Cours 1 : Introduction à la Programmation Logique

March 26, 2024

Séance 1+ et 2, Cours 1 : Introduction à la Programmation Logique

Un plan pour commencer :

1) Histoire et diversité des langages de programmation
2) Démonstration IA de ProLog
3) Structure de ProLog
    - Principe (Delahaye)
    - Lexical
    - Grammaire
    - Unification
    - Exécution
    - Arbre
4) Premiers exemples sur les listes

1 Histoire des langages de programmation

ou brainstorming sur "quels langages de programmation connaissez-vous ? Et quels paradigmes ?"

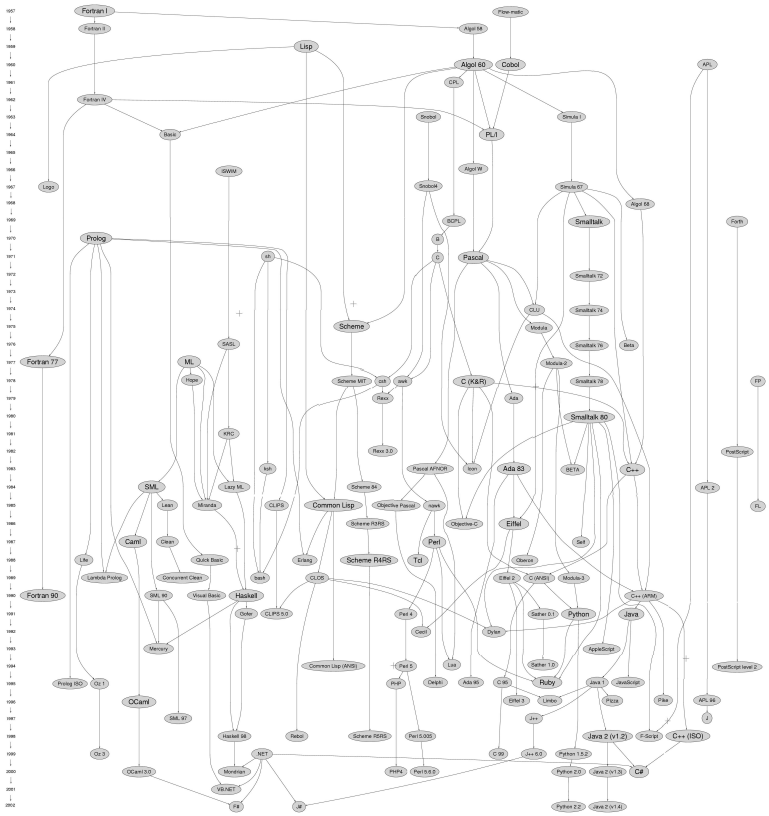
Ceux qui nous intéressent :

- Approche logique
- Approche par contraintes
- Approche parallèle
- Approche fonctionnelle (pure)

Autres perspectives

- Algo vs Données
- Action vs Etat
- IA (symbolique vs statistique) ? Typage ? Performance ?

Pour repérer dans le temps, tous ces paradigmes ou langage, voici une carte des langages trouvée sur le web (il y en a de plus récentes, celle-ci s'arrête en 2002 avant Rust, GoLang, Kotlin,…)



2 Démonstration

avec éléments de syntaxe et principes (délaratifs) de base

A propos de la famille Ingalls

```
       lansford
        /      \
     peter       charles
      / \         |    \   \
  alice  ella   mary laura carrie
    |      |     |     |
  altha   earl  adam  rose
```

voir swish pour une version de secours (interactive aussi)

Sous forme Prolog, cette base de faits sera rédigée de la manière suivante :

[10]:
```
%%writefile parent.pl
parent(lansford,peter).
parent(peter,alice).
parent(alice,altha).
parent(peter,ella).
parent(ella,earl).
parent(lansford,charles).
parent(charles,mary).
parent(mary,adam).
parent(charles,laura).
parent(laura,rose).
parent(charles,carrie).
```

```
Overwriting parent.pl
```

Pour interroger, un petit programme “principal” spécial “parent” :

[11]:
```
%%writefile mainDirect.pl
main :-
  writeln('Qui comme parent ? (charles, mary, ...), suivi d''un point'),
  read(Entree),
  parent(Entree,Sortie),
  write('Enfant : '),
  writeln(Sortie).

:- main.
```

```
Writing mainDirect.pl
```

avec un fichier d’*assemblage* :

[12]:
```
%%writefile pour_progDirect.pl
#include "parent.pl"
#include "mainDirect.pl"
```

```
Writing pour_progDirect.pl
```

Enfin l’assemblage et la requête (à la suite la réponse à l’exécution) :

[13]:
```
!cpp -P -o progDirect.pl pour_progDirect.pl
!echo "mary." | swipl -g halt -s progDirect.pl
```

```
Qui comme parent ? (charles, mary, …), suivi d'un point
Enfant : adam
```

Pour interroger, un petit programme “principal” plus général (le reste est quasi-idem) :

[14]:
```
%%writefile main.pl
main :-
  writeln('Votre requête ?'),
  read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '),
  sort(Resultats,ResultatsTries),
  writeln(ResultatsTries).

:- main.
```

```
Overwriting main.pl
```

[1]:
```
%%writefile pour_prog.pl
#include "parent.pl"
#include "main.pl"
```

```
Overwriting pour_prog.pl
```

[20]:
```
!cpp -P -o prog.pl pour_prog.pl
!echo "parent(mary,adam)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [parent(mary,adam)]
```

Des exemples de requêtes :

[21]:
```
!echo "parent(adam,mary)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : []
```

[22]:
```
!echo "parent(mary,Enfant)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [parent(mary,adam)]
```

[23]:
```
!echo "parent(Parent,mary)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [parent(charles,mary)]
```

On ajoute d'autres règles (grand parent et ancetre) :

[25]:
```
%%writefile grandParent.pl
grandParent(X,Z):-
  parent(X,Y),
  parent(Y,Z).
```

```
Writing grandParent.pl
```

[26]:
```
%%writefile pour_prog.pl
#include "parent.pl"
#include "grandParent.pl"
#include "main.pl"
```

```
Overwriting pour_prog.pl
```

[27]:
```
!cpp -P -o prog.pl pour_prog.pl
!echo "grandParent(charles,PetitEnfant)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [grandParent(charles,adam),grandParent(charles,rose)]
```

[28]:
```
%%writefile ancetre.pl
ancetre(X,Z):-
  parent(X,Y),
  ancetre(Y,Z).
ancetre(X,Z):-
  parent(X,Z).
```

```
Writing ancetre.pl
```

[29]:
```
%%writefile pour_prog.pl
#include "parent.pl"
#include "grandParent.pl"
#include "ancetre.pl"
#include "main.pl"
```

```
Overwriting pour_prog.pl
```

[30]:
```
!cpp -P -o prog.pl pour_prog.pl
!echo "ancetre(charles,EnfantEtCo)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [ancetre(charles,adam),ancetre(charles,carrie),ancetre(cha
rles,laura),ancetre(charles,mary),ancetre(charles,rose)]
```

3 Eléments structurels de la programmation logique

Comment tout cela marche, et comment va-t-on l'utiliser.

Pour commencer, quelques principes directeurs :

3.1 Principes (JP Delahaye) :

1. Enoncer des **Faits**
2. Donner des **règles de raisonnement** se basant sur ces faits pour produire d'autres faits
3. Savoir poser des questions (**requêtes**)

3.2 Eléments lexicaux

- constantes (entier, str, etc.) et identificateurs = commence par une minuscule
- Variables = Commence Par Une Majuscule (_ou _un _souligné "_")
- Listes = liste vide [ ], liste explicite [1, 2, 3] et constructeur de liste [E | L]

Dans les exemples précedents, on a déjà pu voir ces éléments (constantes, Variable), pour les listes et les nombres, voir plus loin (ou à un prochain cours)

3.3 Grammaire

(https://www.complang.tuwien.ac.at/sicstus/sicstus_42.html) :

```
clause --> non-unit-clause | unit-clause
non-unit-clause --> head :- body
unit-clause --> head
head --> [[[[module : head]]]]
| goal
body --> [[[[module : body]]]]
| [...]
| body , body
| goal
goal --> term
term --> functor ( arguments )
| ( subterm )
| { subterm }
| list
| string
| constant
| variable
arguments --> subterm
| subterm , arguments
subterm --> term
list --> []
| [ listexpr ]
listexpr --> subterm
| subterm , listexpr
| subterm | subterm
constant --> atom | number
atom --> name
functor --> name
```

A noter :

- le sous-ensemble présenté s'appelle : pure-prolog (c'est presque sans mot clés !)
- tout ne sera pas utilisé dans **ce cours** (mais ce qui restera suffira pour tout faire, ou presque)
- attention, sur le web, vous trouverez beaucoup plus ; sur le web, on cherche un peu plus d'efficacité que dans ce cours (et moins de concept) ; dans ce cours, on se focalisera sur la déclarativité, les propriétés algorithmiques, ce que les autres langages ne font pas, etc.

3.4 Algorithme d'Unification

(Herbrand algorithm, wwwdh.cs.fau.de/IMMD8/Lectures/LOGIK/isoprolog.pdf)

C'est pour savoir si X=Y (si une expression/un terme ressemble à un/e autre), dans d'autres langage cela s'appelle du matching (franglais). C'est pratique pour le programmeur, sans être essentiel. Pour autant, c'est juste une 1/2 page de code, alors pourquoi s'en priver (?)

Given a set of equations of the form $t_1 = t_2$ apply in any order one of the following non-exclusive steps:

a) If there is an equation of the form:

1) $f = g$ where f and g are different constants, or

2) $f = g$ where f is a constant and g is a compound term, or f is a compound term and g is a constant, or

3) $f(...) = g(...)$ where f and g are different functors, or

4) $f(a_1, a_2, ...a_N) = f(b_1, b_2, ...b_M)$ where N and M are different.

then exit with failure (*not unifiable*).

b) If there is an equation of the form $X = X$, X being a variable, then remove it.

c) If there is an equation of the form $c = c$, c being a constant, then remove it.

d) If there is an equation of the form $f(a_1, a_2, ...a_N) = f(b_1, b_2, ...b_N)$ then replace it by the set of equations $a_i = b_i$.

e) If there is an equation of the form $t = X$, X being a variable and t a non-variable term, then replace it by the equation $X = t$,

f) If there is an equation of the form $X = t$ where:

1) X is a variable and t a term in which the variable X does not occur, and

2) the variable X occurs in some other equation,

then substitute in all other equations every occurrence of the variable X by the term t.

g) If there is an equation of the form $X = t$ such that X is a variable and t is a non-variable term which contains this variable, then exit with failure (*not unifiable, positive occurs-check*).

h) If no transformation can be applied any more, then exit with success ($unifiable$).

3.5 Moteur d'exécution (Deransart)

Là c'est le cœur, mais c'est très court !

Accrochez-vous (et en même temps, cela n'est pas si compliqué, cela devrait bien se passer)

1. Start with a *current goal* which is the initial definite goal G and a *current substitution* which is the empty substitution.
2. If G is `true` then stop (*success*), otherwise
3. Choose a predication `A` in G (*predication-choice*)
4. If `A` is `true`, delete it, and proceed to step (2), otherwise
5. If no freshly renamed clause in P has a head which unifies with `A` then stop (*failure*), otherwise
6. Choose in P a freshly renamed clause `H :- B` whose head unifies with `A` by substitution σ which is the *MGU* of `H` and `A` (*clause-choice*), and
7. Replace in G the predication `A` by the body `B`, flatten and apply the substitution σ to obtain the new current goal, let the new current substitution be the current substitution composed with σ, and proceed to step (2).

A noter :

- les choix (étrange pour une spécification, de laisser cette liberté de choisir)
- en fait tous les choix peuvent être pris mais pas en même temps, il faudra les prendre l'un après l'autre dans un certains ordre
  - le premier : predication-choice (sur le choix du fait à prouver), la stratégie habituelle est de prendre les faits dans l'ordre d'apparition au cours de l'exécution
  - le second : clause-choice (sur la règle de déducion à appliquer), la stratégie habituelle est de prendre les règles dans l'ordre d'apparition dans le fichier où se trouve le programme

3.6 Arbre d'exécution

voir sur un exemple (vidéo, image ou au tableau)

- sur ?- ancetre(charles,X).
- ou sur le prédicat ajout : https://www.youtube.com/watch?v=s74X4cix0NA

4 Premiers programmes

- premier, dernier ; ajouterEnPremier, ajouterEnDernier
- concatener

[31]:
```
%%writefile prog.pl
/* ajouteEnDernier(R,L,E) est vrai ssi R est obtenu à partir de L en ajoutant E␣
 ↪en dernier. */
ajouteEnDernier(E,[],[E]).
ajouteEnDernier(E,[F|L],[F|M]):-ajouteEnDernier(E,L,M).
```

```
/* concatene(D,F,L) est vrai si et seulement la concaténation des listes D et F␣
 ↪donne la liste L. */
concatene([],L,L).
concatene([E|L],M,[E|R]):-concatene(L,M,R).

main :-  writeln('Votre requête ?'), read(Entree),
  findall(Entree,Entree,Resultats),
  write('Liste des résultats : '), writeln(Resultats).
:- main.
```

```
Overwriting prog.pl
```

[32]:
```
!echo "ajouteEnDernier(1,[2,0,2,3],L)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [ajouteEnDernier(1,[2,0,2,3],[2,0,2,3,1])]
```

[33]:
```
!echo "concatene([2,0],[2,4],L)." | swipl -g halt -s prog.pl
```

```
Votre requête ?
Liste des résultats : [concatene([2,0],[2,4],[2,0,2,4])]
```

fin

4.1 Sur Caseine

Prévu 3 pistes **Base de faits** :

- Pour le CM : exercices sur les familles
  - grandParent
  - enfant
  - ancetre
  - generation
- Pour le TD : exercices sur les nombres en chiffres romains
  - plus2
  - moins1
  - plus
  - mult
  - estPair
  - plusGrand
  - max
  - pgcd
- Pour les révisions : exercices sur la géographie en Italie
  - sudEst
  - nord
  - directionSud
  - quartSE