

Représentation et Modélisation des connaissances : Programmation logique

*4^{ème} Année ENSIM
Option IPS*

Youssef SERRESTOU

Représentation et Modélisation des connaissances

Objectifs globaux :

- Donner un aperçu sur la logique mathématique;
- Donner un aperçu sur la programmation logique;
- Montrer les apports de la programmation logique;
- Initiation au langage de programmation Prolog;

Informations diverses

■ Volume horaire :

- 5h cours / 5h TD / 12h TP

■ ECTS de l'UE : 5

■ Enseignant :

- Y.SERRESTOU [youssef.serrestou@univ-lemans.fr]

■ Évaluation :

- 1 Examen final (EF)
- 2 QCM en TP (TP)
- 1 Mini-Projet (MP)
- Note finale : **$NG = 0,4 * EF + 0,3 * TP + 0,3 * MP$**

Informations diverses

■ Langage & logiciels utilisés:

- ❑ **gprolog**
<http://www.gprolog.org/>
- ❑ Amzi ! Prolog – version 5.0 ;
<http://www.amzi.com>
- ❑ SWI-Prolog – version 5.4.7;
<http://www.swi-prolog.org>
- ❑ Ciao Prolog
<http://ciao-lang.org/>

■ Supports pédagogiques :

- ❑ Diapositifs de cours
- ❑ Polycopiés de TD et de TP

Bibliographie

■ Livres :

- ❑ **Logique mathématique**
 - ✓ K. NOUR, R. DAVID et C. RAFFALLI : *Introduction à la logique : Théorie de la démonstration*, Ed. Dunod.
 - ✓ R. OCRI et D. LASCAR : *Logique mathématique, tome 1 & 2*, Ed. Dunod.
 - ✓ B. COURCELLE : *Logique et Informatique, une introduction*, INRIA, Collection Didactique, 1991.
 - ✓ J.Y. GIRARD : *Proof theory and logical complexity*, (Bibliopolis, Napoli, 1987).
 - ✓ J.Y. GIRARD, Y. LAFONT & P. TAYLOR : *Proofs and Types*, Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989.
 - ✓ A. S. TROELSTRA, H. SCHWICHTENBERG : *Basic Proof Theory*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, Second, revised edition 2000.
 - ✓ A.S. TROELSTRA, *Lectures in linear logic*, CSLI Stanford, Lecture Notes Series nr. 29, 1992.
- ❑ **Prolog**
 - ✓ H.. AIT KACI : "Warren' s abstract machine : a tutorial reconstruction", MIT Press, 1991.
 - ✓ P. BLACKBURN, J. Bos, K. STREIGNITZ : *Prolog, tout de suite !*, (free online version : <http://www.learnprolognow.org>), Cahiers de Logique et d'Epistémologie, College Publications, 2007.
 - ✓ I. BRATKO : *Prolog programming for artificial intelligence*, Addison Wesley, 2000.
 - ✓ F.W. CLOCKSIN, C.S. MELLISH : "Programming in Prolog : using the ISO standard", Springer, 2003.
 - ✓ H.COELLO, J.C. COTTA : *Prolog by example. How to learn, teach and use it*, Springer, 1988.

Introduction

Objectifs de cette première partie :

- Placer la logique dans le contexte de l'IA ;
- Donner un aperçu historique ;
- Introduire la programmation logique ;

Plan

- Un peu d'Histoire
- Logique : une approche de l'IA ?
- Rôle de la logique
- Programmation logique
 - Historique
 - Domaines d'applications
 - Paradigmes de programmation

Un peu d'Histoire

- Philosophie (428 av. J.-C. -- présent)
 - Logique et méthodes de raisonnement
 - Esprit comme système physique
 - Fondations de l'apprentissage, du langage et de la rationalité
- Mathématiques (800 -- présent)
 - Représentations formelles et preuves
 - Algorithmes
 - Calcul, (in)décidabilité, (in)solubilité
 - Probabilités
- Économie (1776 -- présent)
 - Théorie de la décision rationnelle
- Neurosciences (1861 -- présent)
 - Étude du fonctionnement du cerveau
- Psychologie (1879 -- présent)
 - Adaptation
 - Phénomène de la perception et du contrôle moteur
 - Techniques expérimentales
- **Mathématique (1920 -- présent)**
 - **Les problèmes d'Hilbert**
 - **La logique mathématique de Gödel, Church, Turing.**
- Ingénierie informatique (1940 -- présent)
 - L'ordinateur comme entité artificielle ayant la meilleure chance de démontrer de l'intelligence

Un peu d'Histoire

- Théorie du contrôle cybernétique (1948 -- présent)
 - Théorie du contrôle (Wiener)
- Linguistique (1957 -- présent)
 - Représentation des connaissances
 - Grammaire
- Les années lumières (euphorie et grands espoirs) 1956-1966
 - démonstration de théorèmes de la logique des propositions
 - Reconnaissance de caractères, La "souris cybernétique", le perceptron (Rosenblatt,58), Dames anglaises (Samuel,59)]
- General Problem Solver (1969): résolveur de problèmes général
 - Projet de traduction automatique (1966: rapport (ALPAC)
 - Weizenbaum, J., (1966) ELIZA- A computer program for the study of natural language communication between man and machine. Communications of the ACM, 9.1:36-45.
- Le renouveau (les premiers systèmes experts) 1969-1979
- L'IA institutionnalisée 1980-aujourd'hui : une industrie (SE, Systèmes d'apprentissage, interfaces ergonomiques, Data Mining, etc.)

Logique : une approche de l'IA ?

- D'un point de vue d'objectif, L'IA peut être considérée comme la science dont le but est de construire des artefacts intelligents
 - Intelligence :
 - Percevoir/Raisonner/Agir/Communiquer
 - Artefact :
 - Machine (système physique et/ou logiciel) manipulant des symboles.
 - Hypothèse : les symboles correspondent à des objets.
 - Construire : Ingénierie
 - Agent (perçoit, raisonne, communique et agit)

Quatre approches de l'IA

Modélisation cognitive
(GPS (Newel & Simon, 61))

Système passant le test de Turing (1950)

Apprendre des K. (s'adapter)
Représenter des K. (mémoriser)
Résoudre des Pb. (raisonner)
Comprendre (communiquer)

| | | | |
|--------|---|------|--|
| Penser | Comme l'humain = approche cognitive | Agir | Comme l'humain = test de Turing |
| | Pour arriver au même résultat que l'humain = approche rationnelle (suivant les lois de la pensée logiciste) | | Agir rationnellement = agent rationnel (cherche à atteindre la meilleure solution) |

Logicisme:
pensée logique
Pascal [1623-1662] (machine à calculer)
Leibnitz [1646-1716] (machine à raisonner)
Babbage [1792-1871] (machine analytique)

Agent rationnel (199X)
agit selon ses croyances pour atteindre des objectifs (pas seulement logique)

Réf. Figure 1.1 p 2 (Russel & Norvig 2003)

Penser comme l'humain

- **Comment fonctionne notre cerveau ?**
 - Modélisation cognitive
- **Requiert des théories scientifiques**
 - Activité interne du cerveau (médecine – biologie)
 - Introspection (tenter de se saisir de ses propres pensées) ou expériences psychologiques → recueil des connaissances
- **Informatisation et implémentation de ces théories et comparaison avec les humains**
 - Reprise de la logique et des connaissances, le but étant d'atteindre le même résultat qu'un humain en suivant les mêmes étapes
- **General Problem Solver (Newell & Simon 1961)**
 - Programme qui résout correctement des problèmes en suivant les mêmes étapes qu'un humain dans la même situation

Penser rationnellement

- **Aristote et ses syllogismes : naissance de la logique**
 - Ex : Socrate est un homme; tous les hommes sont mortels; donc Socrate est mortel
- **Début du XIXème notation précise des assertions et des relations**
 - 1965 programmes qui peuvent résoudre tout problème soluble, dès lors qu'il est formulé en notation logique
- **Difficultés**
 - Représenter dans un système formel des connaissances informelles ou incertaines
 - Engorgement de la puissance de calcul si l'utilisation des relations et des assertions n'est pas optimisée

Agir comme l'humain

- L'agent intelligent devrait posséder la capacité de :
 - Représenter des connaissances
 - Apprendre des connaissances (s'adapter)
 - Apprentissage artificiel (reconnaitances des formes et la fouilles des données)
 - Résoudre des Pb. (raisonner)
 - Comprendre (communiquer)
 - Le traitement du langage naturel

Agir rationnellement

- Lorsque l'on regarde les lois de la pensée, ce qui importe ce sont les inférences : faire la bonne chose
- Ici les inférences font parfois partie de la rationalité, mais pas obligatoirement.
- L'étude l'IA en vue de construire des agents rationnels est la plus large des approche :
 - au-delà des lois de la pensée,
 - ne se contraind pas à la compréhension de l'humain,
 - si l'on réussissait à développer un agent rationnel, il devrait pouvoir passer le test de Turing.

Concrètement qu'est ce que l'IA ?

- **Rechercher** (analyser, résoudre des problèmes, trouver des méthodes de résolution)
- **Représenter** des connaissances (logique, règles, mémoire, cas, langue naturelle, etc.)
- **Mettre en application** les idées 1 et 2 (Systèmes Experts, pilotes automatiques, agents d'interfaces, robots, Data Mining, etc.)

La représentation des connaissances

- Problème central en IA
- Mise en évidence d'un problème en amont :
 - L'acquisition des connaissances
 - La modélisation des connaissances
- Plusieurs formes de représentation
 - Représentation objet
 - Règles de production

Rôle de la logique

- Un formalisme de représentation des connaissances
- Un mécanisme d'inférence : la déduction
- Central pour :
 - **Prolog**
 - **Systèmes à base de connaissances**

La logique

- La logique est un cadre formel qui permet de formaliser, de représenter et de raisonner.
- **Plusieurs logiques:**
 - La logique des propositions
 - La logique des prédicats du 1^{er} ordre
 - La logique des prédicats du 2nd ordre
 - La logique floue
 - ...

PROgrammation LOGique

- 1965 : Méthode de résolution (Alan Robinson) :
- 1970 : création de Prolog (Alain Colmerauer)
- 1973 : 1^{ère} implémentation (P. Roussel)
- 1980 : un des langages de l'Intelligence Artificielle

Domaines d'applications

- Systèmes Experts : Aide à la décision et au diagnostic
 - Diagnostic de pannes
 - Ordonnancement de tâches
- Traitement automatique du langage naturel
 - Analyses syntaxique et sémantique
 - Interrogation de bases de données relationnelles
- Logique mathématique
- Résolution symbolique d'équations
- Planification et allocation de ressources
 - Régulation et optimisation de réseaux
 - Aide à la planification de projets informatiques
- ...

Paradigmes de programmation

- **Programmation impérative**
 - ❑ Pascal, C, Fortran
 - ❑ Blocs d'instructions; ensemble structuré et ordonnés d'instruction
- **Programmation fonctionnelle/applicative**
 - ❑ LISP, CAML
 - ❑ Ensemble de fonctions
- **Programmation objet**
 - ❑ C++, Java
 - ❑ Notion d'entités-messages - vision décentralisée du contrôle
- **Programmation logique**
 - ❑ Prolog
 - ❑ Définir des faits et des règles + exploration systématique d'un arbre de résolution ET/OU

PROLOG (cours 1)

PROgrammation LOGique

Objectifs :

- comprendre « Pourquoi Prolog ? »
- les bases du langage Prolog

Plan

- Les bases du langage Prolog
 - Les faits
 - Les règles
 - Les requêtes
- Modèle d'exécution
 - Unification
 - Arbre de résolution
- Les environnements de programmation

Prolog

- Résoudre un problème en Prolog c'est :
 - Formaliser un problème en terme de logique
 - Utiliser le moteur d'inférences pour faire des démonstrations logiques et répondre à des questions
- ➔ Programmation déclarative (vs procédurale)

Formalisation d'un problème

- Identifier les « objets » du monde à modéliser :
 - Pierre ; le livre ; la table ; etc.
- Définir des relations entre les objets (prédicats) :
 - *est le père de*/2 ; *est posé sur*/2 ;
 - *est heureux*/1 ;
- Écrire l'état initial du monde (faits initiaux) :
 - Luc *est le père de* Pierre ;
 - Le livre *est posé sur* la table ;
- Écrire des relations entre les prédicats :
 - Si X *est en vacances* alors X *est heureux*
 - Si X *donne* Y à Z alors Z *possède* Y
- Poser le problème en termes de but à atteindre :
 - Trouver X tel que X *est heureux*
 - Trouver Y tel que Y *possède* Z

Formalisation d'un problème : Exemple

- Aristote et ses syllogismes
 - Ex : **Socrate est un homme**; tous les hommes sont mortels; donc **Socrate est mortel**
- Identifier les « objets » du monde à modéliser :
 - *Socrate*
- Définir des relations entre les objets (prédicats) :
 - *est un homme* /1 ; *est mortel* /1 ;
- Écrire l'état initial du monde (faits initiaux) :
 - Socrate *est un homme* ;
- Écrire des relations entre les prédicats :
 - **Si X *est un homme* alors X *est mortel* ;**
- Poser le problème en termes de but à atteindre :
 - Trouver X tel que X *est mortel*
 - Trouver Y tel que Y *est un homme*

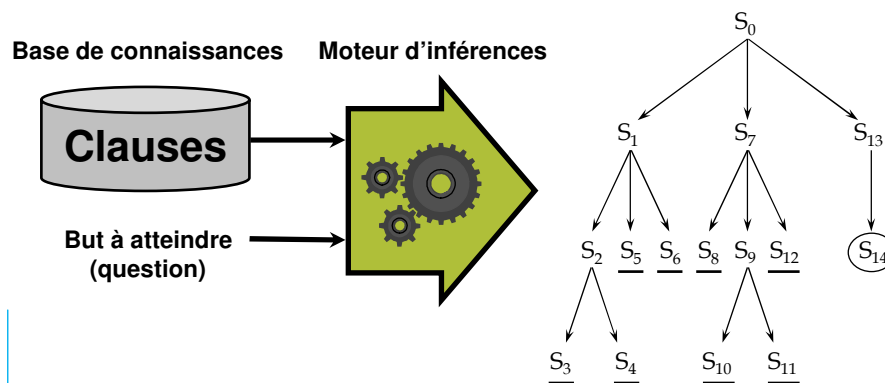
5

Représentation et modélisation des connaissances : Prolog
Y.S

26/04/2022

Moteur d'inférences

- Développer un **arbre ET/OU** en profondeur d'abord :
 - Partir du but et rechercher les conditions (**chaînage arrière**)
 - Mettre en œuvre un retour arrière chronologique (**backtracking**)



6

Représentation et modélisation des connaissances : Prolog
Y.S

Y.S

26/04/2022

Les bases du langage

Objectifs :

appréhender les notions et le fonctionnement
de base de Prolog

Premier exemple

Socrate est un homme, or tous les hommes sont mortels, donc Socrate est mortel

```
/* Les faits */
homme(socrate) .

/* Prédicat */
mortel(X) :- homme(X) .
```

Socrate est un grec, or tous les grecs sont des hommes et tous les hommes sont mortels, donc Socrate est mortel

```
/* Les faits */
grec(socrate) .

/* Prédicats */
homme(X) :- grec(X) .
mortel(X) :- homme(X) .
```


Deuxième exemple

```
/* les entrées */
entree(crudites).
entree(terrines).
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).

/* composition d'un menu simple : une entrée ET un plat ET 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).
```

Troisième exemple

```
/* Le programme famille */

/* Base de faits */
pere(jerome, pierre).
pere(gerard, jerome).
pere(roger, gerard).
pere(roger, martine).

/* Base de règles */

grand_pere(X, Y) :- pere(X, Z), pere(Z, Y).

ancetre(X, Y) :- pere(X, Y); pere(X, Z), ancetre(Z, Y).

/* La définition suivante est identique à la précédente

ancetre(X, Y) :- pere(X, Y).
ancetre(X, Y) :- pere(X, Z), ancetre(Z, Y).

*/
```

Les éléments fondamentaux de Prolog

- Les termes
- Le programme
- Les requêtes

Les termes

- Les constantes
 - Les nombres : 7; - 12,5
 - Les chaînes de caractères : 'Pierre', 'X'
 - Les atomes
 - Les atomes « standards » : atome, crudités
 - Les atomes « protégés » : exit
 - Les atomes « symboliques » : ? + -
- Les variables

X, VARIABLE, _G34, _
- Les structures

nom (arg1, arg2, ... , argn); [a,b]

Le programme

- Un programme Prolog
 - = un ensemble de clauses
 - = des faits + des règles
- Les faits concernent des objets particuliers.
- Les règles concernent des catégories d'objets.

Les faits

- Les faits sont des affirmations qui décrivent des relations ou des propriétés.
- Forme : `prédicat(arg1, arg2, ..., argn)`
- Un prédicat est identifié par son nom et par son arité : `prédicat/n`
- Exemples :

| | |
|---|-------------------------|
| <code>entree(crudites).</code> | → <code>entree/1</code> |
| <code>homme(cesar).</code> | → <code>homme/1</code> |
| <code>pere(jerome, pierre).</code> | → <code>mange/2</code> |
| <code>eleve(albert, 1982, maths, 8).</code> | → <code>eleve/4</code> |

Les règles

- Les règles permettent d'exprimer des conjonctions de buts.
- Forme générale :
 - $F :- F1, F2, \dots, Fn.$
 - si $F1, F2, \dots, Fn$ sont vraies alors la tête est aussi vraie.
 - clause de Horn
- Exemples :
 - « tous les herbivores mangent de l'herbe »
 - « tous les animaux qui volent sont des oiseaux »

Les règles

- Une conjonction de termes → une seule règle
 /* composition d'un menu simple (sans boisson) : une entrée ET un plat ET un dessert */

$$\text{menu_simple}(E, P, D) :- \text{entree}(E), \text{plat}(P), \text{dessert}(D).$$
- Une disjonction de termes → plusieurs règles
 /* le plat de résistance : viande OU poisson */

$$\text{plat}(P) :- \text{viande}(P).$$

$$\text{plat}(P) :- \text{poisson}(P).$$

Exercice

Identifier les objets, les faits, les règles dans le texte suivant:


- La chèvre est un animal herbivore.
- Le loup est un animal cruel.
- 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 et ce qu'il mange.

- Y'a-t-il un animal cruel et que consomme-t-il ?


Solution

- Objets
 - chevre; loup; viande; herbe; eau

- Propriétés et les relations:
 - herbivore(X)
 - cruel(X)
 - carnivore(X)
 - manger(X,Y)
 - boire(X,Y)
 - consommer(X,Y)




ENSIM
 École d'Ingénieurs
 de Nantes Université




Solution

- Les faits
- Les règles

19
Représentation et modélisation des connaissances : Prolog
Y.S | 26/04/2022



ENSIM
 École d'Ingénieurs
 de Nantes Université



Les questions et les requêtes

- Une question a la forme d'une formule atomique pouvant contenir des variables.
 - Question = requête = (clause) but
- Deux types de questions
 - Question close (ne contient pas de variable)
 - Question non-close (contient des variables)
- Exemples:
 - Y'a-il du melon à la carte? :
?- entree(melon) .
 - Y'a-t-il un animal cruel et que consomme-t-il? :
?- cruel(X), consommer(X,Y) .

20
Représentation et modélisation des connaissances : Prolog
Y.S | 26/04/2022

Exemples : interrogation de la carte

- 1er cas : "Y a-t-il du melon à la carte" ?
 ?- entree(melon). clause but ou requête
 Yes il y a "effacement" (satisfaction) de la requête, et passage à la requête suivante.
- 2ème cas : "Y a-t-il du pâté à la carte" ?
 ?- entree(pate).
 No réponse de l'interpréteur, requête non effacée.
- 3ème cas : "Quelles sont les entrées figurant sur la carte" ?
 ?- entree(X). requête avec **variable**
 X = crudites; effacement de la requête avec la 1ère clause du "paquet"
 entrée, 1ère "solution" ou réponse à la question
 X = terrine; autre possibilité, avec la 2ème clause du "paquet" entrée
 X = melon; autre possibilité, avec la 3ème clause
 No il n'y a plus de solution

L'interpréteur fournit ici successivement toutes les valeurs possibles pour la variable X, telles que le remplacement de X par une de ces valeurs dans la requête corresponde à une clause donnée dans la carte (i.e. une clause du programme).

Modèle d'exécution : unification et arbre de résolution

Objectifs:

- comprendre le mécanisme de « raisonnement » du langage Prolog

Unification (1)

■ Exemple :

- $\text{frere}(X,Y) :- \text{homme}(X), \text{enfant}(X,Z), \text{enfant}(Y,Z), X \neq Y.$
où \neq représente le prédicat de différence
- $\text{frere}(\text{patrick}, \text{Qui})$: tentative d'unification avec la tête de la clause $\text{frere}(X,Y)$

■ Définition :

procédé par lequel on essaie de rendre deux formules identiques en donnant des valeurs aux variables qu'elles contiennent.

Unification (2)

■ Règles d'unification

| | |
|--------------------------------------|--|
| variable = terme terme = variable | La variable est liée au terme, même si celui-ci est une autre variable |
| constante = constante | Deux constantes (atomes ou nombres) ne peuvent s'unifier que si elles sont égales |
| structure = structure | Deux structures ne peuvent s'unifier que si elles ont le même prédicat, la même arité, et si leurs arguments peuvent s'unifier deux à deux |

■ Unification de deux structures :

- Consiste à rendre identiques les deux structures :
 - Parcourir (récursivement) les deux structures en parallèle
 - Lier des variables à des termes
 - Obtenir une substitution σ

Unification (3)

- Résultat : c'est un **unificateur** (ou **substitution**), un ensemble d'affectations de variables.
 - Exemple : $\{X=patrick, Qui=Y\}$
- Un terme t_1 est une instance de t_2 s'il existe une substitution σ telle que : $t_1 = \sigma t_2$ (t_2 est un terme plus général que t_1)
- Le résultat n'est pas forcément unique, mais représente l'unificateur le plus général.
- L'unification peut réussir ou échouer.
 - $e(X,X)$ et $e(2,3)$ ne peuvent être unifiés.

Unification (4)

- $a(B,C) = a(2,3)$. donne pour résultat :
YES $\{B=2, C=3\}$
- $a(X,Y,L) = a(Y,2,carole)$. donne pour résultat :
YES $\{X=2, Y=2, L=carole\}$
- $a(X,X,Y) = a(Y,u,v)$. donne pour résultat :
NO

Unification (5)

frere(patrick, Qui)

Unification avec frere(X,Y)

X=patrick et Qui=Y

homme(patrick)
parent(patrick,Z)
parent(Y,Z)
patrick\=Y

Points de choix

- Plusieurs règles concernant une même question :
 - essais consécutifs dans l'ordre de déclaration
 - représentation sous forme d'arbre
 - les nœuds de l'arbre sont appelés **points de choix**

- Exemple :

□ ?- plat(Y).
plat(P) :- viande(P).
plat(P) :- poisson(P).

Arbre de recherche

- On parle d'arbre de recherche d'une question
 - Racine de l'arbre : question
 - Nœud : points de choix (formule à démontrer)
 - Passage d'un nœud vers son fils en considérant l'une des règles et en effectuant une unification et un pas de démonstration
 - Nœuds de gauche à droite dans l'ordre de déclaration des règles
 - Nœuds d'échec
 - Nœuds de succès

Stratégie de Prolog

- Pour résoudre une question, Prolog construit l'arbre de recherche de la question
- Parcours en profondeur d'abord
 - nœud de succès : c'est une solution, Prolog l'affiche et cherche d'autres solutions
 - nœud d'échec : remontée (**backtrack**) dans l'arbre jusqu'à un point de choix possédant des branches non explorées. Si un tel nœud de choix n'existe pas, la démonstration est terminée, il n'y a pas d'autre solution.
- Possibilité de branche infinie et donc de recherche sans terminaison...

Exemple : sur les menus

- Arbres de résolution de :

Quels sont les ancêtres de Pierre ? (réf. Troisième exemple)

Les entrées / sorties

Objectifs :

- écriture à l'écran ou dans un fichier
- lecture à partir du clavier ou d'un fichier

Affichage des termes

- **write\1** : écriture dans le flot de sortie d'atomes

- write(coucou). → coucou
- write(Coucou). → erreur sauf si Coucou est une variable connue

- **nl\0** : passage à la ligne

- **tab\1** : affichage d'espaces

- write(coucou), nl, tab(3), write(coucou)
- coucou
 coucou

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.

- menu:- read(X), write(coucou), write(X).
- test. (rentré par l'utilisateur)
 coucou test
 yes

Sur les fichiers

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

Quelques clauses de base

- Changer de répertoire de travail
`chdir('~/.prolog/tp1') .`
- Charger un fichier
`consult(menu) .`
`consult('menu.pro') .`
Dans Amzi!, revient à lancer le run sur le fichier menu.pro
- Affichage des clauses
`listing .`
`listing(plat) .`
- Quitter
`halt. (sous SWI-prolog)`
`quit. (sous Amzi)`

Le langage PROLOG (cours 2)

Représentation et Modélisation des connaissances

Objectifs :

- Opérateurs & Arithmétique
- Structures de contrôle
- Listes

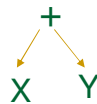
Opérateurs et expressions arithmétiques

Expressions arithmétiques

- Prolog connaît les entiers et les nombres flottants.
- Syntaxe habituelle pour les opérateurs classiques +, -, *, division entière (symbole //), division flottante (symbole /).
- Différence : opérateur infixe **is** qui permet d'évaluer les expressions :
 - ?- X is 3+2. → {X=5}
 - ?- X is 8 // 3. → {X=2}
 - ?- X is 4 * (3+2). → {X=20}

Représentation des expressions

- Les expressions sont représentées par des **arbres** Prolog.
- l'expression (X+Y) est représentée par l'arbre +(X,Y)



- ?- display(2+3 * (sqrt(X)-abs(X))).
 → +(2,*(3,-(sqrt(H1),abs(H1))))
 X = H1 ; no.

- Donc :
 - ?- 3+2 = 2+3. → no, 2 arbres différents
 - ?- 3+2 is 2+3. → no, la partie droite est évaluée
et la partie gauche est un arbre

Expressions arithmétiques

- Expressions et unification : attention à certaines tentatives d'unification
 - la tentative d'unification entre $3+2$ et 5 échouera. En effet, l'expression $3+2$ est un arbre alors que 5 est un nombre.
 - $\langle ?Terme \rangle$ **is** $\langle +Expression \rangle$ s'efface si *Terme* est unifiable avec le résultat de l'évaluation de *Expression*
 - $?- 5 \text{ is } 2 + 3.$ s'efface (yes)
 - $?- N \text{ is } 2 + 3.$ s'efface et donne: $N = 5$
 - L'évaluation des expressions ne fait pas partie de l'algorithme d'unification.

Prédicats de comparaison

- Comparaison pour les expressions arithmétiques :
 - Prédicats binaires et infixés;
 - Évaluation des expressions à gauche et à droite de l'opérateur ;
- | | |
|--------------|-----------------------------------|
| □ $X =:= Y$ | → X est égal à Y |
| □ $X \neq Y$ | → X est différent de Y |
| □ $X < Y$ | → X est strictement inférieur à Y |
| □ $X \leq Y$ | → X est inférieur ou égal à Y |
-
- Exemples :

| | |
|-----------------|-------|
| $5+2 \leq 5+3.$ | → yes |
| $5+2 =:= 5+3.$ | → no |
| $5+2 \neq 5+3.$ | → yes |

Évaluation, identité formelle, unification

■ Évaluation des expressions arithmétiques

$5+3 \text{ } := \text{ } 3+5. \rightarrow \text{yes}$

$5+3 \text{ } := \text{ } 5+3. \rightarrow \text{yes}$

Chaque expression est évaluée avant la comparaison.

■ Identités formelles

$5+3 == 5+3. \rightarrow \text{yes}$ (les termes sont égaux)

$5+3 == 3+5. \rightarrow \text{no}$ (les termes sont différents)

$\langle \text{terme1} \rangle == \langle \text{terme2} \rangle$ s'efface si les 2 termes sont identiques formellement.

■ Unification (il existe une substitution)

$5+3=5+3. \rightarrow \text{yes}$ (termes unifiables)

$5+3=3+5. \rightarrow \text{no}$ (termes non unifiables)

Évaluation, identité formelle, unification

■ ?- A=3.

$\rightarrow A = 3 ; \text{no}$

il existe une substitution $\{A:3\}$

■ ?- A==3.

$\rightarrow \text{no}$

la variable A n'est pas connue, donc ne peut pas être identique à 3

■ ?- A=3, A==3.

$\rightarrow A = 3 ; \text{no}$

■ ?- A==3, A=3.

$\rightarrow \text{no}$

Évaluation, identité formelle, unification

■ Négation :

□ évaluation \models

négation \models → résultats de l'évaluation sont différents

□ identité formelle \equiv

négation \equiv → les termes sont non identiques

□ unification $=$

négation $=$ → les termes ne sont pas unifiables

Les opérateurs (1)

■ Trois types :

□ Opérateur binaire infixé : il figure entre ses 2 arguments et désigne un arbre binaire

$X + Y$ *patrick aime X*

□ Opérateur unaire préfixé : il figure avant son argument et désigne un arbre unaire

$-X$ *rouge(ferrari)*

□ Opérateur unaire suffixé : il figure après son argument et désigne un arbre unaire

$X -$ *ferrari rouge*

| | |
|--|---------------------------------------|
| <code>rouge(ferrari).</code> | <code>ferrari rouge.</code> |
| <code>aime(patrick, X):-rouge(X).</code> | <code>patrick aime X:-X rouge.</code> |

Les opérateurs (2)

- Déclaration des opérateurs :
 - possibilité de modifier la syntaxe Prolog en définissant de nouveaux opérateurs
 - définition : par l'enregistrement de faits de la forme suivante **op(Priorité, Spécification, Nom)**
 - **Nom** : nom de l'opérateur
 - **Priorité** : compris entre 0 (le plus prioritaire) et 1200
 - **Spécification** :
 - type de l'opérateur (infixé, associatif...):
 - Unaire : fx, xf, fy, yf
 - Binaire : xfx, xfy, yfx, yfy

Les opérateurs (3)

- Exemple de déclaration d'un opérateur :
 - **op(1000, xfx, aime)** définit un opérateur infixé non associatif *aime*
 - Dans ce cas, Prolog traduira une expression du type *X aime Y* en le terme *aime(X, Y)*, et si Prolog doit afficher le terme *aime(X, Y)*, il affichera *X aime Y*.

Exemple

- Définition des opérateurs :
 $op(500, xf, est_carnivore).$
 $op(500, xf, est_herbivore).$
 $op(500, xf, est_cruel).$
 $op(600, xfx, mange).$
 $op(600, xfx, boit).$
 $op(600, xfx, consomme).$
- Base de faits :
 $chevre\ est_herbivore$
 $loup\ est_cruel$
 $X\ est_carnivore := X\ est_cruel$
 $X\ mange\ viande := X\ est_carnivore$
 ...

Coupure et contrôle de l'interpréteur

Notion de coupure

- Différents noms : *coupure*, *cut* ou *coupe-choix*
- Introduit un contrôle du programmeur sur l'exécution de ses programmes
 - * en élaguant les branches de l'arbre de recherche
 - * rend les programmes plus simples et efficaces
- Différentes notations : **!** ou **/** sont les plus courantes
- Le coupe-choix permet de signifier à Prolog qu'on ne désire pas conserver les points de choix en attente.

Notion de coupure

- Le coupe-choix permet :
 - d'éliminer des points de choix
 - d'éliminer des tests conditionnels que l'on sait inutile
- Quand Prolog démontre un coupe-choix, il détruit tous les points de choix créés depuis le début de l'exploitation du paquet des clauses du prédicat où le coupe-choix figure.

Exemples

- `menu_simple(E,P,D).`
- `menu_simple(E,P,D),!.`
- `menu_simple(E,P,D),poisson(P).`
- `menu_simple(E,P,D),poisson(P),!.`
- `menu_simple(E,P,D),!,poisson(P).`

Repeat et fail

- Le prédicat *fail/0* est un prédicat qui n'est jamais démontrable, il provoque donc un échec de la démonstration où il figure.
- Le prédicat *repeat/0* est un prédicat prédéfini qui est toujours démontrable mais laisse systématiquement un point de choix derrière lui. Il a une infinité de solutions.
- L'utilisation conjointe de *repeat/0*, *fail/0* et du *coupe-choix* permet de réaliser des boucles.

Prédicats prédéfinis

- **assert/1** : ajout de clauses dans la base de faits (BF)
 - ?-assert([a,b,c,d]).
 - yes
 - ?-listing.
 - [a,b,c,d]
 - yes
- **retract/1** : retrait de clauses de la BF
 - ?-retract([a,b,c,d]).
 - yes
 - ?-listing.
 - yes
 - ?- retract(p(X,Y):-Q), fail. → efface toutes les clauses p/2

La négation en Prolog

- L'argument de **not/1** est défini par
 - not (X) :- ¬X, !, fail .**
 - not (X) .**

Si X s'efface alors not(X) échoue, sinon not(X) réussit.
- Négation par l'échec (**raisonnement en monde fermé**):
 - Un but not(P) est réussi lorsque la résolution du but P échoue
 - Ce qui ne peut pas être prouvé est considéré comme étant faux

La négation en Prolog

■ Deux contraintes :

- Un **not** ne peut intervenir que dans le corps d'une clause
- Pour satisfaire **not(P)**, il faut que toutes les variables de P soient liées

■ Exemples:

```
sportif(X) :- fort(X), not(petit(X)).
```

```
moyen(X) :- personne(X), not(grand(X)), not(petit(X)).
```

■ Contre Exemples :

```
not(grand(X)) :- petit(X).
```

```
moyen(X) :- not(grand(X)), not(petit(X)).
```

Listes

Termes simples et Termes composés

- Termes plus général en Prolog qu'en LPPO
- Termes simples
 - constantes
 - variables
- Termes composés
 - arbres binaires
 - listes

Arbres binaires

- Un arbre binaire est :
 - soit vide : **abv/0**
 - soit composé d'une racine et de 2 sous arbres binaires
ab(Rac, FG, FD)
- les arbres peuvent être utilisés comme données pour des prédicats. Ils peuvent également contenir des variables.
 - Deux arbres sont unifiables si:
 - ils possèdent le même nom
 - ils possèdent la même arité
 - et si leur argument sont unifiables 2 à 2

Listes

- Structure de données
 - traitements récursifs
 - très utilisées
- Définition récursive :
 - la liste vide, représentée par $[]$, est une liste,
 - si T est un terme et L une liste, le terme $.(T, L)$ représente la liste de premier élément T (ou "tête de liste"), et L est la liste privée du premier élément (ou "queue de liste").
 - \bullet est l'opérateur binaire de séquence.

Listes

- Notations
 - $.(T, L)$
 - $[T | L]$, avec l'opérateur $|$ (« cons ») de construction de liste.
- Exemples
 - $.(a, [])$ est représentée aussi par $[a]$, liste d'un élément
 - $.(a, .(b, []))$ équivalent à $[a, b]$ ou $[a | [b]]$
 - $.(a, .(b, .(c, [])))$ équivalent à $[a, b, c]$ ou $[a | [b, c]]$ ou $[a, b | [c]]$

Exemples

- Plusieurs représentations de la même structure de liste :

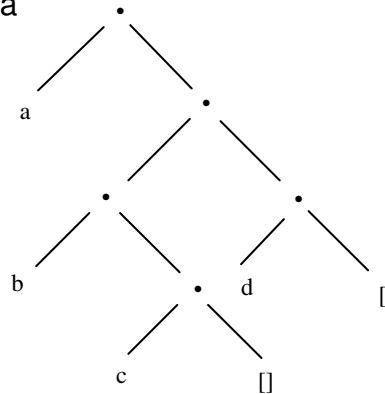
$\text{.(a, .(.(b, .(c, [])), .(d, [])))}$

$[a, [b, c], d]$

$[a \mid [[b, c], d]]$

$[a, [b, c] \mid [d]]$

- dont l'arbre binaire est le suivant :



Unification sur les listes

- Une liste non vide est représentée par $[X \mid Ls]$
- ?- $[[il, fait], beau, [a, paris]] = [X, Y]$.
 - no, une liste de 3 éléments ne peut s'unifier à une liste de 2 éléments
- ?- $[a, [b, c], d] = [X, Y, Z]$.
 - $X = a; Y = [b, c]; Z = d$;no
- ?- $[a, b, c, d] = [a, b \mid L]$.
 - $L = [c, d]$;no
- ?- $[a, [b, c], d] = [a, b \mid L]$.
 - no

Prédicats sur les listes

- appartenance d'un terme à une liste:

```
element (X, [X|_]) .
```

```
element (X, [_|Ls]) :- element (X, Ls) .
```

- concaténation de deux listes

/* concat(L1,L2,LR) : "LR est la liste résultant de la concaténation des deux listes L1 et L2" */

```
concat ([ ], L, L) .
```

```
concat ([X|Ls], L2, [X|Lc]) :- concat (Ls, L2, Lc) .
```

Prédicats sur les listes

- Longueur d'une liste

```
long ([ ], 0) .
```

```
long ([X|Ls], N) :- long (Ls, N1), N is N1 + 1 .
```

- Somme des éléments d'une liste

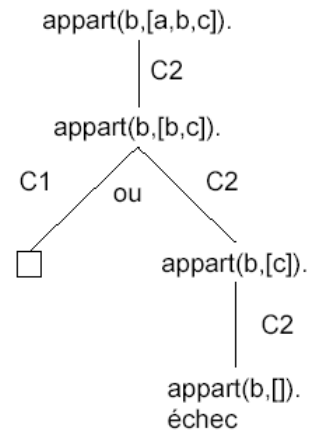
```
somme ([ ], 0) .
```

```
somme ([A|B], C) :- somme (B, D), C is D+A .
```

Coupure et Listes

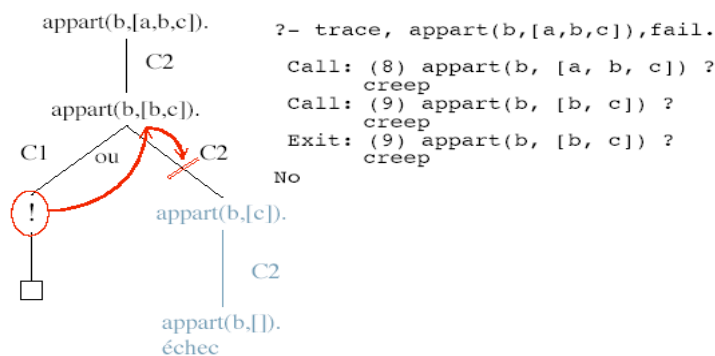
- Appartenance à une liste:
`appart(X, [X|_]) .`
`appart(X, [_|L]) :- appart(X, L)`

- Requête:
b appartient-il à la liste [a,b,c]

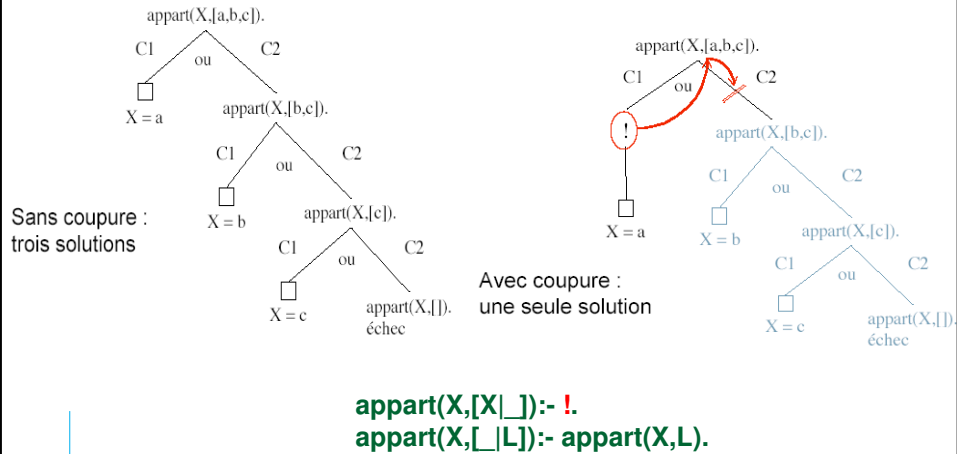


Coupure et Listes (2)

- La coupure interdit le retour arrière sur les points de choix:
`appart(X, [X|_]) :- ! .`
`appart(X, [_|L]) :- appart(X, L) .`



Coupure et Listes (3)



Prédicats sur les listes

Objectif :

Supprimer un élément : **supprimer(+X,+Xs,?Ys)**

version sans coupure :

Exemple de requêtes :

?- **supprimer(a,[b,a,c,d],Y)** .

?- **supprimer(a,X,[b,c,d])** .

Prédicats sur les liste

□ Objectif :

supprimer un élément d'une liste : **supprimer(+X,+Xs,?Ys)**

□ Version avec coupure :

□ Exemple de requêtes :

?- **supprimer(a, [b,a,c,d], Y)** .

Prédicats sur les liste

□ Objectif :

dupliquer les éléments d'une liste : **double(+Xs,?Ys)**

□ Solution :

□ Exemples de requêtes :

?- **double([a,b,c], X)** .

■ **X = [a, a, b, b, c, c] ; No**

?- **double([a,b,c], [a,a,b,b,c,c])** .

?- **double([[a,a],[b,b],[d]], X)** .

?- **double(X, [a,a,b,b,c,c])** .

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le troisième élément d'une liste : **troisieme(+Xs,?T).**

□ Exemples de requêtes :

?- troisieme([a,b,c,d,e,f],F) .

?- troisieme([a],F) .

?- troisieme([a,b],F) .

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le troisième élément d'une liste : **troisieme(+Xs,?T)**

□ Solution :

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le troisième élément d'une liste : **troisieme(+Xs,?T)**.

□ Exemples de requêtes :

?- troisieme([a,b,c,d,e,f],F) .

?- troisieme([a],F) .

?- troisieme([a,b],F) .

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le dernier élément d'une liste : **dernier(+Xs, ?T)**

□ Exemples de requêtes :

?- dernier([a,b,c,d,e,f],F) .

?- dernier([a],F) .

?- dernier([],F) .

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le dernier élément d'une liste :
dernier (+Xs, ?T)

□ Solution :

Prédicats sur les liste

□ Objectif :

Écrire un prédicat permettant d'obtenir le $n^{\text{ième}}$ élément d'une liste :
nelement (+Xs, ?T)

□ Exemples de requêtes :

?- nelement (2, [a, b, c, d, e, f], F) .

?- nelement (3, [a], F) .

?- nelement (-2, [a, b, c, d], F) .

Prédicats sur les liste

□ Solution :

□ Exemples de requêtes :

?- nelement (2, [a,b,c,d,e,f], F) .

?- nelement (3, [a], F) .

?- nelement (-2, [a,b,c,d], F) .

Prédicats sur les liste

□ Objectif :

Dénombrer le nombre d'occurrence d'un élément d'une liste
solution : **occurrences(+X,+L,?N)**

□ Solution :

□ Exemple de requêtes :

?- occurrences (d, [a,d,f,a,d], X) .

?- occurrences (d, [a,d,t,a,x], X) .

Retour sur les fondements de Prolog (cours 3)

Objectifs :

- Retour sur l'unification : quelques précisions
- Lien avec la logique : synthèse

Retour sur l'unification

Précisions

Unificateur

Définition : Un unificateur de deux termes t_1 et t_2 est une substitution σ telle que : $\sigma t_1 = \sigma t_2$

Exemples :

Si $A = \text{point}(X, Y, 12)$ et $B = \text{point}(U, 13, V)$,
alors $V1 = \{X=U, Y=13, V=12\}$ est un unificateur de A et B ,
puisque $V1(A) = V1(B) = \text{point}(U, 13, 12)$.

De même

$V2 = \{X=15, Y=13, V=12\}$ est un autre unificateur de A et B ,
puisque $V2(A) = V2(B) = \text{point}(15, 13, 12)$.

Unificateur le plus général

Lorsque deux termes sont unifiables, ils peuvent admettre plusieurs unificateurs.

Dans l'exemple précédent, l'unificateur $V1$ est préférable à $V2$, le terme **point(U,13,12)** étant plus général que **point(15,13,12)**.

$\Rightarrow V1$: l'unificateur le plus général

Unificateur le plus général

Unificateur le **plus** général (**upg**) :

Un unificateur σ de t_1 et t_2 est un **upg** si pour tout autre unificateur σ' de t_1 et t_2 il existe une autre substitution σ'' telle que : $\sigma' = \sigma'' (\sigma)$

upg (**m.g.u.** – **most general unifier**)

Algorithme d'unification (algorithme de Robinson)

```

fonction upg(t1:terme, t2:terme)
  si t1 == t2 alors retourner({ })
  si t2 est une variable alors permuter t1 et t2
  si t1 est une variable alors
    si t2 contient t1
      alors retourner(échec)
    sinon retourner({ t1 = t2 })
  sinon
    si t1 et t2 sont 2 structures de même prédicat et de même arité
      alors
        soit t1 = pred(u1, ... un) et t2 = pred(v1, ... vn)
        s = { }
        pour i allant de 1 à n faire
          s' = upg(s(ui), s(vi))
          si s' == échec alors retourner(échec)
          s = s' (s)
        sinon retourner(échec)

```

Algorithme de résolution

Quand on pose une question à l'interprète Prolog, celui-ci exécute dynamiquement l'algorithme suivant. L'arbre constitué de l'ensemble des appels récursifs est appelé *arbre de recherche*

```
fonction prouver(lbut:liste, lsubst:liste)
  si lbut == [] alors
    /*la liste de buts est vide donc le but initial est prouvé */
    retourner(lsubst)
  sinon
    soit lbut = [A1, A2, ... An]
    pour toutes les clauses (C :- C1, C2, ... Cr) faire
      /*on suppose que les variables ont été renommées */
      s = upg(A1, C)
      si s != échec alors
        newlbut = [s(C1), s(C2), ... s(Cr), s(A2), ... s(An)]
        newlstbst = lsubst + s
        prouver(newlbut, newlstbst)
```

Lien avec la logique

Synthèse

Clauses de Horn

□ Définition:

une **clause de Horn** possède un et un seul littéral positif

□ Exemples de clauses de Horn :

femme(victoria).

homme(edward).

soeur(X, Y) :- femme(X), parents(X, Mere, Pere), parents(Y, Mere, Pere).

□ Contre exemples :

« Les barbiers rasant tous ceux qui ne se rasant pas eux-mêmes »

$\neg \text{barbier}(X) \vee \text{rase}(X, Y) \vee \text{rase}(Y, Y)$

« Aucun barbier ne rase quelqu'un qui se rase lui-même »

$\neg \text{barbier}(X) \vee \neg \text{rase}(X, Y) \vee \neg \text{rase}(Y, Y)$

Représentation des connaissances

| Écritures logiques | Clauses de Horn | Écritures Prolog |
|----------------------------|-----------------------------|------------------|
| $\Rightarrow D$ | D | $D.$ |
| $A \Rightarrow$ | $\neg A$ | $:- A.$ |
| $A \wedge B \Rightarrow$ | $\neg A \vee \neg B$ | $:- A, B.$ |
| $A \wedge B \Rightarrow C$ | $\neg A \vee \neg B \vee C$ | $C :- A, B.$ |

Représentation des connaissances

Écritures logiques

$\text{homme}(X) \wedge \text{parents}(X, \text{Mere}, \text{Pere}) \wedge$
 $\text{parents}(Y, \text{Mere}, \text{Pere}) \Rightarrow \text{frere}(X, Y)$

Clauses de Horn

$\text{frere}(X, Y) \vee \neg \text{homme}(X) \vee$
 $\neg \text{parents}(X, \text{Mere}, \text{Pere}) \vee$
 $\neg \text{parents}(Y, \text{Mere}, \text{Pere}) .$

Écritures Prolog

$\text{frere}(X, Y) \text{ :- homme}(X),$
 $\text{parents}(X, \text{Mere}, \text{Pere}),$
 $\text{parents}(Y, \text{Mere}, \text{Pere}) .$

Résolution

En logique, on peut toujours transformer une formule par un ensemble de clauses;

Pour démontrer qu'une formule B est une conséquence d'un ensemble de clauses E, on procède par réfutation;

Résolution par réfutation

Pour prouver que H est la conséquence logique de G:

on transforme G et H en ensemble de clauses;

on applique le principe de résolution par réfutation à

$G \vee \neg H$ jusqu'à trouver la clause vide

Ce principe est complet pour les clauses de Horn.

Résolution par réfutation

Procédure : résoudre(S_0 :ensemble des clauses initiales)

$S = S_0$

Répéter

/ sélectionner 2 clauses distinctes solvables C_i et C_j */*

/ ce choix doit être fait en fonction d'une stratégie */*

calculer $C_{ij} = \text{Resolvante}(C_i, C_j)$

$S = S \setminus \{ C_i, C_j \} + \{ C_{ij} \}$

Jusqu'à ce que $S == \emptyset$ */* clause vide */*

Clauses résolvantes ou résolvants

Exemple :

$$C1 = \neg P \vee Q$$

$$C2 = \neg Q \vee R$$

$$C3 = \text{Res}(C1, C2) = \neg P \vee R$$

$$C1 = P$$

$$C2 = \neg P \vee Q$$

$$C3 = \text{Res}(C1, C2) = Q$$

$$C1 = P$$

$$C2 = \neg P$$

$$C3 = \text{Res}(C1, C2) = \emptyset$$

Exemple de problème

Jean et Alain sont deux personnages dont l'humeur est régie par ce principe général assez réaliste :

«Jean et Alain sont de bonne humeur s'ils ont de l'argent et s'ils sont en vacance au soleil, ou bien s'ils réussissent à la fois dans le travail et dans leurs familles respectives »

Par ailleurs, on sait que :

Jean et Alain ont tout deux de l'argent

Jean et Alain réussissent dans leur travail

Jean part en vacances en août et Alain en juillet

Il y a du soleil en août mais on est en juillet

Alain réussit dans sa famille

Question : *qui est heureux ?*

Formalisation du problème

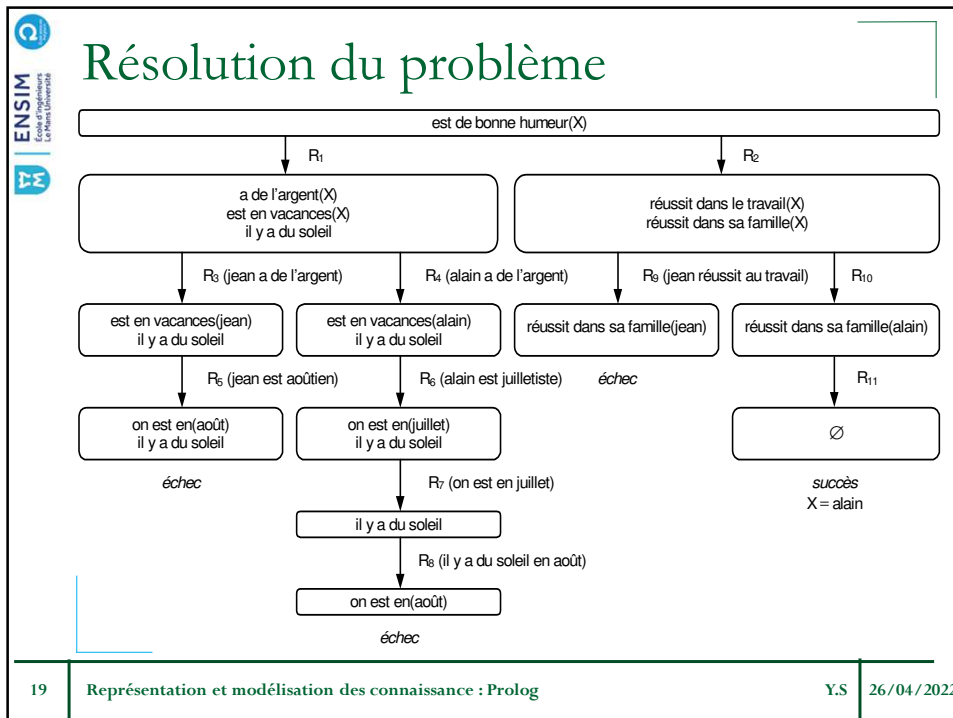
Ensemble de clauses de Horn :

R_1 $\text{est_de_bonne_humeur}(X) \vee \neg \text{a_de_l_argent}(X) \vee$
 $\neg \text{est_en_vacances}(X) \vee \neg \text{il_y_a_du_soleil}.$
 R_2 $\text{est_de_bonne_humeur}(X) \vee \neg \text{reussit_dans_le_travail}(X)$
 $\vee \neg \text{reussit_dans_sa_famille}(X).$
 R_3 $\text{a_de_l_argent}(\text{jean}).$
 R_4 $\text{a_de_l_argent}(\text{alain}).$
 R_5 $\text{est_en_vacances}(\text{jean}) \vee \neg \text{on_est_en}(\text{aout}).$
 R_6 $\text{est_en_vacances}(\text{alain}) \vee \neg \text{on_est_en}(\text{juillet}).$
 R_7 $\text{on_est_en}(\text{juillet}).$
 R_8 $\text{il_y_a_du_soleil} \vee \neg \text{on_est_en}(\text{aout}).$
 R_9 $\text{reussit_dans_le_travail}(\text{jean}).$
 R_{10} $\text{reussit_dans_le_travail}(\text{alain}).$
 R_{11} $\text{reussit_dans_sa_famille}(\text{alain}).$

Résolution du problème

$\text{est_de_bonne_humeur}(X).$

$R_1 \rightarrow \text{a_de_l_argent}(X), \text{est_en_vacances}(X), \text{il_y_a_du_soleil}.$
 $R_3 \rightarrow \text{est_en_vacances}(\text{jean}), \text{il_y_a_du_soleil}.$
 $R_5 \rightarrow \text{on_est_en}(\text{aout}), \text{il_y_a_du_soleil}.$
 $R_4 \rightarrow \text{est_en_vacances}(\text{alain}), \text{il_y_a_du_soleil}.$
 $R_6 \rightarrow \text{on_est_en}(\text{juillet}), \text{il_y_a_du_soleil}.$
 $R_7 \rightarrow \text{il_y_a_du_soleil}.$
 $R_8 \rightarrow \text{on_est_en}(\text{aout}).$
 $R_2 \rightarrow \text{reussit_dans_le_travail}(X), \text{reussit_dans_sa_famille}(X).$
 $R_9 \rightarrow \text{reussit_dans_sa_famille}(\text{jean}).$
 $R_{10} \rightarrow \text{reussit_dans_sa_famille}(\text{alain}).$
 $R_{11} \rightarrow \emptyset$



Résolutions parallèles Prolog/ Réfutation :

■ Clauses Prolog :

D :- A, B.

A :- C.

B.

C.

■ But :

:- D.

Liste de buts = {A, B}

Liste de buts = {C, B}

Liste de buts = {B}

Liste de buts = { }

Clauses de Horn :

$\neg A \vee \neg B \vee D$ (C1)

$\neg C \vee A$ (C2)

B (C3)

C (C4)

But :

$\neg D$ (C5)

C6 = Res(C5, C1) = $\neg A \vee \neg B$

C7 = Res(C6, C2) = $\neg C \vee \neg B$

C8 = Res(C7, C4) = $\neg B$

C9 = Res(C8, C3) = \emptyset

20 Représentation et modélisation des connaissances : Prolog Y.S 26/04/2022

Le langage PROLOG (cours 4)

Représentation et Modélisation des connaissances

Objectif : Implémenter un joueur artificiel dans un jeu à 2 joueurs.

Qu'est-ce qu'un jeu à deux joueurs ?

- Conditions :
 - Deux joueurs adverses;
 - Alternance des coups jusqu'à obtention d'un état terminal {gagnant, perdant, nul};
 - Chacun veut maximiser ses gains et minimiser ceux de l'adversaire (hypothèse pour la modélisation);
 - Connaissance parfaite du jeu à chaque instant;
- Exemples:
 - Jeu de Nim (jeu des allumettes)
 - Tic-Tac-Toe
 - Échecs
 - Dames
- Contre-exemples:
 - jeux de hasard (dès)
 - jeux de cartes

Pourquoi étudier les jeux ?

- Jeux:
 - Micro-mondes pour l'étude de stratégie;
 - Limiter le monde réel à un monde réduit et réaliser un système qui agit sur ce monde réduit.
- Intérêts:
 - Univers limité;
 - Règles spécifiées mais riches en possibilités de déduction;

Jeux à 2 joueurs en Prolog

- 2 composants principaux dans le système:
 - interface de saisie (propre à chaque jeu)
 - saisie des coups des joueurs (humain ou artificiel)
 - affichage de l'état de la partie à un moment donné
 - modélisation du joueur artificiel = moteur de réflexion
 - Partie propre au jeu:
 - Règles du jeu : coup valide ?
 - Connaissance sur le jeu : est-ce que le coup modifie la configuration du jeu ? (prendre une pièce au échec)
 - Partie commune à tous les jeux:
 - construction de l'arbre des coups possibles;
 - méthodes de sélection des coups (heuristique, minimax, alpha-beta...)

Étapes dans la réalisation du jeu

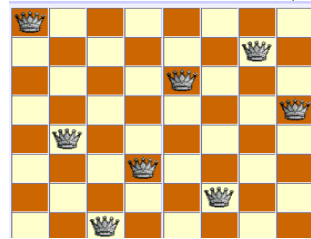
1. Définir la modélisation/représentation du jeu;
1. Gérer la visualisation du jeu ;
1. Modélisation des règles du jeu ;
1. Créer le moteur qui va permettre de lancer le jeu et de faire jouer un joueur contre un joueur artificiel ;
1. Améliorer le « joueur » artificiel : min-max, alpha-beta,...

Modélisation et représentation du jeu

Étape 1 : quelle est la structure de données qui permet de mieux résoudre le problème ?

Exemple des 8 reines

- Comment placer 8 reines sur un échiquier de 8x8 de telle manière qu'aucun couple de reines ne soit en attaque ?
- Rappel : La reine est la pièce la plus mobile et la plus puissante du jeu d'échecs. Elle se déplace d'un nombre quelconque de cases dans toutes les directions (lignes/colonnes/diagonales).



Exemple des 8 reines

- Modélisation du problème :
 - Solution 1 : Tableau à 2 dimensions
`grille[num_ligne][num_colonne]`
 avec `grille[5][2] = "reine";`
 - Solution 2 : Liste de couples de coordonnées
`Reines = [[1,1], [2,7], [3,5], [4,8], [5,2], [6,4], [7,6], [8,3]]`
 - Solution 3 : Liste de numéros de colonne (lignes forcément différentes)
 - la position de X_i dans la liste correspond au numéro de ligne de la reine;
 - la valeur de X_i correspond au numéro de la colonne.`Reines = [1,7,5,8,2,4,6,3]`
 Trouver la solution = Trouver une liste de nombres de 1 à 8 respectant certaines conditions.
- Il n'existe pas une solution meilleure *a priori*
- mais la solution 3 évite le superflu



Exemple le Tic-Tac-Toe

- Grille de 3*3 avec soit :
 - - case vide
 - x pour joueur 1
 - o pour joueur 2
 - Modélisation du problème
 - Solution 1: liste de listes
 $[[x,-,-],[-,o,-],[-,-,-]]$
 - Solution 2 : liste à 9 éléments
 $[x,-,-,-,o,-,-,-,-]$
- ➔ quelle solution choisir sachant qu'on va vouloir tester si une ligne / une colonne est remplie/gagnée ?

Visualisation du jeu

Étape 2

Exemple le Jeu de Nim

21 allumettes, 2 joueurs, allumettes sur la même ligne.

```
/* affichage du coup joué */
joueLeCoup(X) :- nl,write(X),write(' allumettes ont
été prises dans le tas').

/* saisie d'un coup */
saisieUnCoup(X) :- nl,write('veuillez saisir un
coup'), nl, read(X).

/* affichage d'une position */
affichePosition(X) :- nl, write('il reste '),
write(X),write(' allumettes').
```

Tic-Tac-Toe

- % Prédicat : afficheLigne/1

```
afficheLigne([A,B,C]) :- write(A), tab(3),
write(B), tab(3),
write(C), tab(3).
```
- % Prédicat : afficheGrille/1

```
afficheGrille([[A1,B1,C1],[A2,B2,C2],[A3,B3,C3]])
:-
    afficheLigne([A1,B1,C1]), nl,
    afficheLigne([A2,B2,C2]), nl,
    afficheLigne([A3,B3,C3]).
```

→ comment modifier les prédicats pour qu'ils soient plus génériques ?

→ exemple Tic-Tac-Toe TD3

Modélisation des règles du jeu

Étape 3

Règles du jeu

Qu'est-ce qu'un coup valide ?

- ❑ Jeu de Nim & échecs : est-ce que le coup joué respecte les règles du jeu ?
- ❑ Tic-Tac-Toe : est-ce que la case est libre ?
- Qu'est-ce qu'une position gagnante ?
- Qu'est-ce qu'une position perdante ?
- Influence d'un coup sur le jeu ?
 - ❑ Échecs : prendre une pièce de l'adversaire
 - ❑ Othello : retourner les cases de l'adversaire

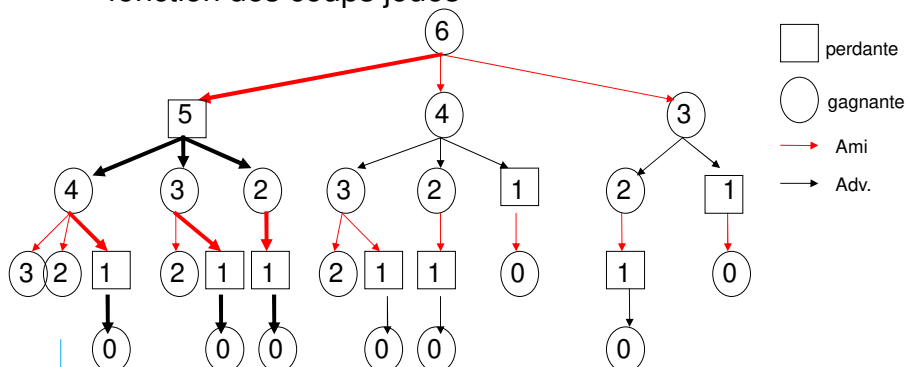
Exemple le Jeu de Nim

- 21 allumettes, 2 joueurs, allumettes sur la même ligne.
- Règles:
 - Prendre 1, 2 ou 3 allumettes;
 - Celui qui prend la dernière a perdu;

```
un(X,A) :- A is X - 1.
deux(X,A) :- A is X - 2.
trois(X,A) :- A is X - 3.
```

Exemple le Jeu de Nim

- Graphe d'états = Arbre de jeu = permet de représenter les différentes positions possibles et leurs liens en fonction des coups joués



Définitions

- **Position perdante** : celui qui joue après atteint *forcément* une position gagnante = toutes les positions « filles » sont gagnantes pour l'autre joueur
 - 5 est une position perdante pour ADV
- **Position gagnante** : celui qui joue après *peut* atteindre une solution perdante = au moins une position « fille » est gagnante
 - 6 est une position gagnante pour Ami

Exemple le Jeu de Nim

- Positions gagnantes et perdantes finales :


```
positionPerdante(1).
positionGagnante(2).
positionGagnante(3).
positionGagnante(4).
```
- Généralisation


```
coupValide(C, Dep, Arr) : la clause fait indiquant qu'un
coup est valide;

positionPerdante(N) :-
    coupValide(C1, N, N1), positionGagnante(N1),
    coupValide(C2, N, N2), positionGagnante(N2),
    coupValide(C3, N, N3), positionGagnante(N3).

positionGagnante(N) :-
    coupValide(C, N, Arr), positionPerdante(Arr).
```

Tic-Tac-Toe

→ TD3 – étape 3 : gestion des coups et du jeu

Moteur du programme (niveau simple)

Étape 4 : lancer le jeu et gérer
l'alternance des coups des joueurs

Principe général

1 - Choix d'un coup pour un joueur

- voir tous les cas possibles
- vérifier leur validité
- sélectionner un coup

2 - Jouer le coup

- modifier le plateau de jeu
- mise à jour

3 - Changer de joueur - retour à 1 avec le 2ième joueur

Exemple du Jeu de Nim

■ Choisir et jouer un coup pour un joueur humain

```
joueJoueur(Depart, Arrivee) :-
    saisieUnCoup(NBAjoue),
    coupValide(NBAjoue, Depart, Arrivee),
    joueLeCoup(NBAjoue).
```

■ Choisir et jouer un coup pour le joueur artificiel

```
joueCPU(Depart, Arrivee) :-
    trouveCoupCPU(Depart, NBAjoue), !,
    coupValide(NBAjoue, Depart, Arrivee),
    joueLeCoup(NBAjoue).
```

```
trouveCoupCPU(X, 1) :- un(X, A), positionPerdante(A).
trouveCoupCPU(X, 2) :- deux(X, A), positionPerdante(A).
trouveCoupCPU(X, 3) :-
    trois(X, A), positionPerdante(A).
```

Exemple du Jeu de Nim

- Changer de joueur

```
campAdverse (joueur, cpu) .
campAdverse (cpu, joueur) .
```

```
moteur (0, Camp) :- nl, write (Camp),
                    write(' a perdu !!!!! ').
```

```
moteur (Nbalumette, cpu) :-
    affichePosition (Nbalumette),
    joueCPU (Nbalumette, Arrivee),
    moteur (Arrivee, joueur) .
```

```
moteur (Nbalumette, joueur) :-
    affichePosition (Nbalumette),
    joueJoueur (Nbalumette, Arrivee),
    moteur (Arrivee, cpu) .
```

Moteur du Tic-Tac-Toe

- moteur(Grille, ListeCoups, Camp) : prend en paramètre une grille dans laquelle tous les coups de la ListeCoups sont jouables, et c'est à Camp de jouer.
- Conditions d'arrêt :
 - Cas gagnant pour le joueur
 - Cas gagnant pour l'adversaire
 - Cas du match nul
- Conditions de jeu
 - C'est à l'ordinateur de jouer
 - C'est à l'utilisateur de jouer

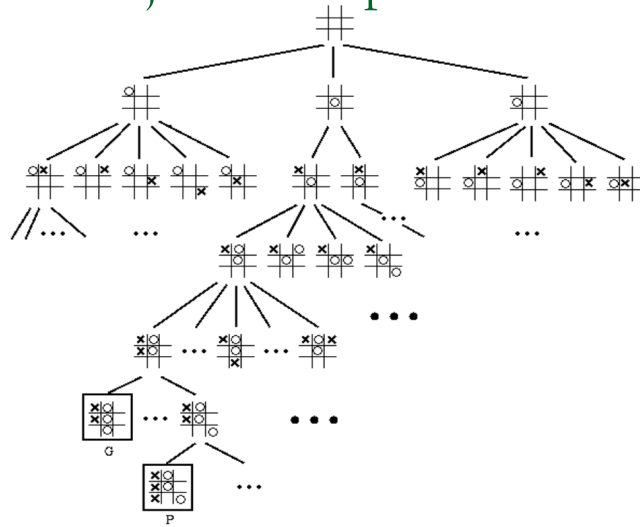
Tic-Tac-Toe

→ TD3 – étape 4 : Moteur de réflexion

Moteur du programme (rajout d'heuristiques dans le choix des coups)

Étape 5

Arbre de jeu : exemple



Définitions

- Arbre de jeu → explosion combinatoire
 - Morpion : arbre de jeu : 362 880 positions
 - Othello : 2^{60} positions
- Arbre de recherche :
 - sous-ensemble de l'arbre de jeu construit dynamiquement;
 - positions terminales ne sont pas forcément celles du jeu
 - Limiter la profondeur de l'arbre de recherche

Exploration dans les arbres de jeux

- Arbre de jeu:
 - **Racine** (prof 0) : position de départ;
 - Nœuds de **profondeur paire** : positions dans lesquelles c'est au joueur 1 de jouer (AMI)
 - Nœuds de **profondeur impaire** : positions dans lesquelles c'est au joueur 2 de jouer (ADV)
 - Les **arcs** issus d'un nœud quelconque représentent les différents coups possibles qui peuvent être joués à partir d'un nœud;
 - Les **feuilles** : positions gagnantes, perdantes ou bloquées.

Fonction d'évaluation

- Cette fonction permet d'évaluer une position non terminale et d'estimer qui peut gagner sans développer de sous-arbres.
 - Soit un nœud n (ie une position), par convention,
 - $f : N \rightarrow] -\infty , + \infty [$
 - $f(n) = -\infty$ si n est perdant pour AMI,
 - $f(n) = +\infty$ si n est gagnant pour AMI,
- Pb : trouver une bonne fonction d'évaluation

MinMax

- MinMax : Maximiser la valeur de la fonction d'évaluation de la situation courante pour AMI et minimiser celle de l'adversaire
- Exemple de fonction d'évaluation pour Tic-Tac-Toe:

$$V = V1 - V2, \text{ où}$$

V1 est la somme :

 - nb d'occurrences de 3X en ligne/col/diag multiplié par 10;
 - nb d'occurrences de 2X en ligne avec la 3ième position vide multiplié par 4;
 - nb d'occurrences d'1X en ligne avec 2 autres positions vides;

V2 la même somme pour les 'O'

MinMax – Tic-Tac-Toe

- Fonction d'évaluation pour une grille
 - Choix de la fonction d'évaluation = nb de lignes encore réalisables(o) – nb de lignes encore réalisables(x)
 - `evalueGrille(Grille, N)` - Évaluer l'état de la grille
 - `compteLignesRéalisables(o, Grille, N1)`
 - `compteLignesRéalisables(x, Grille, N2)`
 - N is $N1 - N2$

MinMax – Tic-Tac-Toe

- Récupérer l'évaluation des coups suivants
 - À partir d'un coup:
 - Il faut jouer les coups suivants
 - Les évaluer, et garder cette valeur dans une liste
 - `parcoursDesCoups` : prédicat récursif qui permet de parcourir tous les coups et de mémoriser leur évaluation (récursivité sur des listes de liste)

Limite du MinMax

- Effet d'horizon : on ne voit pas s'il y a un coup gagnant à la profondeur suivante
- Amélioration : alpha-beta...