PROgrammation LOGique

- Les origines du Prolog (vers 1970):
  - A. Colmerauer et al., Univ. d'Aix-Marseille.
    Un système de communication homme-machine en français
  - R. A. Kowalski, Imperial College, London.
    Predicate logic as a programming languages
- L'age d'or (1982-1992): projet de recherche "Ordinateurs de 5ème génération" au Japon.
- La Programmation par Contraintes (à partir de la moitié des années 80): extension de Prolog. Applications industrielles et intérêt académique.

Un grand nombre d'implementations disponibles. Voir par exemple: `http://fr.wikipedia.org/wiki/Prolog`

La version que nous allons utiliser: ECLiPSe
`http://eclipseclp.org/`

Principale avantage d'ECLiPSe par rapport à d'autres implémentations libres (GNU prolog, SWIProlog, YAP, ...), pour ce cours: traitement satisfaisant de la Programmation par Contraintes.

The user states the problem,
the computer solves it

Claim: La formulation détaillée d'un problème est *très proche* d'un programme Prolog qui le résout.

Raffinement *top-down*: un exemple

Problème: Trouver le minimum d'une liste d'entiers L.

- Quels sont les entités mises en jeu?
  Les entiers et les listes d'entiers
- Quelle est la relation principale, celle liant le problème à sa solution?
  "L'entier N est le minimum de la liste L"
- Dans l'approche top-down, cette relation est progressivement décomposée en relations plus simples, jusqu'à des relations élémentaires.
- Les relations élémentaires sont typiquement les tests d'égalité, différence ou de $\leq$ entre nombres ou plus généralement entre données simples.

Raffinement *top-down*: un exemple

La relation "L'entier N est le minimum de la liste L" n'est pas élémentaire. Il faut la décomposer en d'autres relations plus simples.

La bonne question à se poser est: *quand est-ce que* `N` *est le minimum de* `L`*?*

Réponse: `N` *est le minimum de* `L` *si* `N` *est un élément de* `L` *et si* `N` *est plus petit que tout élément de* `L`.

On a deux nouvelles relations:
"`N` est un élément de `L`" et "`N` est plus petit que tout élément de `L`"
qui décomposent (définissent) la relation "`N` est le minimum de `L`"

Raffinement *top-down*: un exemple

Les relations "`N` est un élément de `L`" et "`N` est plus petit que tout élément de `L`" ne sont pas élémentaires, il faut à leur tour les décomposer .

La bonne question à se poser pour décomposer la première relation est: *quand est-ce que* `N` *est un élément de* `L`*?*

Réponse : `N` *est un élément de* `L` *si* `N` *est égal à la tête de* `L` *ou si* `N` *est un élément du reste de* `L`*.*

De la même manière, pour la deuxième, on obtient la décomposition:
`N` *est plus petit que tout autre élément de* `L` *si* `N` *est plus petit que la tête de* `L` *et si* `N` *est plus petit que tout autre élément du reste de* `L`*.*
et
tout nombre est plus petit que tout élément de la liste vide.
Les derniers prédicats introduits sont l'égalité et le $\leq$ entre entiers, qui sont élémentaires. Le processus de décomposition est terminé.

Raffinement *top-down*: un exemple

Voici le résultat du raffinement top-down du problème de la recherche du minimum d'une liste:

- `N` *est le minimum de* `L` *si* `N` *est un élément de* `L` *et si* `N` *est plus petit que tout autre élément de* `L`.
- `N` *est un élément de* `L` *si* `N` *est égal à la tête de* `L` *ou si* `N` *est un élément du reste de* `L`.
- *tout nombre est plus petit que tout élément de la liste vide*.
- `N` *est plus petit que tout élément de* `L` *si* `N` *est plus petit que la tête de* `L` *et si* `N` *est plus petit que tout autre élément du reste de* `L`.

aux detailles syntaxiques près, il s'agit d'un programme Prolog qui permet de trouver le minimum d'une liste d'entiers.

Raffinement *top-down*: un exemple

Voici le programme Prolog correspondant:

```
minimum(N,L) :- element(N,L) , plus_petit(N,L).
element(N,[M|L]) :-  N=M ; element(N,L).
plus_petit(_,[]).
plus_petit(N,[M|L]) :- N=<M , plus_petit(N,L).
```

Mais attention, on peut décomposer différemment... (et plus efficacement. Comme toujours, plusieurs algorithmes possibles pour résoudre un problème donné).

Raffinement *top-down*: un exemple

La relation "L'entier `N` est le minimum de la liste `L`" peut se décomposer comme suit:

"`N` est le minimum de la liste `[N]`"

"soit `M` la tête de `L`, `H` le reste de `L` et `M` le minimum de la liste `H`. Si `N<M` alors `N` est le minimum de `L`, sinon `M` est le minimum de `L`"

Voici le programme Prolog correspondant:

```
minimum2(N,[N]).
minimum2(N,[N|H]) :- minimum2(M,H) , N < M.
minimum2(M,[N|H]) :- minimum2(M,H) , N >= M.
```

Un programme prolog: assertions et règles

```
/*bio(nom,sexe,ne_en,dcd_en,pere,mere)*/
bio(louis13,h,1601,1643,henri4,marie_medicis).
bio(elisabeth_france,f,1603,1644,henri4,marie_medicis).
bio(louis14,h,1638,1715,louis13,anne_autriche).
/*pere(pere,enfant)*/
pere(X,Y):- bio(Y,_,_,_,X,_).
/*age(personne,age)*/
age(X,Y):-bio(X,_,Z,T,_,_),Y is T-Z.
```

On interroge le programme: buts

Quel est la date de naissance de Louis XIV?

```
bio(louis14,_,X,_,_,_).
```

Qui est le père de Louis XIII?

```
pere(X,louis13).
```

...mais aussi...

```
bio(louis13,_,_,_,X,_).
```

Combien d'année Louis XIV a survecu à son père?

```
bio(louis14,_,_,Y,Z,_),bio(Z,_,_,T,_,_),R is Y-T.
```

Parmi les personnes dont on dispose de la "biographie", qui sont les hommes?

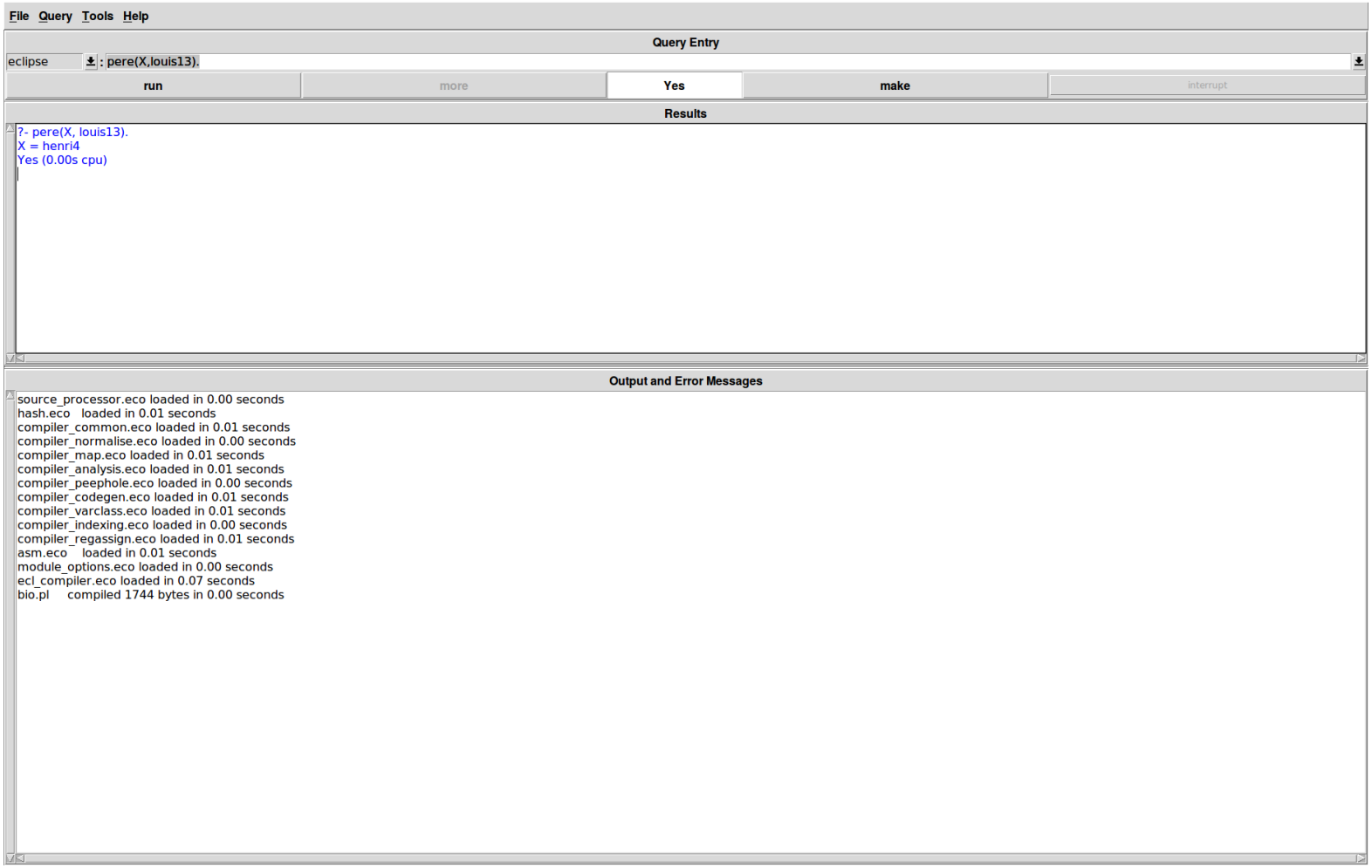
```
bio(X,h,_,_,_,_).
```

Interaction au top level: eclipse

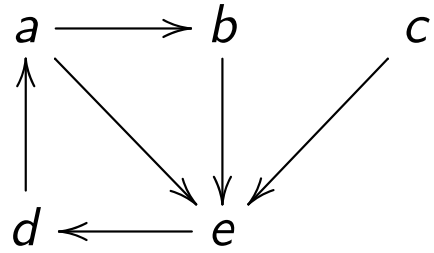
```
/* bio.pl */
bio(louis13,h,1601,1643,henri4,marie_medicis).
bio(elisabeth_france,f,1603,1644,henri4,marie_medicis).
bio(louis14,h,1638,1715,louis13,anne_autriche).
pere(X,Y):-bio(Y,_,_,_,X,_).
age(X,Y):-bio(X,_,T,Z,_,_), Y is Z-T.
# eclipse
[eclipse 1]: compile(bio).
bio.pl compiled 1744 bytes in 0.00 seconds
Yes (0.07s cpu)
[eclipse 2]:  bio(louis14,_,X,_,_,_).
X = 1638
Yes (0.00s cpu)
[eclipse 3]:  pere(X,louis13).
X = henri4
Yes (0.00s cpu)
```

Interaction au top level: `tkeclipse`

```
# tkeclipse&
```



2ème exemple: modélisation d'un graphe orienté




```
/* arc(source,destination) */
arc(a,b).  arc(a,e).  arc(b,e).  arc(c,e).  arc(e,d).
arc(d,a).
/* connexe(source,destination) */
connexe(X,Y):-arc(X,Y).
connexe(X,Y):-arc(X,Z),connexe(Z,Y).
```

Interaction au top level

```
[eclipse 1]:  compile(graph).
Yes (0.07s cpu)
[eclipse 2]:  arc(a,X).
X = b
Yes (0.00s cpu, solution 1, maybe more) ?  ;
X = e
Yes (0.00s cpu, solution 2)
[eclipse 3]:  connexe(a,c).
*** Overflow of the local/control stack!
```

Une solution possible:

```
connexe1(X,Y):-arc(X,Y).
connexe1(X,Y):-aux(X,Y,[]).
aux(X,Y,L):-arc(X,Z), not(membre(Z,L)), aux(Z,Y,[Z|L]).
membre(X,[X|_]).
membre(X,[_|L]):-membre(X,L).
[eclipse 4]:  compile(graph).
Yes (0.00s cpu)
[eclipse 5]:  connexe1(a,c).
No (0.00s cpu)
```

Syntaxe

Les éléments de base d'un programme Prolog sont les *prédicats* et les *termes*.
Dans
`connexe(a,X).`
`connexe` est un (symbole de) prédicat.
`a` est un terme (une constante).
`X` est un terme (une variable).
Il n'y a pas de différence syntaxique entre prédicats et termes: les termes, comme les prédicats, peuvent avoir des arguments:
`p(X,q(X,_,_)).`
ici `q(X,_,_)` est un terme à trois arguments (qui peut désigner arbre binaire étiqueté, par exemple).

Le lexique et la syntaxe (simplifiés) de Prolog

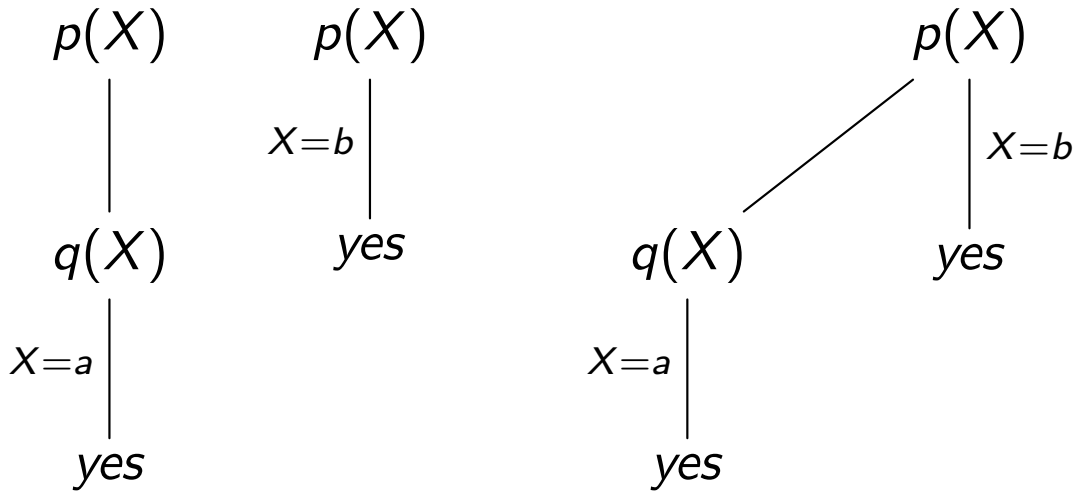
constantes	chaînes de caractères dont le premier est minuscule
variables	chaînes de caractères dont le premier est majuscule, ou _
nombres	en notation décimale
punctuations	:- , . ( ) ; ...

```
programme-prolog ::= clause { clause }
clause ::= assertion | regle
assertion ::= predicat.
predicat ::= constante [ ( liste-termes ) ]
regle ::= predicat :- corps .
corps ::= predicat { , predicat }
liste-termes ::= terme { , terme }
terme ::= terme-simple | terme-complexe
terme-simple ::= constante | variable | nombre
terme-complexe ::= constante ( liste-termes )
```

Un exemple d'exécution

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

but: p(X)



Arbre de dérivation de p(X)

L'unification de termes

Une substitution est une fonction partielle qui associe des termes à des variables, c.à.d. un ensemble de couples `(Var, terme)`.
◇ Par exemple σ_0=`{(X,zero),(Y,succ(T))}` est une substitution.
Le résultat de l'application d'une substitution σ à un terme `t`, notée `t`σ, est le terme `t` dans lequel les variables de σ sont remplacées par les termes correspondants.
◇ Par exemple `add(X,Y)`σ_0 = `add(zero, succ(T))`
Deux termes `t` et `s` sont unifiés par la substitution σ si `t`σ et `s`σ sont identiques.
◇ Par exemple les termes `add(X,3)` et `add(7,Y)` sont unifiés par la substitution `{(X,7),(Y,3)}`.
La substitution σ est l'unificateur le plus général (mgu) des termes `t` et `s` s'il unifie `t` et `s` et si tout autre unificateur de `t` et `s` s'obtient par instanciation de σ.
◇ Par exemple `{(X,3),(Y,123)}` est un unificateur de `f(X,Y)` et `f(3,Y)`, mais ce n'est pas leur mgu. Le mgu est `{(X,3)}`, que l'on peut aussi désigner par `{(X,3),(Y,Y)}`.

L'unification comme prédicat défini en Prolog

Le prédicat binaire infixe = de Prolog calcule les unificateurs (mgu)

```
[eclipse 1]:  f(X)=f(g(a,Y)).
X = g(a, Y)
Y = Y
Yes (0.00s cpu)
[eclipse 2]:  f(X,X)=f(g(a,Y),g(Z,b)).
X = g(a, b)
Y = b
Z = a
Yes (0.00s cpu)
[eclipse 3]:  X=f(X).
X = f(f(f(f(f(f(f(f(f(f(f(f(f(f(...))))))))))))))
Yes (0.00s cpu)
```

Pas de *occur check*!

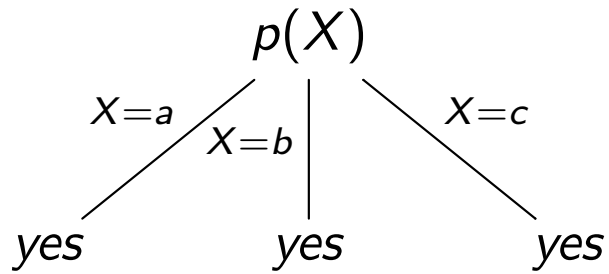
Arbres de dérivation

On considère un programme $\mathtt{P}=\mathtt{c}_1,\ldots,\mathtt{c}_k$ et un but $\mathtt{G}=\mathtt{g}_1,\ldots,\mathtt{g}_n$.
Soit $\mathtt{c}_i = \mathtt{t}_i :- \mathtt{p}_i^1,\mathtt{p}_i^2,\ldots,\mathtt{p}_i^{m_i}$, pour i allant de 1 à k.
(si $m_i = 0$ alors $\mathtt{c}_i$ est une assertion, sinon c'est une règle).
On définit l'**arbre de dérivation** de $\mathtt{G}$ pour $\mathtt{P}$. Cet arbre est potentiellement infini, ses noeuds sont étiquetés par des buts, et ses branches par des substitutions.

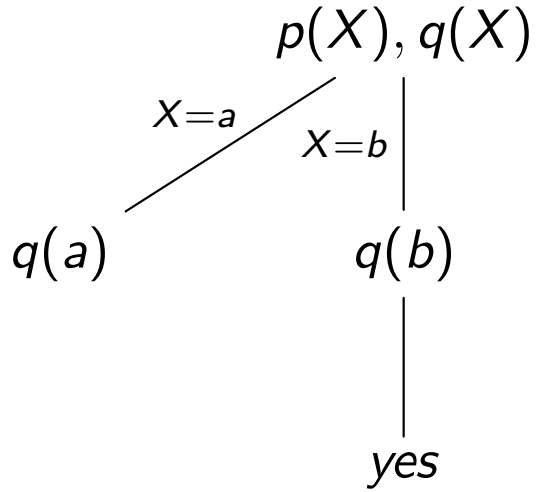
- La racine (c.à.d. le seul noeud de hauteur 0) est $\mathtt{G}$.
- Soit $\mathtt{H}=\mathtt{h}_1,\ldots\mathtt{h}_j$ un noeud de hauteur n, et soit $\mathtt{c}_s$ une clause de $\mathtt{P}$ dont la tête $\mathtt{t}_s$ s'unifie avec $\mathtt{h}_1$, avec mgu σ.
  Alors on crée le noeud $\mathtt{H'}= \mathtt{p}_s^1\sigma,\ldots,\mathtt{p}_s^{m_s}\sigma,\mathtt{h}_2\sigma,\ldots\mathtt{h}_j\sigma$,
  de hauteur $n+1$, et on étiquette la branche de $\mathtt{H}$ à $\mathtt{H'}$ par σ.
- Une feuille est soit un but vide (succes), soit un but dont le premier prédicat ne s'unifie avec la tête d'aucune clause (echec).

Exemples d'arbres de dérivation

```
P = p(a).  p(b).  p(c).
G= p(X)
```




```
P= p(a).  p(b).  q(b).
G= p(X),q(X)
```



Arbres de dérivation, suite et fin

- L'exécution d'un goal `G` pour un programme `P` a comme résultat l'ensemble de feuilles succes de l'arbre de dérivation correspondant.
  Plus précisement, pour chacune de ces feuilles, le résultat est l'ensemble des instanciations des variables de `G` qui se trouvent sur le chemin qui mène de la racine à la feuille en question.
- L'arbre de dérivation est produit de manière préfixe (parcours en profondeur d'abord).
  Donc l'ordre des clause est important. Par exemple, pour
  ```
  p(X):-p(X).
  p(a).
  ```
  le but `p(X)` ne donne aucun résultat (boucle), mais pour
  ```
  p(a).
  p(X):-p(X).
  ```
  le résultat `X=a` est atteint.

Exemple de construction d'un arbre de dérivation

- Programme:

```
p(X,c):-q(X).
p(a,b).
p(a,a):-t(a).
q(X):-t(X).
q(c).
t(b).
```

- But:

```
p(X,Y).
```

(au tableau)

Quelques symboles de fonctions sur les nombres

Plusieurs opérateurs sont predéfinis pour des opérations arithmétiques simples.

- `+` l'addition
- `-` la soustraction
- `*` la multiplication
- `/` la division
- `//` la division entière
- `mod` le reste de la division entière

L'expression `3 + 4` est à considérer en Prolog comme synonyme de `+(3,4)`, un terme d'arité 2 avec symbole fonctionnel + et arguments 3 et 4

A la requête
`X = 4 + 3` Prolog répond
`X = 4 + 3`
Pour évaluer une expression arithmetique il faut utiliser le prédicat (infixe) `is` .

Exemples:

```
[eclipse 1]:  X = 4 + 3.
X = 4 + 3
Yes (0.00s cpu)
[eclipse 2]:  Y is 4 + 3.
Y = 7
Yes (0.00s cpu)
[eclipse 3]:  X is Y + 1.
instantiation fault in +(Y, 1, _279)
Abort
[eclipse 4]:  Y = 2, X is Y + Y.
Y = 2
X = 4
Yes (0.00s cpu)
```

Quelques prédicats sur les nombres

Comparaisons

- `X > Y`
- `X < Y`
- `X >= Y`
- `X =< Y`
- `X =:= Y` `X` et `Y` désignent le même nombre.
- `X =\= Y` `X` et `Y` désignent des nombre différents.

Dans tous ces prédicats, les deux arguments doivent être clos: **ils ne doivent pas contenir des variables non instanciées**. Les deux arguments sont évalués si nécessaire:

```
[eclipse 7]:  3*2+7 mod 4 =< 11-3.
No (0.00s cpu)
```

Plusieurs "égalités"

- `=` unification
- `=:=` test d'égalité entre nombres, avec évaluation.
- `==` test d'égalité *syntaxique* entre termes quelconques.

```
[eclipse 1]:  1 + 3 =:= 3 + 1.
Yes (0.00s cpu)
[eclipse 2]:  1 + 3 = 3 + 1.
No (0.00s cpu)
[eclipse 3]:  1 + A =:= B + 2.
instantiation fault in +(1, A, _366)
Abort
[eclipse 4]:  ?- 1 + A = B + 2.
A = 2
B = 1
Yes (0.00s cpu)
[eclipse 5]:  1 + A == B + 2.
No (0.00s cpu)
[eclipse 6]:  X = 3 + 2, X < 7.
X = 3 + 2
Yes (0.00s cpu)
```

Résumé du premier cours

- Syntaxe de Prolog:
  - constantes, variables, ponctuations,
  - termes, prédicats,
  - assertions, règles, programmes, buts.
- Sémantique (opérationnelle) de Prolog:
  - substitution, unification,
  - arbre de dérivation d'un but P pour un programme G,
  - résultat d'un but: ensemble des feuilles succes de l'arbre de dérivation, et relatives instanciations des variables du but.
- Les nombres en Prolog
  - opérations,
  - relations,
  - plusieurs "égalités": =, =:=, ==.