

Programmation Avancée - Prolog

N. Prcovic

MCours.com

Introduction

- La *programmation logique* est une forme particulière de *programmation déclarative*.
- La programmation déclarative est un paradigme de programmation qui se distingue très nettement de
 - la programmation impérative (C, Pascal, etc)
 - la programmation objet (C++, Java, etc)
 - la programmation fonctionnelle (Lisp, Caml, etc)

Programmation déclarative

- Un programme déclaratif est une suite de déclarations qui constitue une **base de connaissances** dont on ne présuppose pas forcément l'utilisation qu'il en sera fait : on y affirme **ce qui est** mais on ne dit pas **ce qu'il faut en faire**.
- Exemples :
 - "Tout homme est mortel"
"Socrate est un homme"
 - "0 est un entier"
"Le successeur d'un entier est un entier"

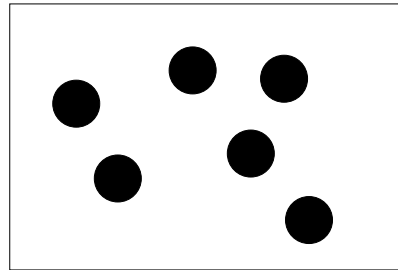
Programmation déclarative

- Pour obtenir une information pouvant être déduite de cette base de connaissances, on a besoin d'un *moteur d'inférence* et d'une *requête*.
- Le moteur d'inférence met en oeuvre une procédure d'extraction de l'information à partir de la requête.
- l'information extraite est :
 - soit explicitement présente dans la base (comme pour les bases de données)
 - soit déduite à partir de données explicitement présentes et de règles de production qui s'appliquent à ces données (ou à celles produites par d'autres règles).

Programmation déclarative

MOTEUR DE RECHERCHE

Base de données

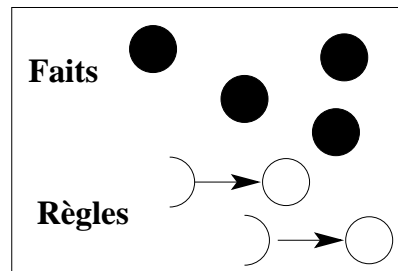


Requete

Résultats

MOTEUR D'INFERENCE

Base de connaissances



Requete

Résultats

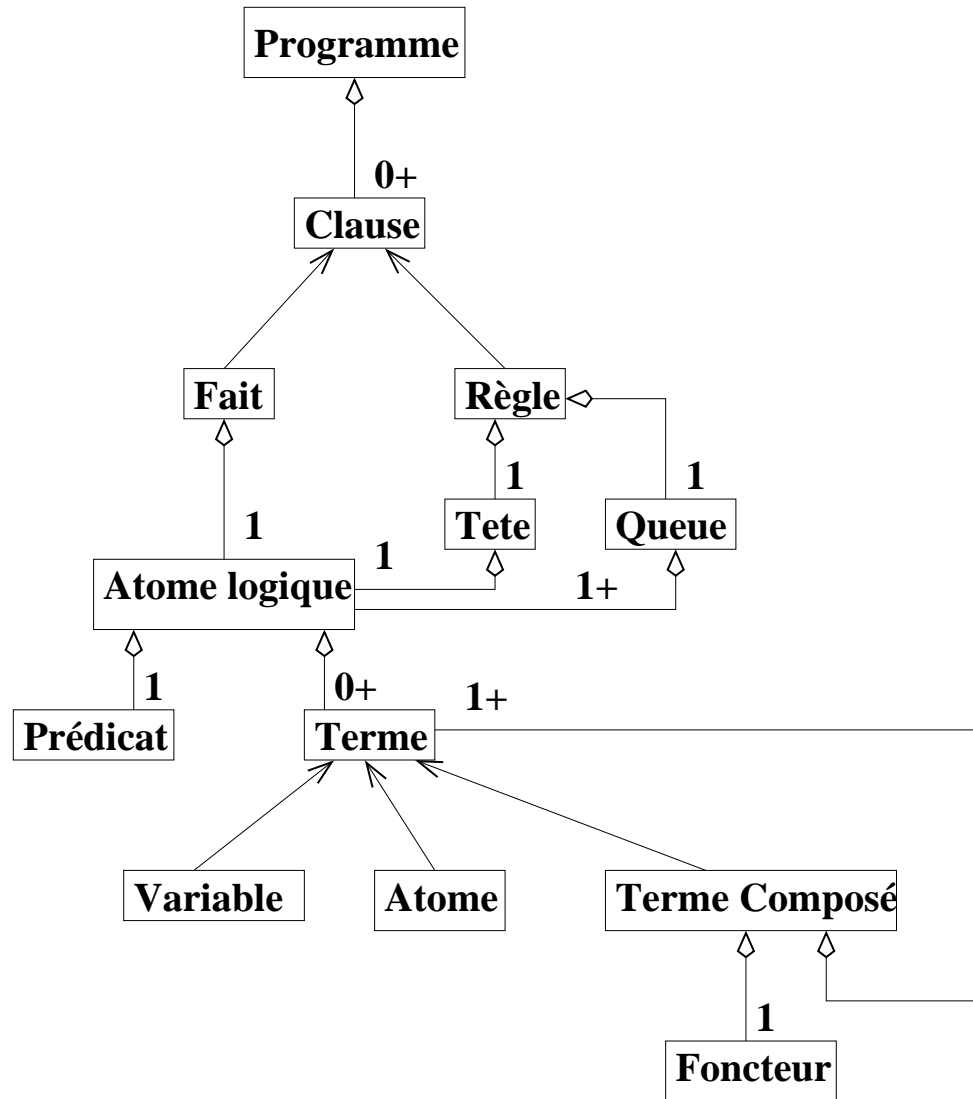
Programmation logique

- Déclaration : formule logique.
- Exemples :
 - homme(socrate)
 $\forall x, \text{homme}(x) \Rightarrow \text{mortel}(x)$
 - entier(0)
 $\forall x, \text{entier}(x) \Rightarrow \text{entier}(s(x))$

Prolog

- Prolog est un langage de programmation logique avec
 - un moteur d'inférence particulier (résolution)
 - des règles particulières (clauses de Horn)
- Calculabilité : aussi puissant que la machine de Turing (ou le lambda-calcul).
- Avantages : rapidité et simplicité pour écrire un programme.
- Inconvénients : temps d'exécution potentiellement très long (si on ne maîtrise pas la façon dont fonctionne le moteur d'inférence).

Syntaxe de Prolog



Syntaxe de Prolog : les termes

- *Terme atomique* \equiv constante (logique des prédicats).
 - Ca peut être un *atome* (symbole), un entier, une chaîne de caractères.
 - Syntaxe d'un atome : suite de lettres, de chiffres et de `_` qui commence par une minuscule ou `_`.
 - Ex : `dupont`, `marseille`.
- *Variable* = inconnue (idem logique des prédicats).
 - Syntaxe : une variable est une suite des lettres et de chiffres qui commence par une majuscule.
 - Ex : `Nom`, `Ville`.
- *Terme composé* \equiv fonction (logique des prédicats). Il s'agit d'un terme composé de plusieurs autres termes.
 - Constitué d'un *foncteur* qui porte sur des termes.
 - Syntaxe : un terme composé est un identifiant suivi d'une suite de termes séparés par des virgules et encadrés par des parenthèses.
 - Ex : `nom(jean, dupont)`, `personne(nom(jeanne, martin), Age)`.

Syntaxe de Prolog : les atomes logiques

- Un *atome logique* comprend un prédicat qui porte sur des termes.
- Il peut être évalué à vrai ou faux (comme dans la logique des prédicats).
- Syntaxe Prolog : la même que pour les termes composés.
- Ex : `entier(s(X))`

Syntaxe de Prolog : les clauses

- Une *clause* correspond à une **clause de Horn** en logique des prédicats (c-à-d une disjonction d'atomes logiques dont un seul maximum peut être positif).
 - Une clause $p_1 \vee \neg p_2 \dots \vee \neg p_n$ peut se réécrire $p_2 \wedge p_3 \wedge \dots \wedge p_n \Rightarrow p_1$.
 - En Prolog, on l'écrira sous la forme:
 $p_1 :- p_2, \dots, p_n .$ où les p_i sont des atomes logiques de Prolog.
- p_1 est la *tête de la clause*.
- $p_2, \dots, p_n .$ est la *queue de la clause*.
- S'il n'y a qu'un seul atome logique (positif) alors la clause est un fait et s'écrit: $p_1 .$

Syntaxe de Prolog : les clauses

- L'apparition d'une variable dans un terme implique qu'elle est quantifiée universellement dans la clause.

- Ex: `beau_parent(X, Y) :- parent(X, Z), maries(Z, Y).`

équivalent à :

$$\forall X, Y, Z : \text{parent}(X, Z) \wedge \text{maries}(Z, Y) \Rightarrow \text{beau_parent}(X, Y)$$

Requête

- Etant donné un programme Prolog (un ensemble de clauses), on peut obtenir un résultat à partir d'une *requête* formulée sous la forme d'une **conjonction d'atomes logiques** $\boxed{?- p_1, \dots, p_n.}$.
- On demande à Prolog : "*Peut-on instancier les variables avec des termes tels que la requête devient une conséquence logique du programme ?*"
- Ex : $\boxed{?- \text{etudiant}(E), \text{date}(D), \text{appris}(E, \text{coursProlog}, D).}$
- A interpréter comme :
 $\exists E, D : \text{etudiant}(E) \wedge \text{date}(D) \wedge \text{appris}(E, \text{coursProlog}, D) ?$
- Si pas de variable : le moteur va simplement indiquer si la requête est ou pas une conséquence logique.

Exemple de requêtes

Avec le programme:

```
homme(socrate).
```

```
mortel(X) :- homme(X).
```

• les requêtes `?- homme(socrate).` ou

`?- mortel(socrate).` retournent `yes`.

• les requêtes `?- homme(X).` ou `?-mortel(X).`

retournent `X=socrate`.

• les requêtes `?- homme(aristote).`,

`?- mortel(platon).` ou `?- philosophe(X).`

retournent `no`.

Sémantique dénotationnelle de Prolog

- Sémantique dénotationnelle = ensemble des faits induits par le programme (via les faits donnés et les règles).
- Remarque : le nombre de faits peut être infini.
Ex : entier(0), entier(s(0)), entier(s(s(0))), etc.
- Un nouveau fait peut s'obtenir par **unification** et **déduction**.

Substitution

- Une *substitution* σ est une fonction de l'ensemble des variables vers l'ensemble des termes.

Ex : $\{ X \rightarrow a, Y \rightarrow f(a,Z), U \rightarrow V \}$.

- On peut étendre l'application t^σ d'une substitution σ à un terme quelconque t ainsi :

- si t est une variable, on applique normalement σ .
- si t est un terme atomique alors $t^\sigma = t$.
- si t est un terme composé $f(t_1, \dots, t_n)$ alors $t^\sigma = f(t_1^\sigma, \dots, t_n^\sigma)$.

Ex : $f(b, X, Y, U)^\sigma = f(b, a, f(a, Z), V)$.

- t^σ est appelé *instance* de t .
- t_1 est *plus général* que t_2 s'il existe une substitution telle que t_2 est l'instance de t_1 .

Unification

- Un *unificateur* σ de deux termes t_1 et t_2 est une substitution telle que $t_1^\sigma = t_2^\sigma$.
- Ex : $\{X \rightarrow h(U), Y \rightarrow b, V \rightarrow a\}$ est un unificateur des termes $f(X, g(a, Y))$ et $f(h(U), g(V, b))$.
- Un *unificateur le plus général* σ de deux termes t_1 et t_2 est un unificateur de t_1 et t_2 telle que tout autre unificateur ρ de t_1 et t_2 est tel que t_1^σ est plus général que t_1^ρ .

L'algorithme de Robinson

```
fonction upg( $t_1, t_2$ )
  si  $t_1 = t_2$  alors retourne  $\emptyset$ 
  si  $t_2$  est une variable alors permuter  $t_1$  et  $t_2$ 
  si  $t_1$  est une variable alors
    si  $t_1$  est un sous-terme de  $t_2$ 
      retourne ECHEC
    sinon
      retourne  $\{t_1 \rightarrow t_2\}$ 
  sinon
    si  $t_1 = f(u_1, \dots, u_n)$  et  $t_2 = f(v_1, \dots, v_n)$ 
       $\sigma \rightarrow \emptyset$ 
      pour  $i$  de 1 a  $n$ 
         $\rho \rightarrow \text{upg}(u_i^\sigma, v_i^\sigma)$ 
        si  $\rho = \text{ECHEC}$  alors retourne ECHEC
         $\sigma \rightarrow \rho \circ \sigma$ 
      sinon
        retourne ECHEC
```

Dénotation d'un programme

- La *dénotation* d'un programme Prolog est l'ensemble des atomes logiques qui peuvent être déduits par une suite d'applications des règles à partir de ses faits.
- Plus précisément :
 - l'ensemble des faits donnés par le programme.
 - si a_1, \dots, a_n font partie de la dénotation, que la règle $c :- b_1, \dots, b_n$ appartient au programme et que σ est le plus grand unificateur de a_1 avec b_1 , a_2 avec b_2 , ..., a_n avec b_n alors c^σ appartient à la dénotation du programme.

Exemple

- Le programme

```
homme(socrate).
```

```
mortel(X) :- homme(X).
```

a pour dénotation

```
{homme(socrate), mortel(socrate)}
```

- Le programme

```
entier(0).
```

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

a pour dénotation

```
{entier(0), entier(s(0)), entier(s(s(0))), ...}
```

Dénotation et requête

- Poser une requête r_1, \dots, r_k à un programme Prolog revient à demander s'il existe un unificateur (le plus général) σ tel que r_1^σ, \dots et r_k^σ font partie de la dénotation du programme (et quel est cet unificateur).
- la requête `?- entier(s(X))`.
a pour résultat `X = 0`
car `entier(s(0))` fait partie de la dénotation du programme.

Sémantique opérationnelle

- Il existe a priori plusieurs façons de déterminer si une requête est déduisible d'un programme Prolog et plusieurs substitutions valides.
- Le moteur d'inférence de Prolog ne fonctionne pas par *chaînage avant* (déduction récursive de faits à partir d'autres faits) mais par *chaînage arrière* (à partir des faits à obtenir, recherche récursive des faits qui permettent de les déduire).

Principe de résolution de Prolog

- On crée la résolvente (suite de buts) $\langle r_1, \dots, r_k \rangle$.
- On essaie de réduire la résolvente à la clause vide \square .
- Pour ceci, on modifie la résolvente courante ainsi : on choisit une clause $t :- q_1, \dots, q_n$ dont la tête t s'unifie avec r_1 grâce à l'unificateur σ et on obtient la résolvente $\langle q_1^\sigma, \dots, q_n^\sigma, r_2^\sigma, \dots, r_k^\sigma \rangle$.
- On continue jusqu'à rencontrer un échec (aucune clause ne peut effacer un but) ou obtenir \square .
- En cas d'échec, on revient sur le dernier choix de clause.
- Les clauses sont choisies dans l'ordre donné par le programme.

Exemple de Résolution

```
triangle(ac, ab, bc).
triangle(ac, ad, cd).
egaux(ac, ab).
egaux(ab, bc).
egaux(ac, bc).
egaux(ac, ad).
angle_droit(ac, ad).
triangle_rectangle(t(C1, C2, C3)) :-
    triangle(C1, C2, C3), angle_droit(C1, C2).
triangle_rectangle(t(C1, C2, C3)) :-
    triangle(C1, C2, C3), angle_droit(C2, C3).
triangle_rectangle(t(C1, C2, C3)) :-
    triangle(C1, C2, C3), angle_droit(C1, C3).
triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C1, C2).
triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C2, C3).
triangle_isocele(t(C1, C2, C3)) :- triangle(C1, C2, C3), egaux(C1, C3).
triangle_equilateral(t(C1, C2, C3)) :-
    triangle(C1, C2, C3), egaux(C1, C2), egaux(C2, C3), egaux(C1, C3).
demi_carre(T) :- triangle_rectangle(T), triangle_isocele(T).
```


Exemple de Résolution

La requête `?- demi_carre(X).` donne la trace de résolution suivante :

- `<demi_carre(X)>`
- `<triangle_rectangle(T), triangle_isocele(T)>` avec $\{X \rightarrow T\}$
- `<triangle(C1, C2, C3), angle_droit(C1, C2), triangle_isocele(t(C1, C2, C3))>` avec $\{T \rightarrow t(C1, C2, C3)\}$
- `<angle_droit(ac, ab), triangle_isocele(t(ac, ab, bc))>` avec $\{C1 \rightarrow ac, C2 \rightarrow ab, C3 \rightarrow bc\}$
- **Echec et retour arrière sur le choix de la clause effaçant `triangle(C1, C2, C3)`**
- `<angle_droit(ac, ad), triangle_isocele(t(ac, ad, cd))>` avec $\{C1 \rightarrow ac, C2 \rightarrow ad, C3 \rightarrow cd\}$
- `<triangle_isocele(t(ac, ad, cd))>`
- `<triangle(ac, ad, cd), egaux(ac, ad)>`
- `<egaux(ac, cd)>`
- `<>`

et on retourne l'instanciation $T=t(ac, ad, cd)$.

Ordre des clauses et des règles

- L'ordre des atomes logiques dans une règle ou dans une requête a de l'importance.

- Ex : il ne faut pas écrire

```
chemin(X, Y) :- chemin(X, Z), arc(Z, Y).
```

qui boucle infiniment mais

```
chemin(X, Y) :- arc(X, Z), chemin(Z, Y).
```

- L'ordre des règles dans le programme a de l'importance.
- Ex : le programme suivant boucle indéfiniment si on lui

fait la requête `?- egaux(ab, ac).` :

```
egaux(X, Y) :- egaux(Y, X).
```

```
egaux(ac, ab).
```

Mais si on intervertit les deux clauses, il donne la réponse `yes`.