $\neg q \vee r$
3. $p \vee q$
4. $\neg r$

Résolution :

- $\neg p$ — *(1) et (4)*
- q — *1ère ligne et (3)*
- $\neg q$ — *(2) et (4)*
- Contradiction ! Le théorème est donc vrai.

Définition

clause de HORN stricte : contient 1 littéral positif et au moins 1 négatif

clause de HORN positive : 1 seul littéral positif

clause de HORN négative : que des littéraux négatifs

L'utilisation de clauses de HORN permet une résolution plus rapide par suppression des conséquences valides (clauses équivalentes).

Résolution de clauses de HORN

Exemple : infirmation de $\{\neg p \vee r, \neg r \vee s, p, \neg s\}$

1. sélection de p et $\neg p \vee r$: $\{r, \neg r \vee s, p, \neg s\}$
2. sélection de r et $\neg r \vee s$: $\{r, s, p, \neg s\}$
3. sélection de $\neg s$ et s : contradiction ! ! !

Interprétation des clauses de HORN

clause de HORN stricte : $q \vee \neg p_1 \vee \neg p_2 \ldots \neg p_n$

équivaut à $\{p_1, p_2, \ldots, p_n\} \Rightarrow q$

C'est une règle !

```
q :- p1, p2, ..., pn.
```

clause de HORN positive : p

C'est un fait !

```
p.
```

clause de HORN négative : $\neg p \vee \neg q \vee \neg r$

Prouver l'inconsistance de cette expression revient à prouver la vérité de ses constitutifs, c'est une question !

```
?- p,q,r.
```


Limitations

Reprenons l'exemple simple :

- si *jour* $\vee$ *dormir*
- et $\neg$*jour*
- alors *dormir*

en logique du premier ordre. . .

- si $jour(X) \vee dormir(X)$
- et $\neg jour(Y)$
- comment faire le lien ?

Le problème

- en logique propositionnelle, une proposition est vraie ou fausse
- en logique des prédicats, un prédicat n'est pas vrai ou faux, il peut être vrai pour certaines valeurs de variables et faux pour d'autres

La solution

- on conserve le principe de résoloution automatique par réfutation
- les clauses sont obtenues par regroupement selon un prédicat commun
- pour que ce regroupement soit vrai, on procède à l'*unification* des variables : on recherche les valeurs cohérentes pour satisfaire les clauses

La démarche de résolution

1. mettre les formules sous forme *prénexe*
2. mettre la forme prénexe sous forme de SKOLEM
3. mettre la forme de SKOLEM sous forme de clauses
4. résoudre les clauses par unification

La forme prénexe

- suppression des implications
- déplacement des quantificateurs à l'extérieur

Exemple

- prédicat :
  $\forall x\ p(x) \land \exists y\ q(y) \rightarrow \exists y(p(y) \land q(y))$
- forme prénexe correspondante :
  $\exists x\ \forall y\ \exists z(\neg p(x) \land \neg q(y)) \lor (p(z) \land q(z))$

Forme prénexe : les règles

$$
\begin{array}{rll}
A \leftrightarrow B & \textit{devient} & (A \rightarrow B) \wedge (B \rightarrow A) \\
A \rightarrow B & \textit{devient} & \neg A \vee B \\
\neg(A \wedge B) & \textit{devient} & \neg A \vee \neg B \\
\neg(A \vee B) & \textit{devient} & \neg A \wedge \neg B \\
\neg\neg A & \textit{devient} & A \\
A \vee (B \wedge C) & \textit{devient} & (A \vee B) \wedge (A \vee C) \\
\neg \forall x A & \textit{devient} & \exists x \neg A \\
\neg \exists x A & \textit{devient} & \forall x \neg A
\end{array}
$$

Forme prénexe : la construction

$\forall x\ p(x) \wedge \exists y\ q(y) \rightarrow \exists y\ (p(y) \wedge q(y))$

1. suppression de $\leftrightarrow$ et $\rightarrow$
   $\neg(\forall x\ p(x) \wedge \exists y\ q(y)) \vee \exists y\ (p(y) \wedge q(y))$
2. renommage des variables
   $\neg(\forall x\ p(x) \wedge \exists y\ q(y)) \vee \exists z\ (p(z) \wedge q(z))$
3. transfert des $\neg$ vers l'intérieur
   $(\exists x\ \neg p(x) \vee \forall y\ \neg q(y)) \vee \exists z\ (p(z) \wedge q(z))$
4. déplacement des quantificateurs
   $\exists x\ \forall y\ \exists z\ ((\neg p(x) \vee \neg q(y)) \vee (p(z) \wedge q(z))$

La forme de SKOLEM

La forme de SKOLEM n'est pas équivalente à la forme initiale, mais si une solution satisfait la forme de SKOLEM alors satisfait aussi la forme initiale. Règles :

- les quantificateurs universels sont sous-entendus
- les quantificateurs existentiels sont remplacés par des fonctions

Exemple

- forme prénexe :
  $\exists x \forall y \exists z((\neg p(x) \vee \neg q(y)) \vee (p(z) \wedge q(z))$
- forme de SKOLEM correspondante :
  - x devient une constante (a) car il ne dépend d'aucune variable
  - z devient une fonction de y ($f(y)$) car il peut dépendre de cette variable

$(\neg p(a) \vee \neg q(y)) \vee (p(f(y)) \wedge q(f(y))$

De la forme initiale à la forme clausale

- formule initiale :
  $\forall x p(x) \wedge \exists y q(y) \rightarrow \exists y(p(y) \wedge q(y))$
- forme prénexe :
  $\exists x \forall y \exists z(\neg p(x) \wedge \neg q(y)) \vee (p(z) \wedge q(z))$
- forme de SKOLEM
  $(\neg p(a) \vee \neg q(y)) \vee (p(f(y)) \wedge q(f(y))$
- forme normale :
  $(p(f(y)) \vee \neg p(a) \vee \neg q(y)) \wedge (q(f(y)) \vee \neg p(a) \vee \neg q(y))$
- forme clausale :
  $\{p(f(y)) \vee \neg p(a) \vee \neg q(y), q(f(y)) \vee \neg p(a) \vee \neg q(y)\}$

Résolution des clauses

- rechercher deux clauses contenant un même prédicat, positif dans l'une, négatif dans l'autre
- tenter de les rendre égaux par unification :
  - une variable peut être remplacée par une constante
  - une variable peut être remplacée par une variable
  - une variable peut être remplacée par une fonction qui ne la contient pas (indépendance)

Le problème abordé

- on possède 2 cubes *a* et *b*
- on pose le cube *b* sur le cube *a* et le cube *a* sur la table
- on veut en déduire que le cube *b* est au-dessus de la table

Les axiomes

- Les axiomes de notre univers :
  - $\forall X \forall Y(sur(X, Y) \rightarrow auDessus(X, Y))$
  - $\forall X \forall Y(auDessus(X, Y) \wedge auDessus(Y, Z) \rightarrow auDessus(X, Z))$
- . . . sous forme normale :
  $\neg sur(U, V) \vee auDessus(U, V)$ *(1)*
  $\neg auDessus(X, Y) \vee \neg auDessus(Y, Z) \vee auDessus(X, Z)$ *(2)*
- Deux axiomes spécifiques à notre exemple
  $sur(b, a)$ *(3)*
  $sur(a, table)$ *(4)*

Démonstration automatique

1. $\neg sur(U, V) \vee auDessus(U, V)$
2. $\neg auDessus(X, Y) \vee \neg auDessus(Y, Z) \vee auDessus(X, Z)$
3. $sur(b, a)$
4. $sur(a, table)$
5. $\neg auDessus(b, table)$
6. (2) et (5) donnent, en posant $X = b$ et $Z = table$:
   $\neg auDessus(b, Y) \vee \neg auDessus(Y, table)$
7. (6) et (1) donnent, en posant $U = Y$ et $V = table$:
   $\neg sur(Y, table) \vee \neg auDessus(b, Y)$
8. (7) et (1) donnent, en posant $U = b$ et $V = Y$:
   $\neg sur(b, Y) \vee \neg sur(Y, table)$
9. (8) et (3) donnent :
   $\neg sur(a, table)$
10. (9) et (4) sont en contradiction : la démonstration est faite !

L'équivalent Prolog

```
sur(a, table).
sur(b, a).
auDessus(X,Z):- sur(X,Z).
auDessus( X, Z) :- auDessus(X, Y), auDessus(Y, Z).
```

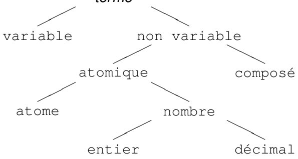
Question posée :

```
?- auDessus(b, table).
yes
?-
```


Les origines de l'idée

- travaux des logiciens (J. HERBRAND, J. ROBINSON...)
- université de Marseille (A. COLMERAUER, PH. ROUSSEL...)
- université d'Edinburgh (R. KOWALSKI, D. WARREN...)

Les termes Prolog



Les variables – 1/2

Une variable (ou inconnue) peut remplacer n'importe quel terme Prolog.

variable instanciée à un terme : la variable a pour valeur ce terme.

variable libre : la variable n'est instanciée à aucun terme.

variables liées : des variables libres peuvent être liées entre elles : dès que l'une d'entre elles sera instanciée à un terme, les autres variables qui lui sont liées le seront aussi.

Les variables – 2/2

Règle lexicale : `[A-Z_][a-zA-Z_0-9]*`

Exemples de variables :

```
X
Nom_compose
_variable
_   (variable
anonyme)
```


Les termes atomiques

- les atomes
- les nombres

Les atomes – 1/2

Atomes : un atome permet de représenter un objet quelconque par un symbole.

Règles lexicales :

- identificateur : `[a-z][a-zA-Z_0-9]*`
- opérateur : `[+-*/^<>=~ :.?#&@]+`
- atome 'quoté' : `'(([^'])|(''))*'`

Les atomes – 2/2

Exemples d'atomes :

identificateur	opérateur	atome 'quoté'
atome	= :=	'ATOME'
bonjour	:? :? :	'ca va?!'
c_est_ca	-->	'c"est ca'

Les nombres

- les entiers
- les réels (le séparateur décimal est le point)

Les termes composés

- Un terme composé permet de d'écrire des données structurées.
- Un terme composé est constitué :
  - d'un atome
  - d'une suite de termes (arguments) ; le nombre d'arguments est appelé arité

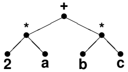
  Le couple **atome/arité** est appelé **foncteur** du terme composé correspondant.

Exemples de termes composés

Foncteur	Terme composé
date/3	date(25,mai,1988)
'etat-civil'/3	'etat-civil'('ac"h',luc, date(1,mars,1965))
c/2	c(3.4,12.7)
c/4	c(a,B,c(1.0,3.5),5.2)
parallele/2	parallele(serie(r1,r2), parallele(r3,c1))

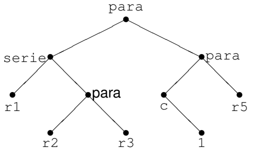
Représentation arborescente – 1/2

2 * a + b * c



Représentation arborescente – 2/2

```
para(serie(r1,para(r2,r3)),para(c(1),r5))
```



Exemple : une fiche d'état civil

individu

état–civil

Nom	:	PETITJEAN
Prénom	:	Albert
Nationalité	:	Française
Sexe	:	masculin

Date de naissance :

Jour	:	15
Mois	:	Février
Année	:	1982

adresse

Rue	:	4 rue Nansouty
Ville	:	Bègles
Code postal	:	33130

Transcription Prolog

% Structures de données

```
% etat_civil(Nom,Prenom,Nationalite,
             Sexe,Date)
% date(Jour,Mois,Annee)
% adresse(Rue,Ville,Code_postal)
```

% Base de données

```
% individu(Etat_civil,Adresse)

individu(etat_civil('PETITJEAN','Albert',
                    francaise,masculin,
                    date(15,'Fevrier',1982)
                   ),
         adresse('4 rue Nansouty','Bègles',33130)
        ).
```

Exemple : les entiers naturels

Le vocabulaire initial de la théorie des entiers naturels comprend :

- une constante z qui représente l'entier 0 (zéro)
- une fonction unaire $s(X)$ qui traduit la notion de successeur d'un entier X
- un prédicat unaire *entier(X)* (*X est un entier*)

Génération :

$$z \in \mathcal{N} \text{ et } \forall x \in \mathcal{N}, s(x) \in \mathcal{N}$$

Addition :

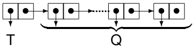
$$\forall x \in \mathcal{N}, x + z = x$$
$$\forall x, y \in \mathcal{N}, x + s(y) = s(x + y)$$

Transcription Prolog

```
entier(z).
entier(s(X)) :- entier(X).

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


Une liste en Prolog

Une liste est une suite ordonnée de termes.



Transcription Prolog

```
liste(0). % 0 equivaut a liste vide}
liste(l(T,Q)) :- liste(Q).

dans(T,l(T,_)).
dans(T,l(_,Q)) :- dans(T,Q).
```

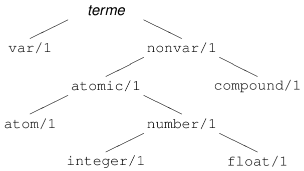
Question possible :

```
?- dans(X,l(a,l(b,l(c,0)))).
X = a
X = b
X = c
true
?-
```

Classification des termes – 1/2

var(T) : T est une variable libre
nonvar(T) : T n'est pas une variable libre
atomic(T) : T est un terme atomique
atom(T) : T est un atome
number(T) : T est un nombre
float(T) : T est un réel
integer(T) : T est un entier
compound(T) : T est un terme composé

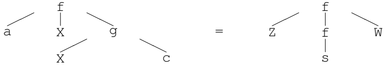
Classification des termes – 2/2



Unification

Unification : mise en correspondance d'arbres syntaxiques

`Terme1 = Terme2`



Comparaison

Par défaut, la comparaison se fait selon l'ordre lexicographique.

*Terme*1 $==$ *Terme*2
*Terme*1 $\backslash ==$ *Terme*2
*Terme*1 @$<$ *Terme*2
*Terme*1 @$=<$ *Terme*2
*Terme*1 @$>=$ *Terme*2
*Terme*1 @$>$ *Terme*2

Inspection des termes

arg(N, T, X) : `X` est le `N`^{ième} argument du terme `T`
functor(T, A, N) : `T` est le terme de foncteur `A/N`

Les opérateurs – 1/2

Un opérateur permet une représentation syntaxique simplifiée d'un terme composé, unaire ou binaire :

en notation préfixée : `~ X` $\equiv$ `~(X)`
en notation postfixée : `X :` $\equiv$ `:(X)`
en notation infixée : `X + Y` $\equiv$ `+(X,Y)`

Aucune fonction n'est *a priori* associée à un opérateur.

Priorité des opérateurs

- Chaque opérateur a une priorité qui va de 1 à 1200.
- La priorité détermine, dans une expression utilisant plusieurs opérateurs, l'ordre de construction du terme composé correspondant.

  `a + b * c` $\equiv$ `a + (b * c)` $\equiv$ `+(a, *(b,c))`

- La priorité d'une expression est celle de son foncteur principal.
- Les parenthèses donnent aux expressions qu'elles englobent une priorité égale à 0.
- La priorité d'un terme atomique est de 0.

Associativité des opérateurs

position	associativité	notation	exemple
infixée	à droite	xfy	a, b, c
	à gauche	yfx	$a + b + c$
	néant	xfx	$x = y$
préfixée	oui	fy	*not not x*
	non	fx	-4
postfixée	oui	yf	
	non	xf	

Quelques opérateurs prédéfinis

priorité	opérateurs	associativité
1200	`:-`	xfx
1200	`:- ?-`	fx
1100	`';'`	xfy
1000	`','`	xfy
900	`help not listing`	fy
700	`= =.. == \==`	xfx
700	`is = := =\= < >`	xfx
500	`+ -`	yfx
400	`* /`	yfx
200	`^`	xfy

Opérateurs personnalisés

⇒ Adapter la syntaxe Prolog à ses besoins
`op(P, A, O).`
permet de définir un opérateur de nom *O*, de priorité *P* et dont l'associativité est définie par *A*
(*A* est un des atomes suivants : `xfx, xfy, yfx, fx, fy, xf, yf`).
Exemple :

```
?- a et b et c.
→ Erreur !
?-op(200, xfy, et).
yes.
?- a et b et c.
→ Ok !
```


Arithmétique : l'évaluation

- *X is Expr*
  X est le résultat de l'évaluation de l'expression arithmétique `Expr`.

```
?- 8 = 5+3. % unification}
false

?- 8 is 5+3 % evaluation
true
?- X is 5+3
X = 8
true
```


Arithmétique : la comparaison

$e_1 \leq e_2$	Expr1	=<	Expr2
$e_1 > e_2$	Expr1	>	Expr2
$e_1 \geq e_2$	Expr1	>=	Expr2
$e_1 = e_2$	Expr1	=:=	Expr2
$e_1 \neq e_2$	Expr1	=\=	Expr2

Exemple : la fonction d'ACKERMAN

$$
\begin{aligned}
f(0,n) &= n+1 \\
f(m,0) &= f(m-1,1) \text{ si } m>0 \\
f(m,n) &= f(m-1,f(m,n-1)) \text{ si } m>0, n>0
\end{aligned}
$$

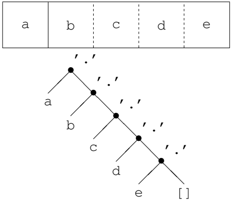
```
f(0,N,A) :- A is N+1.
f(M,0,A) :-
                  M > 0,
                  M1 is M-1,
                  f(M1,1,A).
f(M,N,A) :-
                 M > 0, N > 0,
                 M1 is M-1, N1 is N-1,
                 f(M,N1,A1),
                 f(M1,A1,A).
```


Concept

Représentation standard

liste vide : `[]`
liste non vide : `'.'(Tete,Queue)`

Représentation standard



`'.'(a,'.'(b,'.'(c,'.'(d,'.'(e,[])))))`

Une autre syntaxe – 1/2

$$[T_1, T_2, \dots, T_n|Reste]$$

- $T_1, T_2, \dots, T_n$ représente les n ($n > 0$) premiers termes de la liste
- *Reste* représente la liste des éléments restants
- on a l'équivalence

$$[T_1, T_2, \dots, T_n] \equiv [T_1, T_2, \dots, T_n|[]]$$

Une autre syntaxe – 2/2

```
'.'(a,'.'(b,'.'(c,[])))

≡ [a|[b|[c|[]]]]
≡ [a|[b|[c]]]
≡ [a|[b,c|[]]]
≡ [a|[b,c]]
≡ [a,b|[c|[]]]
≡ [a,b|[c]]
≡ [a,b,c|[]]
≡ [a,b,c]
```


Appartenance à une liste

```
in(T,[T|_]).
in(T,[_|Q]) :- in(T,Q).
```

```
?- in(b,[a,b,c]).
true
?- in(d,[a,b,c]).
false
?- in(X,[a,b,c]).
X = a
X = b
X = c
true
```


Permutation

```
permutation([],[]).
permutation(L,[T|Q]) :-select(T,L,L1),
                       permutation(L1,Q).
select(T,[T|Q],Q).
select(X,[T|Q],[T|L]) :-select(X,Q,L).
```

```
?- permutation([1,2,3],L).
L = [1,2,3] L = [1,3,2]
L = [2,1,3] L = [2,3,1]
L = [3,1,2] L = [3,2,1]
true
```


Tri

```
tri(L1,L2) :-permutation(L1,L2),
          ordonnee(L2).
ordonnee([T]). ordonnee([T1,T2|Q]) :-
          T1 < T2, ordonnee([T2|Q]).
```

```
?- tri([3,1,2],L).
L = [1,2,3]
true
?- tri(L,[1,2,3]).
L = [1,2,3] L = [1,3,2]
L = [2,1,3] L = [2,3,1]
L = [3,1,2] L = [3,2,1]
true
```

Concaténation

```
conc([],L,L).
conc([T|Q],L,[T|QL]) :- conc(Q,L,QL).
```

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


Inversion

```
inverser([],[])).
inverser([T|Q],L) :- inverser(Q,L1),
conc(L1,[T],L).
```

```
?- inverser([a,b,c],L).
L = [c,b,a]
true
?- inverser(L,[c,b,a]).
L = [a,b,c]
!!! Stack Overflow!!!
```


La trace…

```
?- inverser([a,b,c],L).
1 call : inverser([a,b,c],_204)
2 call : inverser([b,c],_328)
3 call : inverser([c],_358)
4 call : inverser([],_388)
4 exit : inverser([],[])
5 call : conc([],[c],_358)
5 exit : conc([],[c],[c])
3 exit : inverser([c],[c])
6 call : conc([c],[b],_328)
7 call : conc([],[b],_632)
7 exit : conc([],[b],[b])
6 exit : conc([c],[b],[c,b])
2 exit : inverser([b,c],[c,b])
8 call : conc([c,b],[a],_204)
9 call : conc([b],[a],_872)
10 call : conc([],[a],_932)
10 exit : conc([],[a],[a])
9 exit : conc([b],[a],[b,a])
8 exit : conc([c,b],[a],[c,b,a])
1 exit : inverser([a,b,c],[c,b,a])
L = [c,b,a]
true
```


Une autre inversion

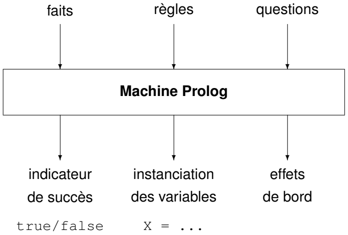
```
inverser(L1,L2) :- inverser(L1,[],L2).
inverser([T|Q],Pile,L) :-
inverser(Q,[T|Pile],L).
inverser([],L,L).
```

```
?- inverser([a,b,c],L).
L = [c,b,a]
true
?- inverser(L,[a,b,c]).
!!! Stack Overflow!!!
```

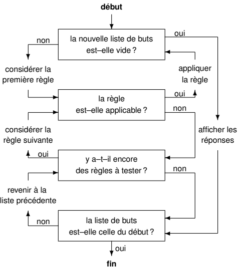
La trace. . .

```
?- inverser([a,b,c],L).
1 call : inverser([a,b,c],_204)
2 call : inverser([a,b,c],[],_204)
3 call : inverser([b,c],[a],_204)
4 call : inverser([c],[b,a],_204)
5 call : inverser([],[c,b,a],_204)
5 exit : inverser([],[c,b,a],[c,b,a])
4 exit : inverser([c],[b,a],[c,b,a])
3 exit : inverser([b,c],[a],[c,b,a])
2 exit : inverser([a,b,c],[],[c,b,a])
1 exit : inverser([a,b,c],[c,b,a])
L = [c,b,a]
true
```


Principe



L'algorithme



Effacement de but – 1/2

1. Chercher une règle (dans l'ordre où elles apparaissent dans le programme) dont la tête s'unifie avec le but à effacer :
   - même foncteur (atome/arité)
   - arguments unifiables
2. Remplacer le but par le corps de la règle applicable en tenant compte des substitutions de variables effectuées lors de l'unification.
   Si le corps de la règle est vide, le but est effacé.

Effacement de but – 2/2

$\{x_1, \ldots, x_r\}$ $[b_1, b_2, \ldots, b_n]$

$\Downarrow$

règle : t :- $q_1, q_2, \ldots, q_m$

unification : $t = b_1$, avec substitution : $\{x_i/t_i, x_j/t_j, \ldots\}$

$\Downarrow$

$\{x_1, \ldots, x_i/t_i, \ldots, x_r\}$ $[q_1, q_2, \ldots, q_m, b_2, \ldots, b_n]$

$\Downarrow$

$\vdots$

$\Downarrow$

$\{x_1/t_1, x_2/t_2, \ldots, x_r/t_r\}$ $[\,]$

Unification – 1/2

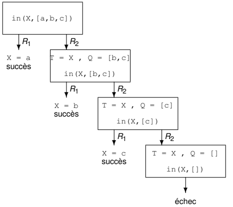
```
u(X,Y) :- var(X), var(Y), X = Y.
u(X,Y) :- var(X), nonvar(Y), X = Y.
u(X,Y) :- nonvar(X), var(Y), X = Y.
u(X,Y) :- atomic(X), atomic(Y),X==Y.
u(X,Y) :- compound(X), compound(Y),
          unifTerme(X,Y).
```


Unification – 2/2

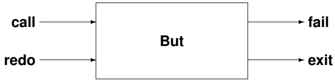
```
unifTerme(X,Y) :-functor(X,F,N),
               functor(Y,F,N),
               unifArgs(N,X,Y).
unifArgs(N,X,Y) :-N > 0,
               unifArg(N,X,Y),
               N1 is N-1,
               unifArgs(N1,X,Y).
               unifArgs(0,X,Y).
unifArg(N,X,Y) :-arg(N,X,ArgX),
               arg(N,Y,ArgY),
               u(ArgX,ArgY).
```


Retour arrière

```
in(T,[T|_]).                    % R1
in(T,[_|Q]) :- in(T,Q).         % R2
```



La trace – 1/3



La trace – 2/3

call *appel initial du but*
: L'entrée par ce port de trace s'effectue avant la première tentative d'unification du but à une tête de clause de la base de clauses.

exit *sortie avec succès du but*
: La sortie par ce port de trace s'effectue lorsque le but a été unifié à une tête de clause et que tous les sous–buts éventuels du corps de la clause ont pu être prouvés.

La trace – 3/3

redo *retour arrière sur un but*
: L'entrée par ce port de trace s'effectue lors d'un retour arrière pour unifier le but à la tête de clause suivante dans la base de clauses.

fail *échec du but initial*
: La sortie par ce port de trace s'effectue lorsque le but ne peut pas être unifié à une tête de clause de la base de clauses, ni à aucun prédicat prédéfini.

Exemple de trace

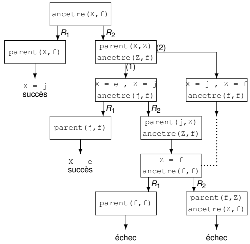
```
?- trace.
Debug mode switched on
true

?- in(X,[a,b,c]).
1 call : in(_192,[a,b,c]) >
1 exit : in(a,[a,b,c])
X = a
1 redo : in(_192,[a,b,c]) >
2 call : in(_192,[b,c]) >
2 exit : in(b,[b,c])
1 exit : in(b,[a,b,c])
X = b
2 redo : in(_192,[b,c]) >
3 call : in(_192,[c]) >
3 exit : in(c,[c])
2 exit : in(c,[b,c])
1 exit : in(c,[a,b,c])
X = c
3 redo : in(_192,[c]) >
4 call : in(_192,[]) >
4 fail : in(_192,[])
true
```

Ordonancement des clauses

L'inversion des clauses ne modifie pas l'arbre de résolution, seul l'ordre des solutions est modifié.

```
parent(e,j). parent(j,f).
                  % ancetre/2 : version 1
ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z),
                  ancetre1(Z,Y).
                  % ancetre/2 : version 2
ancetre2(X,Y) :- parent(X,Z),
                  ancetre2(Z,Y).
ancetre2(X,Y) :- parent(X,Y).
```


Arbre de résolution – ancetre12

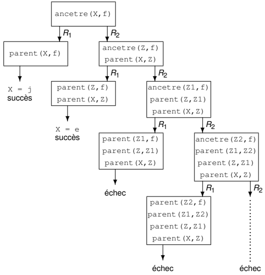


Ordonancement des buts

L'inversion des buts dans une clause modifie l'arbre de résolution.

```
                  % ancetre/2 : version 1
ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z),
                  ancetre1(Z,Y).

                  % ancetre/2 : version 3
ancetre3(X,Y) :- parent(X,Y).
ancetre3(X,Y) :- ancetre3(Z,Y),
                  parent(X,Z).
```


Arbre de résolution – ancetre3



Branches infinies

```
                    % ancetre/2 : version 4
ancetre4(X,Y) :- ancetre4(Z,Y),
                 parent(X,Z).
ancetre4(X,Y) :- parent(X,Y).
```

```
?- ancetre4(X,fabien).
1 call : ancetre4(_192,fabien)
2 call : ancetre4(_248,fabien)
3 call : ancetre4(_278,fabien)
4 call : ancetre4(_308,fabien)
5 call : ancetre4(_338,fabien)
6 call : ancetre4(_368,fabien)
...                                                    % etc

EXCEPTION : stack overflow
```


Les 4 versions de `ancetre/2`

⇐ **Permutations des buts** ⇒

```
ancetre1(X,Y) :- parent(X,Y).
ancetre1(X,Y) :- parent(X,Z),
                 ancetre1(Z,Y).
```

```
?- ancetre1(X,f).
X = j
X = e
true
```

```
ancetre3(X,Y) :- parent(X,Y).
ancetre3(X,Y) :- ancetre3(Z,Y),
                 parent(X,Z).
```

```
?- ancetre3(X,f).
X = j
X = e
... Stack Overflow
```

⇑ ⇑

Permutation des clauses

⇓ ⇓

```
ancetre2(X,Y) :- parent(X,Z),
                 ancetre2(Z,Y).
ancetre2(X,Y) :- parent(X,Y).
```

```
?- ancetre2(X,f).
X = e
X = j
true
```

```
ancetre4(X,Y) :- ancetre4(Z,Y),
                 parent(X,Z).
ancetre4(X,Y) :- parent(X,Y).
```

```
?- ancetre4(X,f).
... Stack Overflow
```

⇐ **Permutations des buts** ⇒

Récursivité non terminale

```
                         % inverser(L1,L2), processus récursif
inverser([],[])).
inverser([T|Q],L) :- inverser(Q,L1), conc(L1,[T],L).
```

```
?- inverser([a,b,c],L).
1 call : inverser([a,b,c],_204)
2 call : inverser([b,c],_328)
3 call : inverser([c],_358)
4 call : inverser([],_388)
4 exit : inverser([],[])
5 call : conc([],[c],_358)
5 exit : conc([],[c],[c])
3 exit : inverser([c],[c])
6 call : conc([c],[b],_328)
7 call : conc([],[b],_632)
7 exit : conc([],[b],[b])
6 exit : conc([c],[b],[c,b])
2 exit : inverser([b,c],[c,b])
8 call : conc([c,b],[a],_204)
9 call : conc([b],[a],_872)
10 call : conc([],[a],_932)
10 exit : conc([],[a],[a])
9 exit : conc([b],[a],[b,a])
8 exit : conc([c,b],[a],[c,b,a])
1 exit : inverser([a,b,c],[c,b,a])
L = [c,b,a]
true
```


Récursivité terminale

```
                         % inverser(L1,L2), processus itératif
inverser(L1,L2) :- inverser(L1,[],L2).
inverser([T|Q],Pile,L) :- inverser(Q,[T|Pile],L).
inverser([],L,L).
```

```
?- inverser([a,b,c],L).
1 call : inverser([a,b,c],_204)
2 call : inverser([a,b,c],[],_204)
3 call : inverser([b,c],[a],_204)
4 call : inverser([c],[b,a],_204)
5 call : inverser([],[c,b,a],_204)
5 exit : inverser([],[c,b,a],[c,b,a])
4 exit : inverser([c],[b,a],[c,b,a])
3 exit : inverser([b,c],[a],[c,b,a])
2 exit : inverser([a,b,c],[],[c,b,a])
1 exit : inverser([a,b,c],[c,b,a])
L = [c,b,a]
true
```

Effets de bord

Certains prédicats ont un comportement procédural ; leurs effets ne sont pas effacés par retour arrière :

Coupe–choix : élagage de l'arbre de résolution

Gestion de la mémoire : ajout et/ou retrait de clauses à l'exécution

Entrées/Sorties : écriture ou lecture de termes

Prédicats de contrôle

! *coupe–choix*
Annule tous les points de choix crées depuis l'appel du but parent.

call (But) *interpréteur*
Evalue **But**.

fail *échec*
Echoue toujours.

\+ But *négation par l'échec*
But n'est pas démontrable.

repeat *point de choix infini*
Réussit toujours même en cas de retour arrière.

true *succès*
Réussit toujours.

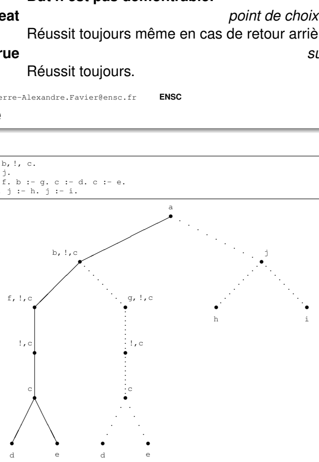
Le coupe-choix

Le coupe–choix (*cut*) a pour but d'élaguer l'arbre de résolution.
Cet élagage *peut* conduire à :

- une plus grande rapidité d'exécution
- moins de place mémoire utilisée
- Dans *tous les cas*, l'interprétation procédurale du programme est modifiée : le programme ne s'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 ou sans coupe–choix.
- Dans *les autres cas*, la signification déclarative du programme est modifiée (coupe–choix " rouge ") : le programme n'a plus la même interprétation logique

Exemple

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



Coupe-choix vert

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

```
?- max(3,2,X).
X = 3
true
?- max(2,3,X).
X = 3
true
?- max(3,2,2).
false
```

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est conservée.

Alternative

```
:- op(900,fx,si).
:- op(850,xfx,alors).
:- op(800,xfx,sinon).
si Cond alors True sinon False :-
                                call(Cond),
                                !,
                                call(True).
si Cond alors True sinon False :-
                                not call(Cond),
                                !,
                                call(False).
                                             % exemple d'utilisation
max(X,Y,Z) :-
                si X > Y alors Z = X
                sinon Z = Y.
```


Coupe-choix rouge

```
max(X,Y,X) :- X > Y,!.
max(X,Y,Y).
```

```
?- max(3,2,X).
X = 3
true
?- max(2,3,X).
X = 3
true
?- max(3,2,2).
true                          %!!!!
```

- La sémantique procédurale du programme est modifiée.
- La sémantique déclarative du programme est modifiée.

Première solution

```
s(1). s(2). s(3).
                    % la première solution
ps(X) :- s(X),!.
                    % la deuxième solution
ds(X) :- ps(Y), s(X), X \== Y,!.
```

```
?- s(X).
X = 1
X = 2
X = 3
true
?- ps(X).
X = 1
true
?- ds(X).
X = 2
true
```

La négation par l’échec

```
                              % non(But)
                  % négation par l'échec
non(But) :- call(But),!,fail.
non(But).
```

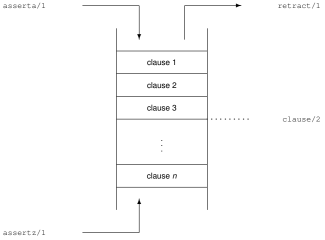
```
?- s(X).
X = 1
X = 2
true
?- non(non(s(X))).
X = _192
true
```


Une boucle de lecture

```
lecture :- repeat,
        read(Terme),
        write('-> '),
        write(Terme),
        nl,
    Terme == fin,!.
```

```
?- lecture.
bonjour.
-> bonjour
salut.
-> salut
fin.
-> fin
true
```


L’espace des clauses – 1/2



L’espace des clauses – 2/2

asserta	(Clause)	*empiler une clause*
assertz	(Clause)	*enfiler une clause*
clause	(Tete,Corps)	*inspecter une clause*
retract	(Clause)	*dépiler (défiler) une clause*

Un chaînage avant

```
deduire :- regle(B,C),
        test(C),
        affirmer(B),!,
        deduire.
deduire.

test((C1,C2)) :-!,
        test(C1), test(C2).
test((C1;C2)) :-!,
        ( test(C1); test(C2) ).
test(C) :-
        regle(C,vrai);
        regle(C,affirme).

affirmer(B) :- not test(B),!,
        assert(regle(B,affirme)),
        write(B),write(' affirme'),nl.
```


Ensemble de solutions

```
findall(T,But,L) :- call(But),
    assertz(sol(T)),
    fail.
findall(T,But,L) :-assertz(sol('c"est
tout')),
    fail.
findall(T,But,L) :- recup(L).

recup([T|Q]) :retract(sol(T)),
    T \== 'c"est tout',
    !,
    recup(Q).
recup([]).
```


findall/3

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

```
?- findall(T,s(T,1),L).
T = _192
L = [b,a,c,a]
true
?- findall(T,s(T,X),L).
T = _192
X = _194
L = [b,a,c,a,d]
true
?- findall(T,s(T,3),L).
T = _192
L = []
true
```


bagof/3

bagof (T,But,L) : `L` est la liste des instances du terme `T` correspondant à chacune des solutions calculées par la résolution de `But`.
Le prédicat `bagof/3` échoue dans le cas où `But` n'admet aucune solution.
Si `But` contient des variables non instanciées n'apparaissant pas dans `T`, `bagof/3` réussit pour chacune des différentes instanciations de cet ensemble de variables.

X ^ But : il existe un `X` tel que `But` soit vrai.
Cette notation est utilisée en deuxième argument du prédicat `bagof/3` pour indiquer que les diverses valeurs de l'argument `T` dans `But` ne provoqueront pas de retour arrière de `bagof/3`.

Exemple

```
?- bagof(T,s(T,1),L).
T = _192
L = [b,a,c,a]
true
?- bagof(T,s(T,X),L).
T = _192 , X = 1 , L = [b,a,c,a]
T = _192 , X = 2 , L = [d]
true
?- bagof(T,X^s(T,X),L).
T = _192 , X = _194 , L = [b,a,c,a,d]
true
?- bagof(T,s(T,3),L).
false
```


`setof/3`

setof (T,But,L) : `L` est la liste des instances du terme `T` , triée dans l'ordre standard défini sur les termes, correspondant à chacune des solutions calculées par la résolution de `But`. La liste triée ne comporte pas d'éléments en double.
Le prédicat `setof/3` échoue dans le cas où `But` n'admet aucune solution.
Si `But` contient des variables non instanciées n'apparaissant pas dans `T`, `setof/3` réussit pour chacune des différentes instanciations de cet ensemble de variables.

```
setof(T,But,OrderedSet) :-bagof(T,But,Bag),
     sort(Bag,OrderedSet).
```


Exemple

```
?- setof(T,s(T,1),L).
T = _192
L = [a,b,c]
true
?- setof(T,s(T,X),L).
T = _192 , X = 1 , L = [a,b,c]
T = _192 , X = 2 , L = [d]
true
?- setof(T,X^s(T,X),L).
T = _192 , X = _194 , L = [a,b,c,d]
true
?- setof(T,s(T,3),L).
false
```


CC81IART - Intelligence Artificielle

Cours 03 : le langage Prolog

Pierre-Alexandre FAVIER

Ecole Nationale Supérieure de Cognitique

IPB
E N S C
BORDEAUX