

# *Programmation logique*

Principal représentant: Prolog (1972) Basé sur la logique des prédicats

Particularités:

- Unification et filtrage
- Retour arrière
- Déclaratif

Extensions: programmation par contraintes

*Relation  $n$ -aire* sur les ensembles  $X_1, \dots, X_n$ : ensemble des tuples de la forme  $\langle x_1, \dots, x_n \rangle$  où  $x_i \in X_i$ .

*Fonction de  $X_1$  vers  $X_2$* : relation binaire qui met en relation au plus 1 élément de  $X_2$  avec chaque élément de  $X_1$ .

Les langages fonctionnels manipulent les relations qui sont des fonctions

Les langages logiques peuvent manipuler n'importe quelle relation

Avantage: permet d'aller "dans les 2 sens"

## *Exemple fruité*

```
genre(pomme, fruit).
```

```
genre(asperge, legume).
```

```
genre(mangue, fruit).
```

```
fruit(X) :- genre(X, fruit).
```

```
both(X) :- genre(X, fruit), genre(X, legume).
```

```
compl(X, Y) :- genre(X, legume), genre(Y, fruit).
```

```
compl(X, Y) :- genre(X, fruit), genre(Y, legume).
```

## Questions-réponses

```
| ?- genre(X, fruit).
```

```
X = pomme ? ;
```

```
X = mangue ? ;
```

```
no
```

```
| ?- genre(pomme, X).
```

```
X = fruit ? ;
```

```
no
```

```
| ?- compl(X, Y), genre(Y, legume).
```

```
X = pomme, Y = asperge ? ;
```

```
X = mangue, Y = asperge ? ;
```

```
no
```

```
| ?- compl(X, X).
```

```
X = tomate ???
```

Les paires s'écrivent:  $[X | Y]$

Les listes s'écrivent:  $[X_1, X_2, \dots, X_n] \equiv [X_1 | [X_2 | [\dots[X_N | []]]]]$

Ou ensemble:  $[X_1, \dots, X_n | Y] \equiv [X_1 | [\dots[X_n | Y]]]$

```
append([], ZS, ZS).
```

```
append([X|XS], YS, [X|ZS]) :- append(XS, YS, ZS).
```

```
| ?- append([X, Y], Z, [1, 2, 3]).
```

```
X = 1, Y = 2, Z = [3] ? ;
```

```
no
```

```
| ? append(X, Y, [1, 2, 3]).
```

# Modèle d'exécution

Programme = ensemble de relations (faits ou règles) + requête

Exécution d'un programme = recherche d'une *preuve* de la requête, étant données les relations existantes

*Prouve*(requête) =

1. Chercher séquentiellement un fait ou une règle applicable.  
Si rien n'est trouvé  $\Rightarrow$  échec.
2. Instancier les variables
3. Pour chaque prémisse: *Prouve*(prémisse)  
Si une des preuves échoue, retourner au point 1.

# Notation logique

$$\frac{}{\text{append}([], ZS, ZS)} \quad \frac{\text{append}(XS, YS, ZS)}{\text{append}([X|XS], YS, [X|ZS])}$$

Cette notation se prête à l'écriture d'arbres de preuve:

$$\frac{\frac{\frac{Z = [3]}{\text{append}([], Z, [3 | []])} \quad Y = 2}{\text{append}([Y | []], Z, [2 | [3 | []]])} \quad X = 1}{\text{append}([X | [Y | []]], Z, [1 | [2 | [3 | []]])}$$

En plus de l'arbre de dérivation final, il y a aussi les échecs

# Notation logique

$$\frac{}{\text{append}([], ZS, ZS)} \qquad \frac{\text{append}(XS, YS, ZS)}{\text{append}([X|XS], YS, [X|ZS])}$$

Cette notation se prête mieux à l'écriture d'arbres de preuve

$$\frac{\frac{\cancel{[Y|[]]} = [] \quad \frac{Z = [3]}{\text{append}([], Z, [3|[]])} \quad Y = 2}{\text{append}([Y|[]], Z, [2|[3|[]]])} \quad X = 1}{\text{append}([X|[Y|[]]], Z, [1|[2|[3|[]]])}}$$

Il est souvent nécessaire d'utiliser plusieurs arbres



La méthode de recherche de preuve est naïve.

```
soit (X, _, X) .
```

```
soit (X, X, _) .
```

```
| ?- soit (X, 1, 1) .
```

```
X = 1 ? ;
```

```
X = 1 ? ;
```

```
no
```

Cela peut mener à des inefficacités et des erreurs

## *Trouver un chemin*

```
lien(a,b) . lien(a,c) . lien(c,d) . lien(c,e) . lien(d,e)
```

```
chemin(Z,Z) .
```

```
chemin(X,Y) :- lien(X,I) , chemin(I,Y) .
```

Requête: chemin(a,e) .

## *Non-terminaison*

Les deux programmes ci-dessous ne sont pas équivalents:

```
chemin(Z, Z) .
```

```
chemin(X, Y) :- lien(X, I), chemin(I, Y) .
```

et

```
chemin(Z, Z) .
```

```
chemin(X, Y) :- chemin(I, Y), lien(X, I) .
```

Requête: `chemin(a, b)` .

$$\langle \text{terme} \rangle ::= \langle \text{variable} \rangle \mid \langle \text{nombre} \rangle \mid \langle \text{symbole} \rangle \\ \mid \langle \text{symbole} \rangle (\langle \text{terme} \rangle \{ , \langle \text{terme} \rangle \})$$

*Terme clos*: terme qui ne contient pas de variable

Un terme non-clos  $T$  désigne un ensemble infini de termes clos (les *instances* de  $T$ ) que l'on peut obtenir en substituant les variables par chacun des termes clos possibles

*Substitution*: liste de couples variable/terme

Prolog utilise l'*unification* pour vérifier que deux termes sont égaux

C'est aussi ainsi que sont passés les paramètres

*Unificateur*: une substitution qui rend deux termes égaux

Exemples:  $\{X = 1, Y = 1\}$  et  $\{X = Y\}$  sont des unificateurs de  $a(X, Y)$  et  $a(Y, X)$

*Unificateur le plus général*: unificateur de  $T1$  et  $T2$  tel que le terme généré  $T$  désigne toutes les instances communes de  $T1$  et  $T2$

# Unification simple

`egal (X, X) .`

`f (X, Y, Z) :- egal (X, Y), egal (Z, 1) .`

`f (X, Y, Z) :- egal (Y, Z), egal (Z, 2) .`

La requête  $f(3, A, A)$  essaie:

1. Unifie  $A$  et 3, puis essaie d'unifier  $A$  et 1  $\Rightarrow$  échec.
2. Unifie  $A$  et  $A$  puis  $A$  et 2.

Lors du retour arrière, les unifications sont défaites

# Structures de données

L'unification peut être utilisée aussi bien pour tester le type d'un terme, construire un terme, ou le déconstruire:

```
cons(X, Y, cons(X, Y)).
```

```
| ?- cons(_, _, A).      %teste si A est un cons
| ?- cons(A, B, C).      %C = cons(A, B)
| ?- cons(A, _, C).      %A = car(C)
| ?- cons(_, B, C).      %B = cdr(C)
```

Attention à distinguer les règles et les termes

```
element (X, noeud (X, _, _)) .  
element (X, noeud (C, G, _)) :-  
    X < C, element (X, G) .  
element (X, noeud (C, _, D)) :-  
    X > C, element (X, D) .
```

```
ins (X, vide, noeud (X, vide, vide)) .  
ins (X, noeud (C, G, D), noeud (C, G2, D)) :-  
    X < C, ins (X, G, G2) .  
ins (X, noeud (C, G, D), noeud (C, G, D2)) :-  
    X > C, ins (X, D, D2) .
```



En réalité beaucoup de relations ne fonctionnent que dans certains sens

Le *mode* indique si l'argument peut être utilisé en entrée et/ou sortie:

- + Un argument qui doit être un *terme clos*
- Un argument qui sera généré (doit être une variable)
- ? Cas général

En Prolog, le cas “-” est rare

L'arithmétique en Prolog est *impure*

```
<var> is <expr>
```

<expr> est une expression évaluée de manière “traditionnelle”

<expr> doit être un *terme clos*

## Arithmétique pure

Si on veut définir une arithmétique pure, c'est possible aussi

```
nat(0).
```

```
nat(N) :- nat(N1), N is N1 + 1.
```

```
num_in(MIN, _MAX, MIN).
```

```
num_in(MIN, MAX, N) :- MIN < MAX, MIN1 is MIN  
                        num_in(MIN1, MAX, N).
```

nat et num\_in sont des *générateurs*

Naïvement

```
add(N, M, NM) :- nat(N), nat(M), nat(NM),  
                NM is N + M.
```

## Naïvement

```
add(N, M, NM) :- nat(N), nat(M), nat(NM),  
                NM is N + M.
```

## Moins naïf:

```
add(N, M, NM) :- nat(LIM),  
                num_in(0, LIM, N),  
                num_in(0, LIM, M),  
                num_in(0, LIM, NM),  
                NM is N + M.
```

# Séquencement et imbrication

```
fact 0 = 1
```

```
fact n = n * fact (n - 1)
```



```
fact (0, 1) .
```

```
fact (N, FN) :- add(X, 1, N),  
                fact(X, FX),  
                mult(N, FX, FN) .
```

Prouver une négation est un problème difficile

Il y a un zone grise entre “prouvable que oui” et “prouvable que non”

⇒ Prolog est incapable de prouver l’inexistence d’une solution

À la place, il utilise l’absence (apparente) de solution

Cela s’appelle *négation par l’échec*

# Négation illogique

```
petit(0).
```

```
petit(1).
```

```
petit(2).
```

```
zero(0).
```

```
| ?- petit(X), \+ zero(X).
```

```
X = 1 ? ;
```

```
X = 2 ? ;
```

```
no
```

```
| ?- \+ zero(X), petit(X).
```

```
no
```



## *L'opérateur de coupure*

! est un opérateur impur qui permet d'influencer directement l'algorithme de recherche de Prolog.

```
rel(X1, ..., Xn) :- toto, titi, !, tata, tutu.  
rel(X1, ..., Xn) :- bibi, bobo.
```

Il s'appelle *cut* parce qu'il coupe une partie de l'arbre de recherche: si les prémisses qui suivent ! échouent, alors la requête entière échoue

Pour résoudre les problèmes de cycles infinis, de négation, de redondance, de performance, ...

## *Négation par coupure*

La négation de Prolog utilise en réalité la coupure

On utilise typiquement la forme suivante:

```
notrel (X1, ..., Xn) :- rel (X1, ..., Xn), !, fail.  
notrel (X1, ..., Xn) .
```

## *Exemple de coupures*

```
couleur(bleu) .  
couleur(orange) .  
couleur(chocolat) .
```

```
aliment(pomme) .  
aliment(orange) .  
aliment(chocolat) .
```

```
deuxsens1(X) :- couleur(X), aliment(X) .  
deuxsens2(X) :- couleur(X), !, aliment(X) .  
deuxsens3(X) :- couleur(X), aliment(X), !.
```

## *Coupures vertes et rouges*

```
% sans opérateur de coupure
```

```
element (X, noeud (X, _, _)) .
```

```
element (X, noeud (C, G, _)) :- X < C, element (X, G) .
```

```
element (X, noeud (C, _, D)) :- X > C, element (X, D) .
```

```
% avec opérateur de coupure
```

```
element (X, noeud (X, _, _)) :- ! .
```

```
element (X, noeud (C, G, _)) :- X < C, !, element (X, G) .
```

```
element (X, noeud (C, _, D)) :- element (X, D) .
```

Création de cycles lors de l'unification:

$$X = cons(1, X)$$

rejeté par l'unification traditionnelle grâce au *occurs check*

Prolog l'accepte pour accélérer le calcul

On peut utiliser `unify_with_occurs_check(T1, T2)`

## *Effets de bord en Prolog*

Les effets de bord sont ajoutés naïvement:

```
natpr(0).
```

```
natpr(N) :- natpr(N1), write(N1), N is N1 + 1.
```

```
| ?- natpr(5).
```

```
001012012301234
```

## *Mélanger les styles*

Programmation fonctionnelle en Prolog: expressions arithmétiques

Programmation impérative en Prolog: pseudo-prédicats *open*, *write*, ...

Programmation logique dans un autre langage: ajout de *variables logiques*, de relations, et d'un solveur.

Peu satisfaisant

# *Programmation logique cachée*

La programmation logique est plus populaire que ne laisse penser la popularité des langages de programmation logique:

- Les *classes de type* en Haskell

Déclarations d'instances sont des règles de programmation logique

- Les *templates* en C++

Les déclarations de *template* correspondent à une règle

La recherche d'instance faite pendant la compilation (ou la vérification de types) correspond à la “recherche de preuve” de Prolog



# ***Curry: programmation logique fonctionnelle***

---

Curry est un mélange de Haskell et Prolog

Syntaxe et système de type inspirés de Haskell

Tentative d'*unifier* les deux styles de programmation

Les prédicats sont vus comme des fonctions booléennes

Les fonctions peuvent renvoyer plusieurs réponses

*Résiduation* par évaluation paresseuse

## *Curry: non-déterminisme*

Le filtrage ne s'arrête pas au premier succès

```
choice x y = x
```

```
choice x y = y
```

Bien sûr, il faut pouvoir refuser un résultat

# Curry: contraintes

Un nouveau type *Success*

Sorte de *Bool* qui n'accepte pas *False*

Contrainte de base:  $(= : =) :: \alpha \rightarrow \alpha \rightarrow \text{Success}$

Combinateurs:

$(\&) :: \text{Success} \rightarrow \text{Success} \rightarrow \text{Success}$

$(\&>) :: \text{Success} \rightarrow \alpha \rightarrow \alpha$

## *Curry: recherche d'instantiation de variables*

```
prelude> X && (Y || (not X))  
X = True, Y = True => True ? ;  
X = True, Y = False => False ? ;  
X = False, Y = _ => False ? ;  
no more  
prelude>
```

Au lieu d'unification, utilise le narrowing

# *Conclusion*

---

Non-déterminisme

Contraintes

Évaluation à l'envers

Modes

Variables logiques

Instantiation par unification, narrowing, ...