Comment modéliser un problème ?

Les entités, les objets dont on parle: les termes, construits à l'aide des symboles de fonction.

Les relations entre termes: les formules, construites à l'aide des symboles de prédicat.

Exemple du TD/TP 1 : Adam aime les pommes. Clara aime les carottes. Olivier aime les oranges. Les pommes sont des fruits. Les oranges sont des fruits. Les carottes sont des légumes. Ceux qui aiment les fruits sont en bonne santé.

Les "objets" dont on parle (symboles de fonction):

`adam/0, pomme/0, clara/0, carotte/0, olivier/0, orange/0`

Les relations entre objets (symboles de prédicat):

`fruit/1, legume/1, aime/2, bonne_sante/1`

Comment modéliser un problème

Le programme correspondant à ce choix de termes et prédicats:

```
aime(adam,pomme).
aime(clara,carotte).
aime(olivier,orange).
fruit(pomme).
fruit(orange).
legumes(carotte).
bonne_sante(X) :- aime(X,Y), fruit(Y).
```

Un choix alternatif

Les objets dont on parle (symboles de fonction):
adam/0, pomme/0, clara/0, carotte/0, olivier/0, orange/0, fruit/0, legume/0
Les relations entre objets (symboles de prédicat):
aime/2, bonne_sante/1, est_instance_de/2
Le programme correspondant:

```
aime(adam,pommes).
aime(clara,carotte).
aime(olivier,orange).
est_instance_de(pomme,fruit).
est_instance_de(orange,fruit).
est_instance_de(carotte,legumes).
bonne_sante(X) :- aime(X,Y), est_instance_de(Y,fruit).
```

Comment modéliser un problème (2)

Un deuxième exemple, où les termes ne sont pas que des constantes. Voici ce qu'il faut modéliser:

Un *arbre binaire* est soit une *feuille* , soit un *noeud* dont les deux fils sont des arbres binaires.

La *hauteur* d'une feuille est 0, et la hauteur d'un noeud est le max des hauteurs de ses fils plus 1.

La *taille* d'une feuille est 0, et la taille d'un noeud est la somme des tailles de ses fils plus 1.

Comment modéliser un problème (3)

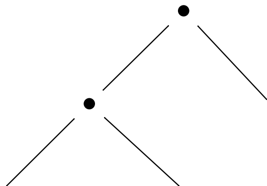
Les symboles de fonction:

`feuille/0, noeud/2`

Un nombre infini de termes. Par exemple

`noeud(noeud(feuille,feuille),feuille)`

qui désigne l'arbre :



Comment modéliser un problème (4)

Les symboles de prédicat:
`hauteur/2, taille/2`
Assertions et règles:

```
hauteur(feuille,0).
hauteur(noeud(T1,T2), R) :- hauteur(T1,H1), hauteur(T2,H2),
                            max(H1,H2,S), R is S+1.


taille(feuille,0).
taille(noeud(T1,T2), R) :- taille(T1,R1), taille(T2,R2),
                           R is R1 + R2 + 1.
```

Comment modéliser un problème (5)

Prolog n'effectue aucun "contrôle de type"; on peut lancer la requête `taille(hauteur(Z,toto),X).`
pour le programme précedent, et il n'y aura aucun message d'erreur (juste une reponse négative).
Nous verrons que dans certain cas il est utile de melanger le niveau des termes et celui des prédicats : des prédicats peuvent avoir d'autres prédicats comme argument.
Pour l'instant, il convient de garder les deux niveaux *strictement* sépares.

Sémantique opérationnelle d'un programme logique

Un *atome* est un symbole de prédicat appliqué à une liste de termes. Il est *clos* s'il ne contient pas de variables non instanciées.
Par exemple `aime(olivier,orange)` est un atome clos, et `connexe(a,X)` n'en est pas un.
Définition (ensemble succes) Soit P un programme logique; *l'ensemble succes* de P, désigné par $[\![P]\!]^{op}$, est l'ensemble de tous les atomes clos a tels que l'arbre de dérivation de racine a possède au moins une branche succes.
Exemple:
L'ensemble succes du programme P suivant:

```
pair(zero).
pair(succ(succ(X)) :- pair(X).
```

est l'ensemble d'atomes:
$[\![P]\!]^{op} = \{\texttt{pair(zero), pair(succ(succ(zero))),....}\}$

Sémantique déclarative d'un programme logique.

Définition:

Soit P un programme logique. La sémantique déclarative de P, désignée par $[\![P]\!]^{dec}$, est l'ensemble des atomes clos qui sont conséquences logiques de P.

Exemple: La sémantique déclarative du programme:

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

est l'ensemble d'atomes:

$[\![P]\!]^{dec} = \{$`homme(socrate), mortel(socrate)` $\}$

Correction et Complétude

Théorème: correction et complétude de la semantique opérationnelle vis à vis de la sémantique déclarative:

Pour tout programme logique P, $[\![P]\!]^{op} = [\![P]\!]^{dec}$.

Problèmes liés à la complétude:

Pour de raisons d'efficacité, l'ensemble des atomes clos sur lesquels l'intrepréteur Prolog termine avec succes pour un programme P, que l'on note ici $[\![P]\!]^{eff}$, est un sous-ensemble de $[\![P]\!]^{op}$: $[\![P]\!]^{eff} \subseteq [\![P]\!]^{op}$,

Exemple: soit P le programme

```
p(X) :- p(X).
p(a).
```

on a: $[\![P]\!]^{op} = [\![P]\!]^{dec} = \{\mathtt{p(a)}\}$

mais $[\![P]\!]^{eff} = \emptyset$

Pourquoi certaines branches de l'arbre de dérivation ne sont pas explorées?

- Non-fairness de la règle de sélection des clauses (comme dans l'exemple précedent).
- Prédicats pré-définis qui élaguent l'arbre de dérivation (typiquement le prédicat de *coupure*).

Conséquence logique ?

Définition: conséquence logique. Une formule F est conséquence logique d'un ensemble de formule S si tout modèle de S est aussi un modèle de F.

Mais qu'est-ce qu'un modèle d'un ensemble de formules?
Mini-cours plus tard: la notion de modèle d'une *théorie du premier ordre*.

Représentation des listes en Prolog

- une liste en Prolog est:
  - soit la constante "liste vide": le terme `[]`
  - soit un terme à deux arguments:
    le premier élément (*head*) et le reste de la liste (*tail*).
    Ce terme s'écrit `[head|tail]`
  - On aurait pu écrire `empty` et list(head,tail)
- `head` est un terme quelconque, `tail` doit être une liste.
  Le listes ne sont pas homogènes:
  `[3|['hello'|[]]]`
  est une liste de longueur 2, dont le premier élément est un entier et le deuxième un string.
- On peut aussi énumérer les éléments d'une liste, séparés par des virgules. La liste précédente peut s'écrire `[3,'hello']`

Représentation des listes (2)

On peut énumérer plusieurs éléments avant le "|".
Par exemple: `[a,b,c] = [a|[b,c]] = [a, b|[c]] = [a,b,c|[]]`

Opérations sur les listes en Prolog

- Appartenance: `member(X,L)` où `X` est un terme et `L` une liste.
- Le but `member(X,L)` réussit si `X` est membre de `L`.
- `member(b,[a, b, c])` et `member([b, c],[a, [b, c]])` sont vrais.
  `member(b,[a, [b,c]])` est faux.
- On peut observer que `X` est membre de `L`, si
  - `X` est le *head* de `L`, ou bien
  - `X` est membre du *tail* de `L`
- En Prolog:

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

Opérations sur les listes (2)

Concaténation:

- `append(L1,L2,L3)` est vrai si et seulement si la liste `L3` est la concatenation de `L1` et `L2`.
- Pour définir `append(L1,L2,L3)` on peut considérer deux cas, en fonction de la forme du premier argument `L1`:
  - Si `L1` est vide, alors `L2` et `L3` doivent être deux listes égales.
  - Si `L1` est non vide, alors `L1` est de la forme `[X|L]`. Dans ce cas concaténer `L1` avec `L2` donne une liste `[X|L3]` où `L3` est la concaténation de `L` et `L2`.
- Cela donne en PROLOG:

```
append([],L,L).
append([X|L],L2,[X|L3]) :- append(L,L2,L3).
```

Opérations sur les listes (3)

Exemples d'utilisation de append:

```
[eclipse 1]:  append([a,b,c],[1,2,3],[a,b,c,1,2,3]).
Yes (0.00s cpu)
[eclipse 2]:  append([a,b,c],[1,2,3],L).
L = [a, b, c, 1, 2, 3]
Yes (0.00s cpu)
[eclipse 3]:  append([a,[b,c],[[]]],[a,[]],L).
L = [a, [b, c], [[]], a, []]
Yes (0.00s cpu)
```

Opérations sur les listes (4)

```
[eclipse 4]:  append(L1,L2,[3,4,b]).
L1 = []
L2 = [3, 4, b]
Yes (0.00s cpu, solution 1, maybe more) ?  ;
L1 = [3]
L2 = [4, b]
Yes (0.00s cpu, solution 2, maybe more) ?  ;
L1 = [3, 4]
L2 = [b]
Yes (0.00s cpu, solution 3, maybe more) ?  ;
L1 = [3, 4, b]
L2 = []
Yes (0.00s cpu, solution 4)
```

On peut redéfinir le prédicat `member` en utilisant append.

```
membre1(X,L) :- append(_,[X|_],L).
```

Opérations sur les listes (5)

Enlever un élément:

- Enlever l'élément X d'une liste L:
  `delete(X,L,L1)`
  où L1 est la liste L sans l'élément X
- ```
  delete(X,[X|Q],Q).
  delete(X,[Y|Q],[Y|Q1]) :- delete(X,Q,Q1).
  ```
- Si X apparaît plusieurs fois dans la liste, des retours en arrière successifs donnent toutes les possibilités.
- `delete(X,L,L1)` échoue si X n'apparaît pas dans L.
  ```

Opérations sur les listes (6)

Exemples d'utilisation de `delete`:

```
[eclipse 5]:  delete(a, L, [c, d, e]).
L = [a, c, d, e]
Yes (0.00s cpu,solution 1,maybe more) ?  ;
L = [c, a, d, e]
Yes (0.00s cpu,solution 2,maybe more) ?  ;
L = [c, d, a, e]
Yes (0.00s cpu,solution 3,maybe more) ?  ;
L = [c, d, e, a]
Yes (0.00s cpu,solution 4)
```

Opérations sur les listes (7)

Insérer un élément dans une liste:

- En utilisant `delete` on peut définir `inserer(X,L,L1)` qui insère l'élément `X` dans la liste `L`, de toute le manières possibles. La liste `L1` est le résultat de l'insertion.
  ```
  inserer(X,L,L1) :- delete(X,L1,L).
  ```
- On peut aussi redéfinir le prédicat `member` en utilisant `delete`. Un élément X est membre d'une liste si on peut l'enlever de cette liste.
  ```
  membre2(X,L) :- delete(X,L,_).
  ```

Opérations sur les listes (8)

Une définition directe de `inserer`:

```
inserer(X,L,[X|L]).
inserer(X,[Y|L],[Y|G]) :- inserer(X,L,G).
[eclipse 3]:  inserer(a,[b,c,d],L).
L = [a, b, c, d]
Yes (0.00s cpu, solution 1, maybe more) ?  ;
L = [b, a, c, d]
Yes (0.00s cpu, solution 2, maybe more) ?  ;
L = [b, c, a, d]
Yes (0.00s cpu, solution 3, maybe more) ?  ;
L = [b, c, d, a]
Yes (0.00s cpu, solution 4)
```

Le module lists de ECLiPSe

```
[eclipse 1]: use_module(library(lists)).
lists.eco loaded in 0.00 seconds
Yes (0.00s cpu)
```

ou bien, dans un fichier source toto.pl

```
:- use_module(library(lists)).
A number of basic list processing predicates (is_list/1,
append/3, member/2, length/2 etc) are available by default
and do not require this library to be loaded.
:- export
checklist/2,
flatten/2,
flatten/3,
collection_to_list/2,
halve/3,
```

```
intersection/3,
maplist/3,
print_list/1,
select/3,
shuffle/2,
splice/3,
subset/2,
union/3.
:- reexport
append/3,
delete/3,
length/2,
member/2,
memberchk/2,
nonmember/2,
subtract/3,
reverse/2
```

Backtrack (retour en arrière)

L'arbre de dérivation est produit de manière *depth first* (profondeur d'abord). Dès qu'il se trouve sur une feuille F, l'interpréteur remonte vers le pêre P de F, et vérifie s'il existe une *nouvelle* manière d'unifier le premier but de P et la tête d'une clause du programme.
Si c'est le cas: on commence à produire une nouvelle branche.
Sinon: on remonte au pêre de P.
On ne trouve pas *une solution*, mais *toutes les solutions* du problème posé par le but à la racine de l'arbre.

Exemple 1: Permutations

```
/* insert(elem, liste_avant_insert, liste_apres_insert) */
insert(X,L,[X|L]).
insert(X,[Y|L],[Y|R]) :- insert(X,L,R).

/* perm(liste_init, liste_permutee) */
perm([],[]).
perm([X|L],R) :- perm(L,S) , insert(X,S,R).
```

Exemple 2: Sélection de k éléments d'une liste

```
/* selection(liste_init, nombre_elts, selection) */
selection(_,0,[]).
selection([X|L],N,[X|R]) :- N>0, P is N-1, selection(L,P,R).
selection([_|L],N,R) :- N>0, selection(L,N,R).
```

Exemple 3: Sous-listes (non vides)

```
/* append(prefixe, suffixe, liste_resultat) */
append([],L,L).
append([X|L],H,[X|R]) :- append(L,H,R).

/* prefixe(prefixe, liste_entiere) */
prefixe(R,L) :- append(R,_,L).

/* sous_liste(sous_liste, liste_entiere) */
sous_liste(R,L):-
    prefixe(P,L),
    prefixe(P1,L),
    append(P,R,P1),
    R\==[].
```

Exemple 4: Fonctions

```
/* fonctions (domaine, codomaine, liste_couples)*/
fonctions([],_,[]).

fonctions([A|L],H,F) :-
    member(B,H),
    fonctions(L,H,F1),
    F = [(A,B)|F1].
```

ou plus court:

```
fonctions([A|L],H,[(A,B)|F1]) :-
    member(B,H),
    fonctions(L,H,F1).
```

Logique du 1er ordre: la syntaxe (1)

Soit V un ensemble dénombrable de symboles de variable (par exemple $V = \{x, y, z, \ldots, x_1, \ldots\}$)
Définition: un alphabet F, P consiste de deux ensembles de symboles: fonctions et prédicat. Les éléments de F et P ont chacun une arité prédéfinie. Les éléments de F d'arité 0 sont les constantes (d'individu), les éléments de P d'arité 0 sont les constantes de prédicat.
Par exemple, un alphabet pour les entiers en notations unaires pourrait être: $F_{int} = \{zero/0, succ/1\}$, $P_{int} = \{pair/1, diviseur/2\}$

Logique du 1er ordre: la syntaxe (2)

Définition: l'ensemble de termes $T_{F,P}$ sur un alphabet F, P donné est défini inductivement par:

- tout élément de V est un terme dans $T_{F,P}$.
- tout élément de F d'arité 0 (donc toute constante) est un terme de $T_{F,P}$
- Si $t_1, \ldots, t_n$ sont des termes de $T_{F,P}$ et $f/n \in F$, alors $f(t_1, \ldots, t_n)$ est un terme de $T_{F,P}$.

Voici quelques termes sur l'alphabet F_{int}, P_{int}:
x, $zero$, $succ(succ(zero))$

Logique du 1er ordre: la syntaxe (3)

Définition: l'ensemble de formules sur un alphabet F, P donné est défini inductivement par:

- Si $p/n \in P$ et $t_1, \ldots, t_n$ sont des termes, alors $p(t_1, ..., t_n)$ est une formule (un atome).
- Si F et G sont des formules, alors $\neg F$, $F \vee G$, $F \wedge G$ $F \rightarrow G$ sont des formules.
- Si $x \in V$ et F est une formule, alors $\forall x\ F$ et $\exists x\ F$ sont des formules.

Voici quelques formules sur l'alphabet F_{int}, P_{int}:
$pair(x)$, $pair(zero) \wedge (\neg pair(succ(zero)))$,
$diviseur(succ(succ(zero)), succ(succ(succ(zero))))$,
$\forall x(pair(x) \rightarrow diviseur(succ(succ(zero)), x))$.

Interprétations (1)

Définition: une intérpretation $[\![\ ]\!]$ d'un langage du premier ordre $L = F, P$ est donnée par:

- un ensemble D (le domaine de l'interprétation).
- Pour tout symbole de fonction f/n de L, une fonction $[\![f]\!] : D^n \to D$.
- Pour tout symbole de prédicat p/n de L, un ensemble $[\![p]\!] \subseteq D^n$.

Trois exemples d'interprétation

Pour le langage des entiers unaires, une interprétation pourrait être:
$D = \mathbb{N}$
$[\![zero]\!]^s = 0$
$[\![succ]\!]^s(n) = n + 1$
$[\![pair]\!]^s = \{n \mid n \text{ est pair}\}$
$[\![diviseur]\!]^s = \{(n, m) \mid n \text{ est un diviseur de } m\}$
celle-ci est l'interprétation standard, mais celle qui suit est aussi une interprétation:
$D = \{0, 1\}$
$[\![zero]\!]^{ns} = 0$
$[\![succ]\!]^{ns}(0) = 1; [\![succ]\!]^{ns}(1) = 1$
$[\![pair]\!]^{ns} = \{0\}$
$[\![diviseur]\!]^{ns} = \{(1, 0)\}$

Interprétation de Herbrand

On peut aussi juste choisir d'interpréter un terme clos comme *lui-même*:

$D = T_{F,P} \setminus V$

$[\![zero]\!]^H = zero$

$[\![succ]\!]^H(x) = succ(x)$

$[\![pair]\!]^H = \{zero, succ(succ(zero)), succ(succ(succ(succ(zero)))), \ldots\}$

$[\![diviseur]\!]^H = \{(s(zero), s(zero)), \ldots\}$

Interprétations (2)

Soit $[\![\ ]\!]$ une interprétation de $L = F, P$, de domaine D. Un environnement est une fonction $\rho : V \to D$

Définition: interprétation des termes Soit $[\![\ ]\!]$ une interprétation de L et $\rho : V \to D$ un environnement. On définit l'intérprétation $[\![\ ]\!]_\rho : Termes \to D$ des termes de L par:

- $[\![x]\!]_\rho = \rho(x)$
- $[\![c]\!]_\rho = [\![c]\!]$, pour $c/0 \in F$.
- $[\![f(t_1, ..., t_n)]\!]_\rho = [\![f]\!]([\![t_1]\!]_\rho, ..., [\![t_n]\!]_\rho)$, pour $f/n \in F$.

Interprétations (3)

Définition: (interprétation des formules) Soit $[\![\ ]\!]$ une interprétation de L et $\rho : V \to D$ un environnement. On définit l'intérprétation $[\![\ ]\!]_\rho : \mathit{Formules} \to \{0, 1\}$ des formules de L par:

- $[\![p(t_1, ..., t_n)]\!]_\rho = \begin{cases} 1 & \text{si } ([\![t_1]\!]_\rho, ..., [\![t_n]\!]_\rho) \in [\![p]\!] \\ 0 & \text{sinon} \end{cases}$
- $[\![\neg A]\!]_\rho$, $[\![A \vee B]\!]_\rho$, $[\![A \wedge B]\!]_\rho$, $[\![A \to B]\!]_\rho$ sont définis en fonction de $[\![A]\!]_\rho$ et $[\![B]\!]_\rho$ avec les tables de verité usuelles.
- $[\![\forall x A]\!]_\rho = 1$ si et seulement si pour tout $d \in D$ $[\![A]\!]_{\rho[x \leftarrow d]} = 1$.
- $[\![\exists x A]\!]_\rho = 1$ si et seulement si il existe $d \in D$ tel que $[\![A]\!]_{\rho[x \leftarrow d]} = 1$.

Modèles

Un modèle d'une formule F est une interprétation $[\![\ ]\!]$ telle que, pour tout ρ, $[\![F]\!]_\rho = 1$.
Un modéle d'un ensemble de formule est une intepprétation qui est un modéle de chaque formule de l'ensemble.

Par exemple, l'interprétation $[\![\ ]\!]^s$ est un modèle de la formule
$\forall x\ pair(x) \rightarrow pair(succ(succ(x)))$
tandis que $[\![\ ]\!]^{ns}$ n'est pas un modèle de cette formule.

Résumé du deuxième cours

- Termes et prédicats en Prolog: choix de modélisation.
- Sémantique de Prolog:
  - Sémantiques opérationnelle, effective et déclarative:
  - Théorème de correction et complétude: $[\![\ ]\!]^{op} = [\![\ ]\!]^{decl}$.
  - $[\![\ ]\!]^{eff} \neq [\![\ ]\!]^{decl}$.
- Les listes en Prolog
- Applications du backtrack