

IA02 : Résolution de Problèmes et Programmation Logique

Programmation Logique : introduction à Prolog

Sylvain Lagrue

sylvain.lagrue@hds.utc.fr

À propos de ce document...

Information	Valeur
Auteur	Sylvain Lagrue (sylvain.lagrue@utc.fr)
Licence	Creative Common CC BY-SA 3.0
Version document	1.3.1

Des coquilles ? sylvain.lagrue@utc.fr ou sur le [forum du cours moodle](#)

I. Introduction

Objectifs du cours

- aborder un nouveau paradigme de programmation (programmation logique et « déclarative »)
- savoir modéliser et résoudre des problèmes simples en Prolog
- connaître les avantages et les limites du langage

“ (D'après Wikipédia) Un paradigme de programmation est une façon d'approcher la programmation informatique et de traiter les solutions aux problèmes et leur formulation dans un langage de programmation approprié.

I. Introduction

Qu'est-ce que Prolog ?

- langage de programmation (à haut niveau d'abstraction)
- basé sur la logique du premier ordre (**Programmation logique**)
- langage interprété (des compilateurs existent)
- orienté requête
- utilisation intensive de la récursivité et de l'unification

I. Introduction

Historique

- 1972 : PROLOG I (Alain Colmerauer et Philippe Roussel)
- 1974 : sémantique de Prolog + restriction aux clauses de Horn (Robert Kowalski et Marteen van Emden)
- 1977 : premier compilateur Prolog (David Warren), syntaxe dite *Edinburgh*
- 1977 : création de Datalog (fragment de Prolog pour les bases de données déductives)
- 1979 : Prolog II



Alain Colmerauer (1941-2017) - crédits : [Wikipédia](#)

I. Introduction

Historique (suite)

- 1983 : création de la machine virtuelle Prolog (la Warren Abstract Machine) et explosion des dialectes...
- 1987 : création d'un groupe de normalisation ISO
- 1989 : Prolog III (ajout d'éléments de programmation par contraintes)
- 1995 : [ISO prolog](#)
- 1996 : Prolog IV (PrologIA)
- 1999 : « naissance » de l'*Answer Set Programming* (ASP)

De nombreux détails sur : <http://www.softwarepreservation.org/projects/prolog>

I. Introduction



De nombreuses implémentations

Commerciales

- B-Prolog : <http://www.probp.com>
- SICStus-Prolog : <https://sicstus.sics.se/>
- ...

Libres

- GNU Prolog : <http://www.gprolog.org/>
- YAP-Prolog : <https://github.com/vscosta/yap-6.3>
- SWI-Prolog : <https://www.swi-prolog.org/>
- Stryer Prolog : <https://github.com/mthom/stryer-prolog>
- ...

Comparatif : https://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations

I. Introduction



Prolog aujourd'hui...

On retrouve la plupart des concepts de prolog dans :

- méthodes de filtrage, de pattern matching et de destruction
- le paradigme de la programmation par contraintes (CP)
- le paradigme de l'Answer Set Programming (ASP)
- les bases de données déductives (Datalog)
- le web sémantique (SPARQL, OWL, etc.)
- le langage de programmation [Erlang](#)
- ...

I. Introduction



Bases théoriques de Prolog

- la logique du premier ordre
- les clauses de Horn
- le principe d'unification (et de plus petit unificateur)
- le principe de résolution
- l'univers de Herbrand
- l'hypothèse du monde clos
- la négation par l'échec

I. Introduction



Exemple de programme

Les règles

```
% Les Schtroumpfs sont bleus
bleu(X) :- schtroumpf(X).

% Les Schtroumpfs sont des lutins
lutin(X) :- schtroumpf(X).

% Les lutins sont petits
petit(X) :- lutin(X).

% Les Schtroumpfs sont tous amis entre eux
ami(X, Y) :- schtroumpf(X), schtroumpf(Y), X \= Y.
```

Les faits

```
% quelques Schtroumpfs
schtroumpf(grand_schtroumpf).
schtroumpf(coquet).
schtroumpf(costaud).
schtroumpf(a_lunette).
schtroumpf(schtroumpfette).
```

I. Introduction



Les requêtes

```
% le Grand Schtroumpf est-il petit ?
?- petit(grand_schtroumpf).
true.
```

```
% Azraël est-il un Schtroumpf ?
?- schtroumpf(azrael).
false.
```

```
% trouve-t-on un ami du Grand Schtroumpf...
?- ami(X, grand_schtroumpf).
X = coquet.
```

```
% quels sont les amis du Grand Schtroumpf ?
?- ami(X, grand_schtroumpf).
X = coquet ;
X = grognon ;
X = costaud ;
X = a_lunette ;
X = schtroumpfette.
```

I. Introduction



```
% donne-moi tous les lutins qui sont ami d'un petit être
% ainsi que le nom de ce petit être...
?- ami(X,Y), lutin(X), petit(Y).
X = grand_schtroumpf,
Y = coquet ;
X = grand_schtroumpf,
Y = grognon ;
X = grand_schtroumpf,
Y = costaud ;
X = grand_schtroumpf,
Y = a_lunette ;
X = grand_schtroumpf,
Y = schtroumpfette ;
X = coquet,
Y = grand_schtroumpf ;
X = coquet,
Y = grognon ;
(...)
X = schtroumpfette,
Y = grand_schtroumpf ;
X = schtroumpfette,
Y = coquet ;
X = schtroumpfette,
Y = grognon ;
X = schtroumpfette,
Y = costaud ;
X = schtroumpfette,
Y = a_lunette.
```

II. Le langage



Commentaires

```
% sur une ligne
/* potentiellement sur plusieurs lignes... */
```

II. Le langage



Les types de base (termes)

- les *atomes*
 - commencent par une minuscule (de préférence) ou entre simples *quotes* (**toto**, **naVariable**, **'Bizarre, mais fonctionne'**)
- les *nombres*
 - entiers (ISO : **12**, **-4** ; SWI : **100_000_000**), peuvent être non bornés
 - les rationnels (**1/2**, **-9/12**, **13/36**)
 - flottants (**-12.94854**, **-0.8761e12**), format IEEE-64 bits
- les *variables*
 - commencent par une majuscule (de préférence) ou par un **_** (**X**, **_na_variable**)
 - une variable spéciale : **_** (variable anonyme, chaque instance dans une règle est considérée comme une variable différente)

II. Le langage



- les *formes fonctionnelles* (*termes composés*)
 - ils sont composés d'un nom (un atome, le foncteur) et de n arguments (il est donc d'arité n)
 - les prédicats et les fonctions (**p(X)**, **odd(12)**, **rel(4, X, 12, f(Y))**)
 - les listes (**[1, 2, 3, 4]**, **["ptn", 12, p, X]**)
 - les chaînes de caractères qui sont des listes de caractères (**"ceci est une chaîne..."**)

Quelques prédicats prédéfinis

aton/1 **number/1** **integer/1** **float/1** **functor/3** etc.

II. Le langage



Format d'un programme Prolog

- un programme est un ensemble de prédicats
- un prédicat est défini par un ensemble de clauses
- une clause est soit un fait soit une règle

Remarque importante

Les prédicats doivent être définis dans un **même bloc**, c'est-à-dire les règles et les faits qui ont le même prédicat en tête doivent toutes être consécutives.

(Contre-)exemple

```
adore(X, Y) :- enfant(X), sucrerie(Y).
sucrerie(bonbon).
enfant(ernest).

adore(X, Y) :- geek(X), oeuvre_de_sciences_fiction(Y).
% Warning + potentielle ERREUR !!
```

II. Le langage



La bonne façon de faire

```
% début bloc
adore(X, Y) :- enfant(X), sucrerie(Y).
adore(X, Y) :- geek(X), oeuvre_de_sciences_fiction(Y).
% fin bloc

sucrerie(bonbon).
enfant(ernest).

% OK
```

II. Le langage



Format d'une règle Prolog

```
H :- B1, B2, ..., Bn.
```

- **H** est la tête de la règle
- **B1, B2, ..., Bn** est le corps de la règle

Ce qui correspond à la formule logique :

$$B_1 \wedge B_2 \wedge \dots \wedge B_n \rightarrow H$$

Mise sous forme de clause :

$$\neg B_1 \vee \neg B_2 \vee \dots \vee \neg B_n \vee H$$

C'est une clause de Horn (c.-à-d. une clause contenant au plus un terme positif)

II. Le langage



Format d'un fait en Prolog

H.

C'est une règle sans corps, équivalente à :

H :- true.

Ce qui correspond à la formule logique :

$$\top \rightarrow H$$

Sous forme de clause :

$$\perp \vee H$$

Ce qui est équivalent à la clause unitaire positive :

$$H$$

C'est aussi une clause de Horn.

II. Le langage



Format d'une requête

Q1, Q2, Q3, ..., Qn.

Ce qui correspond à la formule logique :

$$Q_1 \wedge Q_2 \wedge Q_3 \wedge \dots \wedge Q_n$$

Si on considère sa négation :

$$\neg(Q_1 \wedge Q_2 \wedge Q_3 \wedge \dots \wedge Q_n)$$

$$\neg Q_1 \vee \neg Q_2 \vee \dots \vee \neg Q_n$$

On obtient ainsi une clause de Horn négative (ne comportant que des termes négatifs), appelée également *clause but*.

II. Le langage



Disjonction

L'opérateur ; permet de modéliser un *ou logique* dans le corps des règles.

p(X) :- q(X) ; r(X).

Correspondance logique :

$$\begin{aligned} q(x) \vee r(x) &\rightarrow p(x) \\ \equiv \neg(q(x) \vee r(x)) \vee p(x) \\ \equiv (\neg q(x) \wedge \neg r(x)) \vee p(x) \\ \equiv (\neg q(x) \vee p(x)) \wedge (\neg r(x) \vee p(x)) \\ \equiv (q(x) \rightarrow p(x)) \wedge (r(x) \rightarrow p(x)) \end{aligned}$$

II. Le langage



p(X) :- q(X) ; r(X).

est donc équivalent à

p(X) :- q(X).
p(X) :- r(X).

Remarques

- ; ne peut pas apparaître que dans le corps d'une règle (sinon ce n'est plus une clause de Horn).
- Attention à la lisibilité : **sauf exception, on privilégiera le découpage en 2 règles.**

II. Le langage



Tableau d'équivalence opérateur logique du premier ordre

nom	écriture logique	écriture Prolog
et	\wedge	,
implique	\rightarrow	:-
non	\neg	n'existe pas
ou	\vee	;
tautologie	\top	true
contradiction	\perp	false ou fail
négation par l'échec	n'existe pas	\+ (not/1 est déprécié)

II. Le langage



Quantification

Les règles sont sous forme de Skolem : elles sont toutes quantifiées universellement.

adore(X, Y) :- enfant(X), sucrerie(Y).
sucrerie(bonbon).
enfant(ernest).

est équivalent à la formule suivante :

$$\forall x \forall y (enfant(x) \wedge sucrerie(y) \rightarrow adore(x, y)) \\ \wedge sucrerie(bonbon) \wedge enfant(ernest)$$

III. Unification et résolution



Unification

Règles d'unification en Prolog :

- 2 atomes s'unifient s'ils sont identiques
- 2 nombres s'unifient s'ils sont les mêmes valeurs (et les mêmes types)
- 2 variables non unifiées à une valeur s'unifient toujours
- 1 variable non unifiée et 1 terme s'unifient toujours ; la variable prend la valeur du terme
- une variable unifiée est traitée comme la valeur avec laquelle elle est unifiée
- 2 formes fonctionnelles s'unifient si elles ont le même foncteur, la même arité et si leurs arguments s'unifient récursivement 2 à 2 en partant de la gauche

Toutes les autres tentatives d'unification échouent.

On peut trouver l'algorithme ISO à l'adresse suivante : <https://www.deransart.fr/prolog/unification.html>

III. Unification et résolution



Opérateurs et unification

- le prédicat infix binaire `=` renvoie vrai *ssi* il réussit à unifier ses arguments

```
?- X = 12.  
true.
```

- le prédicat prédéfini `\=` renvoie vrai *ssi* ses 2 arguments ne peuvent pas s'unifier

```
?- douze \= 12.  
true.
```

```
?- Douze \= 12.  
false.
```

Quelques remarques

1. L'opérateur `=` n'est pas une affectation (par exemple on peut écrire `12 = X`)
2. Le prédicat `\=` peut être utilisé pour représenter le concept de différent au sens large (\neq), même si on peut lui préférer le prédicat `dif/2`
3. quand cela est possible, on préférera fusionner des variables plutôt que d'utiliser `X = Y` (surtout dans les requêtes)

III. Unification et résolution



plutôt que :

```
?- ant(X, Y), schtroumpf(Z), schtroumpf(T), X = Z, Y = T.
```

on préférera :

```
?- ant(X, Y), schtroumpf(X), schtroumpf(Y).
```

et plutôt que :

```
aine(X, Y) :- schtroumpf(X), Y = schtroumpfette.
```

on préférera :

```
aine(X, schtroumpfette) :- schtroumpf(X).
```

III. Unification et résolution



mini-exercice

```
?- X = YZ.
```

```
true.
```

```
?- 12 = 12.0.
```

```
false.
```

```
?- [1, 2, 3] = [1, 2, 3].
```

```
true.
```

```
?- "Totor le castor" = "Totor le castor".
```

```
false.
```

III. Unification et résolution



Exercice

On désire modéliser une famille en ne considérant les prédicats `pere/2` et `mere/2`, écrire en Prolog :

1. un prédicat `grand-pere-paternel/2`
2. un prédicat `grand-pere/2`
3. un prédicat `aieul/2`

III. Unification et résolution



Solution

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

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

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

III. Unification et résolution



La résolution en Prolog (SLD-resolution)

Présentation

- Selective Linear Definite clause resolution
- Proposée par Robert Kowalski et Marteen van Emdem
- Dans le cas pour les clauses de Horn : algorithme complet (pour la réfutation) et adéquat (cf. Foundations of Inductive Logic Programming, ch. 7, LNCS 1228 Springer, 1997)

Principe

- On prend la négation des requêtes et ainsi obtenir des clauses de Horn négatives qui composent les buts à atteindre
- On cherche à obtenir la contradiction
- Recherche en profondeur d'abord par chaînage arrière
- *Backtrack* (retour arrière) en cas d'échec ou de demande d'une autre solution
- Les choix font toujours dans l'ordre d'écriture des clauses

III. Unification et résolution



Algorithme simplifié en pseudo-code

```
procedure prouver(buts: liste d'atomes logique,  $\sigma$ : une substitution)
début
  si buts = [] alors
    /* le but est prouvé */
    afficher  $\sigma$ 
    demander si continuation
    si non continuation alors
      exit
    sinon
      buts = [B1, B2, ..., Br]
      pour chaque clause Cl = H :- B'1, B'2, ..., B'n
         $\sigma'$  := unifie(B1, H)
        si  $\sigma'$  n'échoue pas
           $\sigma$  :=  $\sigma \circ \sigma'$ 
          prouver([ $\sigma$ (B'1),  $\sigma$ (B'2), ...,  $\sigma$ (B'n),  $\sigma$ (B2),  $\sigma'$ (B3),...,  $\sigma'$ (Br)],  $\sigma$ )
fin
```

III. Unification et résolution



Exemple

Programme

```
% R1
ami(X, Y) :- schtroumpf(X), schtroumpf(Y), dif(X, Y).

% R2
schtroumpf(grand_schtroumpf).

% R3
schtroumpf(coquet).

% R4
schtroumpf(costaud).
```

Requête

?- ami(grand_schtroumpf, Y).

Ce qui revient à la clause but :

$\neg \text{ami}(\text{grand_schtroumpf}, y)$

III. Unification et résolution



- (R1) $\text{ami}(X, Y) :- s(X), s(Y), \text{dif}(X, Y).$
- (R2) $s(g_s).$
- (R3) $s(coq).$
- (R4) $s(cos).$

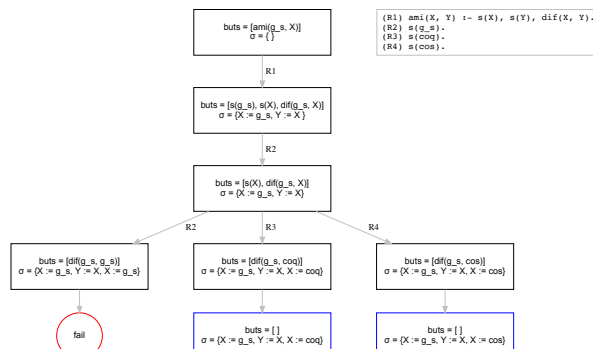
prouver([ami(g_s, X)], { })

- Buts = [ami(g_s, X)]; $\sigma = \{ \}$
 - (R1) Buts = [s(g_s), s(X), dif(g_s, X)]; $\sigma = \{ X := g_s, Y := X \}$
 - (R2) Buts = [s(X), dif(g_s, X)]; $\sigma = \{ X := g_s, Y := X \}$
 - (R2) Buts = [s(g_s), dif(g_s, g_s)]; $\sigma = \{ X := g_s, Y := X, X := g_s \}$
 - *failure* \Rightarrow backtrack
 - (R3) Buts = [dif(g_s, coq)]; $\sigma = \{ X := g_s, Y := X, X := g_s_coq \}$
 - Buts = []; $\sigma = \{ X := Y, X := g_s, X := coq \}$

III. Unification et résolution



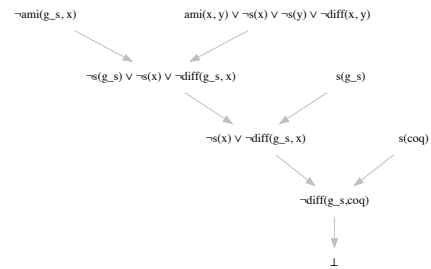
Arbre d'exécution



III. Unification et résolution



Arbre de réfutation (pour le cas X = coq)



III. Unification et résolution



Remarques

- La stratégie de recherche n'est pas directement complète : on peut avoir une suite infinie d'appels récurifs
- La stratégie de recherche dépend de :
 - l'ordre de définition des clauses dans un bloc (on considère les clauses selon leur ordre d'apparition dans le bloc)
 - l'ordre des termes dans le corps d'une clause (on prouve les atomes logiques selon leur ordre d'apparition dans la clause)

Prolog est sensible à la syntaxe, ce qui l'éloigne un peu de l'idéal purement déclaratif.

III. Unification et résolution



La négation par l'échec

- La négation logique **n'existe pas** en Prolog ; seule existe la négation par l'échec
- On la note $\backslash +$ (l'ancienne forme **not** est dépréciée car source de confusion)
- $\backslash + F$ est vrai s'il n'est pas possible de déduire **F** (si la démonstration de **F** échoue)
- Liens théoriques forts avec l'hypothèse du monde clos et la complétion de Clark, le raisonnement non-monotone, la logique des défauts de Reiter, la *circumscription* de McCarthy et enfin la théorie des modèles stables
- Généralisation de l'algorithme de résolution à la *SLDNF* (Selective Linear Definite clause resolution with Negation as Failure)

III. Unification et résolution



Exemple

```
% Les schtroumps (dont on n'arrive pas à déduire qu'ils sont noirs) sont bleus
bleu(X) :- schtroumpf(X), \+ noir(X).
```

```
% Les schtroumpfs infectés sont noirs
noir(X) :- infecté(X), schtroumpf(X).
```

```
schtroumpf(grand_schtroumpf).
schtroumpf(grognon).
schtroumpf(schtroumpfette).
infecté(grognon).
```

```
?- bleu(X). % avec la négation logique, on n'obtiendrait rien !
X = grand_schtroumpf ;
X = schtroumpfette.

?- noir(X).
X = grognon.
```

N.B. : La logique du premier ordre est monotone : si $H_1, \dots, H_n \models F$ alors $H_1, \dots, H_n, G \models F$

Ce n'est pas le cas avec la négation par l'échec : si on ajoute `infecté(grand_schtroumpf)`, on ne déduira plus `bleu(grand_schtroumpf)` !

IV. Les listes



Principe

- Ensembles ordonnés d'éléments
- Les éléments peuvent être de types différents
- Composées d'un tête et d'un reste
- La liste vide est notée `[]`
- Exemple de liste : `[A, 1, [2, 3], f(X)]`
- Les **constructeurs de liste** (*cons*) sont les formes fonctionnelles :
 - `.` (historique)
 - `[|]` (à utiliser préférentiellement)
- le premier élément est la tête de la liste, le second le reste de la liste

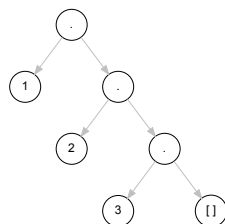
IV. Les listes



Exemple

La liste `[1, 2, 3]` correspond aux formes fonctionnelles `.(1, .(2, .(3, [])))` et `[1 | [2 | [3 | []]]]`

Sous forme d'arbre (*peigne*) :



IV. Les listes



Listes et unification

N.B. : On trouve les notions équivalentes de filtrage, destructuration, etc.

```
% unification de tous les éléments de la liste
?- [A, B, C] = [1, 2, 3].
```

```
A = 1,
B = 2,
C = 3.
```

```
% unification de la tête et du reste
?- [T|R] = [1, 2, 3].
```

```
T = 1
R = [2, 3]
```

```
% unification avec la liste vide
?- [T|R] = [].
false.
```

IV. Les listes



```
% unification de la tête seulement
?- [T|_] = [1, 2, 3].
T = 1.
```

```
% unification avec une liste d'un seul élément
?- [T|R] = [1].
T = 1.
R = [].
```

```
% unification avec les 2 premiers éléments de la liste et du reste
?- [T1, T2|R] = [1, 2, 3].
T1 = 1,
T2 = 2,
R = [3].
```

IV. Les listes



Exemple : un prédicat de concaténation

Question : Définir un prédicat `concat(L1, L2, L3)` qui concatène les listes `L1` et `L2` et qui unifie le résultat avec `L3`. On utilisera pour cela l'unification et la récursivité.

Idee : pour concaténer `L1` et `L2`, il faut prendre le reste de la liste `L1`, le concaténer avec `L2` et ajouter la tête de `L1` au résultat.

```
% cas de base (pour explication)
concat(L1, L2, L3) :- L1 = [], L3 = L2.
```

```
% en beaucoup plus élégant...
concat([], L, L).
```

```
% cas général (pour explication)
concat(L1, L2, L3) :-
    [T|R] = L1,
    concat(R, L2, LTmp),
    L3 = [T|LTmp].
```

```
% cas général beaucoup plus élégant :
concat([T|R], L2, [T|LTmp]) :- concat(R, L2, LTmp).
```

IV. Les listes



Exemples d'utilisation

```
?- concat([1, 2], [3, 4, 5], L).
L = [1, 2, 3, 4, 5].
```

```
?- concat([1, 2], [3, 4, 5], [1, 2, 3, 4, 5]).
true.
```

```
?- concat([1, 2], L, [1, 2, 3, 4, 5]).
L = [3, 4, 5].
```

```
?- concat(L, [3, 4, 5], [1, 2, 3, 4, 5]).
L = [1, 2].
```

IV. Les listes



```
?- concat(L1, L2, [1, 2, 3, 4, 5]).
L1 = [],
L2 = [1, 2, 3, 4, 5] ;
```

```
L1 = [1],
L2 = [2, 3, 4, 5] ;
```

```
L1 = [1, 2],
L2 = [3, 4, 5] ;
```

```
L1 = [1, 2, 3],
L2 = [4, 5] ;
```

```
L1 = [1, 2, 3, 4],
L2 = [5] ;
```

```
L1 = [1, 2, 3, 4, 5],
L2 = [] ;
false. % pas d'autre solution
```

⇒ Une même définition aboutit à 5 utilisations différentes !

Remarques et convention pour les signatures des prédicats



Remarques

- En prolog, on ne « renvoie » rien (pas de *return*)
- Mais certains arguments peuvent être utilisés pour « stocker » le résultat

Écriture pour documenter les prédicats (PAS DANS LE CODE)

- `pred/2` signifie que `pred` est un prédicat d'arité 2

Convention de notation pour les signatures dans les documentations

- `+` : la variable doit être instanciée avant l'appel (*in*)
- `-` : la variable sera unifiée à la fin de l'appel (*out*)
- `?` : la variable peut être instanciée avant l'appel (*in/out*)

Exemple : `pred(+X, ?Y)` est également un prédicat d'arité 2, mais son premier argument doit être instancié avant appel et son deuxième sera instancié s'il ne l'a pas été avant.

D'autres notations existent et sont expliquées ici : <https://www.swi-prolog.org/pldoc/man?section=preddesc>

V. L'arithmétique en Prolog



Les prédicats arithmétiques

- Prédicats de comparaison (ISO) : `<` `>` `=<` `>=` `=` `:=` qui correspondent respectivement à `<` `>` `≤` `≥` `=`
- Ne fonctionnent qu'avec des numériques (entiers, rationnels et flottants)
- Doivent être instanciés au moment de l'appel

```
?- 12 := 13.
false.
```

```
?- 12 < 13.
true.
```

```
?- 12 := 12.0.
true.
```

```
?- X = 12, X < 13.
true.
```

```
?- X < 13, X = 12.
% ERROR: Arguments are not sufficiently instantiated
```


V. L'arithmétique en Prolog



La forme `is`

- Permet d'unifier une variable (à gauche) avec l'expression arithmétique à droite (renvoie vrai si tout se passe bien)
- La partie gauche doit être une variable non-instantiée
- Tous les éléments de la partie droite doivent être instanciés au moment de l'évaluation
- Si la partie gauche est instanciée, alors elle doit être un numérique et avoir la même valeur que la partie de droite

V. L'arithmétique en Prolog



Exemples

```
?- X is 1 + 2.  
X = 3.
```

```
?- 1 + 2 is X.  
% ERROR: Arguments are not sufficiently instantiated
```

```
?- 12 is 11 + 1.  
true.
```

```
?- 11 + 1 is 12.  
false.
```

```
?- X = 11 + 1, Y is X.  
X = 11+1,  
Y = 12.
```

V. L'arithmétique en Prolog



Les fonctions mathématiques

Opérateurs se trouvant à droite de l'opérateur `is` ou comme argument d'un prédicat de comparaison

- constantes : `pi` `e`...
- opérateurs : `-` (unaire) `-` (binaire) `+` `*` `/` `mod` (modulo sur réels/rationnels) `//` (division entière) `rem` (reste de la division entière) `^` (puissance)...
- fonctions : `min` `max` `abs` `round` `cos` `sin` `acos`...

Plus de précisions : <https://www.swi-prolog.org/pldoc/man?section=arith>

V. L'arithmétique en Prolog



Exemples de définitions de prédicats arithmétiques

Le prédicat incrément: `inc(+X, ?Y)`

```
inc(X, Y) :- Y is X + 1.
```

```
?- inc(12, X).  
X = 13.
```

```
?- inc(1, 2).  
true.
```

```
?- inc(X, 12).  
% ERROR: Arguments are not sufficiently instantiated
```

V. L'arithmétique en Prolog



Le prédicat factoriel `fac(+N, ?Res)`

```
fac(0, 1).  
fac(N, Res) :-  
    N > 0,  
    N2 is N - 1,  
    fac(N2, Res2),  
    Res is N * Res2.
```

```
?- fac(5, X).  
X = 120.
```

```
?- fac(5, 120).  
true.
```

N. B. : la condition `N > 0` est nécessaire pour ne pas avoir une boucle infinie...

VI. Concepts avancés



Prédicats de control de l'exécution

Le prédicat prédéfini ! (`cut`)

Utilité

- Contrôler le *backtrack*
- S'arrêter à la première solution trouvée
- Limiter le non-déterminisme de Prolog
- Augmenter les performances (en temps et en mémoire)

Problèmes

- Fait perdre sa forme déclarative à Prolog
- N'assure plus la sémantique de la résolution (*red cut* vs. *green cut*)

VI. Concepts avancés



Principe du prédicat !

- Prédicat qui est toujours vrai
- Coupe toutes les clauses alternative en dessous de lui
- Coupe toutes les solutions alternatives des sous-buts à gauche *cut*
- Possibilité de retour arrière sur les sous-buts à la droite du *cut*

Syntaxe

```
p(X) :- q(X), !, r(X).  
p(X) :- s(X).
```

VI. Concepts avancés



Exemple

```
ami(X, Y) :- schtroumpf(X), schtroumpf(Y), dif(X, Y).  
ami(johan, pirlouit).  
ami(pirlouit, johan).  
  
schtroumpf(grand_schtroumpf).  
schtroumpf(coquet).  
schtroumpf(costaud).
```

```
?- ami(X, Y).  
X = grand_schtroumpf,  
Y = coquet ;  
X = grand_schtroumpf,  
Y = costaud ;  
X = coquet,  
Y = grand_schtroumpf ;  
X = coquet,  
Y = costaud ;  
X = costaud,  
Y = grand_schtroumpf ;  
X = costaud,  
Y = coquet ;  
X = johan,  
Y = pirlouit ;  
X = pirlouit,  
Y = johan.
```

VI. Concepts avancés



Exemple (suite)

```
ami2(X, Y) :- schtroumpf(X), schtroumpf(Y), dif(X, Y), !.  
ami2(johan, pirlouit).  
ami2(pirlouit, johan).  
  
schtroumpf(grand_schtroumpf).  
schtroumpf(coquet).  
schtroumpf(costaud).
```

```
?- ami2(X, Y).  
X = grand_schtroumpf,  
Y = coquet.
```

VI. Concepts avancés



Exemple (suite et fin)

```
ami3(X, Y) :- schtroumpf(X), !, schtroumpf(Y), dif(X, Y).  
ami3(johan, pirlouit).  
ami3(pirlouit, johan).  
  
schtroumpf(grand_schtroumpf).  
schtroumpf(coquet).  
schtroumpf(costaud).
```

```
?- ami3(X, Y).  
X = grand_schtroumpf,  
Y = coquet ;  
X = grand_schtroumpf,  
Y = costaud.
```

VI. Concepts avancés



Réécriture de factoriel avec un cut

```
fac(0, 1) :- !.  
fac(N, Res) :-  
  N2 is N - 1,  
  fac(N2, Res2),  
  Res is N * Res2.
```

Pour s'avancer en TD...



- Définir un prédicat **empty(+L)** qui est vrai seulement si son argument est une liste vide
- Définir un prédicat **head(+L, X)** qui unifie X avec la tête de la liste L

```
empty(X) :- X = [].
```

```
% version plus élégante  
empty([]).
```

```
head1(L, X) :- L = [_|_], X = _.
```

```
% comme R n'est pas utilisée...  
head2(L, X) :- L = [_|_], X = _.
```

```
% version BEAUCOUP plus élégante  
head3([_|_], _).
```

VII. Conclusion



Prolog

- Langage précurseur
- Paradigme de la programmation logique
- Paradigme de la programmation déclarative
- Les écritures peuvent être très élégantes...
- Le ! et la sensibilité à l'ordre des clauses et des termes mettent à mal le côté déclaratif du langage

La programmation logique aujourd'hui

- Le web sémantique et les logiques de description
- SPARQL
- La programmation par contrainte
- L'Answer Set Programming



Annexe : prédicats complémentaires



pas au programme

Annexe : prédicats complémentaires



pas au programme

Modification dynamique de la base de prédicats

- ajout en début/fin de base : **asserta(+Term)** et **assertz(+Term)**
- suppression : **retract(+Term)** et **retractall(+Head)**

```
?- assertz(ami(pirlouit, grand_schtroumpf)).
true.

?- ami(pirlouit, X).
X = johan ;
X = grand_schtroumpf.
```

Annexe : prédicats complémentaires



pas au programme

Les entrées/sorties

- <https://www.swi-prolog.org/pldoc/man?section=IO>
- **read/1** **write/1** **nl/1** **tab/1**

```
?- read(X), write("Bonjour "), write(X), nl.
|: "Sylvain".
Bonjour Sylvain
X = "Sylvain".
```

Annexe : prédicats complémentaires



pas au programme

Autres prédicats de contrôle

- <https://www.swi-prolog.org/pldoc/man?section=control>
- **repeat**
- **->** et ***->**

```
boucle_menu :-
    repeat,
    menu(C),
    (C = 0 ->
        (write("Au revoir..."), !)
        ;
        (write("Alors on continue !"), nl)).

menu(Choix) :-
    nl, write("Quitter (o/n): "),
    read(Choix), nl.
```

Annexe : prédicats complémentaires



pas au programme

Trouver toutes les solutions d'un but

- <https://www.swi-prolog.org/pldoc/man?section=allsolutions>
- `findall(+Template, :Goal, -Bag)`
- `bagof(+Template, :Goal, -Bag)`
- `setof(+Template, +Goal, -Set)`

```
?- findall(X, ami(X,Y), L).
L = [grand_schtroumpf, grand_schtroumpf, coquet, coquet, costaud, costaud, johan, pirlouit].

?- bagof(X, ami(grand_schtroumpf,X), L).
L = [coquet, costaud].
```