

# IA02 Intelligence Artificielle PROgrammation LOGique

Antoine Jouglet (antoine.jouglet@hds.utc.fr)

## But du cours

- Comprendre le fonctionnement de Prolog
- Maîtriser les concepts de base de programmation Prolog
- Connaître la syntaxe de Prolog
- Savoir programmer des algorithmes de recherche élémentaires
- Quand utiliser Prolog ?

## Plan du Cours

- Une introduction a Prolog ;
- Représentation des connaissance ;
- Les calculs et les I/O sous Prolog ;
- L'effacement ;
- La récursivité ;
- La coupure (terme d'arrêt) ! ;
- Les listes ;
- Les structures complexes ;
- La résolution de problèmes avec Prolog ;
- *etc...*

## Une Introduction à Prolog

4

- Historique ;
- Bibliographie ;
- Un autre mode de programmation ;
- Des domaines d'applications multiples ;
- Base Théorique (rappel) ;

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Historique

5

- Le PROLOG est né d'un projet, dont le but n'était pas de faire un langage de programmation mais de traiter les langages naturels.
- Ce projet a donné naissance à un PROLOG préliminaire à la fin de 1971, et un PROLOG plus définitif fin 1972.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Historique

6

- Travaux théoriques fondateurs :
    - Le principe de Résolution de Alan Robinson ;
    - Les clauses de Horn.
  - Acteurs principaux :
    - Alain Colmerauer (Université de Aix-Marseille)
    - Philippe Roussel (Université de Aix-Marseille)
- puis
- Robert Kowalski (Université d'Edimburgh)

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Historique

7

- Début 70's.
- Colmerauer et Roussel travaille sur le traitement du langage naturel en s'appuyant sur la démonstration automatique.
- Travaux théoriques de Kowalski : utilisation de la logique formelle comme base d'un langage de programmation.
- 1972 : premier **interpréteur Prolog**  
A. Colmerauer et P. Roussel (Univ. Aix-Marseille).
- 1977 : premier **compilateur Prolog**  
D. Warren (Univ. Edimburgh).

IA02

Programmation Logique

## Bibliographie

8

- Quelques supports de cours
  - L. Sterling & E. Shapiro, *The Art of Prolog*, MIT Press
  - P. Bellot, *Objectif Prolog*, Masson
  - N. Ford, *Programmer en Prolog*, Dunod
  - M. Bramer, *Logic Programming with Prolog*, Springer

IA02

Programmation Logique

## Bibliographie

9

Une liste très complète sur le Prolog :

[www.cetus-links.org/oo\\_prolog.html](http://www.cetus-links.org/oo_prolog.html)

Interpréteurs prolog commerciaux :

- B-Prolog : [www.probp.com](http://www.probp.com)
- SICStus-Prolog : [www.sics.se/isl/sicstuswww/site/index.html](http://www.sics.se/isl/sicstuswww/site/index.html)

Interpréteurs prolog freeware :

- SWI Prolog : [www.swi-prolog.org](http://www.swi-prolog.org)
- **GNU Prolog** : [www.gprolog.org](http://www.gprolog.org)

Et bien d'autres : Amzi! Prolog, Ciao Prolog, Open Prolog, PD Prolog, Turbo Prolog, Visual Prolog, W-Prolog, Yap Prolog, etc.

IA02

Programmation Logique

## Fonctionnement

10

- Prolog est **principalement un langage interprété**, (l'ordinateur exécute immédiatement les commandes saisies par l'utilisateur) : le code source est immédiatement traduit en code machine.
- **pratique** dans le cas du prototypage (il est possible de tester le code sans devoir recompiler à chaque fois).
- **lent** et utilise plus de mémoire.
- Dans le cas d'un **langage compilé**, le code source est traduit dans son intégralité en un programme exécutable. Le développement est plus long, mais un programme exécutable est nettement plus rapide.
- Prolog offre un bon **compromis** puisqu'il est possible de le compiler lorsque le développement est terminé.

IA02

Programmation Logique

## Fonctionnement

11

Pratiquement, Prolog consiste :

- en un interprète où on peut saisir des expressions après le prompt ( ?- ),
- et un moteur d'inférence qui teste si ces expressions sont vraies ou fausses (par unification) en parcourant une base de faits et de règles (par backtracking).

IA02

Programmation Logique

## Paradigmes de Programmation ???

12

- **Paradigme** : « Modèle théorique de pensée qui oriente la recherche et la réflexion scientifiques »
- **Paradigme de programmation** :  
Un outil de pensée qui oriente la façon d'analyser, de concevoir et de coder un programme :
  - programmation impérative,
  - la programmation fonctionnelle,
  - la programmation déclarative,
  - la programmation orientée objet.

IA02

Programmation Logique

## Caractéristiques de prolog

13

- Langage purement déclaratif : on spécifie les **propriétés** du résultat du programme et non pas le **processus** pour arriver à ce résultat (aspect **opérationnel**). Facilité de compréhension – facilité d'écriture.
- Nouveaux concepts de programmation
  - Pas de procédure
  - Pas de test, pas de boucle
  - Pas d'affectation
  - Les instructions sont remplacées par des règles qui sont "interprétées" par un moteur d'inférences
  - Parcours d'arbres
- La syntaxe est simple et élégante mais il faut savoir manipuler correctement la **logique**, la **récurtivité** et le **backtrack**.

IA02

Programmation Logique

## Le Pari de PROLOG

14

- Langage de haut niveau (qui peut s'avérer inefficace) mais qui permet de coder succinctement des programmes qui dans un langage "classique" nécessiteraient un lourd développement.
- Pari gagné... Les domaines d'application sont nombreux :
  - Démonstration automatique
  - Résolution de problèmes combinatoires
  - Base de données
  - Systèmes experts
  - Compilation
  - Traitement des langages

IA02

Programmation Logique

## Structure Tripartiste

15

- La base de faits
- La base de règles
- Le moteur d'inférence

Le **moteur d'inférence** ou de déduction permet de gérer le déroulement d'un raisonnement.

IA02

Programmation Logique

## La base de données et le moteur d'inférence

16

Le **moteur d'inférence** met en relation :

- La **base de faits**, ou les données de départ : constituent les informations élémentaires nécessaires au déroulement du raisonnement.
- La **base de connaissances** composée de règles de déduction (si ... alors ...)

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Le principe de résolution

17

(J.A Robinson)

$$\begin{array}{c} \neg P \vee Q \\ P \vee R \end{array} \quad \Bigg| \quad Q \vee R$$

- Le principe de résolution est **valide** ;
- Il généralise les autres règles.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Les clauses de Horn

18

- Prolog est basé sur l'écriture de **clauses de Horn** (clauses ayant au plus un littéral positif)

$$\neg P_1 \vee \neg P_2 \vee \neg P_3 \vee Q \quad (\equiv \quad P_1 \wedge P_2 \wedge P_3 \Rightarrow Q)$$

( $P_1 \wedge P_2 \wedge P_3$  est la queue, Q la tête)

- Une clause contenant une conclusion est une **règle** ;
- Une clause sans queue est un **fait** ou une **assertion**.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Propriétés du calcul propositionnel

19

- Le calcul des prédicats muni du principe de résolution et de l'unification est **complet**.
- $G$  est conséquence logique de  $H_1, H_2, \dots, H_n$  ssi il existe une réfutation par résolution des clauses issues de  $H_1, H_2, \dots, H_n$  et  $\neg G$
- Un ensemble  $S$  de clauses est **insatisfaisable** ssi
$$S \xrightarrow{\text{résolution}} \{\}$$
- Démonstration : par l'absurde = méthode de résolution par **réfutation**

$$S \vdash C \text{ ssi } S \wedge \{\neg C\} \xrightarrow{\text{résolution}} \{\}$$

IA02

Programmation Logique

## Interprétation procédurale des Clauses de Horn

20

Kowalski :

$Q$  si  $P_1$  et  $P_2$  et ... et  $P_n$   
peut être lu comme une procédure d'un programme récursif où  $Q$  est la tête de la procédure et les  $P_i$  sont son corps.

$Q$  est vrai si  $P_1$  et  $P_2$  et ... et  $P_n$  sont vrais.

Pour résoudre (exécuter)  $Q$ ,  
résoudre (exécuter)  $P_1$  et  $P_2$  et ... et  $P_n$ .

IA02

Programmation Logique

## Plan du cours

21

- Une introduction a Prolog
- Représentation des connaissances**
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## Représentation des connaissances

22

- Syntaxe d'un programme Prolog ;
- La sémantique des règles ;
- Questionner la base de données Prolog ;

IA02

Programmation Logique

---

---

---

---

---

---

---

## Syntaxe PROLOG

23

- Un **programme** est un ensemble de **prédicats** ;
- Un **prédicat** est défini par un ensemble de **clauses** ;
- Une **clause** est soit un **fait** soit une **règle**.

IA02

Programmation Logique

---

---

---

---

---

---

---

## Syntaxe PROLOG : faits

24

La syntaxe d'un **fait** est :

**pred**(**arg**<sub>1</sub>, **arg**<sub>2</sub>, ..., **arg**<sub>N</sub>).

- **pred** est le nom du prédicat (il doit commencer par une minuscule) ;
- **N** est l'arité du fait ;
- **arg**<sub>1</sub>, ..., **arg**<sub>N</sub> sont les arguments (les termes) ;
- Un argument est soit une **variable**, soit une **constante**, soit une structure plus complexe (...) ;
- Le caractère "." est la marque syntaxique de la fin d'une clause.

IA02

Programmation Logique

---

---

---

---

---

---

---



## Règle

25

- Une **règle** a la même syntaxe qu'un fait auquel on a ajouté une queue :

`pred(arg1, arg2, ..., argN) :- but1, but2, ..., butN.`

- le caractère ":-" indique le début de la queue de la règle.
- `but1, but2, ..., butN` est la liste des buts à effacer.

Un but est :

- soit le caractère de coupure "!" ;
- soit un groupe syntaxique ayant la même structure que la tête d'une clause.
- les buts sont séparés par des virgules qui expriment la conjonction (ET)
- le caractère "." est la marque syntaxique de la fin d'une clause.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Syntaxe PROLOG : règles

26

Les expressions suivantes sont équivalentes.

`A :- B1, B2, ..., Bn.` et `A :- B1,  
B2,  
...,  
Bn.`

NB:

La conclusion **A** (la tête de la règle) peut être utilisée comme but d'une autre règle ou de cette même règle.

**A** et **B<sub>i</sub>** sont définis par des constantes, des variables ou des prédicats.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Syntaxe PROLOG : règles

27

- Exemples

`% faits  
homme(socrate).  
femme(lola).`

`% règles  
mortel(X) :- homme(X).  
jeune_femme(X) :- jeune(X), femme(X).`

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Les Termes

28

⇒ Objets manipulés par un programme Prolog  
(les "données" du programme)

On distingue trois sortes de termes:

- Les **variables** ;
- Les **termes élémentaires** (constantes) ;
- Les **termes composés** (de termes simples).

IA02

Programmation Logique

## Les Termes

29

- Les **variables** : objets inconnus de l'univers.

Chaîne alpha-numérique commençant par une  
**majuscule** ou par **\_**.

Variable anonyme : « **\_** » objet dont on ne souhaite  
pas connaître la valeur.

Une variable représente toujours le même objet tout au  
long de sa durée de vie et ne peut pas changer de  
valeur.

IA02

Programmation Logique

## Les Termes

30

- Les **termes élémentaires** (constantes) : objets simples  
connus de l'univers.

3 sortes :

- les **nombres** : entiers ou flottants ;
- les **identificateurs** (atomes) : chaîne alpha-  
numérique commençant par une **minuscule** ;  
ex: toto, ax12, jean\_Paul\_2
- les chaînes de caractères entre quotes (ex.  
'Toto');

- Les **termes composés** (de termes simples) : objets  
composés (structurés) de l'univers.

ex: adresse(18, 'rue des lilas', Ville).

IA02

Programmation Logique

## Syntaxe

31

- Considérons l'énoncé
  - Socrate est un homme
  - Tout homme est mortel
  - Socrate est-il mortel ?

| Calcul des prédicats                                      | Prolog                              |
|---|-------------------------------------|
| $\exists x, \text{homme}(x)$                              | <code>homme(socrate).</code>        |
| $\forall x, \text{homme}(x) \rightarrow \text{mortel}(x)$ | <code>mortel(X) :- homme(X).</code> |
|   | <code>?- mortel(socrate).</code>    |

IA02

Programmation Logique

## Des exemples de clauses

32

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

professeur(pierre, jean).

est_eleve(X, Y):- professeur(Y, X).

jeune_femme(X) :- jeune(X),
                  femme(X).
```

IA02

Programmation Logique

## Sémantique des règles

33

- Une règle doit toujours pouvoir s'exprimer clairement (il est facile d'écrire n'importe quoi).
- Il faut choisir une sémantique et s'y tenir :
  - Que veut dire l'assertion `professeur(pierre, jean)?`  
*i.e.*, pierre est le professeur de jean ou jean est le professeur de pierre ?
  - Cohérence des nouvelles règles :  
Que signifie `eleve(X, Y) :- professeur(Y, X).` ?

IA02

Programmation Logique

## Un programme PROLOG

34

- Un programme Prolog est constitué d'une suite de clauses regroupées en paquets (**ordre des paquets non significatif**).
- Chaque paquet définit un prédicat : constitué d'un ens. de clauses dont l'atome de tête a le même symbole de prédicat (**ordre significatif**). *Intuitivement*, deux clauses d'un même paquet sont liées par un « OU » logique.

- Par exemple, le prédicat `personne` défini par les deux clauses:

```
personne(X) :- homme(X).
```

```
personne(X) :- femme(X).
```

= "pour tout X, `personne(X)` est vrai si `homme(X)` est vrai ou `femme(X)` est vrai".

IA02

Programmation Logique

## Exécution d'un programme PROLOG

35

- Exécution = consiste à poser une **question** à l'interprète PROLOG.
- Une **question** (ou but ou activant) est une suite d'atomes logiques séparés par des virgules.
- La **réponse** de Prolog est « yes » si la question est une conséquence logique du programme, ou « no » si la question n'est pas une conséquence logique du programme.

IA02

Programmation Logique

## Exécution d'un programme PROLOG

36

- Une **question** peut comporter des variables, quantifiées existentiellement.
- **Réponse** : l'ensemble des valeurs des variables pour lesquelles la question est une conséquence logique du programme.
- Ex : la question `?- pere(toto,X).`  
« est-ce qu'il existe un `x` tel que `pere(toto,X)` soit vrai ? »  
Réponse : si **yes**, `x` s'unifie avec une des valeurs qui vérifient cette relation.  
= un des enfants de toto... si toto est effectivement père.

IA02

Programmation Logique

## Quelques propriétés

37

- la recherche réalisée par Prolog est une recherche en **profondeur d'abord**.
- on peut obtenir plusieurs solutions pour une même requête :  
*Utiliser « ; » pour avoir toutes les réponses (toutes les unifications possibles).*
- les seuls résultats possibles : **yes** ou **no**
  - pas de fonction, les réponses sont obtenues par **unification** uniquement

IA02

Programmation Logique

## Quelques prédicats de base ...

38

halt est un prédicat qui renvoie toujours vrai et qui permet de sortir de l'interpréteur.

Equivalent à ^D ou end\_of\_file.

Soit un fichier "mon\_prog.pl". Les prédicats suivants permettent de charger mon\_prog.pl en mémoire.

```
consult(mon_prog).  
consult('mon_prog').  
consult('mon_prog.pl').  
[mon_prog].  
['mon_prog'].  
['mon_prog.pl'].
```

IA02

Programmation Logique

## Quelques prédicats de base ...

39

```
?-working_directory(X).
```

permet d'unifier le répertoire courant de travail avec X.

```
?-change_directory('d:\\prolog\\').
```

ou

```
?-change_directory('d:/prolog/').
```

permet de faire en sorte que 'd:\\prolog\\' soit le répertoire courant de travail (où trouver ses programmes...).

```
?-directory_files('d:\\prolog\\',L).
```

permet d'unifier les fichiers qui appartiennent au répertoire 'd:\\prolog\\' avec la variable L.

IA02

Programmation Logique

## Aujourd'hui, Demain, ...

40 Soient les assertions suivantes : ( 'aujourd'hui.pl' )

```
aujourd'hui(mardi).  
demain(lundi,mardi).  
demain(mardi,mercredi).  
demain(mercredi,jeudi).  
demain(jeudi,vendredi).  
demain(vendredi,samedi).  
demain(samedi,dimanche).
```

- Comment poser la question : « Quel jour sommes nous ? ».
- Comment compléter le programme prolog pour poser les questions :
  - Quel jour sommes nous demain ?
  - Quel jour étions nous hier ?

IA02

Programmation Logique

## Les villes

41 Soient les assertions suivantes :

```
habite(olivier,paris).  
habite(delphine,bordeaux).  
habite(francois,frankfort).  
habite(estelle,lille).  
habite(claire,casablanca).
```

- Décrire d'autres assertions du même type.
- Comment poser les questions :
  - « qui habite où ? »
  - « où habite olivier ? »
  - « qui habite lille ? »Décrire d'autres assertions
- On ajoute des assertions du type capitale(paris).  
Créer une règle qui permet de trouver les personnes qui habitent une capitale.

IA02

Programmation Logique

## Les liens familiaux

42

Détailler une base de données Prolog permettant d'exprimer tous les liens familiaux (homme, femme, enfant, fils, tante, cousin, arrière grand-mère ...).

IA02

Programmation Logique

## Plan du cours

43

- Une introduction a Prolog
- Représentation des connaissance
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## Calculs et I/O sous PROLOG

44

- L'arithmétique sous Prolog,
  - La primitive "is"
  - Les opérateurs de comparaison
- Les Entrées / Sorties

IA02

Programmation Logique

## Les calculs : is / 2

45

- Comment intégrer l'évaluation d'expressions numériques au schéma logique de Prolog?
- Utilisation du prédicat prédéfini "is / 2"  
`X is <expression arithmétique>`
- Quelques exemples

|                                   |                                   |
|-----------------------------------|-----------------------------------|
| <code>?- X is 2 + 2.</code>       | <code>?- X is 3.2 * 4.0.</code>   |
| <code>X = 4</code>                | <code>X = 12.8</code>             |
| <code>?- X is 3 * (4 + 2).</code> | <code>?- X is (8 / 4) / 2.</code> |
| <code>X = 18</code>               | <code>X = 1</code>                |
- D'autres fonctions arithmétiques : `sin()`, `cos()`, `asin()`, `acos()`, `exp()`, `ln()`, `round()`, `integer`, `float`, ...

IA02

Programmation Logique

## Les calculs : is / 2

46

- Attention : toutes les variables de l'expression doivent avoir été unifiées avec **une valeur numérique** avant l'évaluation de l'expression.

IA02

Programmation Logique

---

---

---

---

---

---

---

## Les calculs : is / 2

47

- Le premier argument de **is** peut éventuellement être une constante ou une variable déjà unifiée avec une valeur numérique.
- Dans ce cas, si le résultat de l'expression arithmétique (le 2ème argument de **is**) est égal au premier argument de **is**, le prédicat est vrai, sinon il échoue (il est faux).

IA02

Programmation Logique

---

---

---

---

---

---

---

## Les calculs : is / 2

48

Résumé : le second argument du prédicat **is** est **évalué** et sa valeur est alors **unifiée** avec le premier argument :

- Si le premier argument est une variable non unifiée, elle est alors unifiée avec l'évaluation du second argument et le prédicat réussit (**vrai**).
- Si le premier argument est une valeur numérique ou une variable unifiée avec une valeur numérique, l'unification réussit seulement si les deux valeurs sont égales (**vrai**); sinon elle échoue (**faux**).

IA02

Programmation Logique

---

---

---

---

---

---

---



## Les comparaisons

49

Prolog permet d'effectuer des comparaisons entre deux expressions  $E1$  et  $E2$  :

$E1 > E2$ ,  $E1 < E2$ ,  $E1 \geq E2$ ,  $E1 \leq E2$ ,  
 $E1 =:= E2$ ,  $E1 \neq E2$ .

Les expressions  $E1$  et  $E2$  sont évaluées. Les prédicats échouent ou réussissent suivant le résultat de la comparaison.

Attention, toutes les variables de  $E1$  et  $E2$  doivent avoir été préalablement unifiées avec une valeur numérique.

IA02

Programmation Logique

## Les calculs (exercices)

50

- `val_abs(X, Y)`.
- `maximum(X, Y, MAX)`. (maximum de 2 termes, de 3 termes)
- `minimum(X, Y, MIN)`. (minimum de 2 termes, de 3 termes)
- `prix_ttc(HT, TTC)`.
- `solve(A,B,C,X)`. Résolution d'une équation du second degré.

IA02

Programmation Logique

## Les sorties

51

- Le prédicat `write/1` écrit son argument sur la console.
- Le prédicat `nl/0` effectue un saut de ligne
- Le prédicat `tab/1` écrit sur la console un nombre d'espaces spécifiés par son premier argument
- Exemple

|   |   |
|---|---|
| <pre>voir1(X, Y) :-<br/>    write(X),<br/>    write(' and '),<br/>    write(Y).<br/>?- voir1(4, socrate).<br/>4 and socrate</pre> | <pre>voir2(X, Y) :-<br/>    write(X),<br/>    nl,<br/>    write(Y).<br/>?- voir2(4, socrate).<br/>4<br/>socrate</pre> |
|---|---|

IA02

Programmation Logique

## Les entrées

52

- Le prédicat `read(X)` lit tous les caractères de l'entrée standard jusqu'au premier "." rencontré puis unifie l'expression rencontrée avec `X`.
- Exemple:  

```
?- read(X).  
1 is 4 - 3.  
X = 1 is 4 - 3
```

IA02

Programmation Logique

## Hello World

53

- Ecrire un programme qui permet de demander le prénom de l'utilisateur puis de dire « bonjour » à l'utilisateur ...

IA02

Programmation Logique

## Plan du cours

54

- Une introduction à Prolog
- Représentation des connaissances
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## L'effacement

55

- L'interpréteur
- Principe de base
- Chaînage avant / arrière
- Algorithme d'effacement
- Exemple

IA02

Programmation Logique

## L'interpréteur

56

- Consiste à rechercher toutes les solutions, impliquées par une ou plusieurs questions, en appliquant toutes les règles et assertions.
- Interpréteur Prolog =
  - Démonstrateur de théorème.
  - Moteur d'inférence d'ordre 1.

IA02

Programmation Logique

## L'interpréteur

57

- Question = conjonction de relations à satisfaire, appelées buts.
- Réponses = **non** ou **oui** sous les contraintes que doivent satisfaire les variables, pour respecter les relations exprimées par des règles du programme.
- Fonctionnement non déterministe : le programmeur ne contrôle ni les valeurs affectées aux variables ni le déroulement du programme.

IA02

Programmation Logique

## Principe de base

58

- Programmeur Prolog = analyste purement logique, exprimant toutes les contraintes d'un problème.  
⇒ ne cherche pas à décrire un algorithme donnant une solution.
- Soit une question = conjonction de buts  $q_1 \ q_2 \ \dots \ q_n$   
l'interpréteur va chercher à satisfaire tous ces buts séquentiellement.
- Quand un but est satisfait, il est effacé de la liste.

IA02

Programmation Logique

## Principe de base

59

- Deux types de règles
  - Assertions  $p.$
  - Règles  $p :- c_1 \ c_2 \ \dots \ c_n.$
- Moteur en chaînage arrière
  - liste de buts initiaux
  - nouveaux buts ajoutés
  - backtrack pour avoir toutes les autres solutions (défaire toutes les affectations, faites lors du choix précédent)

IA02

Programmation Logique

60

## Signification d'un programme Prolog

IA02

Programmation Logique

## Définitions préliminaires

61

- **Substitution** : fonction de l'ensemble des variables dans l'ensemble des termes.

**ex** :  $s = \{ \langle X, Y \rangle, \langle Z, f(a, Y) \rangle \}$

Par extension, une substitution peut être appliquée à un atome logique.

**ex** :  $s(p(X, f(Y, Z))) = p(s(X), f(s(Y), s(Z))) = p(Y, f(Y, f(a, Y)))$

- **Instance** : Une instance d'un atome logique  $A$  est le résultat  $s(A)$  de l'application d'une substitution  $s$  sur  $A$ .  
**ex** : `pere(toto, paul)` est une instance de `pere(toto, X)`.

IA02

Programmation Logique

## L'unification sous PROLOG

62

- L'opérateur d'unification sous PROLOG est :

=

- 2 atomes s'unifient si ils sont identiques.
- 2 termes composés s'unifient si ils ont le même foncteur (nom du prédicat) et la même arité et si leurs arguments s'unifient 2 à 2 en partant de la gauche.
- 2 nombres s'unifient si ils sont les mêmes.
- 2 variables non unifiées à une valeur s'unifient toujours.
- 1 variable non unifiée et 1 terme s'unifient toujours : la variable prend la valeur du terme.
- Une variable unifiée est traitée comme la valeur avec laquelle elle est unifiée.
- Cas des listes... (plus tard)
- Toutes les autres tentatives d'unification échouent si on les tente.

IA02

Programmation Logique

## Dénotation d'un programme PROLOG

63

- La **dénotation** d'un programme Prolog  $P$  est l'ensemble des atomes logiques qui sont des conséquences logiques de  $P$ .
- La « **réponse** » de Prolog à une question est l'**ensemble des instances** de cette question qui font partie de la dénotation.
- Cet ensemble peut être "calculé" par une **approche ascendante**, dite en **chaînage avant** :
  - on part des faits (relations qui sont vraies sans condition),
  - on applique itérativement toutes les règles conditionnelles pour déduire de nouvelles relations ... jusqu'à ce qu'on ait tout déduit.

IA02

Programmation Logique

## Dénotation d'un programme PROLOG

64

Ex: P : parent(paul,jean). parent(jean,anne).  
parent(anne,marie). homme(paul). homme(jean).  
pere(X,Y):-parent(X,Y),homme(X).  
grand\_pere(X,Y):-pere(X,Z),parent(Z,Y).

- L'ens. des faits dans P est

E0={parent(paul,jean), parent(jean,anne),  
parent(anne,marie), homme(paul), homme(jean) }

- À partir de E0 et P, on déduit l'ens. des nouvelles relations vraies

E1={pere(paul,jean), pere(jean,anne)}

- À partir de E0, E1 et P, on déduit l'ens. des nouvelles relations vraies

E2={grand\_pere(paul,anne), grand\_pere(jean,marie)}

- À partir de E0, E1, E2 et P, on ne peut plus rien déduire de nouveau.

- L'union de E0, E1 et E2 constitue la dénotation (l'ensemble des conséquences logiques) de P.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Dénotation d'un programme PROLOG

65

- la dénotation d'un programme est **souvent un ensemble infini** et n'est donc pas calculable de façon finie.

Ex: P :  
nombre(0).  
nombre(X) : - nombre(Y), X is Y+1.

L'ens. des atomes logiques vrais sans condition dans P est

E0={ nombre(0) }

à partir de E0 et P, on déduit

E1={ nombre(1) }

à partir de E0, E1 et P, on déduit

E2={ nombre(1), nombre(2) }

à partir de E0, E1, E2 et P, on déduit

E3={ nombre(1), nombre(2), nombre(3) }

etc ...

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Signification opérationnelle

66

- D'une façon générale, on ne peut pas calculer l'ensemble des conséquences logiques d'un programme par **l'approche ascendante** : ce calcul est trop coûteux ou infini.
- En revanche, on peut démontrer qu'un but (composé d'une suite d'atomes logiques) est une conséquence logique du programme, en utilisant une **approche descendante**, dite en **chaînage arrière**.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Chaînage arrière

67 Pour prouver un but composé d'une suite d'atomes logiques  
ex: But = [A\_1, A\_2, ..., A\_n]

- l'interprète Prolog commence par prouver le premier de ces atomes logiques (A\_1) : il cherche une clause dans P dont l'atome de tête s'unifie avec le premier atome logique à prouver  
ex: la clause A'\_0 :- A'\_1, A'\_2, ..., A'\_r telle que  $\text{upg}(A_1, A'_0) = s$
- Puis l'interprète Prolog remplace le premier atome logique à prouver (A\_1) dans le but par les atomes logiques du corps de la clause, en leur appliquant la substitution (s). Le nouveau but à prouver devient  
But = [s(A'\_1), s(A'\_2), ..., s(A'\_r), s(A\_2), ..., s(A\_n)]
- **L'interprète Prolog recommence alors ce processus, jusqu'à ce que le but à prouver soit vide**, c'est à dire jusqu'à ce qu'il n'y ait plus rien à prouver.
- A ce moment, l'interpréteur **Prolog a prouvé le but initial**.
- Si le but initial comportait des variables, il affiche la valeur de ces variables obtenue en leur appliquant les substitutions successivement utilisées pour la preuve.

IA02

Programmation Logique

## Algorithme d'effacement

68

- A chaque étape de l'algorithme, Prolog gère une liste (une conjonction) de buts à satisfaire ;
- Moteur en chaînage arrière ;
- Pour satisfaire chaque but, les règles sont appliquées
  - Le but courant est remplacé par une conjonction de buts impliqués par **l'une** des règles,
  - Lorsque **la liste de buts est vide**, on a trouvé une **solution**,
  - Si **l'un des buts n'est pas effaçable**, on **remet en cause** (backtrack...) le choix de l'une des règles appliquées.

IA02

Programmation Logique

## Algorithme d'effacement

69

Ce processus est résumé par la fonction suivante :

```
procedure prouver(But: liste d'atomes logiques ){
  si But = [ ] alors {
    /* le but initial est prouvé */
    /* afficher les valeurs des variables du but initial */
  }sinon {
    soit But = [A_1, A_2, ..., A_n];
    pour toute clause (A'_0 :- A'_1, A'_2, ..., A'_r) de P {
      s <- upg(A_1, A'_0);
      si s != echec alors {
        prouver([s(A'_1), ..., s(A'_r), s(A_2), .. s(A_n)]);
      }
    }
  }
}
```

IA02

Programmation Logique

## Bactracking : retour en arrière

70

Le **bactracking** est le processus qui consiste à revenir en arrière sur un but précédant pour essayer de le re-satisfaire, c'est-à-dire pour essayer de trouver un autre moyen de le satisfaire.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Chaînage arrière

71

- Il existe généralement plusieurs clauses dans le programme Prolog dont l'atome de tête s'unifie avec le premier atome logique à prouver.
- L'interprète Prolog va successivement répéter ce processus de preuve pour chacune des clauses candidates.  
⇒ plusieurs réponses à un but. Utilisation de ";".
- Quand on pose une question à l'interprète Prolog, celui-ci exécute dynamiquement l'algorithme précédent.
- L'arbre constitué de l'ensemble des appels récursifs à la procédure prouver (but) est appelé **arbre de recherche**.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Chaînage arrière

72

### Remarques

- la stratégie de recherche n'est pas directement complète : on peut avoir une suite infinie d'appels récursifs,
- la stratégie de recherche dépend
  - de l'ordre de définition des clauses dans un paquet  
on considère les clauses selon leur **ordre** d'apparition dans le **paquet**
  - de l'ordre des atomes logiques dans le corps d'une clause  
on prouve les atomes logiques selon leur **ordre** d'apparition dans la **clause**

IA02

Programmation Logique

---

---

---

---

---

---

---

---



## Exemple

73

- Base de faits :
  - R1 viande(gril).
  - R2 viande(poulet).
  - R3 poisson(bar).
  - R4 poisson(sole).
  - R5 plat(P) :- viande(P).
  - R6 plat(P) :- poisson(P).
- Question :  
plat(P), diff(P, gril).

IA02

Programmation Logique

---

---

---

---

---

---

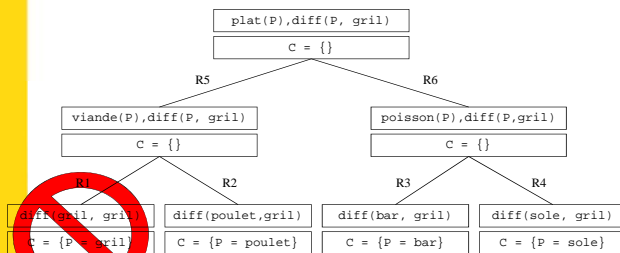
---

---

## Exemple

74

- R1 viande(gril).
- R2 viande(poulet).
- R3 poisson(bar).
- R4 poisson(sole).
- R5 plat(P) :- viande(P).
- R6 plat(P) :- poisson(P).



IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Plan du cours

75

- Une introduction a Prolog
- Représentation des connaissance
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## La récursivité en PROLOG

76

- Élément fondamental de Prolog.
- Deux exemples d'utilisation.
- Simuler une boucle en Prolog.

IA02

Programmation Logique

## L'importance de la récursivité

77

- Le **même terme** peut à la fois apparaître dans la **tête** et dans la **queue** d'une même règle.
  - Moyen puissant de décrire des règles complexes
- En Prolog Pas de boucle, pas d'instruction de saut.
  - Tout est récursif !
- Attention, aux erreurs de conception.

IA02

Programmation Logique

## Des exemples de récursivité

78

- Factorielle.

```
fact(0, 1).
fact(N, Res) :- N > 0,
                M is N - 1,
                fact(M, Tmp),
                Res is Tmp * N.
```

  - Pourquoi " $N > 0$ " est-il très important ?
- Puissance ( $a^b$ ),
- Somme des  $n$  premiers carrés entiers,
- Suite de Fibonacci ( $u_0 = 0$ ,  $u_1 = 1$ ,  $u_n = u_{n-1} + u_{n-2}$ ),
- PGCD.

IA02

Programmation Logique

## Comment mimer une boucle

79

- On veut mimer `for I = X to I = Y do ...`

```
boucle(I, K) : - I <= K,  
    ...,  
    Nouveau_I is I + 1,  
    boucle(Nouveau_I, K).  
boucle(I, K) : - I > K.
```

- L'appel se fait alors par  
`boucle(X, Y).`

IA02

Programmation Logique

## Plan du cours

80

- Une introduction a Prolog
- Représentation des connaissance
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt) et prédicats de contrôle
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## Prédicats de contrôle du système

81

- La nécessité de contrôler le système
- Le fonctionnement de la coupure
- Des exemples
- La négation, la double négation

IA02

Programmation Logique

## Contrôler le système

82

Comment faire pour

- S'arrêter à la première solution trouvée ?
- "Séparer" deux recherches arborescentes ?
- Interdire le backtrack ?
- Contrôler le non-déterminisme de Prolog ?

Il existe un terme d'arrêt (une **coupure**): "!"

IA02

Programmation Logique

## La coupure !

83

- Le prédicat prédéfini « ! » (prononcer "cut") permet de changer le contrôle d'un programme et d'empêcher le retour en arrière.
- Les conséquences du **cut** sont d'améliorer l'efficacité et d'exprimer le déterminisme;

### ATTENTION !

- Il peut introduire des erreurs de programmation difficiles à détecter.
- Il faut donc l'utiliser avec précaution et seulement lorsque c'est vraiment utile.

IA02

Programmation Logique

## La coupure !

84

- Soit une clause  $P :- B_1, \dots, B_k, !, B_{k+2}, \dots, B_n$ .
- Si le but courant  $G$  s'unifie avec la tête  $P$  de la clause et si les buts  $B_1, \dots, B_k$  réussissent, la coupure a l'effet suivant :
  - toute autre clause pour  $P$  qui pourrait s'unifier avec  $G$  est ignorée par la suite,
  - Au cas où  $B_i$  échouerait pour  $i > k$ , la remontée ne se fait que jusque la coupure  $!$ ,
  - Les autres choix restant pour les calculs des  $B_i$ ,  $i \leq k$  sont élagués de l'arbre de recherche,
  - Si la remontée s'effectue jusqu'à la coupure, alors la coupure échoue et la recherche se poursuit avec le dernier choix fait avant que  $G$  n'ait choisi la clause.

IA02

Programmation Logique

## La coupure !

85

Conséquences :

- ! coupe toutes les clauses alternative en dessous de lui ;
- ! coupe toutes les solutions alternatives des sous - buts à gauche du cut ;
- En revanche, possibilité de retour arrière sur les sous buts à la droite du cut.

$P :- A_1, \dots, A_m.$

~~$P :- B_1, \dots, B_k, !, B_{k+2}, \dots, B_n.$~~

~~$P :- C_1, \dots, C_m.$~~

IA02

Programmation Logique

## La coupure !

86

- Contrôle du backtrack
  - Soit une fonction  $f$  dont une définition Prolog peut être :

$f(X, 0) :- X < 3.$

$f(X, 2) :- 3 \leq X, X < 6.$

$f(X, 4) :- 6 \leq X.$

- Que se passe-t-il si on pose la question ?
- ?–  $f(1, Y), Y > 2.$

IA02

Programmation Logique

?–  $f(1, Y), Y > 2.$

87

$f(X, 0) :- X < 3.$

$f(X, 2) :- 3 \leq X, X < 6.$

$f(X, 4) :- 6 \leq X.$

Effacement de  $f(1, Y)$  ?

Première règle :  $Y = 0$

Effacement de  $Y > 2$  ? :– **échec**

Deuxième règle : **échec**

Troisième règle : **échec**

IA02

Programmation Logique

## La coupure !

88

- On appelle **but père** le but ayant permis d'unifier la clause contenant la coupure (! cut)
- L'effet du **cut** est de **couper** tous les points de choix restant depuis le but père. Les autres alternatives restent en place

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- 3 <= X, X < 6, !.  
f(X, 4) :- 6 <= X.
```

- ?- f(1,Y), Y > 2.

IA02

Programmation Logique

## Si...alors...sinon

89

- Le **cut** peut servir à exprimer des **conditions mutuellement exclusives** et ainsi **simplifier** l'écriture.
- La clause suivant un **cut** peut être considérée comme un **sinon**

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- 3 <= X, X < 6, !.  
f(X, 4) :- 6 <= X.
```

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- X < 6, !.  
f(X, 4).
```

IA02

Programmation Logique

## Exemple

90

```
b(1).  
b(2).
```

```
?- b(X), b(Y).
```

```
X = 1, Y = 1 ;  
X = 1, Y = 2 ;  
X = 2, Y = 1 ;  
X = 2, Y = 2 ;  
No
```

```
?- b(X), b(Y), !.
```

```
X = 1, Y = 1 ;  
No
```

```
?- b(X), !, b(Y).
```

```
X = 1, Y = 1 ;  
X = 1, Y = 2 ;  
No
```

IA02

Programmation Logique

## Exemple

91

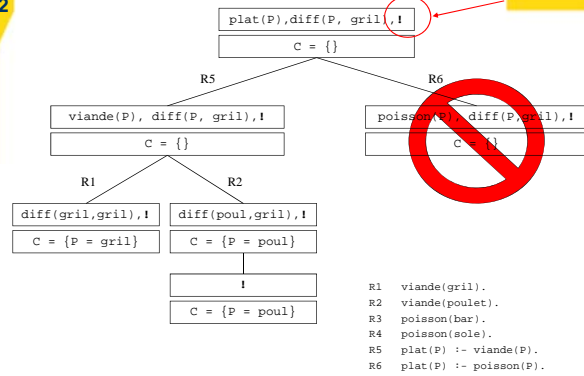
- Base de faits:
  - R1 viande(gril).
  - R2 viande(poul).
  - R3 poisson(bar).
  - R4 poisson(sole).
  - R5 plat(P) :- viande(P).
  - R6 plat(P) :- poisson(P).
- Question (version 1):  
plat(P), diff(P, gril).
- Question (version 2):  
plat(P), diff(P, gril), !.
- Question (version 3):  
plat(P), !, diff(P, gril).

IA02

Programmation Logique

## Exemple V2

92

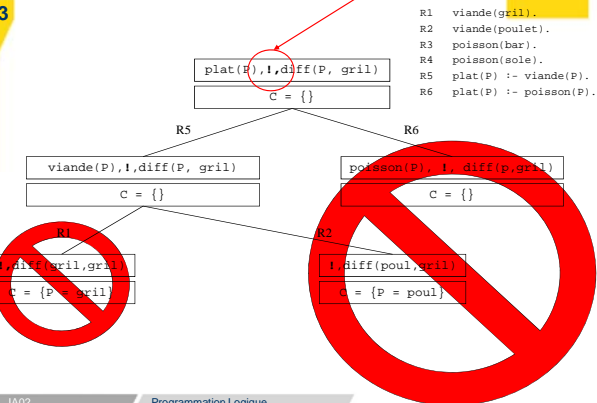


IA02

Programmation Logique

## Exemple V3

93



IA02

Programmation Logique

## Le cut : un usage délicat

94

- **Green cut** : la sémantique déclarative du programme **n'est pas** modifiée,
  - on peut enlever le **cut** le programme fonctionnera toujours ;
- **Red cut** : la sémantique déclarative du programme **est** modifiée,
  - Le retrait du **cut** conduit à un programme au fonctionnement erroné.

IA02

Programmation Logique

## Exemple de cut rouge

95

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- X < 6, !.  
f(X, 4).
```

IA02

Programmation Logique

## Exemple de cut vert

96

```
f(X, 0) :- X < 3, !.  
f(X, 2) :- 3 =< X, X < 6, !.  
f(X, 4) :- 6 =< X.
```

IA02

Programmation Logique



## Autre exemple de coupure

97

Précédente version de factorielle :

```
fact(0, 1).  
fact(N, Res) :- N > 0,  
                M is N - 1,  
                fact(M, Tmp),  
                Res is Tmp * N.
```

Nouvelle version de factorielle :

```
fact(0, 1) :- !.  
fact(N, Res) :- M is N - 1,  
                fact(M, Tmp),  
                Res is Tmp * N.
```

IA02

Programmation Logique

## La coupure en résumé

98

En résumé, les utilités de la coupure sont :

- ✓ éliminer 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.

IA02

Programmation Logique

## Autres prédicats de contrôle

99

- **true** est un but qui **réussit** toujours  
 $p(A, B).$        $\equiv$        $p(A, B) \text{ :- true.}$
- **fail** est un but qui **échoue** toujours
- **repeat** est un but qui **réussit** toujours mais qui laisse systématiquement un point de choix derrière lui.  
Il a une infinité de solutions :

```
repeat.  
repeat.  
repeat.  
...
```

IA02

Programmation Logique

## La négation par l'échec

100

Si  $F$  est une formule, sa négation est notée  $\neg F$ .

Prolog pratique la **négation par l'échec**, c'est-à-dire que pour démontrer  $\neg F$  Prolog va tenter de démontrer  $F$ .

Si la démonstration de  $F$  échoue, Prolog considère que  $\neg F$  est démontrée.

Pour Prolog,  $\neg F$  signifie que la formule  $F$  n'est pas démontrable, et non que c'est une formule fausse. C'est ce que l'on appelle **l'hypothèse du monde clos**.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Usage du Cut : La négation

101

On cherche à ce que le but  $\text{non}(P)$  s'efface lorsque  $P$  ne s'efface pas.

```
non(P) : -P,                (R1)
        !,
        fail.
non(_).                    (R2)
```

Si  $P$  s'efface (règle R1) alors  $\text{non}(P)$  ne doit pas s'effacer. Pour ce faire, on supprime tous les choix en attente par "!" (i.e., la règle R2 ne sera pas essayée) et on provoque un échec.

Si  $P$  ne s'efface pas, on examine R2 qui est toujours vérifiée.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## La double négation

102

Étant donné un but  $P$ , que signifie  $\text{non}(\text{non}(P))$  ?

- Si  $P$  s'efface, qu'en est-il de  $\text{non}(\text{non}(P))$  ?
- Les variables de  $P$  sont-elles unifiées si  $\text{non}(\text{non}(P))$  est effacé ?
- Que se passe-t-il en cas d'échec ?

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Les villes

103

Soient les assertions suivantes :

```
habite(virginie,amiens).  
habite(delphine,lille).
```

Que se passe t-il si on pose les questions :

```
?- non(habite(delphine,X)).  
?- non(non(habite(virginie,X))).  
?- non(habite(X,Y)).  
?- non(non(habite(X,Y))).
```

IA02

Programmation Logique

## Unification vs. Différence

104

### • L'unification :

Prédicat binaire :  $x = y$ .

Pour démontrer  $x = y$ , Prolog unifie  $x$  et  $y$  ;

S'il réussit, la démonstration est réussie, et le calcul continue avec les valeurs des variables déterminées par l'unification.  $?- x = 2$ . YES

$?- x = 2, y = 3, x = y$ . NO

### • La différence est définie comme le contraire de l'unification.

Elle est notée :  $x \neq y$ . Elle signifie que  $x$  et  $y$  ne sont pas unifiables, et non qu'ils sont différents.

Ex :  $z \neq 3$ . Sera faux car  $z$  et  $3$  sont unifiables.

IA02

Programmation Logique

## Usage du Cut : La différence

105

On cherche à ce que le but  $\text{diff}(X,Y)$  s'efface lorsque  $x$  est différent de  $y$ .

```
diff(X,X) :- !, fail.      (R1)  
diff(_,_) .               (R2)
```

Si  $x = y$ , il y a unification avec la tête de R1. Le cut est exécuté. Plus de retour en arrière possible. Puis provocation de l'échec avec `fail`.

Sinon le but réussi toujours quelques soient  $x$  et  $y$  avec R2.

IA02

Programmation Logique

## Usage de repeat : Exemple

106

```
boucle_menu:- repeat, menu, !.  
menu:- nl, write('1. Choix 1'),nl,  
        write('2. Choix 2'),nl,  
        write('3. Choix 3'),nl,  
        write('4. Terminer'),nl,  
        write('Entrer un choix '),  
        read(Choix),nl, appel(Choix),  
        Choix=4, nl.  
  
appel(1):- write('Vous avez choisi 1'),!.  
appel(2):- write('Vous avez choisi 2'),!.  
appel(3):- write('Vous avez choisi 3'),!.  
appel(4):- write('Au revoir'),!.  
appel(_):- write('Vous avez mal choisi').
```

IA02

Programmation Logique

## Plan du cours

107

- Une introduction a Prolog
- Représentation des connaissance
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## Les listes

108

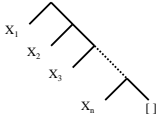
- Représentation
- Parcours
- Manipulation

IA02

Programmation Logique

## Représentation

- 109
- Comme en Lisp, les listes jouent un grand rôle en Prolog.
  - Une liste est un ensemble ordonné.
  - La liste est un terme composé particulier de symbole de fonction "." et d'arité 2:
    - **premier argument** : tête de la liste
    - **deuxième argument** : queue de la liste.
  - La liste vide est notée "[ ]".
  - Les listes sont représentées par des peignes (arbre dont chaque branche de gauche n'a pas de ramification).



IA02

Programmation Logique

## Notations

- 110
- la liste  $.(X, L)$  est également notée  $[X|L]$ ,
  - la liste  $.(X1, .(X2, L))$  est également notée  $[X1, X2|L]$ ,
  - la liste  $.(X1, .(X2, \dots, .(Xn, L) \dots))$  est également notée  $[X1, X2, \dots, Xn|L]$ ,
  - la liste  $.(X1, .(X2, \dots, .(Xn, [ ]) \dots))$  est également notée  $[X1, X2, X3, \dots, Xn]$ .

Ex:

la liste  $[a, b, c] = .(a, .(b, .(c, [ ])))$

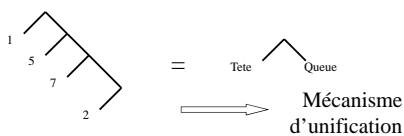
la liste  $[a, b|L]$  correspond à la liste  $.(a, .(b, L))$

IA02

Programmation Logique

## Equation sur une liste

- 111
- La notation  $[Tete|Queue]$  représente une liste dont la tête (i.e., le premier élément) est **Tete** et dont la queue (i.e., le reste de la liste) est **Queue**.
  - Considérons l'équation  $[Tete|Queue] = [1, 5, 7, 2]$ .  
La solution est **Tete = 1** et **Queue = [5, 7, 2]**.



IA02

Programmation Logique

## Exemples

112 ?- [a, b] = [X | Y].  
X = a, Y = [b]

?- [a] = [X | Y].  
X = a, Y = []

?- [a, [b]] = [X | Y].  
X = a, Y = [[b]]

?- [a, b, c, d] = [X, Y | Z].  
X = a, Y = b, Z = [c, d]

?- [[a, b, c], d, e] = [X | Y].  
X = [a, b, c], Y = [d, e]

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Structure récursive des listes

113 la liste **Liste** = [x|L] est composée d'un élément de tête x et d'une queue de liste L qui est elle-même une liste.

⇒ les relations Prolog qui manipulent les listes seront généralement définies par :

- une ou plusieurs clauses récursives, définissant la relation sur la liste [x|L] en fonction de la relation sur la queue de liste L,
- une ou plusieurs clauses non récursives assurant la terminaison de la manipulation, et définissant la relation pour une liste particulière (par exemple, la liste vide, ou la liste dont l'élément de tête vérifie une certaine condition...).

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Manipulation des listes

114 Quelques opérations élémentaires sur les listes:

- ✓ Imprimer une liste
- ✓ Déterminer si un élément est dans une liste
- ✓ Créer une liste contenant un élément sur deux d'une autre liste.
- ✓ Ajouter un élément en tête et en queue d'une liste
- ✓ Renverser une liste
- ✓ Concaténer une liste
- ✓ ...

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Imprimer une liste

115

L'utilisation de la notation  $[Tete|Queue]$  permet d'itérer sur les éléments de la liste.

```
imprime([]).  
imprime([T|Q]) :- write(T), nl, imprime(Q).  
  
?- imprime([7, 8, 9]).  
7  
8  
9
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Élément de ...

116

$x$  fait-il partie d'une liste  $L$  ?

- Utiliser la notation  $L = [T|Q]$
- Itérer récursivement sur tous les éléments de la liste
- Si  $x = T$  alors  $x$  est élément de  $L$ .
- Sinon, tester si  $x$  est élément de  $Q$ .

```
R1      element(X, [X|Q]).  
R2      element(X, [T|Q]) :- element(X, Q).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Exercice : Effacement de element

117

- Soit la question  
    `element(X, [7, 8, 9])`
- Appliquer une par une les règles R1 et R2
- Dessiner l'arbre de recherche de Prolog
- Reprendre les questions avec  
    `element(X, [7, 8, 9]), !.`

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Liste des éléments de rang pair

- 118
- Soit  $L$  une liste dont on veut extraire un élément sur deux.
  - On prend le premier élément qui est mis dans une autre liste. Le second élément est enlevé et on itère.

```
un_sur_deux([], []).
un_sur_deux([X], [X]).
un_sur_deux([X|Q], [X|R]) :- Q = [_|NewQ],
                             un_sur_deux(NewQ, R).
```

Plus élégant :

```
un_sur_deux([], []).
un_sur_deux([X], [X]).
un_sur_deux([X, _|Q], [X|R]) :- un_sur_deux(Q, R).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Concaténer deux listes

119

- Soient deux listes  $L1$  et  $L2$  que l'on veut concaténer dans  $L3$  : `concat(L1, L2, L3)`.

```
concat([], L, L).
concat([T|Q], L, [T|R]) :- concat(Q, L, R).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Renverser une liste (I)

120

- Soit une liste  $L$  que l'on veut renverser dans  $LR$ .
- Si  $[T|Q]=L$  alors  $LR$  est la concaténation de la liste  $Q$  renversée et de  $[T]$  :

```
renverser([], []).
renverser([T|Q], R) :- renverser(Q, QR),
                      concat(QR, [T], R).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---



## Renverser une liste (II)

121

- A quoi sert, dans le code suivant, le deuxième argument du prédicat renverser ?

```
renverser([], L, L).  
renverser([T|Q], L, R) :- renverser(Q, [T|L], R).  
  
?- renverser([1, 2, 3], [], X).  
X = [3,2,1]
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Les structures complexes

122

- Listes et sous-listes
- Arbres
- Equations

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Listes et sous - listes

123

- En prolog, le niveau de liste est infini (limité par la mémoire).
- On peut utiliser plusieurs niveaux de listes et sous-listes pour représenter des objets complexes.
- Ex : pour représenter une famille :

```
[pierre, claire, [fabrice, sebastien, helene]]
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Exemple : les menus (I)

124

- Soit une liste  $L$  qui représente tous les plats disponibles dans un menu :  
 $L = [\text{sardine}, \text{pate}, \text{melon}, \text{celeri}, \text{poulet}, \text{roti}, \text{steak}, \text{merlan}, \text{colin}, \text{loup}, \text{tarte}, \text{gateau}, \text{orange}, \text{pomme}, \text{banane}]$
- Mauvais, car il n'y a plus de distinction entre entrées, viandes, poissons, desserts, ...
- Si on utilisait «  $\text{element}(X, L)$  », on aurait l'énumération indifférenciée de tous les mets ...

IA02

Programmation Logique

## Exemple : les menus (II)

125

- On peut découper la liste  $L$  en sous-listes.
- Chaque sous-liste rassemble les mets d'une même catégorie.
- La sous-liste des plats peut elle même être séparée en viandes et poissons ....

```
L=[
  [sardine,pate,melon,celeri],
  [ [poulet,roti,steak],[merlan,colin,loup] ],
  [tarte, gateau, orange, pomme, banane]
]
```

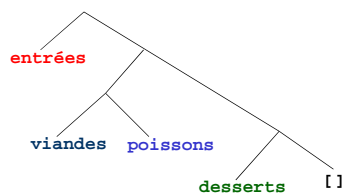
IA02

Programmation Logique

## Exemple : les menus (III)

126

- Représentation :  
 $L = [ [\text{sardine}, \text{pate}, \text{melon}, \text{celeri}], [[\text{poulet}, \text{roti}, \text{steak}], [\text{merlan}, \text{colin}, \text{loup}]], [\text{tarte}, \text{gateau}, \text{orange}, \text{pomme}, \text{banane}]]$



IA02

Programmation Logique

## Les menus (IV) : les équations

127

- $L = [X | Y]$  a pour solution :

```
X=[sardine,pate,melon,celeri]
Y=[[poulet,roti,steak],[merlan,colin,loup]],
  [tarte, gateau, orange, pomme, banane]]]
```

- $L = [E, [V, P], D]$  a pour solution :

```
E=[sardine,pate,melon,celeri]
V=[poulet,roti,steak]
P=[merlan,colin,loup]
D=[tarte, gateau, orange, pomme, banane]
```

IA02

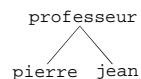
Programmation Logique

## Les arbres

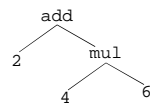
128

- Toute relation ou règle peut être représentée par un arbre

- professeur(pierre, jean)



- add(2, mul(4, 6))



- $[a, b, [c, d]] \equiv .(a, .(b, .(.c, .(d, [])) , []))$

IA02

Programmation Logique

## Les arbres

129

- Un arbre est composé :
  - d'un nœud  $x$
  - de sous – arbres  $a_1, a_2, \dots, a_n$

IA02

Programmation Logique

## PROLOG et les arbres

130

- L'interprète Prolog manipule des arbres ;
- Recherche à unifier (faire correspondre) des arbres ;
- Remplace des variables par des valeurs qui représentent des solutions (contraintes) ;
- Possibilité de résoudre des équations entre des structures d'arbres :  
 $a = a' , a \neq a'$

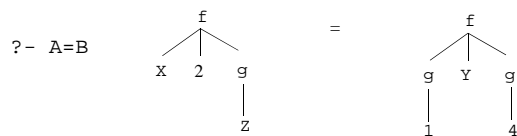
IA02

Programmation Logique

## Egalité entre deux arbres

131

- $A = f(X, 2, g(Z))$
- $B = f(g(1), Y, g(4))$



Solution :  $\{X=g(1) , Y=2 , Z=4\}$

IA02

Programmation Logique

## Conditions d'égalité entre deux arbres

132

- En prolog, les prédicats sont des arbres
- Soient  $A=(x, a_1, a_2, \dots, a_n)$   
 $B=(y, b_1, b_2, \dots, b_m)$
- $A$  est égal à  $B$  si et seulement si :
  - ✓  $n = m$  (même nombre de sous arbres)
  - ✓  $x = y$  (même tête)
  - ✓  $\forall i \in [1, n], a_i = b_i$  (les sous arbres sont égaux)

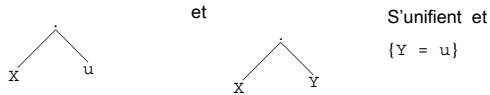
IA02

Programmation Logique

## Unification

133

- Unification entre A et B : trouver des valeurs que doivent prendre les valeurs A et B, pour rendre égaux les deux arbres



- L'ensemble des égalités de type  $\{Variable = valeur\}$  est appelé une substitution.

IA02

Programmation Logique

## Exemple : la famille

134

```
famille( indiv(thomas,hackett,date(31,mai,1966)),
        indiv(louise,hackett,date(27,dec,1966)),
        [ indiv(gregoire,hackett,date(4,fev,1994)),
          indiv(olivier,hackett,date(5,fev,1995))] ).

famille( indiv(philippe,dupond,date(4,sept,1949)),
        indiv(marie,dupond,date(14,fev,1952)),
        [ indiv(tom,dupond,date(7,oct,1976)),
          indiv(pierre,dupond,date(6,oct,1977)),
          indiv(valerie,dupond,date(22,fev,1980)),
          indiv(brigitte,dupond,date(20,sept,1987))] ).

? - famille(P,M,E).
? - famille(indiv(philippe,dupond,_), indiv(F,_,_), _)
```

Quels sont les enfants de Mr Hackett ?  
Quelles sont les familles de plus de 3 enfants ?  
Quelles sont les familles dont le premier enfant a plus de 20 ans ?

IA02

Programmation Logique

## Plan du cours

135

- Une introduction a Prolog
- Représentation des connaissance
- Les calculs et les I/O sous Prolog
- L'effacement
- La récursivité
- La coupure (terme d'arrêt)
- Les listes
- Les structures complexes
- La résolution de problèmes avec Prolog

IA02

Programmation Logique

## Résolution de problèmes

136

- Exploration de graphes d'espace d'états ;
- Génération - et - test ;
- Expression de contraintes ;
- Utilisation du backtrack ;
- Solutions incrémentales ;
- Boucles ;
- Multi - récursivité.

IA02

Programmation Logique

## Exploration des graphes d'espace d'états

137

- Les graphes d'espace d'états sont utilisés pour représenter la solution de certains problèmes.
- Les nœuds du graphe sont les états du problème.
- Il y a un arc entre deux nœuds **Etat1** et **Etat2** s'il y a une règle de transition (ou mouvement) qui transforme **Etat1** en **Etat2**.
- Résoudre le problème : trouver un chemin d'un état initial donné à un état solution souhaité, en appliquant une suite de règles de transition.

IA02

Programmation Logique

## Cadre pour l'exploration

138

**Solve(EtatInit, EtatFinal, History, Mvts)**

**Mvts** est la suite des mouvements pour atteindre **EtatFinal** à partir de **EtatInit**, où **History** contient les états visités auparavant.

```
solve(EtatFinal,EtatFinal,_,[]).
solve(Etat, EtatFinal, History,[Mvt|Mvts]):-
    move(Etat, Mvt, EtatSucc),
    valid(EtatSucc), % respecte les contraintes du problème
    \+element(EtatSucc,History), % non visité
    solve(EtatSucc,EtatFinal,[EtatSucc|History],Mvts).
```

IA02

Programmation Logique

## Exemple : les carafes d'eau

139

Il y a deux carafes de capacité égales à 8 et 5 litres, sans marques, et le problème est de puiser exactement 4 litres d'eau d'une auge contenant 20 litres (ou une autre grande quantité). Les opérations consistent à remplir une carafe depuis l'auge, vider une carafe dans l'auge et transférer le contenu d'une carafe dans l'autre jusqu'à ce que, soit la carafe qui verse soit complètement vide, soit l'autre carafe soit remplie à ras le bord.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Exemple : les carafes d'eau

140

```
solve(EtatFinal,EtatFinal,_,[]).
solve(Etat, EtatFinal, History,[Mvt|Mvts]):-
  move(Etat, Mvt, EtatSucc),
  valid(EtatSucc),
  not(element(EtatSucc,History)),
  solve(EtatSucc,EtatFinal,[EtatSucc|History],Mvts).
```

```
etat_init(volumes(0,0)).
etat_final(volumes(4,_)).
etat_final(volumes(_,4)).
```

```
capacite(1,8).
capacite(2,5).
```

```
valid(_).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Exemple : les carafes d'eau

141

```
move(volumes(_,V2), vider(1), volumes(0,V2)).
move(volumes(V1,_), vider(2), volumes(V1,0)).
move(volumes(_,V2), remplir(1), volumes(C1,V2)):-capacite(1,C1).
move(volumes(V1,_), remplir(2), volumes(V1,C2)):-capacite(2,C2).
```

```
move(volumes(V1,V2), transf(2,1), volumes(W1,W2)):-
  Vtot is V1+V2, capacite(1,C1),
  Exces is Vtot-C1, ajuster(Vtot, Exces, W1, W2).
```

```
move(volumes(V1,V2), transf(1,2), volumes(W1,W2)):-
  Vtot is V1+V2, capacite(2,C2),
  Exces is Vtot-C2, ajuster(Vtot, Exces, W2, W1).
```

```
ajuster(Vtot, Exces, Vtot, 0) :- Exces <= 0.
ajuster(Vtot, Exces, V, Exces) :- Exces > 0, V is Vtot-Exces.
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Génération - et - test

142

- Technique courante dans la conception d'algorithme et la programmation.
- Soit un problème **II**, dans la génération-et-test :
  - Un processus engendre l'ensemble des candidats susceptibles de résoudre le problème posé ;
  - Un autre processus teste ces candidats en essayant de trouver le ou les candidats qui résolvent vraiment le problème.

IA02

Programmation Logique

## Génération - et - test

143

- Il est facile de rédiger des programmes Prolog qui mettent en œuvre la technique de génération-et-test.
- On a typiquement une conjonction de deux buts :
  - Un but s'occupe de la génération des candidats ;
  - Un but teste si la solution est acceptable.

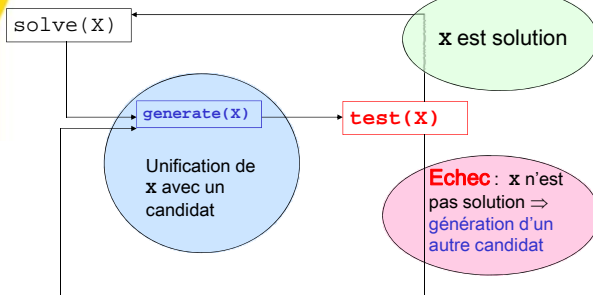
```
solve(X):- generate(X) , test(X).
```

IA02

Programmation Logique

```
solve(X):- generate(X) , test(X).
```

144



Arrêt quand le testeur trouve avec succès une solution ayant la propriété distinguée, ou quand le générateur a épuisé toutes les solutions (échec).

IA02

Programmation Logique



## Un générateur simple : `element_de`

145

- Un programme à solutions multiples utilisé couramment comme générateur est le programme `element`.
- `element(X, [a, b, c])` donnera successivement les solutions `X=a`, `X=b`, `X=c` que l'on peut utiliser par la suite avec un programme test.

IA02

Programmation Logique

## Exemple

146

- Tester si deux listes ont un élément en commun :

```
intersection(Xs,Ys) :-  
    element(X, Xs),  
    element(X, Ys).
```

- `element` est à la fois générateur et testeur.

IA02

Programmation Logique

## Rendre la monnaie

147

Utilisation du  
backtrack pour  
proposer  
d'éventuelles  
solutions.

Pose de la  
contrainte.

```
chiffre(X) :-  
    element(X, [0, 1, 2, 3, 4, 5,  
                6, 7, 8, 9]).  
  
change(Cinq, Dix, Vingt,  
       Cinquante, Somme) :-  
    chiffre(Cinq),  
    chiffre(Dix),  
    chiffre(Vingt),  
    chiffre(Cinquante),  
    Somme =:= (5 * Cinq  
               + 10 * Dix  
               + 20 * Vingt  
               + 50 * Cinquante).
```

IA02

Programmation Logique

## SEND + MORE = MONEY

148

$$\begin{array}{r} \text{S E N D} \\ + \text{M O R E} \\ \hline = \text{M O N E Y} \end{array}$$

Avec S, M, E, N, D, O, R, Y tous différents

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## SEND + MORE = MONEY

149

- Backtrack.
- Expression de contraintes.
- ORDRE des contraintes !!!

```
probleme([S,E,N,D,M,O,R,Y]):-  
  chiffre(S),chiffre(E),chiffre(N),  
  chiffre(D),chiffre(M),chiffre(O),  
  chiffre(R),chiffre(Y),  
  diffrts([S,E,N,D,M,O,R,Y]),  
  V1 is 1000 * S + 100 * E +  
    10 * N + D,  
  V2 is 1000 * M + 100 * O +  
    10 * R + E,  
  Somme is 10000 * M + 1000 * O +  
    100 * N + 10 * E + Y,  
  Somme == V1 + V2.
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Coloriage de cartes

150

- Problème : colorier une carte planaire de façon qu'aucune paire de régions adjacentes ne soit de la même couleur.
- Une conjecture célèbre (ouverte pendant 100 ans), fut démontrée en 1976, affirmant que quatre couleurs suffisent pour colorier n'importe quelle carte planaire.

IA02

Programmation Logique

---

---

---

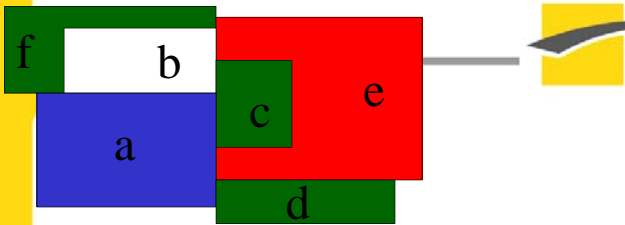
---

---

---

---

---



```

regions([a, b, c, d, e, f]).
couleurs([bleu, vert, rouge, blanc]).
  adjacent(a,f). adjacent(a,b). adjacent(a,c). adjacent(a,e).
  adjacent(a,d). adjacent(f,b). adjacent(b,e). adjacent(b,e).
  adjacent(c,e). adjacent(e,d).
  ...
unColoriage(L) :-
  regions(R), couleurs(C), colorie(R,C,[],L).

```

IA02 Programmation Logique

---

---

---

---

---

---

---

---

## Coloriages de cartes

152

```

colorie([], _, L, L).

colorie([Region|Q], Colors, SolutionCour, Res) :-
  element(C, Colors),
  pasEnConflit(Region, C, SolutionCour),
  colorie(Q, Colors, [(Region,C)|SolutionCour], Res).

pasEnConflit(_, _, []).

pasEnConflit(Reg, C, [(AutreReg,Cbis)|SolutionCour]) :-
  \+adjacent(Reg, AutreReg),!,
  pasEnConflit(Reg, C, SolutionCour).

pasEnConflit(Reg, C, [(AutreReg,Cbis)|SolutionCour]) :-
  C \= Cbis,
  pasEnConflit(Reg, C, SolutionCour).

```

IA02 Programmation Logique

---

---

---

---

---

---

---

---

## Modification de la base

153

Les prédicats de base de données permettent de manipuler les faits et les règles de la base de données.

- Ajout dans la base
  - `asserta/1`
  - `assertz/1`
- Retrait dans la base
  - `retract/1`
  - `retractall/1`
- Attention :
  - Modification dynamique**
  - Ne modifie pas la base de données statique**

IA02 Programmation Logique

---

---

---

---

---

---

---

---

### Les prédicats `asserta/1` et `assertz/1`

154

- `asserta(P)` et `assertz(P)` permettent d'ajouter `P` à la base de données.
- `asserta/1` : ajout au début de la base de données
- On ajoute en fin avec `assertz/1` : ajout en fin de base

On peut aussi ajouter des clauses :

```
asserta((P :- B, C, D))
```

⇒ modification dynamique du programme.

IA02

Programmation Logique

### Le prédicat `retract/1`

155

`retract(P)` permet de retirer `P` de la base de données.

État de la base :

```
personne(jean).
personne(isa).
personne(françois).
```

```
?- retract(personne(isa)).
```

```
personne(jean).
personne(françois).
```

```
?- retract(personne(X)).
```

```
X = jean ; X = françois ; No
```

```
rien
```

IA02

Programmation Logique

### Le prédicat `retractall/1`

156

Le prédicat `retractall(Terme)` permet de retirer tous les termes `Terme`.

```
?- retractall(personne(_)).
```

Retire tous les prédicats `"personne(_)"`.

IA02

Programmation Logique

## Comment garder toutes les solutions

157

Chercher toutes les instances **x** (et les conserver) telles que **but (x)** s'efface.

- ✓ Éviter de répéter des calculs inutiles ;
- ✓ Très utile pour résoudre des problèmes de recherche.

Deux prédicats **setof** et **bagof** permettent de chercher toutes les instances **x** telles que **but (x)** s'efface.

- **setof(X, But, Instances)** s'efface lorsque **Instances** est l'ensemble des instances **x** pour lesquelles **But** s'efface.
- **bagof** est une variante dans laquelle les instances identiques sont conservées.

IA02

Programmation Logique

## Setof, bagof

158

```
homme(socrate).  
homme(socrate).  
homme(robert).  
homme(charles).
```

```
?- setof(X, homme(X), Hommes).  
Hommes=[charles,robert,socrate]
```

```
?- bagof(X, homme(X), Hommes).  
Hommes=[socrate,socrate,robert,charles]
```

IA02

Programmation Logique

## Setof

159

```
chiffre(X) :- element(X, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).  
change(Cinq, Dix, Vingt, Cinquante, Somme) :-  
    chiffre(Cinq),chiffre(Dix),  
    chiffre(Vingt),chiffre(Cinquante),  
    Somme is  
    (5 * Cinq + 10 * Dix + 20 * Vingt + 50 * Cinquante).
```

```
-? setof([C,D,V,Ci],change(C,D,V,Ci,50),S).
```

```
S=[[0,0,0,1],[0,1,2,0],[0,3,1,0],[0,5,0,0],[2,  
0,2,0],[2,2,1,0],[2,4,0,0],[4,1,1,0],[4,3,0,0]  
,[6,0,1,0],[6,2,0,0],[8,1,0,0]]
```

Permet : d'avoir toutes les solutions;

IA02

Programmation Logique

## Coder un setof

160

1<sup>ère</sup> idée :

```
mysetof(X, But, L) :- mysetof(X, But, [], L).  
mysetof(X, But, Tmp, L) :-  
    But,  
    not(element(X, Tmp)),  
    !,  
    mysetof(X, But, [X|Tmp], L).  
mysetof(X, But, L, L).
```

la variable **x** est instanciée et donc le code n'est pas correct.

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Coder un setof

161

Boucle INTERDITE ! (instanciation de **x**).  
Comment s'assurer de la nouveauté de chaque effacement de But ?  
Il faut une mémoire non soumise au backtracking.

=> **CREATION DYNAMIQUE DE FAITS !**

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Modification dynamique de la base

162

Création d'un fait.

```
assert(toto(10,2,[])). <=> toto(10,2,[]).
```

Suppression d'une règle (la première dans la base).

```
retract(toto(X, Y, Z)).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

---

## Coder un setof

163

```
mysetof(X, But) :-
    But,
    solution(L),
    \+element(X, L),
    retractall(solution(_)),
    asserta(solution([X|L])),
    fail.

mysetof(X, But, _) :-
    retractall(solution(_)),
    asserta(solution([])),
    mysetof(X, But).

mysetof(_, _, L) :-
    solution(L),
    retracall(solution(_)).
```

IA02

Programmation Logique

## Plus court chemin dans un graphe

164

```
pcc(X,Y,G,_,VC,Copt,Vopt):-
    chemin(X,Y,G,CN,VN),
    VN<VC,!,
    pcc(X,Y,G,CN,VN,Copt,Vopt).

pcc(_,_,_,Copt,Vopt,Copt,Vopt):-!.

pcc(X,Y,G,Copt,Vopt):-
    chemin(X,Y,G,CC,VC),!,
    pcc(X,Y,G,CC,VC,Copt,Vopt).
```

IA02

Programmation Logique

## Plus court chemin dans un graphe

165

```
pcc2(X,Y,G,_,_):-
    retractall(solution(_,_)),
    asserta(solution(99999,[])),
    chemin(X,Y,G,CN,VN),
    solution(VC,_),
    VN<VC,
    retractall(solution(_,_)),
    asserta(solution(VN,CN)),
    fail.

pcc2(_,_,_,Copt,Vopt):-
    solution(Vopt,Copt),
    retractall(solution(_,_)),!.
```

IA02

Programmation Logique

## Entrées / sorties ... la suite

- 166
- Le prolog standard ne connaît que l'ASCII. Les entrées/sorties sont primitives.
  - Affichage de caractères et de chaînes : `put/1` s'utilise en donnant le code ASCII d'un caractère :  
`put(65).` (eq. `put_char('A')`)  
affiche : A
  - Lecture de caractères : `get/1`, `get0/1`. et `get_char/1`.  
`get/1` et `get0/1` prennent en argument un terme unifié avec le code ASCII du caractère lu. `get/1` ne lit que les caractères de code compris entre 33 et 126.  
`get_char/1` s'unifie avec le caractère.

IA02

Programmation Logique

## Les fichiers

- 167
- Ouvrir et fermer des fichiers en lecture :
- `see(fichier).` : ouverture en lecture du fichier `fichier`. Le fichier devient le flux d'entrée courant.
  - `see(user).` : le flux courant d'entrée redevient l'utilisateur - le clavier.
  - `see(fichier).` : rappelle `fichier` à l'endroit où il était s'il n'a pas été fermé. Et il redevient le flux courant.
  - `seen.` : fermeture du fichier ouvert en lecture. L'utilisateur devient le flux courant.

IA02

Programmation Logique

## Les fichiers

- 168
- Ouvrir et fermer des fichiers en écriture :
- `tell(fichier).` : ouverture en écriture du fichier `fichier`. Le fichier devient le flux de sortie courant.
  - `tell(user).` : le flux courant de sortie redevient l'écran.
  - `tell(fichier).` : rappelle `fichier` à l'endroit où il était s'il n'a pas été fermé. Et il redevient le flux courant de sortie.
  - `told.` : fermeture du fichier ouvert en écriture. L'écran devient le flux courant.

IA02

Programmation Logique



## Les fichiers

169

Ouverture d'un fichier : prédicat `open/3`

- argument 1 : nom du fichier
- argument 2 : mode d'ouverture `write`, `append` ou `read`.
- argument 3 : variable qui va recevoir un identificateur de fichier (*flux* ou *stream*).

En écriture :

mode `write` : le contenu est effacé avant écriture.

mode `append` : écriture à partir de la fin du fichier.

IA02

Programmation Logique

## Les fichiers

170

- Tous les prédicats `read`, `write` et autres admettent un second argument : le flux identifiant le fichier.

Ex : `write(Flux, X)`. où `Flux` est un identificateur de fichier.

Ex : `read(Flux, X)`, `get(Flux, X)`, `get0(Flux, X)`.

Fermeture du fichier : le prédicat `close/1` qui prend en argument le flux associé au fichier.

IA02

Programmation Logique

## Les fichiers

171

- Exemple : écriture dans un fichier

```
ecrire(T) :-  
    % ouverture  
    open('essai.txt', append, Flux),  
    % écriture  
    write(Flux, T), nl(Flux),  
    % fermeture  
    close(Flux).
```

IA02

Programmation Logique

## Les fichiers

172

- Exemple : lecture d'un fichier

```
read_file(File,List):-
    see(File),
    read_list(List), seen, !.

read_list([X|L]):-
    get_char(X),
    X \= end_of_file,
    !,
    read_list(L).
read_list([]).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

## Les fichiers : lecture d'un dictionnaire de mots

173

- Fichier 'liste\_mots.txt':  
*arbre voiture chien fleur maison fusée*

```
?- read_dico('liste_mots.txt',L).
```

```
L = [[a, r, b, r, e], [v, o, i, t, u, r, e],
      [c, h, i, e, n], [f, l, e, u, r, e],
      [m, a, i, s, o, n], [f, u, s, e, e]]
```

IA02

Programmation Logique

---

---

---

---

---

---

---

## Les fichiers : lecture d'un dictionnaire de mots

174

```
read_dico(File,Dico):-
    see(File),
    retractall(fin(_)), asserta(fin(0)),
    read_dico(Dico), seen,!.

read_dico([]):-fin(1).
read_dico([Mot|L]):-read_mot(Mot),!,read_dico(L).

read_mot([]):-peek_char(X), X=end_of_file, !,
    asserta(fin(1)).
read_mot([]):-peek_char(X), X=' ', !, get_char(X).
read_mot([X|M]):-get_char(X),!,read_mot(M).
```

IA02

Programmation Logique

---

---

---

---

---

---

---

## Lecture d'un dictionnaire de mots commençants par...

175

- Fichier 'liste\_mots.pl':  
*arbre voiture chien fleure maison fusée peur phalène pharaon  
phare pharmacie phase phebus phenix savon*

```
?- read_dico('liste_mots.pl',L,[p,h,a]).
```

```
L = [[p,h,a,l,e,n,e], [p,h,a,r,a,o,n],  
      [p,h,a,r,e], [p,h,a,r,m,a,c,i,e],  
      [p,h,a,s,e]]
```

IA02

Programmation Logique

---

---

---

---

---

---

---

## Lecture d'un dictionnaire de mots commençants par...

176

```
read_dico_begin([],_):-fin(1).  
read_dico_begin(Res,Begin):-  
    read_mot(Mot),!,  
    read_dico_begin(L,Begin),  
    ajout(Mot,Begin,L,Res).  
  
meme_debut(_,[]).  
meme_debut([T|L1],[T|L2]):-meme_debut(L1,L2).  
  
ajout(Mot,Begin,L,Res):-meme_debut(Mot,Begin),  
                        concat([Mot],L,Res).  
ajout(_,_ ,L,L).
```

IA02

Programmation Logique

---

---

---

---

---

---

---