Cours de Programmation Déclarative et Bases de Données

Prolog - introduction

Nicolas Jouandeau

n@up8.edu

2022

programmation logique et Prolog

- idée abstraite de Bob Kowalski dans les années 70
- développé en tant que langage de programmation en 1975
  par Alain Colmerauer et Philippe Roussel
  (Univ. Marseille, Campus Lumini)
- sur le principe de résolution de la logique du 1er ordre
  (variables, prédicats (i.e. relations), connecteurs logiques (et, ou),
  quantificateurs ("Pour tout" universel, "Il existe" existanciel))
- déduction = une forme de calcul
- un programme logique = un ensemble fini de faits et de règles
- un programme = description du résultat sans le moyen d'y arriver
  (i.e. un programme déclaratif)
- concis = permet d'écrire des programmes avec moins de lignes que
  d'autres langages
- flexible = modifier le programme est possible en ajoutant une règle

repose sur l'hypothèse du monde clos

- tout ce qui est vrai est connu
- ce qui est faux n'est pas mentionné
- tout est faux jusqu'à preuve du contraire

permet d'énoncer des problèmes sous forme logique

- un programme est un ensemble de déclarations
  (*i.e.* un ensemble de disjointions de clauses)
  exemple avec 4 clauses : $\mathtt{Q} \vee (\mathtt{S} \Leftarrow \mathtt{Q} \wedge \mathtt{P}) \vee (\mathtt{S} \Leftarrow \mathtt{Q} \wedge \mathtt{R}) \vee (\bot \Leftarrow \mathtt{P}' \wedge \mathtt{Q})$
- les clauses sont des faits ou des règles de déduction
- par résolution logique, on obtient une réponse à une question
- une question peut demander
  - la vérification d'un fait (ex: $\mathtt{Q}$ est il vrai?)
  - s'il existe un élément qui satisfait une propriété énoncée
    (ex: ce qui satisfait la propriété $\mathtt{S}$ est vrai)

la programmation logique est fondée sur la théorie des clauses de Horn

- possède une résolution en temps linéaire
- est Turing complète

clauses de Horn

- un littéral = 1 formule atomique ou sa négation
- un littéral positif = une proposition ($\mathtt{A}$)
- un littéral négatif = la négation d'une proposition ($\neg\mathtt{A}$)
- une clause = 1 disjonction de littéraux
- une clause de Horn = si au + 1 littéral est positif
  exemple avec 4 clauses : $\{\{\mathtt{Q}\},\{\neg\mathtt{Q},\neg\mathtt{P},\mathtt{S}\},\ \{\neg\mathtt{Q},\neg\mathtt{R},\mathtt{S}\},\ \{\neg\mathtt{P}',\neg\mathtt{Q}\}\}$
  par réécriture
  - $\{\neg\mathtt{Q},\neg\mathtt{P},\mathtt{S}\}$ équivaut à $\mathtt{S}\vee\neg\mathtt{Q}\vee\neg\mathtt{P}=\mathtt{S}\vee\neg(\mathtt{Q}\wedge\mathtt{P})$ équivaut à $\mathtt{S}\Leftarrow\mathtt{Q}\wedge\mathtt{P}$
  - $\{\neg\mathtt{P}',\neg\mathtt{Q}\}=\{\neg\mathtt{P}',\neg\mathtt{Q},\bot\}$ équivaut à $\bot\vee\neg\mathtt{P}'\vee\neg\mathtt{Q}$ équivaut à $\bot\Leftarrow\mathtt{P}'\wedge\mathtt{Q}$

  $\mathtt{Q}\vee\ (\mathtt{S}\Leftarrow\mathtt{Q}\wedge\mathtt{P})\vee\ (\mathtt{S}\Leftarrow\mathtt{Q}\wedge\mathtt{R})\vee\ (\bot\Leftarrow\mathtt{P}'\wedge\mathtt{Q})$

programmation déclarative

- déclare une situation correspondant au problème à résoudre
- aucune instruction à exécuter
- est compatible avec la structuration objet (Prolog++, LogTalk, ...)
- à la différence de
  - la programmation impérative
    (proche du fonctionnement des ordinateurs)
  - la programmation fonctionnelle
    (composition de fonctions et récursivité, pas de machine à états)

swi-prolog

- implémentation du langage Prolog (langage logique non typé)
- installation : `sudo apt-get install swi-prolog`
- `https://www.swi-prolog.org/`

syntaxe

- des termes, des faits, des règles et des questions
- objets manipulés sont définis par des termes
  (atomes, nombres, variables, structures)
- atome
  - une chaîne de caractères commençant par une minuscule
    `un  deUX  jouer_Aux_Echecs`
  - une chaîne de caractères arbitraire entre quotes
    `'Blind Chess'  '?!&ŝ&#$ &'`
- nombre (entier ou réel ou symbole prédéfini ou booléen)
  `1  2.0  pi  true  false`
- variable
  - une chaîne de caractères commençant par une majuscule ou _
    `Un  _deUX`
  - une variable nommée _ est anonyme (i.e. non utilisable)
- structure (un nom, des parenthèses et des termes)
  - une chaîne de caractères commençant par une minuscule
    `doublet(1,2)  triplet(1,2.0,'hello')`

syntaxe (suite)

- atomes + nombres = constantes
- constantes + variables = termes simples
- structures = termes complexes
- prédicat
  - un nom et une arité (i.e. un nombre d'arguments)
    `proche(A,B)` est d'arité 2
    `proche/2` indique que l'on parle de ce prédicat
  - un prédicat peut être unaire
    `magicien(mario)`
  - un prédicat peut être d'arité zéro
    `pluie` ou `pluie()`
  - si deux prédicats ont le même nom et des arités différentes, alors ce sont deux prédicats différents
    `ami(mario,luigi)`
    `ami(mario)`
    `ami/2` signifie "est l'ami de"
    `ami/1` signifie "est un ami"

syntaxe (fin)

- on termine un fait ou une règle par un "`.`"
- fait = prédicat vrai
- règle
  - est strictement une clause de Horn composée de prédicats
  - déclare le comportement du programme
  - l'ordre des prédicats est déterminant
    - `sauter :- carapace.`
      se lit `sauter` si `carapace`
    - `sauter :- carapace,yoshi.`
      si `carapace` puis `yoshi`.
    - `appliquer_f(A,B,C,D) :- C is 1/A, D is A/B.`
      `C` reçoit `1/A` puis `D` reçoit `A/B`
- question
  - commence par "`?-`" (en mode interactif)
  - `appliquer_f/4` est il satisfait pour ces valeurs ?
    `?- appliquer_f(10,20,30,40).`
  - que faut il pour satisfaire `appliquer_f/4` ici ?
    `?- appliquer_f(10,20,A,B).`

remarques

- dans le corps d'une règle, les prédicats sont évalués dans le sens de lecture (i.e. de gauche à droite)
- la portée d'une constante est l'ensemble des règles
- la portée d'une variable est limitée à la règle dans laquelle elle est
- le ET entre prédicats est défini sur une même ligne
- le OU entre règles est défini sur deux lignes ou avec un ";"

```
dir(mario,droite);dir(mario,gauche).
```

est équivalent à

```
dir(mario,droite).
dir(mario,gauche).
```

quantificateurs

- universel : ∀
  - pour tout `x`, `f` est satisfait si `g` et `h` sont satisfaits
    ```
    f(X):-g(X),h(X).
    ```
  - pour tout `a`, `f` n'est pas satisfait
    ```
    not(f(a)).
    ```
- existanciel : ∃
  - il existe un `a` qui satisfait `f`
    ```
    f(a).
    ```

dist.pl

```
dist(paris, marseille, 797).
dist(paris, nantes, 355).
dist(paris, rouen, 122).
dist(paris, metz, 315).
dist(X,Y,D) :- dist(paris,X,DX), dist(paris,Y,DY), D is DX+DY.
```

quelques explications

- les 4 premières lignes du programme sont des faits
- la dernière ligne du programme est une règle
- dans une règle
  - ":-" se lit SI
  - "," se lit ET
  - ";" se lit OU
- la conclusion est en tête (i.e. nommé tête de la règle)
- les conditions sont dans la suite (i.e. nommé corps de la règle)

exécution

```
$> swipl
?- consult(dist).
true.
?- dist(paris,nantes,D).
D = 355 .
?- dist(nantes,marseille,X).
X = 1152 .
?- ^D
$>
```

dist.pl

```
dist(paris, marseille, 797).
dist(paris, nantes, 355).
dist(paris, rouen, 122).
dist(paris, metz, 315).
dist(X,Y,D) :- dist(paris,X,DX), dist(paris,Y,DY), D is DX+DY.
```

quelques explications

- le prédicat `consult/1` charge du code Prolog dans l'interpréteur (`consult(dist).` s'écrit également `[dist].`)
- on peut charger plusieurs codes
- les codes correspondent à une base de faits et de règles
- on peut ensuite poser des questions
- `swipl` utilise la base pour vérifier/démontrer/répondre à une question

dist.pl

```
dist(paris, marseille, 797).
dist(paris, nantes, 355).
dist(paris, rouen, 122).
dist(paris, metz, 315).
dist(X,Y,D) :- dist(paris,X,DX), dist(paris,Y,DY), D is DX+DY.
```

exécution

```
?- dist(nantes,paris,D).
ERROR: Out of local stack
```

ajouter une règle après la dernière ligne

```
dist(X,Y,D) :- dist(Y,X,D).
```

nouvelle exécution

```
?- dist(nantes,paris,D).
ERROR: Out of local stack
Exception: (1,522,927) dist(paris, paris, _2362) ?
```

sortir du traitement de l'erreur

- appuyer sur a

sortir de l'interpréteur Prolog

- utiliser la commande `halt.` ou `ctrl-d`

exécution interactive : visualiser les phases de l'exécution

- usuellement pour concevoir les programmes
  - lancer l'interpréteur Prolog avec la commande `swipl`
  - charger un ou plusieurs codes déclarant des faits et des règles
  - poser des questions
  - sortir de l'interpréteur avec le prédicat `halt/0`
- si plusieurs réponses à une question
  - appuyer sur "`;`" pour avoir la réponse suivante
  - appuyer sur "`entrée`" ou "`.`" pour arrêter la résolution

schéma d'exécution en mode interactif

```
$> swipl
% charger un ou plusieurs fichiers de faits et règles
?- consult(<UN_FICHIER>).
true.
% poser une ou plusieurs questions
?- <PREDICAT>(<AVEC_OU_SANS_ARGUMENTS>).
<UNE_OU_PLUSIEURS_REPONSES>
% sortir de Prolog
?- halt.
$>
```

correction du problème

- on remarque `dist(paris,paris,...`
- le démonstrateur recherche des distances entre `paris` et `paris`
- on ajoute la contrainte `X` différent de `Y` dans `dist(X,Y,D)`

nouveau `dist.pl`

```
dist(paris, marseille, 797).
dist(paris, nantes, 355).
dist(paris, rouen, 122).
dist(paris, metz, 315).
dist(X,Y,D) :- X\=Y, dist(paris,X,DX), dist(paris,Y,DY), \
  D is DX+DY.
```

question de satisfaction d'un fait (réponse vrai/faux)

```
?- dist(paris,nantes,300).
false.
?- dist(paris,nantes,355).
true .
```

question avec une réponse (une valeur par variable)

```
?- dist(paris,nantes,X).
X = 355 .
?- dist(paris,nantes,X).
X = 355 ;
false.
```

question avec plusieurs réponses (n valeurs par variable)

```
?- dist(paris,X,Y).
X = marseille,
Y = 797 ;
X = nantes,
Y = 355 .
```

suivi d'exécution

- mode `trace` permet de suivre pas à pas une exécution
- entrer dans le mode `trace` puis évaluer un prédicat
  `?- trace,<PREDICAT>(<...>).`
- sortir du mode `trace`
  `?- notrace,nodebug.`

dans le mode `trace`

- "`a`" pour abort = fin du calcul et retour à l'interpréteur en mode `trace`
- "`l`" pour leap = fin du calcul et retour à l'interpréteur en mode `debug`
- "`s`" pour skip = exécute l'appel sans tracer la décomposition en profondeur de l'appel courant et passe à l'appel suivant; permet une exécution pas à pas sans aller en profondeur
- "`e`" pour exit = fin du calcul et sortie de l'interpréteur

dans le mode `debug`

- il est possible de définir des points d'arrêt ou d'arrêter le programme sur des modifications de variables, en plaçant des drapeaux

exemple d'ensemble de considérations

- la chèvre est un animal herbivore
- le loup est un animal cruel
- un animal cruel est carnivore
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

exemple de question

- y a t il un animal cruel et que consomme t il ?

identifions termes, faits et règles

les termes

- la chèvre est un animal herbivore
- le loup est un animal cruel
- un animal cruel est carnivore
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

exemple de question

- y a t il un animal cruel et que consomme t il ?

les prédicats définissant les faits

- la chèvre est un animal herbivore
- le loup est un animal cruel
- un animal cruel est carnivore
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

les règles

- la chèvre est un animal herbivore
- le loup est un animal cruel
- un animal cruel est carnivore
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

les règles (suite)

- la chèvre est un animal herbivor
- le loup est un animal cruel
- un animal cruel est carnivore
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
mange(X,viande):-carnivore(X).
mange(X,herbe):-herbivore(X).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

les règles (suite)

- la chèvre est un animal h
- le loup est un animal cru
- un animal cruel est carni
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
mange(X,viande):-carnivore(X).
mange(X,herbe):-herbivore(X).
mange(X,Y):-carnivore(X),herbivore(Y).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

les règles (suite)

- la chèvre est un animal l
- le loup est un animal cru
- un animal cruel est carn
- un animal carnivore mange de la viande
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
mange(X,viande):-carnivore(X).
mange(X,herbe):-herbivore(X).
mange(X,Y):-carnivore(X),herbivore(Y).
boit(X,eau):-carnivore(X);herbivore(X).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

les règles (fin)

- la chèvre est un animal h
- le loup est un animal cru
- un animal cruel est carn
- un animal carnivore mar
- un animal herbivore mange de l'herbe
- un animal carnivore mange des animaux herbivores
- les carnivores et les herbivores boivent de l'eau
- un animal consomme ce qu'il boit ou ce qu'il mange

code Prolog

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
mange(X,viande):-carnivore(X).
mange(X,herbe):-herbivore(X).
mange(X,Y):-carnivore(X),herbivore(Y).
boit(X,eau):-carnivore(X);herbivore(X).
consomme(X,Y):-boit(X,Y);mange(X,Y).
```

exemple de question

- y a t il un animal cruel et que consomme t il ?

herbivores-et-carnivores.pl

```
herbivore(chevre).
cruel(loup).
carnivore(X):-cruel(X).
mange(X,viande):-carnivore(X).
mange(X,herbe):-herbivore(X).
mange(X,Y):-carnivore(X),herbivore(Y).
boit(X,eau):-carnivore(X);herbivore(X).
consomme(X,Y):-boit(X,Y);mange(X,Y).
```

question

```
?- cruel(X),consomme(X,Y).
X = loup,
Y = eau ;
X = loup,
Y = viande ;
X = loup,
Y = chevre.
```

résultat

- le loup est cruel et consomme de l'eau, de la viande et de la chèvre

ajouter des commentaires dans le code Prolog

- sur une ligne, sont précédés de `%`
- sur plusieurs lignes, sont encadrés par `/*...*/`

exemple

```
/* commentaire sur
   plusieurs
   lignes */
f(a).
% commentaire sur une ligne
g(a).
```

afficher des messages pendant l'exécution

- le prédicat `write/1` affiche un message sur `stdout`
- le prédicat `nl/0` affiche un retour à la ligne

exemple

```
f(1).
g(2.0).
h("trois\n").

f1():-f(X),write(X),nl.
g1():-g(X),write(X),nl.
h1():-h(X),write(X).

f2(X):-f(X),write(X),nl.
g2(X):-g(X),write(X).
h2(X):-h(X),write(X),nl.
```

exécution

```
?- f1.
1
true.

?- g1.
2.0
true.

?- h1.
trois
true.
```

exécution

```
?- f2(X).
1
X = 1.

?- g2(X).
2.0
X = 2.0.

?- h2(X).
trois

X = "trois\n".
```

afficher des messages avec un format

- le prédicat `writef/2` affiche un message selon un format
  - le format est composé de deux parties
  - la 1ère partie structure l'affichage
  - la 2ième partie liste les valeurs attendues dans la 1ère partie
  - `%w` affiche l'élément suivant de la liste
  - `[A,B,C]` est la liste des variables `A`, `B`, `C`
  - `[1,2.0,'trois']` est la liste des valeurs `1`, `2.0`, `'trois'`
  - ci-dessous les variables `X`, `Y`, `Z` valent `1`, `2.0`, `'trois'`

exemple

```
f3():-f(X),g(Y),h(Z),writef('... %w %w %w', [X, Y, Z]).
```

exécution

```
?- f3.
... 1 2.0 trois
true.
```

exécuter directement un programme

- ajouter un prédicat `go/0`
- demander à `swipl` d'appeler `go`
- demander à `swipl` de terminer par `halt.`

exemple1.pl

```
f(X):-Y is X+1, writef('(%w %w)\n', [X,Y]).

go:-f(10).
```

exécution

```
$> swipl -s exemple1.pl -g go -t halt
(10 11)
$>
```

exécuter directement un programme avec des arguments

- utiliser le prédicat `current_prolog_flag`
  (qui récupère une liste d'arguments)

exemple2.pl

```
f([X]):-writef('(%w %w %w)\n', [X,X,X]).

go:-current_prolog_flag(argv, Argv),f(Argv).
```

exécution

```
$> swipl -s exemple2.pl -g go -t halt 10
(10 10 10)
$>
```

récupérer toutes les réponses possibles

- utiliser le prédicat `fail/0` (qui poursuit l'évaluation)

exemple3.pl

```
f([X]):-writef('(%w %w)\n', [X,X]),fail.
f([X]):-writef('(%w %w %w)\n', [X,X,X]).

go:-current_prolog_flag(argv, Argv),f(Argv).
```

exécution

```
$> swipl -s exemple3.pl -g go -t halt 10
(10 10)
(10 10 10)
$>
```