Introduction à l'Intelligence Artificielle

Mélanie COURTINE
melanie.courtine@univ-paris13.fr

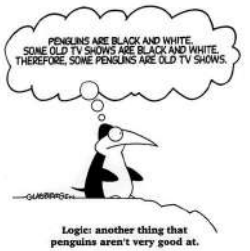
Plan

Evaluations
1 examen
1 travail à rendre en Prolog

- Logique des propositions et des prédicats
- Langage Prolog
- Histoire de l'Intelligence Artificielle
  - Des mythes à la réalité
- Résolution de problèmes : exploration intelligente des solutions
  - Modélisation sous forme de graphes d'états
  - Recherche de solutions optimales
  - Définition d'heuristiques
- Systèmes formels, experts et à base de connaissances
- Représentation des connaissances
  - De la logique aux langages complexes
- Extraire des connaissances
  - Induction d'arbres de décision et de règles
  - Classification automatique
  - Fouille de données

Bibliographie

- L. Sterling et E. Shapiro (1990) *L'art de Prolog.* Masson.
- F. Giannesini, H. Kanoui, R. Pasero et M. Van Caneghem (1995) *Prolog*. Interéditions.
- P. Bellot (1997) *Objectif Prolog*. Dunod.
- I. Bratko (2009) *PROLOG Programming for AI*. Addison Wesley.
- L. Gacôgne (2009) *Prolog Programmation par l'exemple*. Editions Hermann.



Prolog : une introduction

Styles de programmation (1)

- Programmation **impérative** (et programmation objet)
  - Basée sur la notion de blocs d'instructions
    - Séquencement des calculs **spécifié**
      - Contrôle total du flot d'exécution

    Ex : Pascal, C, …
- Programmation **objet**
  - Notion d'entités-messages

    Ex : Smalltalk, C++, Java, …
- Programmation **fonctionnelle**
  - Tout est **fonction**
    - Syntaxe dépouillée
    - Base mathématique forte : λ-calcul
  - **Contrôle délégué** à l'interprète
    - Utilisation intensive de la récursivité

    Ex : LISP, Scheme, …

Styles de programmation (2)

- Programmation **logique**
  - Tout est **logique**
    - Syntaxe simple et dépouillée
    - Base théorique : **calcul des prédicats**
    - La logique est un **cadre formel** qui permet de formaliser, de représenter et de raisonner.
  - Encore plus de contrôle donné à la machine
    - Récurrence
    - Non déterminisme

    Ex : Prolog

PROgrammation LOGique

- **1930** : Calcul des prédicats (J. Herbrand)
- **1945** : Principe de résolution (J.A. Robinson)
- **1969** : Kowalski montre que la logique peut être utilisée comme langage de programmation
- **1972** : Premier interprète Prolog (A. Colmerauer et P. Roussel)
- **1977** : Premier compilateur Prolog (D.H.D. Warren)
- **1980** : Prolog = le langage de l'Intelligence Artificielle
- **1990** : Prolog = le langage la programmation par contraintes

Domaines d'application

- Logique mathématique
- Traitement des langages naturelles
- Conception assistée par ordinateur
- Résolution symbolique d'équations
- Analyse de structures biochimiques
- Planification, allocation de ressources
- Systèmes-experts
- Aide à la décision
- Aide au dépannage
- …

La logique

- En logique – calcul des propositions, calcul des prédicats, et autres – les formules ont :
  - La **syntaxe** est liée à l'écriture des formules
    - Elle définit la façon dont les formules doivent être écrites pour former le langage de la logique considérée
    - Analogie : la syntaxe d'un langage de programmation définit la façon dont les instructions doivent être écrites
  - La **sémantique** est liée à la façon d'associer un sens aux formules
    - En calcul des propositions, ce sens est donné par une valeur de vérité : la formule peut être vraie ou fausse
- La logique permet de **représenter** des connaissances et de **raisonner** sur ces connaissances

Prolog

- Résoudre un problème en Prolog
  - Formaliser un problème en terme de logique
  - Utiliser le moteur d'inférence pour faire des démonstrations logiques et répondre à des questions
- Mais on peut comprendre Prolog et l'apprendre sans connaître la logique

⇨ Programmation déclarative (vs procédurale)

Logique des propositions

Comment écrire les formules ?
Comment déterminer la valeur de vérité d'une formule ?
Comment démontrer de nouveaux résultats ?

Le calcul des propositions

- Le calcul des propositions manipule des **formules propositionnelles** (ou propositions) formées à partir de formules élémentaires appelées **atomes** (ou variables propositionnelles) et de **connecteurs** booléens.

Définition

Un **atome** est un énoncé élémentaire (qu'on ne cherchera pas à décomposer).

- Par convention, les atomes par des **lettres minuscules**.
- Exemple :

```
p = il_fait_beau
q = strasbourg_est_en_france
r = les_poules_ont_des_dents
```

Connecteurs logiques

Définition

Les **connecteurs** booléens utilisés en calcul des propositions sont les suivants :

- ¬ *(non)* : négation
- ∧ *(et)* : conjonction
- ∨ *(ou)* : disjonction
- → *(implique* ou *si…alors…)* : implication
- ↔ *(équivalent* ou *si et seulement si)* : équivalence

Propositions

Définition

Les **propositions** (ou formules propositionnelles) sont définies comme suit :

- un atome est une proposition,
- si A est une proposition alors (¬A) est une proposition,
- si A et B sont des propositions alors (A ∧ B), (A ∨ B), (A → B) et (A ↔ B) sont des propositions.
- La proposition vide est notée □ et définie par (a∧¬a)

- Par convention, les propositions par des **majuscules** ou des lettres grecques
- Attention aux **ambiguïtés** :
  - Ordre de **dominance des connecteurs** : ↔, →, ∧, ∨, ¬
  - La proposition `p→ q↔ ¬r` doit se lire `( (p→ q) ↔ (¬r) )`

Limites du calcul propositionnel

- Modéliser :
  - Les chandelles sont faites pour éclairer
  - Quelques chandelles éclairent très mal
  - Quelques objets qui sont faits pour éclairer le font très mal

Impossible

Logique des prédicats

Une modélisation

- Les chandelles sont faites pour éclairer

$$\forall x, \quad chandelle(x) \rightarrow éclaire(x)$$

- Quelques chandelles éclairent très mal

$$\exists x, \quad chandelle(x) \wedge éclaireMal(x)$$

- Quelques objets qui sont faits pour éclairer le font très mal

$$\exists x, \quad éclaire(x) \wedge éclaireMal(x)$$

La logique des prédicats

- Le langage des **prédicats du premier ordre** (LP1) est un langage formel qui permet de :
  - **exprimer** des connaissances (complexes) avec rigueur
  - **combiner** les connaissances pour engendrer de nouvelles

Syntaxe

- Des **variables** (x ,y, …)
- Des **constantes**(a ,b, …)
- Des **fonctions** (f, g, …)
  - les fonctions d'**arité** 0 sont appelées des **constantes**
- Des relations (**prédicats**) (R, S, éclaire, …)
- Des **connecteurs logiques** (¬, ∧, ∨, → et ↔)
- Des **quantificateurs** (∀ et ∃)

Formules

- Un atome est une **formule**
- Si **F** et **G** sont des formules et **x** une variable, alors les expressions suivantes sont des **formules**
  - ¬(**F**)
  - (**F**) ∧ (**G**) et (**F**) ∨ (**G**)
  - (**F**) → (**G**) et (**F**) ↔ (**G**)
  - ∀**x** (**F**) et ∃**x** (**G**)

Inconvénients de la LP1

- La logique du premier ordre
  - Ne permet pas :
    - d'exprimer des nuances
    - De décider avec des informations manquantes
  - Contrairement à l'homme qui a des raisonnements par défaut
- Recours à d'autres logiques :
  - Logiques multivaluées, modales, non monotones, temporelle, floue, …

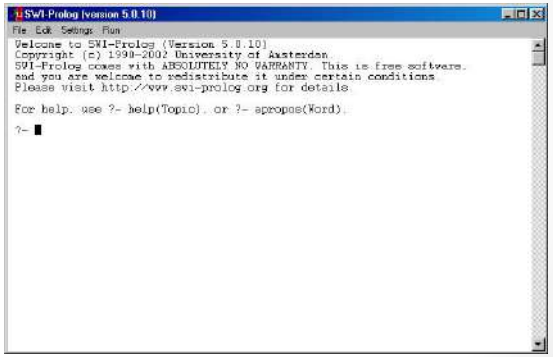
Les fondamentaux de Prolog

Les environnements Prolog

- Ce sont des systèmes interactifs
- Environnements
  - Quintus Prolog
  - BIM Prolog
  - SWI-Prolog
  - Turbo-Prolog
- Une session de travail
  - Une fenêtre SWI-Prolog
  - Une fenêtre d'édition : Xemacs
  - Une fenêtre de shell

SWI-Prolog

- SWI-Prolog : http://www.swi-prolog.org



Les clauses de base

- Afficher le répertoire de travail et ses fichiers

```
pwd.
ls.
```

- Changer de répertoire de travail

```
chdir('~/prolog/tp1').
```

- Charger un fichier

```
consult(menu).
consult('menu.pl').
['menu.pl'].
['menu.pl',fichier1].
```

- Affichage des clauses

```
listing.
listing(plat).
```

- Quitter

```
halt.
```


Communication homme-machine

- **Enoncé :**
  - *Tout psychiatre est une personne.*
  - *Chaque personne qu'il analyse est malade.*
  - *Jacques est un psychiatre à Paris.*
  - *Est-ce que Jacques est une personne ?*
  - *Où est Jacques ?*
  - *Est-ce que Jacques est malade ?*
- **Réponses du système :**
  - *Oui.*
  - *A Paris.*
  - *Je ne sais pas.*

PROLOG

- Prolog est un langage de **programmation déclaratif**.
- Il permet de **représenter** et de **manipuler des connaissances** sur un **domaine** particulier.
  - **Représentation** = logique des prédicats du premier ordre
  - **Domaine** = ensemble d'objets
  - **Connaissances** = ensemble de relations entre des objets, pour décrire leurs propriétés, leurs interactions.

Représentation de connaissances

- *« Jean est le père de Paul »* signifie qu'une relation *« est père de »* lie les objets *Jean* et *Paul*.
- En PROLOG, on écrirait :
  - `est-pere-de('Jean','Paul').`
  - ou plus simplement : `pere(jean,paul)`.
- *« Qui est le père de Paul ? »* revient à :
  - chercher si la relation *« est père de »* lie *Paul* à un autre objet qui sera la réponse à la question

La programmation déclarative

- Les **faits** concernent des objets particuliers
- Les **règles** concernent les catégories d'objets
  - Elles permettent d'établir de nouveaux faits
    - Une conjonction de termes => une seule règle
    - Une disjonction de termes => plusieurs règles
- Les **questions** permettent de déterminer si un fait est ou peut être établi/vérifié.

Programme Prolog

/* .... */ : les commentaires de blocs
% ..... : commentaire de fin de ligne

Trois sortes de connaissances :

Programme Prolog
= un ensemble de clauses
= des **faits +** des **règles**

- Faits : **P(…).** où P est un prédicat
  - Clause de Horn réduite à un littéral positif
- Règles : **P(…) :- Q(…), …, R(…).**
  - Clause de Horn complète

Fichier texte avec l'extension pl

Résoudre un problème en Prolog
= poser une question

- Questions : **S(…), …, T(…).**
  - Clause de Horn sans littéral positif

sous Prolog

Les faits

- Les **faits** sont des affirmations qui décrivent des relations ou des propriétés
  - C'est une clause de Horn réduite à un littéral positif
- Forme : **predicat (arg1, arg2, …, argn).**
- Un prédicat est identifié par son nom et son arité : **predicat/n**
- Ex :
  - homme(sartre).
  - mange(vache, herbe).
  - eleve(albert, 1982, maths, 8).

Les règles

- Les **règles** permettent d'exprimer des conjonctions de buts
  - Clause de Horn complète
  - C'est une combinaison de faits à partir desquels nous pouvons déduire un nouveau fait
- Forme : **tete :- C1, C2, …, Cn.**
  - si C1, C2 .. et Cn sont vraies alors la tête est aussi vraie.
- Ex :
  - « tous les herbivores mangent de l'herbe »
  - « tous les animaux qui volent sont des oiseaux »
  - Prédicat fils/2

Les termes

- Les **constantes**
  - Les nombres
  - Les atomes
    - Les atomes **standards**
      - chaînes de caractères commençant par une **minuscule**
    - Les atomes **protégés**
      - chaînes de caractères entre ' '
    - Les atomes **symboliques** : + - * / \ ^ < > = : . ? @ # $ &
- Les **variables**
  - chaînes de caractères commençant par une **majuscule** ou _
  - La variable indéterminée (singleton) : _
- Les structures/relations/**prédicats** d'arité n :

  `nom (arg1, arg2, …argn)`

Le menu

```
/* La carte est donnée ("clauses faits") */
/* les entrées */
entree(crudites).
entree(terrine).
entree(melon).

/* les viandes (avec légumes associés) */
viande(steack).
viande(poulet).
viande(gigot).

/* les poissons (avec légumes associés) */
poisson(bar).
poisson(saumon).

/* les desserts */
dessert(sorbet).
dessert(creme).
dessert(tarte).
/* Fin de la carte */

/* Composition d'un menu simple (sans boisson) : une entrée, un plat, un
  dessert */
menu_simple(E, P, D) :- entree(E), plat(P), dessert(D).
/* le plat (de résistance) : viande OU poisson */
plat(P) :- viande(P).
plat(P) :- poisson(P).
```


Les questions et les requêtes

- Une **question** a la forme d'une formule atomique pouvant contenir des variables

  question = requête = but

- Deux types de questions :
  - Les questions **fermées**
  - Les questions **ouvertes**
    - Elles contiennent des variables
- **;** permet d'avoir la solution suivante

La stratégie de résolution (1)

- Exécuter un programme
  - ⇔ Résoudre un but
  - ⇔ Poser une question
  - ⇔ Demander une preuve d'une expression
  - ⇔ Trouver toutes les valeurs des variables qui apparaissent dans la question et amènent à une contradiction
- La règle de résolution en utilisant la technique de backtracking
  - Recherche en profondeur
  - Retour-arrière

La stratégie de résolution (2)

- L'utilisateur ne spécifie que le « **quoi** » et pas le « **comment** »
- **Avantages** :
  - souplesse et pouvoir d'expression du langage
  - rapidité de développement d'un système
  - simplification de la maintenance
- **Inconvénients** :
  - Bonne maîtrise des stratégies de résolution de PROLOG
  - Expertise dans l'ordre des faits et règles à inoculer au système

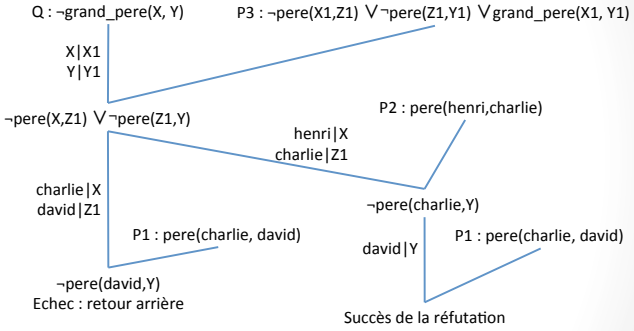
Grand_père

- Soit le programme :

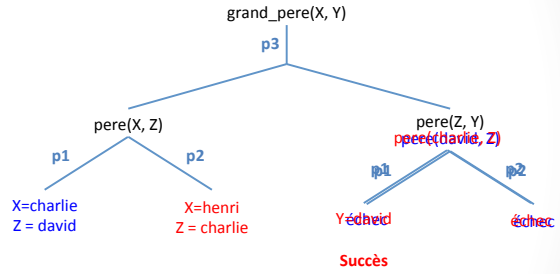
```
P1 : pere(charlie, david).
P2 : pere(henri, charlie).
P3 : grand_pere(X,Y) :- pere(X,Z), pere(Z,Y).
```

- Soit la question
  - `grand_pere(X,Y).`
- Réponse :
  - X=henri, Y=david

Graphe de résolution



Arbre et/ou de recherche



Prolog n'est pas équivalent à la logique

- Soit la règle `p :- q, r.`
  - L'ordre a de l'importance :
    - Pour résoudre p, résoudre d'abord le sous-problème q, puis le sous-problème r
    - L'interpréteur considère les clauses les unes après les autres, dans l'ordre dans lesquelles elles se trouvent
  - Recommandation :
    - commencer toujours les prédicats les plus simples à résoudre

- Exemple :
  - `ancetre(X,X).`
  - `ancetre(X,Y) :- pere(X,Z), ancetre(Z,Y).`
  - et pas : `ancetre(X,Y) :- ancetre(Z,Y), pere(X,Z).`

Carnivores et herbivores

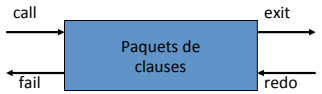
- *La chèvre est un animal herbivore.*
- *Le loup est un animal carnivore.*
- *Un animal cruel est carnivore.*
- *Un animal carnivore mange de la viande.*
- *Un animal herbivore mange de l'herbe.*
- *Un animal carnivore mange des animaux herbivores.*
- *Les herbivores et les carnivores boivent de l'eau.*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

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

Formaliser le problème et la question. Quelle est la réponse ?

Modèle d'exécution

- Boîte de Byrd



s : « skip » : sauter un prédicat
<return> : avancer en profondeur dans le débogueur (creep)
a : quitter le débogueur

- Activation : `trace.`
- Désactivation : `notrace.`

Exemple de trace

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

```
?- trace, p(X).
   Call: (7) p(_G157) ? creep
   Call: (8) q(_G157) ? creep
   Exit: (8) q(a) ? creep
   Exit: (7) p(a) ? creep

X = a ;
   Redo: (8) q(_G157) ? creep
   Exit: (8) q(b) ? creep
   Exit: (7) p(b) ? creep

X = b ;
   Fail: (8) q(_G157) ? creep
   Redo: (7) p(_G157) ? creep
   Exit: (7) p(c) ? creep

X = c ;
   Fail: (7) p(_G157) ? creep

No
```

L'unification

- L'**unification** :

  Procédé par lequel on essaie de rendre deux formules identiques en donnant des valeurs aux variables qu'elles contiennent.
  - Deux termes peuvent être ou ne pas être unifiables
  - Substitution : σ={(V1,T1), (V2,T2), …(Vn, Tn)}
    - La substitution n'est pas forcément unique
    - On cherche l'unificateur le plus général

→ Deux termes sont unifiables ssi il existe une subtitution σ telle que σ(T1) = σ(T2)

Unifiables ou non unifiables (1)

- Deux atomes sont **unifiables** ssi ils sont identiques
  - un m.g.u. est la substitution vide
- Un atome b et une variable X sont unifiables
  - un m.g.u. est V={X=b}
- Un atome et une structure ne sont pas unifiables
- Deux variables sont unifiables :
  - Si elles sont identiques, X et X par exemple
    - un m.g.u. est la substitution vide
  - Si les deux variables sont de noms différents, X et Y sont unifiables
    - les m.g.u. possibles sont V1={X=Y} ou V2={Y=X}

Unifiables ou non unifiables (2)

- Soit une variable X et une structure f(T1,T2,…,Tn) à unifier.
  - Si la variable X apparaît dans la structure f(T1,T2, …,Tn), alors les deux termes ne sont pas unifiables.
  - Dans le cas contraire, les deux termes sont unifiables
    - un m.g.u est V={X=f(T1,T2,…,Tn)}
  - Pour que deux structures soient unifiables, il est nécessaire qu'elles aient le même nom et soient de même arité

L'unification

Exercice

- Les termes suivants sont-ils unifiables ?
  1. pierre et marie
  2. X et jean
  3. X et Y
  4. X et X
  5. pierre et aime(X, Y)
  6. père(X, Y) et aime(U, jean)
  7. père(X, Y) et père(U, jean)
  8. X et parent(X, paul)
  9. parent(Y, Z) et parent(X, paul)
  10. plus(s(X), Y, s(Z)) et plus(U, V, s(s(s(0))))

Liste

- Une **liste** est une suite de termes de longueur quelconque
- Elle est définie récursivement de la manière suivante :
  - la liste vide [] est une liste
  - Si T est un terme et L une liste alors le terme [T|L] représente la liste dont le premier élément est T et L représente la liste privée de son premier élément.
- Prédicat prédéfini : `is_list\1`
- Rq : la liste vide est la fin de n'importe quelle liste

Des exemples de listes

- [a]
- [a | [b,c]]
- [a, [b,c], d]
- [[a],[b],c]
- [a, [b, [c]]]
- [[a,b], [b,c]]
- [[a,b,c],[b,a,c] | [c,b,a]]
- [[a,b,c],[b,a,c], [c,b,a]]
- [[il, fait], beau, [a, paris]]
- Rq : [a, b, c] = [a|[b,c]] = [a, b|[c]] = [a,b,c|[]]

Exercice

Unification sur les listes

- Les termes suivants sont-ils unifiables ?
  1. [[il, fait], beau,[a,paris]] = [X,Y]
  2. [a, b] = [X]
  3. [a, [b, c], d] = [X, Y, Z]
  4. [a, [b, c], d] = [X, Y]
  5. [a, [b, c], d] = [X | Y]
  6. [a, [b, c], d] = [a | L]
  7. [a, [b, c], d] = [a, b | L]
  8. [a, b, c, d] = [a, b | L]
  9. [a, b, c, d | L1] = [a, b | L2]

Expressions arithmétiques

- Termes des expressions arithmétiques :
  - Des nombres : 1, 2.3, -1, pi, e, cputime
  - Des variables : X, Nb, _nb
  - Des fonctions ou opérateurs prédéfinis :
    - Opérateurs unaires : -, abs, sign, sin, cos, tan, exp, sqrt, log, log10
    - Opérateurs binaires : +, -, *, /, **, min, max
    - Fonctions à arg IntExpr : //, mod
    - Fonctions à arg Int : random(N)
      - nombre aléatoire i entre 0 ≤ i < N

Comparateurs arithmétiques

<+Expr1> ***<comp>*** *<+Expr2>*

- <comp> =:=, =\=, =<, <, >, >=
- Expr1 et Expr2 doivent être évaluables au moment de l'effacement
  - =:= signifie X et Y ont même valeur
- *Ex :*
  - ?- X + 2 < X + 4 .
  - ?- X = 2, X + 2 < X + 4 .

Prédicat prédéfini `is/2`

<?Terme> ***is*** *<+Expression>*

- s'efface si Terme est unifiable avec le résultat de l'évaluation de Expr.
  - Terme est soit un nombre, soit une variable.
  - Expr doit être évaluable au moment de l'effacement.
- Exemple :
  - ?- 5 is 2 + 3 .
  - ?- N is 2 + 3 .
  - ?- T is sin(pi/2) .
  - ?- X is 2, X + 2 < X + 4 .

Utiliser le bon comparateur

- Unifiabilité des termes : `=/2`
- Non unifiabilité des termes : `\=/2`
- Identité formelle des termes : `==/2`
- Non identité formelle des termes : `\==/2`
- Unification avec le résultat : `is/2`

Unification, identité formelle, comparateur et évaluateur

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

?- 1+2 = 2+1.
?- 1+2 == 2+1.
?- 1+2 =:= 2+1.
?- 1+2 is 2+1.

?- 0 = (10 mod 5).
?- 0 == (10 mod 5).
?- 0 =:= (10 mod 5).
?- 0 is (10 mod 5).
```

Exercices d'arithmétique

1. Prédicats « pair » et « impair »
   - N est pair si N est divisible par 2
2. Prédicat `div2/2`
   - div2(NN,N) s'efface si N est égal à la moitié de NN

La récursivité

- **Prédicat récursif**
  - Prédicat dont la définition fait appel (directement ou non) à lui-même
- **Programmation récursive**
  - **Décomposition récursive en sous-problème** : Ramener le problème, considéré à une complexité donnée, au même problème mais de complexité inferieure (principe de récurrence)
  - **Cas particuliers** : Chercher les cas particuliers éventuels ne rentrant pas dans le schéma récursif
  - **Cas d'arrêts** : Chercher les cas d'arrêts de la récursivité
    - **Remarque** : Les cas d'arrêts sont des cas particuliers, mais certains cas particuliers peuvent ne pas correspondre à l'arrêt de la récurrence générale
  - S'assurer de la **convergence de la récursivité** : En Prolog cela revient à chercher les modes d'utilisation
  - **Programmer** et vérifier les modes d'utilisation (jeux d'essais complets)

Exercices de récursivité

Sur les entiers

Pour chacun des cas, définir les éléments suivants :
- Décomposer récursive en sous-problème
- Identifier les cas particuliers et les cas d'arrêts
- Chercher les modes d'utilisation
- Programmer et vérifier les modes d'utilisation (jeux d'essais complets)

1. La suite de Fibonacci
   - fibonacci(1) = fibonacci(2) = 1
   - fibonacci(N) = fibonacci(n-1) + fibonnaci(n-2)
2. Prédicat « factorielle »
   - factorielle (0) = 1
   - factorielle (N) = N * factorielle (N-1) pour N>0
3. Prédicat « pair »

Exercices de récursivité

Sur les listes

Pour chacun des cas, définir les éléments suivants :
- Décomposer récursive en sous-problème
- Identifier les cas particuliers et les cas d'arrêts
- Chercher les modes d'utilisation
- Programmer et vérifier les modes d'utilisation (jeux d'essais complets)

1. Soit E un élément, L une liste, dernier(L,E) est vérifié si E est le dernier élément de la liste L
2. Soit E un élément, L une liste, appartient(L,E) est vérifié si E appartient à la liste L
3. Soit N un entier, L une liste, nombreElement(L,E) est vérifié si N est le nombre d'éléments contenus dans la liste L

Prédicats prédéfinis

- **`append(A,B,C)`** : pour la concaténation de listes

```
?- append([a,b,c],[d,e],L).
L = [a, b, c, d, e]
?- append(_,[X],[a,b,c,d]).
X = d
?- append(L2,L3,[b,c,a,d,e]),append(L1,[a],L2).
L2 = [b, c, a]
L3 = [d, e]
L1 = [b, c]
```

- **`member(A,L)`** : A est un élément de L

```
?- member(b,[a,b,c]).
true
?- member(X,[a,b,c]).
X = a ;
X = b ;
X = c.
```

- **`reverse(L,LR)`** : LR est la liste en sens inverse de L
- **`length(L,N)`** : N est la longueur de L

Système formel de Péano

- « Arithmétique formelle »
- Symboles de base :
  - Constante : 0
  - Fonction unaire : s
- Définition récursive des entiers formels
  - 0 est un entier formel
  - Si N est un entier formel alors s(N) est un entier formel
- En Prolog :

```
entierf(0).
entierf(s(X)) :- entierf(X).
```


Exercice

Somme d'entiers formels

- La somme de 2 entiers formels :
  - Prédicat : sommef/3
  - sommef(X,Y,Z) signifie « X+Y=Z »
  - La somme d'un entier X et de 0 est X.
  - La somme d'un entier X et de s(Y) donne le successeur de la somme de X et Y.

Affichage des termes

- **`write`**\1 : écriture dans le flot de sortie d'atomes
- **`display`**\1 : écriture dans le flot de sortie en notation préfixe
- **`writef`**\1 : écriture d'une chaîne de caractères
- **`nl`**\0 : passage à la ligne
- **`write_nl`**\1 : écriture d'atomes avec passage à la ligne
- **`tab`**\1 : affichage d'espaces
- **`put`**\1 : s'utilise en donnant le code ASCII d'un caractère

Lecture des termes

- **read**\1 : lit un terme au clavier et l'unifie avec son argument. Le terme lu doit être obligatoirement suivi d'un point.
- **get0**\1 : si X est une variable, get0(X) instancie X au code ASCII du caractère tapé. Sinon ce but réussit ou échoue suivant que X est ou n'est pas le code ASCII du premier caractère disponible sur le flot d'entrée.
- **get**\1 : ne considère que les caractères imprimables.

Sur les fichiers

- Ouverture d'un fichier :
  `open(FileName,Code,Flux).`
- Fermeture d'un fichier :
  `close(Flux).`
- Utilisation de tous les prédicats d'entrée/sortie vus précédemment en ajoutant comme premier argument le flux.

Les prédicats de contrôle

- Notion de contrôle sur le traitement des points de choix
  - Forcer le retour arrière : **Fail**
  - Créer des points de choix : **Repeat**
  - Eliminer des points de choix : **Cut, !**
- Utilisation combinée de ces prédicats :
  - Permet un meilleur contrôle des boucles
  - Permet de traiter une forme de la négation

L'échec

- Prédicat prédéfini : `fail/0`
  - Le prédicat fail n'est jamais démontrable
  - Le but fail ne s'efface jamais
  - Il provoque donc un échec de la démonstration où il figure
  - Cet échec de l'effacement entraîne un retour arrière

Afficher la liste des pères

```
liste_pere :- pere(X,Y),
              write(X),
              writef(« a pour pere »),
              write(Y),
              nl,
              fail.
liste_pere.
```

← Permet l'effacement final

donne l'affichage de « X a pour père Y » pour tous les jeux d'instances du couple (X,Y) pour lesquels pere(X,Y) s'efface.

La répétition

- Prédicat prédéfini : **`repeat/0`**
  - Le prédicat repeat est toujours démontrable
  - Il laisse systématiquement un point de choix derrière lui
  - Il y a une infinité de solutions
  - Il introduit indéfiniment un nouveau point de choix à chaque effacement
- Une façon de faire une **boucle** pour répéter un traitement en Prolog consiste à utiliser le repeat et le fail

Boucle d'interaction

```
boucle :- repeat, saisie, fail.
saisie :- writef(« Donnée suivante : »),
          read(X),
          write(X).
```

➡ L'échec de l'effacement de fail entraîne un retour arrière sur le point de choix introduit par repeat et donc la répétition du traitement des buts compris entre le repeat et le fail
et ce indéfiniment : boucle infinie !!

La coupure (1)

- Prédicat prédéfini : **`!/0`**
- La gestion des points de choix en Prolog
  - Lorsque l'interpréteur utilise une clause d'un paquet pour effacer un but :

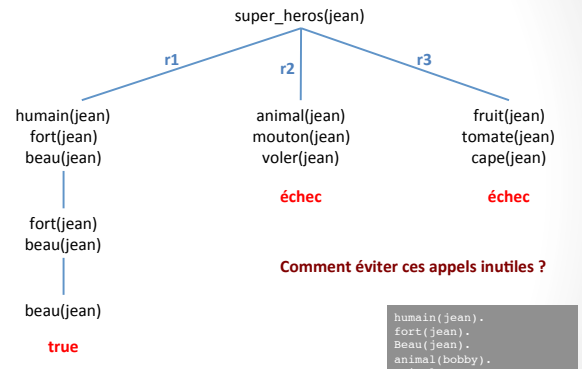
    si celle-ci n'est pas la dernière du paquet,
    il mémorise l'information pour pouvoir revenir ultérieurement sur le choix de clauses effectué
- Le coupe-choix permet de supprimer les points de choix « en attente »
  - Élagage de l'arbre de résolution en supprimant les possibilités non encore envisagées

Super-héros (1)

- Différents types de super héros :
  - humain→ un humain fort et beau
  - animal →un mouton volant
  - fruit→ une tomate avec une cape
    - (r1) `super_heros(X) :- humain(X), fort(X), beau(X).`
    - (r2) `super_heros(X) :- animal(X), mouton(X), voler(X).`
    - (r3) `super_heros(X) :- fruit(X), tomate(X), cape(X).`
- Une base de faits :

```
humain(jean).
fort(jean).
beau(jean).
animal(bobby).
animal(peggy).
cochon(peggy).
marche_au_plafond(peggy).
```


Super-héros (2)



Comment éviter ces appels inutiles ?

```
humain(jean).
fort(jean).
Beau(jean).
animal(bobby).
animal(peggy).
cochon(peggy).
marche_au_plafond(peggy).
```

Attention : cela suppose que si Jean est un humain, il n'y a pas d'animal ou de fruit nommés Jean

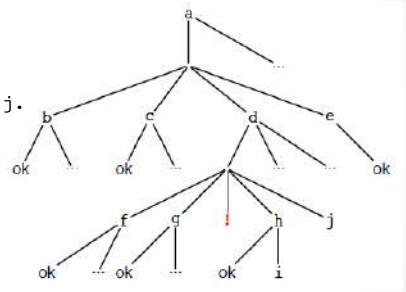
La coupure (2)

- Sert à
  - interdire l'exploration de certaines branches
  - améliorer l'efficacité d'un programme
  - confirmer un choix que l'on sait être le seul ou plus pertinent
  - écrire un "si alors sinon"
- `p :- a, b, ..., !, ..., s.`

  → Tous les choix mémorisés depuis l'appel de la tête de clause jusqu'à l'exécution du ! sont supprimés

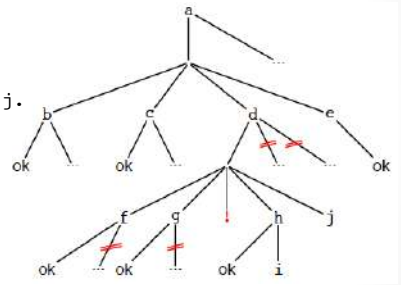
Exemple de coupure (1)

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
d :- ...
d :- ...
f.
f :- ...
g.
g :- ...
h.
h :- i.
e.
```



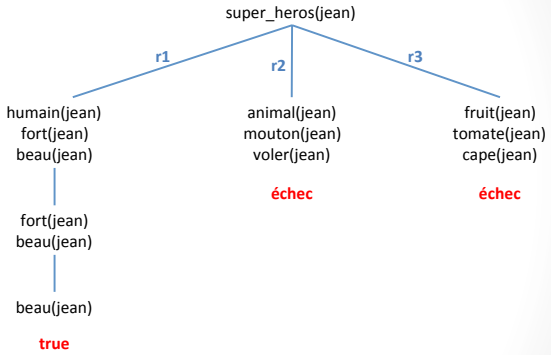
Exemple de coupure (2)

```
a :- b,c,d,e.
a :- ...
b.
b :- ...
c.
c :- ...
d :- f,g, !, h,j.
(d :- ...)
(d :- ... )
f.
(f :- ... )
g.
(g :- ... )
h.
h :- i.
e.
```

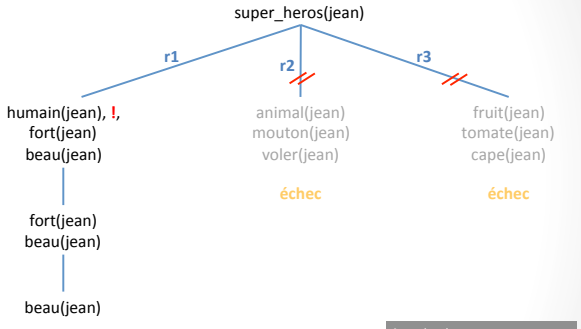


La coupure supprime les points de choix sur les sommets aînés et sur le sommet père

Super-héros (2)



Super-héros (3)




```
humain(jean).
fort(jean).
animal(bobby).
animal(peggy).
cochon(peggy).
marche_au_plafond(peggy).
```


Si … Alors … Sinon …

- La coupure peut être utilisée pour obtenir le schéma général :

  Si <t> alors <s> sinon <a>

- Soit en Prolog :

```
ifThenElse(T,A,S) :- T, !, A.
ifThenElse(_,_,S) :- S.
```

- Rq : pas de retour arrière sur T (seule la première solution sera obtenue)

Utilisation de la coupure

- La coupure produit un « effet de bord »
- Importance de la place de la coupure dans le corps de la clause et de la place de la clause contenant la coupure dans le paquet
- Quand utiliser la coupure :
  - Modifier/réduire le nombre de solutions cherchées : **coupure forte**
  - Économiser des essais de clauses inutiles, sans modifier l'ensemble des solutions : **coupure faible**
  - Économiser des tests coûteux ou éviter des tests infaisables : cas de la **négation**

Danger de la coupure

- Soit le programme suivant :

```
enfants(helene, 3).      enfants(helene, 3) :- !.
enfants(corinne, 1).     enfants(corinne,1) :- !.
enfants(X, 0).           enfants(X, 0).
```

- Soit la question : `enfants(helene, N).`

  N=3 et N=0 N=3

- Soit la question : `enfants(helene, 0).`

  Yes Yes

- Le programme correct serait :

```
enfants(helene,N) :- !, N=3.
```


La coupure en résumé

- En résumé, les utilités du coupure sont :
  - Eliminer les points de choix menant à des échecs certains
  - Supprimer certains tests d'exclusion mutuelle dans les clauses
  - Permettre de n'obtenir que la première solution de la démonstration
  - Assurer la terminaison de certains programmes
  - Contrôler et diriger la démonstration

La négation (1)

- Problème de la négation et de la différence en Prolog
  - Pas de moyen en Prolog de démontrer qu'une formule n'est pas déductible.
- La **négation** par l'échec
  - Si *F* est une formule, sa négation est notée *not(F)*
  - Prolog pratique la **négation par l'échec**
    - Pour démontrer *not(F)*, Prolog va tenter de démontrer *F*
    - Si la démonstration de *F* échoue, Prolog considère que *not(F)* est démontrée
  - Pour Prolog, *not(F)* signifie que la formule F n'est pas démontrable, et non que c'est une formule fausse.
    - **L'hypothèse du monde clos**.

La négation (2)

- p(X) est vrai si X vérifie la propriété p
- Comment écrire not(p(X)), qui est vrai si p(X) est faux ?

```
not(p(X)) :- p(X), !, fail.
not(p(X)).
```

- Si p(X) est vrai, retourne Faux et ne déclenche pas la règle 2 ; sinon retourner Vrai.

Les ensembles de solutions

- **findall/3**, **bagof/3**, **setof/3** :
  - La résolution Prolog peut toutes les solutions satisfaisant un ensemble de buts
  - Mais lorsqu'une nouvelle solution est générée, la solution précédente est perdue
  - Les prédicats bagof, setof et findall permettent de collecter ses solutions dans une liste

Le prédicat **findall/3**

findall(+Motif, +But, -Liste)

- Il créé une liste des instanciations de *Motif* par retours arrières sur *But* et unifie le résultat dans *Liste*. Si le *But* n'a pas de solution, findall retournera la liste vide: **[]**.
- Exemple :

```
eleve(françois, 2, info).
eleve(isa, 2, info).
eleve(françois, 3, math).
?- findall(X, eleve(X, 2, info), B).
> B = [françois, isa].
?- findall(X, eleve(X,Y,Z),B).
> B = [françois, isa, françois].
```


Le prédicat **bagof/3**

bagof(+Motif, +But, -Liste)

- Il se comporte comme `findall`, à la différence que :
  - `bagof` échoue si *But* n'a pas de solution
  - a un comportement différent concernant les variables libres de Motif
  - Exemple :

```
eleve(françois, info, 2). eleve(isa, info, 2).
eleve(françois, info, 3). eleve(paul, math, 3).
masculin(françois). masculin(paul). feminin(isa).
?-bagof(X, eleve(X, info, 2), B).
> B = [francois, isa].
?-bagof(X, eleve(X, info, Y), B).
> B = [françois, isa], Y=2;
> B = [françois], Y=3;
```

`setof(+Motif, +But, -Liste)`

- idem que bagof, sauf que la liste est triée et les doublons exclus

Prédicats prédéfinis

- Vérifier **le type d'un terme**
  - `integer(X)` <=> est-ce que X est un entier ?
  - `float(X)` <=> est-ce que X est un flottant ?
  - `number(X)` <=> est-ce que X est un nombre ?
  - `atom(X)` <=> est-ce un atome ?
  - `atomic(X)` <=> est-ce un atome ou un nombre ?
  - `var(X)` <=> est-ce une variable non instanciée ?
  - `nonvar(X)` <=> (le contraire de var) <=> X est un terme autre qu'une variable, ou une variable instancié.
  - `compound(X)` <=> est-ce un terme composé ?
  - `ground(X)` <=> est-ce un terme clos ?
- Prédicats de **manipulation des termes**
  - functor(Terme, Foncteur, Arité)
  - arg(N, Terme, X) <=> unifie X à l 'argument numéro N de terme

RUE DES PROBLÈMES

Géant, nain et cie

1. *Un géant est une personne de grande taille et de grande force.*
2. *Un nain est une personne de petite taille.*
3. *Un joueur est une personne qui aime la compétition.*
4. *Un homme d'affaires est un joueur, a une personnalité non amicale et n'a pas d'amis.*
5. *Une personne de petite taille a une personnalité amicale.*
6. *Une personne dont la personnalité est amicale a au moins un ami.*
7. *Un solitaire est une personne qui n'a pas d'ami.*

- Montrer qu'un nain a au moins un ami et qu'un homme d'affaires est un solitaire.

RUE DES PROBLÈMES

Les règles des animaux

1. *Si l'animal a des poils, alors c'est un mammifère.*
2. *Si l'animal donne du lait, alors c'est un mammifère.*
3. *Si l'animal a des plumes, alors c'est un oiseau.*
4. *Si l'animal vole, et s'il pond des œufs, alors c'est un oiseau.*
5. *Si l'animal est un mammifère, et s'il mange de la viande, alors c'est un carnivore.*
6. *Si l'animal est un mammifère, et s'il a des dents pointues, et s'il a des griffes, et ses yeux pointent vers l'avant, alors c'est un carnivore.*
7. *Si l'animal est un mammifère, et s'il a des sabots, alors c'est un ongulé .*
8. *Si l'animal est un mammifère, et s'il rumine, alors c'est un ongulé .*
9. *Si l'animal est un carnivore, et s'il est de couleur fauve, et s'il a des taches sombres, alors c'est un guépard.*
10. *Si l'animal est un carnivore, et s'il est de couleur fauve, et s'il a des rayures noires, alors c'est un tigre.*
11. *Si l'animal est un ongulé , et s'il a de longues pattes, et s'il a un long cou, et s'il est de couleur fauve, et s'il a des taches sombres, alors c'est une girafe.*
12. *Si l'animal est un ongulé, et s'il est blanc, et s'il a des rayures noires, alors c'est un zèbre.*
13. *Si l'animal est un oiseau, et s'il ne vole pas, et s'il a de longues pattes, et s'il a un long cou, et s'il est blanc et noir, alors c'est une autruche.*
14. *Si l'animal est un oiseau, et s'il ne vole pas, et s'il nage, et s'il est blanc et noir, alors c'est un manchot.*
15. *Si l'animal est un oiseau, et s'il vole bien, alors c'est un albatros.*

RUE DES PROBLÈMES

Un petit tour à Roland Garros

- On considère le programme Prolog suivant :

```
soleil.
lobes(kafelnikov).
petit(pioline).
meilleur(sampras).
gagnant(X,Y) :- meilleur(X).
gagnant(X,Y) :- plus_malin(X,Y).
gagnant(X,Y) :- plus_chanceux(X,Y).
plus_malin(X,_Y)   :- soleil, lobes(X).
plus_malin(X,Y)    :- petit(Y), lobes(X).
plus_chanceux(_X,pioline).
```

- Requête : `?- gagnant(X,_Y).`

RUE DES PROBLÈMES

Dès que le vent soufflera

- On considère le programme Prolog suivant :

```
navigue(garlier,mardi).
navigue(desjoyeux,mercredi).
navigue(desjoyeux,jeudi).
navigue(ledam,J).
temps(mardi,calme).
temps(mercredi,calme).
temps(jeudi,grostemps).
aime(ledam,grostemps).
aime(X,calme).
chance(desjoyeux,mercredi).
gagne(X,J) :- navigue(X,J), temps(J,T),
  aime(X,T).
gagne(X,J) :- navigue(X,J), chance(X,J).
```

- Requête : `?- gagne(X,mercredi).`