Polycopié de cours

Programmation logique

Licence informatique 2e année

Dominique Py

2018-2019

Préambule

Ce document est le support de cours du module « Programmation logique », proposé en deuxième année de licence informatique, qui consiste en une initiation à la logique et à la programmation logique en Prolog. Il ne nécessite pas de prérequis.

Le premier chapitre présente les rudiments de la logique classique (logique des propositions, logique des prédicats) et la méthode de résolution, qui sont les fondements théoriques du langage Prolog. Bien que les définitions et théorèmes de ce chapitre aient été limités au minimum indispensable, cette partie du cours est relativement difficile en raison du vocabulaire nouveau qui y est abordé. Cependant, la méthode de résolution elle-même est simple à comprendre et les exercices d'application doivent permettre au lecteur de se familiariser progressivement avec ces notions.

Le chapitre 2 étudie ensuite les bases du langage Prolog : clauses, unification, stratégie de résolution, ainsi que quelques éléments d'arithmétique et la négation. A l'issue de ce chapitre, le lecteur sera en mesure d'écrire des programmes élémentaires en Prolog.

Les chapitres 3 et 4 abordent les caractéristiques qui donnent toute sa puissance à Prolog : la récursivité, mécanisme utilisé pour effectuer des itérations, et les listes, principales structures de données. Ces deux chapitres permettent d'aborder des programmes Prolog plus complexes et de comprendre les notions essentielles de la programmation logique.

Enfin, les chapitres 5 et 6 viennent compléter ces bases en présentant quelques prédicats prédéfinis parmi les plus fréquemment utilisés, puis en offrant une introduction à l'analyse du langage via les grammaires à clauses définies (DCG), qui constituent l'une des applications les plus connues de Prolog.

Ce cours visant essentiellement une initiation à la programmation logique, les notions plus complexes (programmation par contraintes, parallélisme, modularité) sont laissées de côté. De même, les bibliothèques et les nombreux prédicats prédéfinis ne sont pas tous présentés ici, mais le lecteur curieux pourra compléter son information sur les manuels de référence disponibles en ligne.

Chapitre 1 – Logique classique

1) Introduction

La logique classique considère des *énoncés* qui possèdent une valeur de vérité : vrai ou faux, à l'exclusion de toute autre valeur. Par exemple : « il pleut » et « s'il pleut, je prends mon parapluie » sont des énoncés qui peuvent être vrais ou faux.

Comme tout langage, la logique possède une *syntaxe* et une *sémantique*. La syntaxe est l'ensemble des règles du langage formel dans lequel on exprime les énoncés. La sémantique permet de déterminer la valeur de vérité d'un énoncé donnée selon différentes *interprétations*, également appelées *mondes possibles*. Par exemple, l'énoncé « il pleut » pourra être vrai à un moment et un lieu donnés (un monde possible) et faux dans un autre monde possible. Un énoncé comme « s'il pleut, Watson prend son imperméable » pourra être vrai dans tous les mondes possibles. La logique offre des outils de raisonnement qui permettent d'effectuer des déductions sur des énoncés et de garantir la validité de ces déductions.

La logique comporte différents niveaux, appelés ordres. On distingue :
- la logique d'ordre 0, ou logique des propositions, ou encore calcul des propositions. Elle raisonne sur des variables propositionnelles valant vrai ou faux ;
- la logique d'ordre 1, ou logique des prédicats, ou encore calcul des prédicats. Outre les variables propositionnelles, elle utilise des prédicats et des quantificateurs ;
- les logiques d'ordre supérieur, aussi appelées logiques non classiques, telles que la logique modale qui utilise des modalités (croire, penser), la logique floue (énoncés incertains), …

2) Logique des propositions

La logique des propositions est la logique la plus simple : elle permet d'effectuer des raisonnements sur des énoncés tels que « il pleut » ou « s'il pleut je prends mon parapluie ».

1) Syntaxe

Les énoncés sont exprimés à l'aide de *formules* ou *expressions* dont le vocabulaire est composé de :
- un ensemble fini de variables propositionnelles V = {p, q, r…} qui sont les *propositions atomiques* ;
- les constantes vrai et faux ;
- les connecteurs logiques $\wedge$ (et), $\vee$ (ou), $\rceil$ (non), $\Rightarrow$ (implication), $\Leftrightarrow$ (équivalence) ;
- les parenthèses ( ).

Les *formules bien formées*, ou *expressions bien formées*, sont définies ainsi :
- toutes les propositions atomiques sont des formules bien formées ;
- si A est une formule bien formée, alors $\rceil$A est une formule bien formée ;
- si A et B sont des formules bien formées, alors $A\vee B$, $A\wedge B$, $A\Rightarrow B$ et $A\Leftrightarrow B$ sont des formules bien formées.

Pour simplifier, on les appelle couramment « formules ».

2) Sémantique

La *valeur de vérité* d'une formule se détermine à partir de la valeur de vérité des propositions atomiques qu'elle comporte et des tables de vérité des connecteurs logiques. Comme il existe deux valeurs possibles pour chaque variable propositionnelle, une formule comportant *n* variables possède 2^n *interprétations*, ou *mondes possibles*.

Tables de vérité des connecteurs logiques

A	**B**	**$A \vee B$**	**$A \wedge B$**	**$A \Rightarrow B$**	**$A \Leftrightarrow B$**
vrai	vrai	vrai	vrai	vrai	vrai
vrai	faux	vrai	faux	faux	faux
faux	vrai	vrai	faux	vrai	faux
faux	faux	faux	faux	vrai	vrai

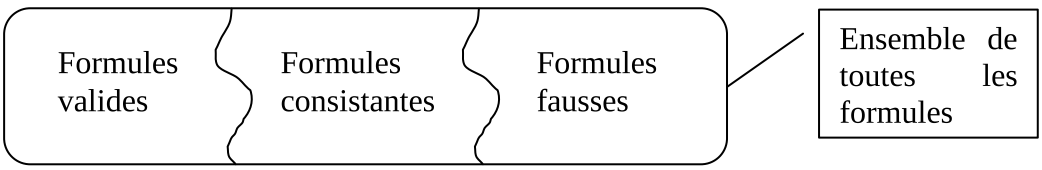
Noter que :
- $A \Rightarrow B$ équivaut à $\neg A \vee B$
- $A \Leftrightarrow B$ équivaut à $(A \Rightarrow B) \wedge (B \Rightarrow A)$ ou encore à $(A \wedge B) \vee (\neg A \wedge \neg B)$

L'implication $A \Rightarrow B$ sert généralement à traduire les énoncés de la forme « si A, alors B ». Mais en logique, $A \Rightarrow B$ équivaut à $\neg A \vee B$: c'est donc vrai lorsque A est faux (cf. table ci-dessus). Un énoncé tel que « si 2+2=5 alors la Terre est plate » serait donc vrai du point de vue de la logique, parce que son hypothèse 2+2=5 est fausse, même si cette affirmation n'a guère de sens en français.

3) Propriétés des formules

En fonction de ses interprétations, une formule logique peut être consistante, inconsistante ou valide.

- Une formule est **consistante** (ou **satisfiable**) s'il existe au moins une interprétation de ses variables qui la rend vraie. Par exemple, $P \Rightarrow (R \wedge S)$ est vrai dans {P=vrai, R=vrai, S=vrai}.
- Une formule est **inconsistante** (ou **insatisfiable**, ou plus simplement fausse) si elle est fausse dans toutes les interprétations possibles. Par exemple, $(P \wedge \neg P)$ est fausse.
- Une formule est **valide** si elle est vraie dans toutes les interprétations possibles. Par exemple, $(P \vee \neg P)$ est valide. On dit que c'est une **tautologie**.



Quelques tautologies courantes

Identité	$A \Rightarrow A$ et $A \Leftrightarrow A$
Tiers exclu	$A \vee \neg A$
Non contradiction	$\neg(A \wedge \neg A)$
Double négation	$A \Leftrightarrow \neg(\neg A)$

Équivalences logiques

Formules équivalentes		Propriété
$A \vee A$	A	Idempotence
$A \wedge A$	A	
$A \wedge B$	$B \wedge A$	Commutativité
$A \vee B$	$B \vee A$	
$A \wedge (B \wedge C)$	$(A \wedge B) \wedge C$	Associativité
$A \vee (B \vee C)$	$(A \vee B) \vee C$	
$(A \wedge B) \vee C$	$(A \vee C) \wedge (B \vee C)$	Distributivité
$(A \vee B) \wedge C$	$(A \wedge C) \vee (B \wedge C)$	
$\rceil(A \wedge B)$	$\rceil A \vee \rceil B$	Lois de De Morgan
$\rceil(A \vee B)$	$\rceil A \wedge \rceil B$	
$A \Rightarrow B$	$\rceil B \Rightarrow \rceil A$	Contraposée

Ces équivalences peuvent être facilement démontrées en construisant leur table de vérité.

Exercice d'application

Démontrer les tautologies suivantes en construisant leur table de vérité :

$A \Rightarrow (B \Rightarrow C)$ équivaut à $(A \Rightarrow B) \Rightarrow (A \Rightarrow C)$

$A \Rightarrow (B \Rightarrow C)$ équivaut à $B \Rightarrow (A \Rightarrow C)$

4) Conséquence logique

En logique, on s'intéresse aux raisonnements qu'il est possible d'effectuer, à partir de connaissances acquises, pour produire de nouvelles connaissances. Que peut-on déduire, de quelle manière, et comment être certain de la validité de la déduction ?
Le principe du raisonnement logique repose sur la relation de conséquence logique entre des énoncés, qui permet de faire des déductions valides à partir d'un ensemble d'hypothèses, ou *prémisses*. Il se schématise ainsi : si l'on sait A et si l'on sait que A implique B, alors on peut déduire B.

$$\frac{\text{Pluie} \qquad \text{Pluie} \Rightarrow \text{Parapluie}}{\text{Parapluie}}$$

Définition
Soit $\{F_1, \dots F_n\}$ un ensemble de formules.
Une formule C est une **conséquence logique** de $\{F_1, \dots F_n\}$, noté $\{F_1, \dots F_n\} \vdash C$, si toute interprétation qui rend vraies les formules $\{F_1, \dots F_n\}$ rend aussi vrai la formule C.

On peut ainsi vérifier que {Pluie, Pluie $\Rightarrow$ Parapluie } $\vdash$ Parapluie en construisant la table de vérité de ces trois formules.

La notion de tautologie est liée à celle de conséquence logique. En effet, une formule est une tautologie si on peut la déduire de l'ensemble vide : $\{\} \vdash A$ (noté aussi $\vdash A$).

Théorème de déduction

$$\{F_i\} \vdash C \text{ si et seulement si } \vdash \{F_i\} \Rightarrow C$$

Autrement dit, C se déduit de l'ensemble de formules $\{F_i\}$ si et seulement si $\{F_i\} \Rightarrow C$ est une tautologie. Ce théorème possède un corollaire, qui est à la base de la méthode de résolution qui sera vue ultérieurement, le théorème de réfutation.

Théorème de réfutation

$$\{F_i\} \vdash C \text{ si et seulement si } \{F_i\} \cup (\rceil C) \text{ est inconsistant}$$

Exemple

On considère le raisonnement suivant : « Si Bill est malade, alors il est à l'infirmerie ou il est rentré chez lui. Or, Bill n'est pas à l'infirmerie et il n'est pas rentré chez lui. Donc, Bill n'est pas malade. »

On choisit la notation suivante :

- Bill est malade	M
- Bill est à l'infirmerie	F
- Bill est rentré chez lui	R
- H_1 : Si Bill est malade, il est à l'infirmerie ou rentré chez lui	$M \Rightarrow (F \vee R)$
- H_2 : Bill n'est pas à l'infirmerie	$\rceil F$
- H_3 : Bill n'est pas rentré chez lui	$\rceil R$
- C : Bill n'est pas malade	$\rceil M$

Peut-on déduire logiquement C de $\{H_1, H_2, H_3\}$? Autrement dit, a-t-on $\{H_1, H_2, H_3\} \vdash C$? Pour le vérifier, on construit la table de vérité de ces formules.

M	**F**	**R**	**H1 :** $M \Rightarrow (F \vee R)$	**H2 :** $\rceil F$	**H3 :** $\rceil R$	**C :** $\rceil M$
vrai	vrai	vrai	vrai	faux	faux	faux
vrai	vrai	faux	vrai	faux	vrai	faux
vrai	faux	vrai	vrai	vrai	faux	faux
vrai	faux	faux	faux	vrai	vrai	faux
faux	vrai	vrai	vrai	faux	faux	vrai
faux	vrai	faux	vrai	faux	vrai	vrai
faux	faux	vrai	vrai	vrai	faux	vrai
faux	faux	faux	**vrai**	**vrai**	**vrai**	**vrai**

On constate que, lorsque H_1, H_2 et H_3 sont vrais, C est également vrai. La formule $\rceil M$ est donc une conséquence logique de $\{M \Rightarrow (F \vee R), \rceil F, \rceil R\}$.

Construire la table de vérité d'une formule permet de vérifier la validité d'une déduction logique. Cependant, cette méthode est très coûteuse puisque, avec *n* variables, la table contient 2^n lignes. Elle n'est réalisable en pratique que pour de petites valeurs de *n*.

5) Principe de résolution

Il est possible de faire des déductions logiques sans passer par la sémantique (comme on l'a fait jusqu'à présent), mais en utilisant des méthodes purement syntaxiques. Pour cela, on applique des règles d'inférences, dont les deux plus connues sont le *modus ponens* et le *modus tollens*.

Modus ponens $$\frac{A \qquad\qquad A \Rightarrow B}{B}$$

Modus tollens $$\frac{\neg B \qquad\qquad \neg B \Rightarrow \neg A}{\neg A}$$

Le *modus ponens* et le *modus tollens* sont des cas particuliers de la règle de résolution :

$$\frac{A \vee B \qquad\qquad \neg A \vee C}{B \vee C}$$

Exemple
On sait que Zoé travaille sur PC ou bien qu'elle travaille sur Mac. On sait également que si Zoé travaille sur Mac, alors elle est en salle 130. Que peut-on en conclure ?

On choisit la notation suivante :
- Zoé travaille sur PC P
- Zoé travaille sur Mac M
- Zoé est en salle 130 S
- H1 : Zoé travaille sur PC ou sur Mac $P \vee M$
- H2 : Si Zoé travaille sur Mac, elle est salle 130 $M \Rightarrow S$ ou $\neg M \vee S$

On applique la règle de résolution aux deux hypothèses et on en déduit que Zoé travaille sur PC ou bien elle est en salle 130 :

$$\frac{P \vee M \qquad\qquad \neg M \vee S}{P \vee S}$$

Sur cet exemple simple, la résolution est immédiate. Le principe de résolution s'applique également à des formules plus complexes, mais celles-ci doivent au préalable être mises sous la forme d'un ensemble de clauses.

Définitions
- ✓ Un *littéral* est une variable propositionnelle (ex : p) ou sa négation (ex : $\neg p$).
- ✓ Une *clause* est une disjonction de littéraux (ex : $p \vee q \vee \neg r$).
- ✓ La clause vide est notée $\perp$
- ✓ Un *ensemble de clauses* est la conjonction de toutes les clauses qu'il contient.
  Par exemple : $(p \vee q \vee \neg r) \wedge (p \vee q \vee \neg r) \wedge (s \vee \neg r)$

Pour faire une démonstration en utilisant le principe de résolution, il faut, au préalable, transformer l'énoncé en un ensemble de clauses équivalent : c'est ce qu'on appelle la mise sous *forme normale conjonctive*. Cette transformation est toujours possible.

Propriété
Toute formule admet une *forme clausale* qui lui est équivalente.

En logique des propositions, la mise sous forme normale conjonctive se déroule en quatre étapes qui utilisent les équivalences logiques vues précédemment.

Mise sous forme normale conjonctive en logique des propositions

1. Remplacer partout $A \Leftrightarrow B$ par $(A \Rightarrow B \wedge B \Rightarrow A)$
2. Remplacer partout $A \Rightarrow B$ par $(\neg A \vee B)$
3. Faire rentrer les négations dans les parenthèses jusqu'à les appliquer aux littéraux
    a. remplacer $\neg\neg A$ par A
    b. remplacer $\neg(A \wedge B)$ par $\neg A \vee \neg B$
    c. remplacer $\neg(A \vee B)$ par $\neg A \wedge \neg B$
4. Distribuer le $\vee$ sur le $\wedge$ pour obtenir des clauses
    a. remplacer $A \vee (B \wedge C)$ par $((A \vee B) \wedge (A \vee C))$

L'expression obtenue est une forme clausale (un ensemble de clauses). Cet ensemble de clauses peut ensuite être simplifié :

– Les clauses comportant deux littéraux opposés valent vrai et peuvent être supprimées
  ex : $(p \vee q \vee \neg p)$ peut être supprimé
– Les répétitions du même littéral dans une clause peuvent être supprimées
  ex : $(p \vee q \vee \neg p)$ équivaut à $(p \vee q)$
– Si une clause C est incluse dans une clause C', la clause C' peut être supprimée car la valeur de l'ensemble ne dépend que de C
  ex : $C = (p \vee \neg r)$ et $C' = (p \vee q \vee \neg r)$

Exercice d'application

Dans le désert, un voyageur arrive devant deux pistes. Chacune est gardée par un sphinx. Chaque piste peut soit conduire à une oasis, soit se perdre dans le désert.
- Le sphinx de droite dit : « Au moins une des deux pistes conduit à une oasis ».
- Le sphinx de gauche dit : « La piste de droite va dans le désert ».
- Les sphinx disent tous les deux la vérité, ou mentent tous les deux.

On choisit la notation suivante :

D	La piste de droite conduit à une oasis
G	La piste de gauche conduit à une oasis
$A = D \vee G$	Le sphinx de droite dit « Au moins une des deux pistes conduit à une oasis »
$B = \neg D$	Le sphinx de gauche dit « La piste de droite va dans le désert »
$A \Leftrightarrow B$	Les sphinx disent tous les deux la vérité, ou mentent tous les deux

Mettre $A \Leftrightarrow B$ sous forme normale conjonctive et en déduire quelle piste le voyageur doit prendre pour trouver une oasis

La règle de résolution se généralise à deux clauses de taille quelconque. Elle permet de calculer leur *résolvante*.

$$\textit{clauses} \longrightarrow \frac{A \vee x1 \vee \ldots xn \qquad \neg A \vee y1 \vee \ldots\ yn}{x1 \vee \ldots xn \vee y1 \vee \ldots\ yn} \longleftarrow \textit{résolvante}$$

Si les clauses sont réduites à un littéral, la résolvante est la clause vide, notée $\perp$

$$\frac{A \qquad\qquad \neg A}{\perp}$$

6) Méthode de résolution

La méthode de résolution, proposée par A. Robinson en 1965, permet de démonter une conclusion à partir d'un ensemble d'hypothèses en s'appuyant sur le théorème de réfutation :

$\{F_i\} \vdash C$ ssi $\{F_i\}$ U $\{\neg C\}$ est inconsistant

L'énoncé est transformé en un ensemble de clauses, puis on lui ajoute la négation de la conclusion. On cherche alors à mettre en évidence une contradiction dans l'ensemble obtenu, en appliquant la règle de résolution autant de fois que nécessaire.

Résolution par réfutation

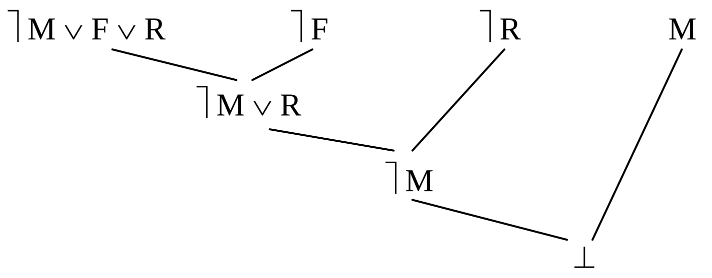
1. Mettre les hypothèses sous forme d'un ensemble de clauses F
2. Ajouter la négation de la conclusion $\neg C$
3. Répéter
   a. Choisir deux nouvelles clauses dont on peut calculer la résolvante
   b. Ajouter la résolvante à l'ensemble
4. Jusqu'à ce que
   a. Soit on obtient la résolvante vide : F implique C
   b. Soit on ne peut plus calculer de nouvelle résolvante: C n'est pas une conséquence logique de F

Exemple

On reprend l'exemple de Bill : H = $\{M \Rightarrow (F \vee R), \neg F, \neg R\}$, C = $\neg M$

H est mis sous forme clausale H' = $\{\neg M \vee F \vee R, \neg F, \neg R\}$

On ajoute à H' la négation de la conclusion : $\{\neg M \vee F \vee R, \neg F, \neg R, M\}$



La *clôture de la résolution* d'un ensemble de clauses E est l'ensemble de toutes les clauses qu'on peut déduire de E. Cette clôture est un ensemble fini, puisque le nombre de littéraux est fini. Par conséquence, l'algorithme termine toujours.

7) Chaînage avant et chaînage arrière

La méthode de résolution transforme toutes les formules en clauses, c'est-à-dire en disjonction de littéraux. Or, une connaissance donnée sous la forme A => B contient une connaissance implicite de contrôle. En la traduisant par $\neg A \vee B$, on perd cette information. Conserver la forme initiale permet d'orienter la résolution pour la rendre plus efficace: le raisonnement est guidé par les connaissances. Le chaînage avant et le chaînage arrière sont deux modes d'inférence, basés sur la règle de résolution, qui exploitent ce principe. Ils utilisent des formules particulières, appelées *clauses de Horn*.

- Une **clause de Horn** est une clause possédant *au plus* un littéral positif
- Une **clause de Horn stricte** est une clause possédant *exactement* un littéral positif et au moins un littéral négatif. Ces clauses peuvent s'écrire (prémisses $\Rightarrow$ but)

  ex: $\neg p$ V $\neg q$ V r équivaut à (p/\q) => r

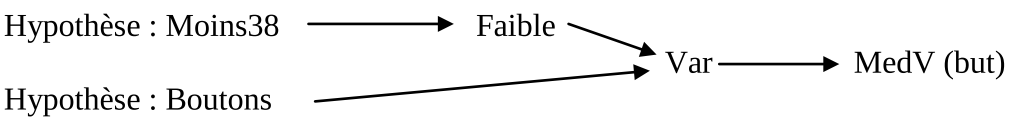
L'intérêt des clauses de Horn est pratique : beaucoup d'énoncés sont présentés sous la forme d'implications logiques. Cependant, les clauses de Horn sont une restriction de la logique et tous les énoncés ne peuvent pas être mis sous cette forme (par exemple, $A \vee B$ ne peut pas être mis sous la forme d'une clause de Horn). Les chaînages avant et arrière ne permettent donc pas de faire toutes les déductions qui sont possibles avec la méthode de résolution.

Exemple

Une personne qui a une forte fièvre, de la toux et des boutons a la rougeole. Une personne qui a une faible fièvre et des boutons a la varicelle. Une fièvre inférieure à 38° est faible, une fièvre supérieure à 38° est forte. Pour soigner la rougeole on donne le médicament R, pour soigner la varicelle le médicament V. Bill a 37,5° de fièvre et des boutons.

	Règle	**Clause de Horn équivalente**
1.	Forte $\wedge$ Toux $\wedge$ Boutons $\Rightarrow$ Rou	$\neg$Forte $\vee$ $\neg$Toux $\vee$ $\neg$Boutons $\vee$ Rou
2.	Faible $\wedge$ Boutons $\Rightarrow$ Var	$\neg$Faible $\vee$ $\neg$Boutons $\vee$ Var
3.	Moins38 $\Rightarrow$ Faible	$\neg$Moins38 $\vee$ Faible
4.	Plus38 $\Rightarrow$ Forte	$\neg$Plus38 $\vee$ Forte
5.	Rou $\Rightarrow$ MedR	$\neg$Rou $\vee$ MedR
6.	Var $\Rightarrow$ MedV	$\neg$Var $\vee$ MedV
7.	Moins38	
8.	Boutons	

Le chaînage avant consiste à appliquer les règles aux faits connus, à l'aide du *modus ponens*, pour en déduire de nouveaux faits. On s'arrête soit quand on a démontré le but, soit quand on ne peut plus déduire aucun fait nouveau (saturation). Lorsque la base de faits est finie, l'algorithme termine obligatoirement par un succès ou un échec.



Le chaînage arrière consiste à remplacer le but par des sous-buts, à l'aide du modus tollens, jusqu'à se ramener aux faits initiaux. On s'arrête soit quand on a éliminé tous les sous-buts (succès), soit quand on échoue à démontrer un sous-but. De la même manière que pour le chainage avant, l'algorithme termine toujours.

$$\begin{tikzcd}
 & & \text{Faible} \arrow[r] & \text{Moins38 (hypothèse)} \\
\text{But : MedV} \arrow[r] & \text{Var} \arrow[ur] \arrow[dr] & & \\
 & & \text{Boutons (hypothèse)} &
\end{tikzcd}$$

L'intérêt d'étudier le raisonnement en chaînage arrière tient au fait que ce mode d'inférence est à la base du langage Prolog. En Prolog :

- on exprime les règles sous forme de clauses de Horn strictes (ex: $p \wedge q \wedge r \Rightarrow s$)
- on exprime les faits sous forme de littéraux positifs (ex: p, q, r)
- on donne le but à établir (ex: s?) et Prolog essaie de le démontrer en appliquant les règles

2) Logique des prédicats

La logique des propositions a un pouvoir d'expression relativement limité. En particulier, elle ne permet pas d'exprimer aisément des énoncés de la forme « Tous les X sont Y ». Par exemple, pour exprimer la propriété « tous les chats sont gris », il faudrait autant d'énoncés qu'il existe de chats. La logique des prédicats est plus expressive: elle permet de représenter des objets ainsi que des fonctions ou des relations sur ces objets.

1) Syntaxe

Les énoncés sont formés à partir de :

- un ensemble fini de constantes {a, b, c…} ;
- un ensemble fini de variables {X, Y, Z…} ;
- un ensemble fini de fonctions {f, g, h…} dont chacune possède une *arité* (nombre d'arguments) strictement positive ;
- un ensemble fini de prédicats {P, Q, R…} dont chacun possède une *arité* positive ;
- les connecteurs logiques ($\wedge$, $\vee$, …) et les parenthèses ;
- les quantificateurs $\forall$ (quel que soit) et $\exists$ (il existe).

Un *terme* est une expression logique qui désigne un objet.

- les constantes et les variables sont des termes
  ex : bill, zoé, félix, X, Y
- si *f* est une fonction d'arité *n* et (t_1, t_2, … t_n) des termes, alors $f(t_1, t_2, \dots t_n)$ est un terme
  ex : adresse(zoé), couleur(félix), f(X,Y), adresse(père(bill))

Une *formule atomique* est définie par:

- les prédicats d'arité 0 (les propositions) sont des formules atomiques
  ex : Toux, Forte, Faible
- si *P* est un prédicat d'arité *n* et (t_1, t_2, … t_n) des termes, alors $P(t_1, t_2, \dots t_n)$ est une formule atomique
  ex : EstGris(félix), Amis(bill, zoé)

Les *formules* sont définies par :
– toute formule atomique est une formule ;
– si A est un formule, alors ¬A est une formule ;
– si A et B sont des formule, alors A \/ B, A /\ B, A => B, A <=> B sont des formules ;
– si A est une formule et X une variable, alors (∃X) A et (∀X) A sont des formules.

Un quantificateur s'applique à une variable
– le quantificateur *universel* ∀ sert à exprimer le fait que tous les objets sur lesquels s'applique une formule vérifient cette formule
  ex : « tous les chats sont gris » se traduit par (∀X) Chat(X) => Gris(X)
– le quantificateur *existentiel* ∃ sert à exprimer le fait qu'il existe au moins un objet vérifiant une formule donnée
  ex : « il existe une panthère noire » se traduit par (∃X) Panthère(X) /\ Noire(X)

Quelques expressions classiques et leur traduction logique :

Tous les A sont B: (∀X) A(X) => B(X)
« tous les chats sont gris » devient (∀X) Chat(X) => Gris(X)

Seuls les A sont B: (∀X) B(X) => A(X)
« seuls les majeurs peuvent voter » devient (∀X) DroitVote(X) => Majeur(X)

Aucun A n'est B: (∀X) A(X) => ¬B(X)
« aucun arbre ne marche » devient (∀X) Arbre(X) => ¬Marche(X)

Quelques A sont B: (∃X) A(X) /\ B(X)
« il existe une panthère noire » devient (∃X) Panthère(X) /\ Noire(X)

Équivalences logiques

Formules équivalentes		**Propriété**
(∃X) A(X)	(∃Y) A(Y)	Variables muettes
(∀X) A(X)	(∀Y) A(Y)	
¬(∃X) A(X)	(∀X) ¬A(X)	Négation des quantificateurs
¬(∀X) A(X)	(∃X) ¬A(X)	
(∀X) A(X)	¬(∃X) ¬A(X)	
(∃X) A(X)	¬(∀X) ¬A(X)	

2) Déductions logiques

La méthode de résolution, utilisée jusqu'ici en logique des propositions, s'applique aussi en logique des prédicats. Cela nécessite de définir, dans le cadre de la logique des prédicats, d'une part la méthode de mise sous forme conjonctive, d'autre part la méthode de calcul d'une résolvante avec des variables. Cette dernière fait appel à la méthode *d'unification.*

La méthode de mise sous forme normale conjonctive est très proche de celle employée ne logique des propositions. Elle comporte ici huit étapes : les étapes 1, 2, 3 et 7 correspondent aux étapes 1, 2, 3 et 4 de la méthode applicable en logique des propositions, les étapes 4, 5, 6 et 8 sont spécifiques aux prédicats et permettent d'éliminer les quantificateurs et de renommer les variables.

Mise sous forme normale conjonctive en logique des prédicats

1. Remplacer partout $A \Leftrightarrow B$ par $(A \Rightarrow B \wedge B \Rightarrow A)$
2. Remplacer partout $A \Rightarrow B$ par $(\neg A \vee B)$
3. Faire rentrer les négations dans les parenthèses jusqu'à les appliquer aux littéraux
    a. remplacer $\neg\neg A$ par A
    b. remplacer $\neg(A \wedge B)$ par $\neg A \vee \neg B$
    c. remplacer $\neg(A \vee B)$ par $\neg A \wedge \neg B$
    d. remplacer $\neg(\exists X)\ A(X)$ par $(\forall X)\ \neg A(X)$
    e. remplacer $\neg(\forall X)\ A(X)$ par $(\exists X)\ \neg A(X)$
4. Déplacer tous les quantificateurs à gauche de la formule sans changer leur ordre relatif
5. Éliminer les quantificateurs existentiels : les $(\exists X)$ sont supprimés et toute occurrence de X est remplacée soit par une constante, soit par une fonction f comportant autant d'arguments qu'il y a de quantificateurs universels à gauche de $(\exists X)$
6. Éliminer les quantificateurs universels
7. Distribuer le $\vee$ sur le $\wedge$ pour obtenir des clauses
    a. remplacer $A \vee (B \wedge C)$ par $((A \vee B) \wedge (A \vee C))$
8. Renommer les variables (en appliquant la loi des variables muettes) de telle sorte que deux clauses distinctes comportent des variables distinctes

Exemple

Soit la formule : $(\exists X)\ ((P(X) \vee Q(X,a))) \Rightarrow ((\forall Y)(\exists Z)\ R(X,Y,Z))$

Étapes 1 et 2 : remplacer $(A \Rightarrow B)$ par $(\neg A \vee B)$

$(\exists X)\ \neg((P(X) \vee Q(X,a))) \vee ((\forall Y)(\exists Z)\ R(X,Y,Z))$

Étape 3 : faire entrer les négations

$(\exists X)\ (\neg P(X) \wedge \neg Q(X,a)) \vee ((\forall Y)(\exists Z)\ R(X,Y,Z))$

Etape 4 : déplacer les quantificateurs à gauche

$(\exists X)\ (\forall Y)(\exists Z)\ (\neg P(X) \wedge \neg Q(X,a)) \vee R(X,Y,Z)$

Etape 5 : éliminer les quantificateurs existentiels

$(\forall Y)(\neg P(cst1) \wedge \neg Q(cst1,a)) \vee R(cst1,Y, fct1(Y))$

Etape 6 : éliminer les quantificateurs universels

$(\neg P(cst1) \wedge \neg Q(cst1,a)) \vee R(cst1,Y,fct1(Y))$

Etape 7 : distribuer le $\vee$ sur le $\wedge$

$(\neg P(cst1) \vee R(cst1,Y,fct1(Y)) \wedge (\neg Q(cst1,a) \vee R(cst1,Y,fct1(Y)))$

Etape 8 : Renommer les variables

$(\neg P(cst1) \vee R(cst1,Y,fct1(Y1)) \wedge (\neg Q(cst1,a) \vee R(cst1,Y,fct1(Y2)))$

On obtient un ensemble de deux clauses:
{ $\neg$P(cst1) $\vee$ R(cst1,Y,fct1(Y1) , $\neg$Q(cst1,a) $\vee$ R(cst1,Y,fct1(Y2) }
En logique des prédicats, le calcul d'une résolvante devient :

$$\frac{A(\ldots) \vee B(\ldots) \qquad \neg A(\ldots) \vee C(\ldots)}{B(\ldots) \vee C(\ldots)}$$

Deux questions se posent alors : sous quelles conditions peut-on calculer la résolvante ? S'il est possible de la calculer, que vaut cette résolvante ? La réponse passe par l'utilisation d'un algorithme d'unification de termes.

L'*unification* consiste à rendre identiques deux formules, sous certaines conditions. Pour unifier deux formules, il faut trouver une *substitution*, c'est-à-dire une manière de remplacer certaines variables par des termes, qui rendra les deux formules identiques. Autrement dit, on a A $\neq$ B mais subst(A) = subst(B).

Définition
Une substitution est un ensemble de couples (terme/variable) où chaque couple *t/v* signifie que toute occurrence de la variable *v* sera remplacée par le terme *t* dans l'expression à transformer. On note E_s l'expression E à laquelle on a appliqué la substitution s

Exemples

Expression E	Substitution S	Expression substituée E_s
P(X)	{Z/X}	P(Z)
P(X)	{zoe/X}	P(zoe)
P(X)	{couleur(Z)/X}	P(couleur(Z))
P(X)	{couleur(mer)/X}	P(couleur(mer))
R(X,X)	{zoe/X}	R(zoe,zoe)
P(X) $\vee$ T(X,Y)	{zoe/X}	P(zoe) $\vee$ T(zoe,Y)
P(X) $\wedge$ Q(Y)	{zoe/X, tom/Y}	P(zoe) $\wedge$ Q(tom)
P(X) $\wedge$ Q(Y)	{tom/X, tom/Y}	P(tom) $\wedge$ Q(tom)

Définition
Deux formules A et B sont unifiables si et seulement si il existe une substitution *s* telle que A_s=B_s. On dit que *s* est un *unifieur* de A et B.

Remarque: si A et B sont unifiables, ils possèdent (parmi tous leurs unifieurs) un unifieur le plus général (noté p.g.u.), au nom des variables près. C'est l'unifieur *s* tel que, pour tout autre unifieur *r*, il existe une substitution *r'* telle que $A_r = (A_s)_{r'}$ et $B_r = (B_s)_{r'}$.

Par exemple, avec A = P(X, f(Y), Z)) et B = P((X, f(a), Z)), le p.g.u. est *s* = {a/Y}. L'unifieur *r* = {a/Y, X/Z} n'est pas le plus général car l'unifieur *r'* = {X/Z} combiné à *s* équivaut à *r*.

Intuitivement, le p.g.u. est l'unifieur qui comporte le moins de substitutions. Par la suite, on considère uniquement les unifieurs les plus généraux.

Exemples d'unifications

E1	**E2**	**plus général unifieur s**	**$E1_s = E2_s$**
P(X)	P(Y)	{Y/X} ou (X/Y}	P(X) ou P(Y)
P(X)	P(tom)	{tom/X}	P(tom)
R(X,tom)	R(zoe,Y)	{zoe/X, tom/Y}	R(zoe,tom)
R(X,X)	R(tom,Y)	{tom/X, tom/Y}	R(tom,tom)
P(X) ∨ R(X,tom)	P(zoe) ∨ R(Y,Z)	{zoe/X, zoe/Y, tom/Z}	P(zoe) ∨ R(zoe,tom)
P(couleur(X))	P(Y)	{couleur(X)/Y}	P(couleur(X))
P(couleur(mer))	P(Y)	{couleur(mer)/Y}	P(couleur(mer))
P(couleur(mer))	P(couleur(X))	{mer/X}	P(couleur(mer))

Contre-exemples (expressions pour lesquelles il n'existe aucun unifieur)

E1	**E2**	**échec**
P(tom)	P(zoe)	pas de variables
P(X)	T(X)	P ≠ T
P(X,X)	P(X)	arités différentes
P(X) ∨ Q(X)	P(X) ∧ Q(X)	connecteurs différents
P(couleur(X))	P(taille(X))	fonctions différentes
R(X,X)	R(tom,zoe)	substitution inconsistante
T(X,Y,tom)	T(Y,zoe,X)	substitution inconsistante

En logique des prédicats, le calcul d'une résolvante devient donc :

$$\frac{A \vee B \qquad \neg A' \vee C}{(B \vee C)_s} \qquad \text{avec } A_s = A'_s$$

On peut calculer la résolvante de deux clauses de la forme $(A \vee B)$ et $(\neg A' \vee C)$ s'il est possible d'unifier A et A' par une substitution. Cette résolvante est la disjonction des deux clauses, à laquelle on applique cette substitution.

Exemple

On cherche à démontrer le raisonnement « Tous les hommes sont mortels, or Socrate est un homme, donc Socrate est mortel ». On le représente à l'aide de la constante « Socrate » et de deux prédicats :

homme(X): vrai si X est un homme
mortel(X): vrai si X est mortel

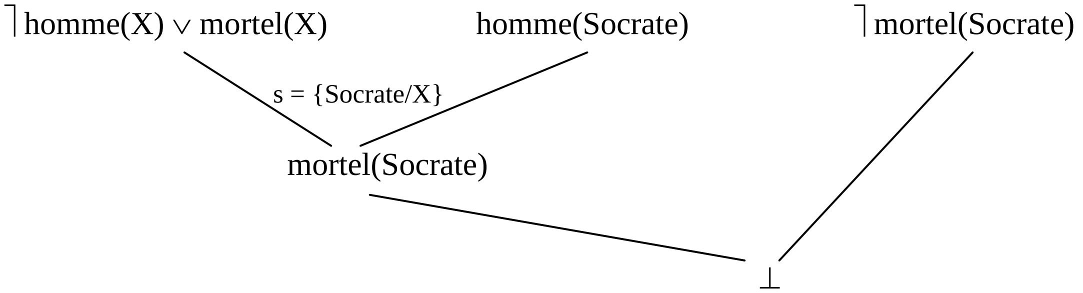
On a les formules :

$(\forall X)$ homme(X) ⇒ mortel(X)	première hypothèse
homme(Socrate)	seconde hypothèse
mortel(Socrate)	but

Après mise sous forme normale conjonctive et ajout de la négation du but, on obtient l'ensemble de clauses :

$\{ \neg \text{homme}(X) \vee \text{mortel}(X), \text{homme}(Socrate), \neg \text{mortel}(Socrate) \}$

Résolution



Il existe une contradiction entre l'ensemble des hypothèses et la négation de la conclusion, autrement dit, $\{H_i \cup \neg C\}$ est inconsistant. L'énoncé « Socrate est mortel » est donc une conséquence logique des deux hypothèses.

Exercice d'application

Formuler cet énoncé en logique du premier ordre et démontrer la conclusion: « La nuit, tous les chats sont gris. Félix est un chat. Il fait nuit. Donc, Félix est gris. »

Chapitre 2 – Le langage Prolog

1) Introduction

Prolog, pour PROgrammation LOGique, est un langage qui a été conçu par Alain Colmerauer et Philippe Roussel à Marseille en 1972. Il est basé sur la logique du premier ordre : un programme Prolog est un ensemble de clauses de Horn et son exécution s'appuie sur le principe de résolution de Robinson.

C'est un langage déclaratif : le programmeur exprime des connaissances en décrivant des objets et leurs relations, puis effectue des requêtes, mais la stratégie de résolution est laissée à Prolog. La programmation en Prolog est donc très différente de la programmation dans un langage impératif (C, Java) ou fonctionnel (Lisp, Scheme, CamL). C'est aussi un langage de haut niveau : il est concis, permet du prototypage rapide. En revanche, il est moins efficace que les langages impératifs (mais il est possible de l'interfacer avec un langage impératif). De manière générale, c'est un langage adapté à l'intelligence artificielle, aux bases de données relationnelles et à l'analyse du langage naturel.

La plupart des implémentations de Prolog reposent sur un interpréteur, plutôt que sur un compilateur : le programme n'est compilé que partiellement au moment du chargement, puis il est analysé dynamiquement par l'interpréteur et les expressions sont évaluées au fur et à mesure de l'exécution. Cela rend les programmes plus lents, mais aussi plus facilement portables. Il existe pour Prolog de nombreuses implémentations, souvent gratuites, telles que SWI Prolog (http://www.swi-prolog.org/) ou GNU-Prolog (http://www.gprolog.org/) qui seront utilisées en TP.

2) Clauses

Un programme Prolog est constitué d'un ensemble de clauses de Horn, c'est-à-dire de clauses comportant au plus un littéral positif. Prolog utilise trois sortes de clauses.

- Les faits : un littéral positif.
- Les règles : un ensemble de prémisses qui implique une conclusion. L'ensemble des prémisses est appelé le *corps* de la règle, la conclusion est la *tête* de la règle.
- Les questions : un littéral négatif, ou une conjonction de littéraux négatifs.

Syntaxe

Depuis 1995, la norme ISO-Prolog a permis de standardiser la syntaxe utilisée par les différentes implémentations de Prolog.

- Une clause est terminée par un point.
- La tête et le corps d'une règle sont reliés par « :- »
- Les éléments du corps sont séparés par des virgules.

```
dimanche.
soleil.
balade :- dimanche, soleil.
```

Ce programme est composé de trois clauses :
- deux faits
- une règle

A partir de ce programme, l'utilisateur peut effectuer des requêtes dans l'interpréteur.

```
?- dimanche.
yes
?- soleil
yes
?- balade.
yes
?- cinema.
no
```

Les clauses de Horn ne permettent pas d'exprimer le « ou » dans les prémisses. Une règle de la forme P1 $\vee$ P2 $\Rightarrow$ C se traduira en Prolog à l'aide de deux clauses : P1 $\Rightarrow$ C et P2 $\Rightarrow$ C.

```
dimanche.
soleil.
promenade :- weekend, soleil.
weekend :- samedi.
weekend :- dimanche.
```

On part en promenade si
c'est le week-end ET il y a du soleil

C'est le week-end si c'est samedi
OU si c'est dimanche

```
?- weekend.
yes
?- promenade.
yes
```

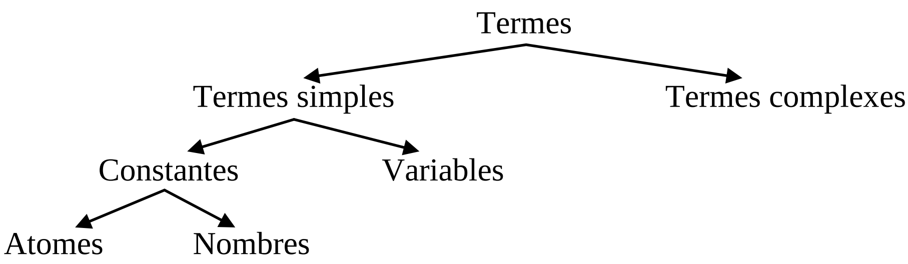
3) Les objets en Prolog

Dans les exemples précédents, les prédicats n'ont pas d'arguments : ce sont des variables propositionnelles). En Prolog, les prédicats peuvent aussi avoir des arguments, qui sont des termes (constantes, variables ou termes complexes).

Définitions

- ✓ Un **atome** est une séquence de lettres et de chiffres commençant par une *minuscule*
  ex : x, titi, k7video, jean_paul, anneLise
- ✓ Un **nombre** est un entier ou un réel (les plus couramment utilisés)
  ex : 52, -17, 486, 321.521, -58.2
- ✓ Les atomes et les nombres forment l'ensemble des **constantes**
- ✓ Une **variable** est une séquence de lettres et de chiffres commençant par une *capitale* ou par un *souligné* (variable anonyme). Attention : les variables système sont toujours de la forme _nombre
  ex : X, Titi, K7video, Jean_Paul, AnneLise, _truc, _854721
- ✓ Les variables et les constantes forment l'ensemble des **termes simples**
- ✓ Un **terme complexe** est construit à partir d'un foncteur et de ses arguments, qui sont eux-mêmes des termes
  ex : carre(11), pere(X), age(titi), enfant(adam,eve), pere(pere(pere(Y))
- ✓ L'**arité** d'un prédicat ou d'un foncteur est le *nombre d'arguments* qu'il possède. On note *nom_prédicat*/*valeur_arité* un prédicat ou un foncteur d'arité donnée
  ex : carre/1, enfant/2

La hiérarchie des objets en Prolog se représente par l'arborescence suivante.



Attention : en Prolog, deux prédicats ou deux foncteurs ayant le même nom mais des arités différentes sont considérés comme des objets différents. Par exemple, truc(X) et truc(X,Y) sont deux objets distincts.

4) L'unification

C'est l'unification, associée au principe de résolution, qui permet à Prolog d'effectuer des déductions complexes. Deux prédicats P et Q sont unifiables s'ils sont identiques, ou bien s'il existe une substitution s telle que $(P)_s = (Q)_s$.

Exemples et contre-exemples

P	**Q**	**s**	**échec**
femme(X)	femme(zoe)	{zoe/X}	-
canard(X)	canard(Y)	{X/Y} ou {Y/X}	-
joue(zoe,X)	joue(Y,piano)	{zoe/Y, piano/X}	-
homme(X)	homme(pere(zoe))	{pere(zoe)/X}	-
femme(zoe)	femme(nina)	-	zoe ≠ nina
homme(X)	homme(X,Y)	-	Arités différentes
joue(zoe,X)	joue(mozart,Y)	-	zoe ≠ mozart
joue(zoe, piano)	joue(X,X)	-	s inconsistante
homme(tom)	homme(pere(zoe))	-	tom ≠ père(zoe)

5) Stratégie de résolution

La stratégie de résolution de Prolog consiste à chercher à démontrer le but de toutes les manières possibles. Ainsi, pour démontrer un but P, Prolog essaie successivement de résoudre toutes les clauses dont la tête peut s'unifier avec P. Par conséquence, en Prolog pur, l'ordre d'écriture des clauses n'a pas d'importance: les réponses données par Prolog seront les mêmes quel que soit l'ordre des clauses pour un prédicat donné.

```
canard(donald).             /* Donald est un canard                 */
canard(daisy).              /* Daisy est un canard                  */
canari(titi).               /* Titi est un canari                   */
oiseau(X) :- canard(X).     /* X est un oiseau si X est un canard   */
oiseau(X) :- canari(X).     /*               ou si X est un canari  */
```

```
?- canard(daisy).
yes
?- canard(mickey).
no
?- oiseau(donald).      /* Donald est un oiseau car Donald est un canard */
yes
?- oiseau(titi).        /* Titi est un oiseau car Titi est un canari */
yes
?- oiseau(minnie).      /* Minnie n'est pas un oiseau car        */
no                      /* elle n'est ni un canard, ni un canari */

?- oiseau(X).           /* Quels sont tous les oiseaux ?*/
X = donald
X = daisy
X = titi
```

Les prédicats prédéfinis **trace** et **notrace** permettent respectivement d'activer et de désactiver la trace. Ils sont très utiles en phase de débogage.

```
| ?- trace.
The debugger will first creep -- showing everything (trace)
| ?- oiseau(X).
      1    1  Call: oiseau(_36) ?
      2    2  Call: canard(_36) ?
      2    2  Exit: canard(donald) ?
      1    1  Exit: oiseau(donald) ?
X = donald
      1    1  Redo: oiseau(donald) ?
      2    2  Redo: canari(titi) ?
      2    2  Exit: canari(titi) ?
      1    1  Exit: oiseau(titi) ?
X = titi
(16 ms) yes
| ?- notrace.
The debugger is switched off
```

En Prolog, les prédicats sont généralement **réversibles**: tout paramètre est indifféremment un paramètre d'entrée ou de sortie, selon le contexte.

```
joue(mozart, piano).    /* Mozart joue du piano */
joue(tim, guitare).     /* Tim joue de la guitare */
joue(zoe, guitare).     /* Zoé joue de la guitare */
joue(zoe, piano).       /* Zoé joue du piano */
joue(nina, basse).      /* Nina joue de la basse */
chante(tim).            /* Tim chante */
```

```
?- joue(zoe,piano).     /* Est-ce que zoé joue du piano? */
yes

?- joue(mozart,basse).  /* Est-ce que mozart joue de la basse? */
no

?- joue(X,piano).       /* Qui joue du piano? */
X = mozart
X = zoe
```

```
?- joue(zoe,X).        /* De quels instruments joue zoé? */
X = guitare
X = piano

?- joue(X,Y).          /* Qui joue de quoi? */
X=mozart Y=piano
X=tim Y=guitare
X=zoe Y=guitare
X=zoe Y=piano
X=nina Y=basse
```

Exercice d'application

Pour la base « musique » ci-dessus, définir les prédicats suivants :
- choriste/1: un choriste sait chanter et jouer d'un instrument
- polyvalent/1: un joueur polyvalent sait jouer de deux instruments différents
- duo/2: un duo est formé de deux personnes qui jouent du même instrument
- groupe/3: un groupe est composé d'un chanteur, d'un guitariste et d'un bassiste. Le chanteur peut être aussi le guitariste ou le bassiste, mais il faut deux instrumentistes différents.

6) Arithmétique en Prolog

Pour faire des calculs arithmétiques en Prolog, on utilise l'opérateur prédéfini is **:**

variable is *expression*

évalue l'expression et l'affecte à la variable libre**,**

constante is *expression*

évalue l'expression et **réussit si le résultat est égal à la constante, échoue sinon.**

```
?- X is 3+2.
X=5
?- X is 5*2+4.
X=14
?- Y is 5, X is 2*Y.
Y=5
X=10
?- 5 is 3+2
yes
```

Attention, le prédicat is *n'est pas* réversible : le membre gauche ne peut pas être une expression et le membre droit ne doit pas contenir de variable libre.

```
?- 3+2 is X.      /* Ces trois expressions provoquent une erreur */
?- 3+Y is X.
?- X is Y+2.
```

Exemple

```
/* La somme de X et Y vaut Z */
somme(X,Y,Z):- Z is X+Y.
```

```
?- somme(2,3,5).
yes
?- somme(2,2,5).
no
?- somme(2,3,X).
X = 5
?- somme(2,X,5).
uncaught exception: error(instantiation_error,(is)/2)
```

Les comparateurs de nombres sont < (inférieur), > (supérieur), =< (inférieur ou égal), >= (supérieur ou égal). Attention à la syntaxe : dans le comparateur « inférieur ou égal », le signe égal *précède* le signe supérieur tandis que dans le comparateur « supérieur ou égal », le signe égal *suit* le signe supérieur.

```
age(tim, 22).
age(zoe, 23).
age(nina, 18).
plusjeune(X,Y):- age(X,A), age(Y,B), A<B.
```

```
?- plusjeune(tim,zoe).  /* Tim est-il plus jeune que Zoé ? */
yes

?- plusjeune(zoe,tim).  /* Zoé est-elle plus jeune que Tim ? */
no

?- plusjeune(nina,X).   /* Qui est plus âgé que Nina ? */
X=tim
X=zoe
```

Exercice d'application

Définir les prédicats suivants
- sup/2: vrai si X est strictement supérieur à Y
- inf/2: vrai si X est inférieur ou égal à Y
- diff/3: Z vaut la différence entre X et Y
- abs/2: Y vaut la valeur absolue de X
- diffbis/3: réécrire *diff* en utilisant abs/2

L'opérateur *div* calcule le quotient de la division entière d'un entier par un autre et l'opérateur *mod* calcule le reste de la division entière d'un entier par un autre.

```
?- X is 9 div 2
X = 4

?- X is 42 mod 10
X = 2
```

7) Les foncteurs

En Prolog, les foncteurs sont utilisés pour construire des termes complexes. Par exemple, avec le foncteur age/1 et la constante zoe, on construit le terme complexe age(zoe). Cette notion diffère des fonctions en programmation impérative, car le terme complexe n'est jamais évalué. Par exemple, il n'est pas possible d'évaluer carre(4) pour rendre 16.

Les foncteurs permettent de regrouper des constantes ou des termes en un seul terme complexe: ils sont plutôt l'équivalent des *structures* en langage C. Quelques exemples de termes complexes: max(age(pere(zoe),carre(8)), fraction(5,8), heure(12,30,5).

Exemple
On veut manipuler des pièces du jeu d'échec. Chaque pièce est caractérisée par son nom (reine, roi, fou, tour, cavalier, pion) et sa couleur (noir, blanc). On définit un foncteur pièce/2 dont le premier argument représente le nom de la pièce et le second sa couleur :

piece(reine,blanc)
piece(cavalier,noir)
piece(fou,noir)
…

```
/* Liste des noms des pièces */
est_nom(reine).
est_nom(roi)                  .
est_nom(fou).
est_nom(tour)
est_nom(cavalier)
est_nom(pion).

/* Liste des couleurs */
est_couleur(blanc).
est_couleur(noir).

/* Enumération de toutes les pièces du jeu */
est_piece(piece(N,C)):- est_nom(N), est_couleur(C).
```

```
?- est_piece(P).              /* Quelles sont les pièces? */
P = piece(reine, blanc)
P = piece(reine, noir)
P = piece(roi, blanc)
…
P = piece(pion,noir)
```

Si l'on veut représenter des pièces sur un échiquier, en utilisant la notation usuelle aux échecs, il suffit d'ajouter deux arguments, l'un pour la colonne (une lettre dans [a..h]) et l'autre pour la ligne (un chiffre dans [1..8]) :

piece(reine, noir, d,8)
piece(roi, noir, e, 8)
…

Exercice d'application

On considère un jeu de 32 cartes. Chaque carte a une hauteur et une couleur.
- Les hauteurs sont, par ordre croissant: sept, huit, neuf, dix, valet, dame, roi, as.
- Les couleurs sont, par ordre croissant: carreau, cœur, trèfle, pique.
On définit un foncteur carte/2 dont le premier argument est la hauteur et le second la couleur

Écrire un prédicat est_carte/1 qui énumère les cartes du jeu

8) La négation

Les clauses de Horn possèdent au plus un littéral positif. Une règle Prolog ne contient donc que des prémisses positives :

P :- P1, P2, P3 correspond à $\neg P1 \vee \neg P2 \vee \neg P3 \vee P$

Or, certaines déductions reposent sur des prémisses négatives. Par exemple, « si c'est l'été et qu'il ne pleut pas, le voisin arrose son jardin » se traduit en logique par :

été $\wedge \neg$pluie $\Rightarrow$ arrose

Traduire cette règle en Prolog nécessiterait d'employer la négation pour un fait :

P :- P1 $\wedge \neg$P2

En Prolog, l'opérateur **\=** permet d'exprimer $X \neq Y$, qui est une forme de négation. Pour généraliser la négation à un prédicat quelconque, on utilise l'opérateur **\+** qui correspond au « non » logique.

Exemple

```
joue(tim, guitare).
chante(callas).
chante(tim).

/* Une cantatrice chante mais ne joue d'aucun instrument*/
cantatrice(X) :- chante(X), \+ joue(X,Y).
```

```
?- cantatrice(tim).
no
? - cantatrice(X).
X = callas.
```

La négation fonctionne de la manière suivante : quand l'interpréteur rencontre \+ P, il essaie de résoudre P : s'il réussit, il renvoie faux ; s'il échoue, il renvoie vrai.

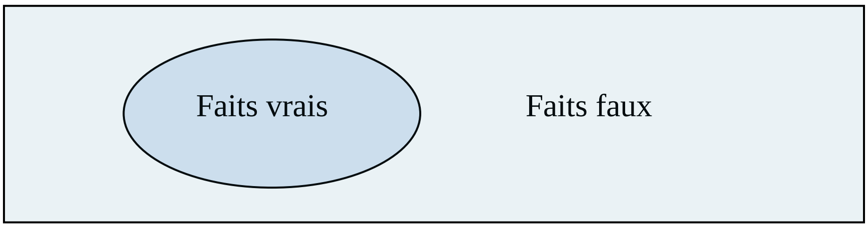
Attention: la négation en Prolog n'a pas le même sens que la négation logique :

- en logique, $\neg$P est vrai si et seulement si P est faux ;
- en Prolog, \+ P est vrai si et seulement si P *n'est pas démontrable*.

Si P n'est pas démontrable (et ⌉P non plus) :

- en logique, on *ne peut pas prouver* que P est faux ;
- la négation de Prolog considère que P est faux.

En effet, Prolog repose sur **l'hypothèse du monde clos**: tout ce qui n'est pas explicitement déclaré est considéré comme faux.



Exercice d'application

1) On prend la base de faits « vide-greniers » : des personnes vendent et achètent des objets. Une personne qui vend un objet fixe son prix de vente. Une personne qui cherche à acheter un objet fixe le prix maximum qu'elle est prête à payer. On représente ces informations à l'aide des prédicats vend(*nom*, *objet*, *prix*) et achete (*nom*, *objet*, *budget_max*).

```
vend(paul, guitare, 50).        achete(emma, livre, 5).
vend(loic, livre, 10).          achete(loic,dvd,20).
…                               …
```

- Écrire un prédicat invendu(X,Y) qui vaut vrai si X vend Y mais personne ne veut acheter Y (indépendamment du prix fixé).

- Écrire un prédicat echoue(X,Y) qui vaut vrai si X veut acheter Y mais personne n'a de Y à vendre.

2) On suppose disponibles les prédicats :

max(A, B, C, D) D est le maximum de A, B et C
min(A, B, C, D) D est le minimum de A, B et C

- En utilisant la négation, écrire un prédicat mediane(A, B, C, D) tel que D est la médiane de A, B et C.

Chapitre 3 – La récursivité

Puisque les structures itératives (*while*, *for*) n'existent pas en Prolog, la répétition d'un même traitement s'exprime obligatoirement par la récursivité.

1) Un relation récursive simple

Dans la base « famille », on veut exprimer la relation X-est-ancêtre-de-Y, définie de la manière suivante :
- si X est le parent de Y, il est son ancêtre,
- si X est le parent d'un ancêtre de Y, il est l'ancêtre de Y.

```
parent(adam, mona).
parent(mona, homer).
parent(homer, bart).
parent(homer, lisa).
parent(homer, maggie).
ancetre(X,Y) :- parent(X,Y).
ancetre(X,Y) :- parent(X,Z), ancetre(Z,Y).
```

`?- ancetre(mona, Y).`		*De qui Mona est-elle l'ancêtre ?*
`Y = homer`	*fils*	(= *qui sont ses descendants?*)
`Y = bart`	*petit-fils*	
`Y = lisa`	*petite-fille*	
`Y = maggie`	*petite-fille*	
`?- ancetre(X, lisa).`		*Qui sont les ancêtres de Lisa ?*
`X = homer`	*père*	
`X = mona`	*grand-mère*	
`X = adam`	*arrière-grand-père*	

Le prédicat ancêtre calcule la *fermeture transitive* de la relation ancêtre. C'est un cas simple de récursivité.

Dans l'écriture d'un prédicat récursif, on trouve au moins une clause non récursive, qui traduit le cas d'arrêt de la récursivité. Dans une clause récursive, le choix de l'ordre des prédicats dans le corps de la règle et le choix des paramètres doivent être faits très soigneusement : un mauvais choix peut facilement provoquer une boucle infinie.

Premier exemple : l'appel récursif placé en premier provoque une boucle infinie
ancetre(X,Y) :- parent(X,Y).
ancetre(X,Y) :- **ancetre(X,Z)**, parent(Z,Y).

Second exemple : le passage des paramètres à l'identique provoque une boucle infinie
ancetre(X,Y) :- parent(X,Y).
ancetre(**X,Y**) :- parent(X,Z), ancetre(**X,Y**).

Exercice d’application

On reprend l’exemple du jeu de 32 cartes. Les hauteurs sont, par ordre croissant: sept, huit, neuf, dix, valet, dame, roi, as. Les couleurs sont, par ordre croissant: carreau, cœur, trèfle, pique.

- Écrire le prédicat precede/2 qui définit la relation entre deux hauteurs consécutives ainsi qu’entre deux couleurs consécutives
- Écrire le prédicat inf/2 qui vaut vrai si la première valeur est inférieure à la seconde (fermeture transitive du précédent)
- Écrire le prédicat inferieure/2 qui vaut vrai si la première carte est inférieure à la seconde. Une carte X est inférieure à une carte Y si sa valeur est inférieure ou bien si sa valeur est identique et sa couleur inférieure

2) Récursivité terminale et non terminale

Comme en programmation impérative, la plupart des traitements récursifs en Prolog peuvent s’écrire de manière récursive *terminale* (l’appel récursif est le dernier traitement effectué par la clause récursive) ou récursive *non terminale* (des traitements sont effectués après le retour de l’appel récursif). En général, la version récursive non terminale est plus intuitive car plus proche de la définition, tandis que la version récursive terminale possède un argument de plus que la version non terminale. Attention : certains traitements sur les listes en Prolog ne peuvent être effectués qu’en utilisant la récursivité terminale.

Exemple
Écrire un prédicat somme(X,S) tel que S est la somme des entiers de 1 à X.

Définition récursive informelle (non terminale):

somme(0) = 0
somme(X) = somme(X-1) + X si X > 0

```
/* Version récursive non terminale */
somme(0,0).
somme(X, R1):- X>0, X1 is X-1, somme(X1,R), R1 is R+X.
```

Pour écrire une version récursive terminale, on ajoute un paramètre A qui joue le rôle d’accumulateur à chaque appel récursif et qui vaut 0 à l’appel.

Définition récursive informelle (terminale)

somme(X, A) = somme(X-1, A+X) si X > 0
somme(0, A) = A

```
/* Version récursive terminale */
somme(0, A, A).
somme(X, A, R):- X>0, X1 is X-1, A1 is A+X, somme(X1, A1, R).
```

Attention: ces prédicats *ne sont pas* réversibles car ils utilisent le prédicat **is** qui n’est pas réversible.

```
?- somme(5,S).
S = 15

?- somme(5,15).
yes

?- somme(N,15).
uncaught exception: error(instantiation_error,(is)/2)
```

Exercices d'application

Écrire un prédicat mult(A, B, M) tel que M est le produit de A par B, en ne faisant que des additions et des soustractions :
- de manière récursive non terminale,
- de manière récursive terminale (mult/4).

En utilisant les opérateurs div et mod, écrire un prédicat sommechiffres(X,S) tel que S est la somme des chiffres de X.

Variante : écrire un prédicat sommechiffres(X, A, S) qui effectue le même calcul de manière récursive terminale.

3) La coupure

Le contrôle en Prolog repose sur le *non-déterminisme* et le *backtrack* (retour-arrière): l'interpréteur essaie de résoudre le but de toutes les manières possibles, en cas d'échec sur une clause il revient en arrière pour essayer la clause suivante.

La coupure (notée « ! ») est un opérateur de contrôle qui supprime, dans un paquet de clauses, les points de choix en suspens. C'est un prédicat prédéfini qui réussit toujours.

Exemple

```
/* Sans coupure */             /* Avec coupure */
p(X):- q(X), r(X).             p(X):- q(X), !, r(X).
p(X):- s(X).                   p(X):- s(X).
q(a).                          q(a).
q(b).                          q(b).
r(a).                          r(a).
r(b).                          r(b).
s(c).                          s(c).
```

```
?- p(X).                       ?- p(X).
X = a                          X = a
X = b
X = c
```

La coupure permet d'éliminer des points de choix menant à des échecs et de réduire le nombre de tests lorsque les conditions sont exclusives. Cependant, lorsqu'on utilise la coupure, l'ordre des clauses devient significatif : il n'est plus possible de permuter deux clauses (dans l'exemple ci-dessus, si on permute les deux clauses définissant p(X), le résultat sera différent). De plus, la coupure peut faire perdre des solutions. C'est donc un opérateur à utiliser avec prudence.

Exemple

Le prédicat max(A,B,C) est tel que C vaut le plus grand de A et de B.

```
/* Définition de max/3 sans coupure */
max(A,A,A).             /* si A=B   */
max(A,B,A):- A > B.     /* si A>B   */
max(A,B,B):- B < A.     /* si B>A   */
```

L'utilisation de la coupure permet de n'écrire qu'une seule condition

```
/* Définition de max3 avec coupure */
max(A,B,A):- A > B, !.  /* si A>B   */
max(A,B,B).             /* sinon    */
```

Exercice d'application

En utilisant la coupure, réécrire le prédicat pgcd(A,B,C), tel que C est le pgcd de A et de B.

La coupure n'est pas spécifique à la récursivité, comme le montrent les exemples précédents. Mais elle est souvent utilisée dans le cas d'arrêt de la récursivité, pour éviter des tests dans la clause récursive.

```
/* Calcul de la somme des entiers de 1 à N */

/* Sans coupure */
somme(0,0).
somme(X, R1):- X>0, X1 is X-1, somme(X1,R), R1 is R+X.

/* Avec coupure */
somme(0,0):- !.
somme(X, R1):- X1 is X-1, somme(X1,R), R1 is R+X.
```

Chapitre 4 – Les listes

En Prolog, la liste est la seule structure de données disponible. Les parcours de listes sont donc un aspect essentiel des programmes Prolog, qu'il est indispensable de bien maîtriser. Ces parcours s'effectuent de manière récursive.

1) Définition

Une liste est une séquence finie d'éléments quelconques. Les éléments sont entourés de crochets et séparés par des virgules. Par exemple :

[zoe, tom]
[5, licence, X, -24.50, pere(bill)]
[daisy, [X,Y,Z], [picsou]]

La liste vide est notée []. Toute liste non vide est composée d'une **tête** et d'une **queue** : la tête est le premier élément, la queue est la liste (éventuellement vide) composée de tous les autres éléments.

Exemples

– [zoe, tim, tom]	tête = zoe	queue = [tim, tom]
– [5]	tête = 5	queue = []
– [[], [riri, fifi, loulou]]	tête = []	queue = [[riri, fifi, loulou]]

L'opérateur | permet de décomposer une liste en une tête et une queue. Ainsi, l'expression [X|Y] désigne une liste dont la tête est X et la queue est Y.

Si on unifie [riri,fifi,loulou] et [A|B]	A = riri,	B = [fifi,loulou]
Si on unifie [daisy] et [A|B]	A = daisy,	B = []

Mais

[] et [A|B] ne s'unifient pas
toto et [A|B] ne s'unifient pas
[5,6] et [A] ne s'unifient pas

Exercice d'application

Quel est le résultat de l'unification des couples de termes suivants ?

[5,6,12,4]	et	[A,B,C]
[[5]]	et	[A|B]
[[5]]	et	[[A|B]]
[5,6,8]	et	[A|B]
[X,5|R]	et	[8,Y|Z]
[X,5|R]	et	[8|Z]
[4,2]	et	[X,Y|Z]

2) Parcours récursif de listes

Certains traitements sur liste sont utilisés très fréquemment : test d'appartenance d'un élément à une liste, dénombrement des éléments d'une liste, concaténation de deux listes, inversion de l'ordre des éléments d'une liste…

1) Membres d'une liste

Le prédicat membre(E, L) vaut vrai si l'élément E appartient à la liste L.

```
/* E appartient à la liste soit s'il est la tête de la liste */
/* soit s'il appartient à la queue de la liste */
membre(E, [E|L]).
membre(E,[X|L]) :- membre(E,L).
```

```
?- membre(tim,[zoe, tim,bill]).
yes
?- membre(E,[zoe, tim,bill]).
E = zoe
E = tim
E = bill
```

Attention: si on utilise la coupure ou la négation, le prédicat n'est plus réversible !

```
membre(E, [E|L]):- !.              /* Version NON réversible */
membre(E,[X|L]) :- membre(E,L).

membre(E, [E|L]).                  /* Version NON réversible */
membre(E,[X|L]) :- E\=X, membre(E,L).
```

```
?- membre(E,[zoe, tim,bill]).
E = zoe
```

2) Taille d'une liste

Le prédicat taille(L,N) vaut vrai si N est le nombre d'éléments de la liste L.

```
/* N vaut la taille de la liste L*/
taille([], 0).
taille([E|L], N) :- taille(L,M), N is M+1.
```

```
?- taille([zoe,tim,bill],3).
yes
?- taille([zoe,tim,bill],N).
N = 3
?- taille([],N).
N = 0
/* Le premier paramètre doit être instancié */
?- taille(L,3).
L=[_,_,_]
… boucle infinie…
```

3) Concaténation de deux listes

Concaténer deux listes consiste à mettre leurs éléments dans une troisième liste, sans changer leur ordre. Par exemple, la concaténation de L1=[1,2] et de L2=[3,4,5] vaut L3=[1,2,3,4,5]. Cette opération est plus délicate qu'il n'y paraît à première vue, du fait de la représentation de listes en Prolog, qui se décomposent en une tête et une queue.

En effet, si on écrit simplement :

conc(L1,L2,[L1|L2]).

alors L1 devient la tête de la liste dont L2 est la queue :

conc([1,2],[3,4,5],[[1,2],3,4,5]).

L'écriture avec une virgule à la place de la barre verticale ne réussit pas davantage :

conc(L1,L2,[L1,L2]).

forme la liste des listes L1 et L2 :

conc([1,2],[3,4,5],[[1,2],[3,4,5]]).

La bonne méthode consiste à prendre un par un les éléments de L1, en commençant par le dernier, pour les placer en tête de L2 :
- si L1 est vide, le résultat vaut L2,
- sinon, on concatène la queue de L1 avec L2 pour obtenir L3 et le résultat vaut la liste formée de la tête X de L1 et de L3.

```
/* L3 est la concaténation de L1 et L2*/
conc([],L2,L2).
conc([X|L1],L2,[X|L3]) :- conc(L1,L2,L3).
```

```
?- conc([2,4,6],[1,3,5],L).
L = [2,4,6,1,3,5]

/* conc permet de calculer le préfixe ou le suffixe */

?- conc(L1,[1,3,5], [2,4,6,1,3,5]).
L1 = [2,4,6]
?- conc([2,4,6],L2, [2,4,6,1,3,5]).
L2 = [1,3,5]
```

4) Inversion de l'ordre des éléments d'une liste

Renverser une liste consiste à produire la liste constituée de ses éléments rangés dans l'ordre inverse. Par exemple, renverser [1,2,3,4] produit la liste [4,3,2,1]. Contrairement à la plupart des traitements sur les listes, qui peuvent s'effectuer aisément de manière récursive non terminale, l'inversion d'une liste nécessite le recours à un paramètre supplémentaire et se fait classiquement de manière récursive terminale.

En effet, si on écrit :

renverser([],[]).
renverser ([X|L1], [L2|X]):- renverser (L1,L2).

le résultat est une liste de listes :

?- renverser([1,2,3],N).
N = [[[[]|3]|2]|1]

La bonne méthode consiste à introduire un troisième paramètre, qui vaut initialement la liste vide et qui sert d'accumulateur. A chaque étape, on ajoute la tête de la liste à renverser dans cet accumulateur. Lorsque la liste à renverser est vide, le résultat vaut l'accumulateur.

```
renverser([],L,L).
renverser([X|L1],L2,L3):- renverser(L1,[X|L2],L3).
```

```
?- renverser([1,2,3,4],[],L).
L = [4,3,2,1]
?- renverser([1,2,3,4],[],[4,3,2,1]).
yes
```

Pour simplifier l'appel de renverser, on peut introduire un prédicat renverser/2 qui se borne à appeler renverser/3 avec la liste vide en deuxième paramètre.

```
renverser(L1,L2):- renverser(L1,[],L2).
renverser([],L,L).
renverser([X|L1],L2,L3):- renverser(L1,[X|L2],L3).
```

```
?- renverser([1,2,3,4],L).
L = [4,3,2,1]
?- renverser([1,2,3,4],[4,3,2,1]).
yes
```

3) Application : le problème des deux cruches

Énoncé du problème

On dispose de deux cruches d'une contenance respective de 5 litres et 8 litres (notées C5 et C8), initialement vides, ainsi que d'une source d'eau infinie. On peut remplir entièrement une cruche, la vider entièrement, ou transvaser le contenu d'une cruche dans une autre, en s'arrêtant avant qu'elle ne déborde, le cas échéant.
On veut mesurer exactement quatre litres d'eau, mais les cruches ne sont pas graduées! Quelle séquence de manipulations faut-il effectuer pour obtenir exactement quatre litres dans C8?

On représente un état du problème à l'aide du foncteur etat/2 à deux arguments enteirs : etat(C5, C8). L'état initial est représenté par etat(0,0), l'état final par etat(0,4). Deux faits Prolog décrivent ces deux états :

initial(etat(0,0)).
final(etat(4,0)).

On représente les actions à l'aide du prédicat transition/3 :
- le premier argument est l'état courant,
- le deuxième argument est le nouvel état,
- le troisième argument est le nom de l'action,
- le corps de la clause exprime les conditions à remplir pour pouvoir exécuter l'action.

transition(E_courant,E_nouveau,Action) :- conditions.

La résolution est récursive:

- s’il existe une transition qui fait passer de l’état courant à l’état final, arrêt avec succès,
- sinon, on applique une transition qui fait passer de l’état courant à un nouvel état, puis on essaie de résoudre le problème à partir de ce nouvel état.

Le prédicat resoudre/2 prend en argument un état et renvoie la liste des actions qui font passer de cet état à l’état final. Enfin, le prédicat cruche/1 lance la résolution.

```
initial(etat(0,0)).
final(etat(4,0)).

cruche(S):- initial(EI), resoudre(EI,S).

resoudre(EF,[]):- final(EF), !.
resoudre(EC,[A|S]):- transition(EC,EN,A), resoudre(EN,S).

/* Remplir une cruche */
transition(etat(C5,C8), etat(5,C8), remplir(5)):- C5\=5.
transition(etat(C5,C8), etat(C5,8), remplir(8)):- C8\=8.

/* Vider une cruche */
transition(etat(C5,C8), etat(0,C8), vider(5)):- C5\=0.
transition(etat(C5,C8), etat(C5,0), vider(8)):- C8\=0.

/* Verser C5 dans C8 */
transition(etat(C5,C8),etat(0,Total),verser(5,8)):-
      C5\=0, C8\=8, Total is C5+C8, Total=<8.
transition(etat(C5,C8),etat(Reste,8),verser(5,8)):-
      C5\=0, C8\=8, Total is C5+C8, Total>8, Reste is Total-8.

/* Verser C8 dans C5 */
transition(etat(C5,C8),etat(Total,0),verser(8,5)):-
      C5\=5, C8\=0, Total is C5+C8, Total=<5.
transition(etat(C5,C8),etat(5,Reste),verser(8,5)):-
      C5\=5, C8\=0, Total is C5+C8, Total>5, Reste is Total-5.
```

A l’exécution, ce programme part dans une boucle infinie. En effet, il existe des actions réciproques (remplir et vider la même cruche) qui peuvent être répétées sans fin. Pour éviter ces boucles, il faut interdire qu’un même état apparaisse deux fois dans une solution. On mémorise les états rencontrés dans une liste et on vérifie, à chaque transition, que le nouvel état n’appartient pas à la liste

```
/* On initialise la liste avec l’état initial */
cruche(S):- initial(EI), resoudre(EI, [EI] ,S).

/* Après avoir calculé le nouvel état
      - on vérifie qu’il est absent de la liste
      - on l’ajoute à la liste pour l’appel récursif */
resoudre(EF,L,[]):- final(EF), !.
resoudre(EC,L,[A|S]):-
      transition(EC,EN,A),absent(EN,L),resoudre(EN,[EN|L],S).
```

On obtient ainsi plusieurs solutions, dont la plus courte comporte douze actions :

S = [remplir(5), verser(5,8), remplir(5), verser(5,8), vider(8), verser(5,8), remplir(5), verser(5,8), remplir(5), verser(5,8), vider(8), verser(5,8)]

Exercices d’application

- Écrire un prédicat memeTaille/2 qui vaut vrai si les deux listes passées en argument ont même longueur.

- Écrire un prédicat absent/2 qui vaut vrai si l’élément X est absent de la liste L (sans utiliser membre/2).

- Écrire un prédicat sous_ens/2 qui vaut vrai si la liste L1 est un sous-ensemble de la liste L2.

- Écrire un prédicat disjointes/2 qui vaut vrai si la liste L1 n’a aucun élément commun avec la liste L2.

- Écrire un prédicat prefixe/2 qui vaut vrai si la liste L1 est un préfixe de la liste L2 (par exemple, prefixe([1,2],[1,2,3,4]) vaut vrai).

- Écrire un prédicat classer/2 tel que le premier argument est une liste d’entiers et le second une liste de foncteurs pair/1 et impair/1 appliqués à chaque entier selon sa parité

    classer([5,2,4],L) unifie L avec [impair(5),pair(2),pair(4)]

- Écrire un prédicat elever/2 tel que le premier argument est une liste de foncteurs carre/1 et cube/1 appliqués à des entiers et le second la liste des nombre élevés, selon le cas, au carré ou au cube

    elever([carre(5),cube(2),cube(4)],L) unifie L avec [25,8,64]

- A l’aide du foncteur paire/2, écrire un prédicat couples/3 tel que les deux premiers arguments sont des listes de même longueur et le troisième est la liste des couples de valeurs de même rang

    couples([a,b,c],[1,2,3],L) unifie L avec [paire(a,1),paire(b,2),paire(c,3)]

Chapitre 5 – Prédicats prédéfinis

Aujourd'hui, les implémentations de Prolog proposent un grand nombre de prédicats prédéfinis, qui sont détaillés dans le manuel de référence. Ce chapitre n'en présente qu'une petite partie, parmi les plus souvent utilisés.

1) Affichage

La plupart des programmes simples ne nécessitent pas d'affichage, mais est parfois utile d'écrire des messages ou des termes. Trois prédicats prédéfinis peuvent être utilisés :
- nl/0 affiche un retour à la ligne,
- write/1 affiche le terme passé en paramètre,
- format/2 affiche la chaîne passée en premier paramètre, le second étant une liste vide.

```
afficher(X):- format(″L'argument est ″,[]), write(X), nl.
```

```
?- afficher(5).
L'argument est 5

?- afficher(tom).
L'argument est tom
```

2) Vérification du type d'un argument

Les prédicats qui testent les types rendent vrai si l'argument est du type correspondant :
- var/1 vaut vrai si l'argument est une variable libre,
- nonvar/1 vaut vrai si l'argument n'est pas une variable libre,
- integer/1 vaut vrai si l'argument est un entier,
- float/1 vaut vrai si l'argument est un réel,
- string/1 vaut vrai si l'argument est une chaîne de caractères,
- atom/1 vaut vrai si l'argument est un atome,
- compound/1 vaut vrai si l'argument est un terme composé
- ground/1 vaut vrai si l'argument est un terme qui ne contient aucune variable libre.

3) Arithmétique

Ces opérations arithmétiques prédéfinies s'utilisent avec **is** :
- gcd/2 calcule le pgcd des deux arguments entiers,
- abs/1 calcule la valeur absolue de l'entier,
- sign/1 calcule le signe (1 ou -1) de l'entier,
- max/2 calcule le maximum de deux entiers,
- min/2 calcule le minimum de deux entiers,
- sqrt/1 calcule la racine carrée de l'argument
- random/1 renvoie un entier aléatoire entre 0 et l'argument.

4) Traitement des listes

Certains des prédicats écrits aux chapitres précédents existent aussi en version prédéfinie : c’est le cas de member/2 qui correspond à membre/2 et de length/2 qui correspond à taille/2. Il existe également :
- sort/2 qui trie la liste passée en premier argument, en supprimant les doublons, et l’unifie avec le second argument,
- msort/2 qui fait la même chose en conservant les doublons

5) Ensemble de solutions

Le mécanisme de backtrack en Prolog ne permet d’énumérer les valeurs satisfaisant une propriété donnée, mais pas de produire leur liste. Pour cela, on utilise :
- findall/3 qui unifie le troisième argument avec la liste des éléments X qui vérifient la propriété passée en deuxième argument. Le premier argument est l’élément X.
- bagof/3 qui fait la même chose mais énumère tous n-uplets solutions lorsque la propriété contient plusieurs variables libres.

```
joue(mozart, piano).
joue(zoe, guitare).
joue(zoe, basse).
joue(tim, guitare).
joue(nina, basse).
chante(tim).
chante(callas).
```

```
?- findall(X,chante(X),L).
L = [tim,callas]

?- bagof(X,joue(X,Y),L).
L = [zoe,nina]
Y = basse
L = [zoe,tim]
Y = guitare
L = [mozart]
Y = piano
```

6) Modification de la base de faits

Il est possible d’ajouter ou de supprimer dynamiquement des clauses au programme :
- asserta/1 ajoute, en tête du paquet de clauses, la clause passée en argument,
- assertz/1 idem en fin du paquet de clauses,
- retract/1 supprime la clause passée en argument.

```
?- chien(X).
no
?- asserta(chien(milou)), assertz(chien(belle)).
yes
?- chien(X).
X = milou
X = belle
?- retract(chien(milou)).
yes
?- chien(X).
X = belle
```

Chapitre 6 – Grammaires à clauses définies

Les grammaires à clauses définies, ou DCG (pour *Definite Clause Grammars*) sont un formalisme qui permet d'exprimer aisément en Prolog des grammaires formelles, afin d'analyser ou de générer des phrases dans un langage donné. Les DCG sont étroitement liées à Prolog : en effet, l'objectif initial de l'inventeur de Prolog était de pouvoir analyser le langage naturel.
Les DCG permettent :
- d'analyser une phrase et de vérifier qu'elle satisfait la syntaxe du langage,
- de générer tout ou partie des phrases d'un langage donné.

Exemple

On définit un langage dans lequel une phrase est composée d'un sujet, d'un verbe et d'un adverbe. Le vocabulaire très simple comprend deux noms, deux verbes et deux adverbes.

```
phrase --> nompropre, verbe, adverbe.
nompropre --> [zoe].
nompropre --> [tom].
verbe --> [marche].
verbe --> [ecrit].
adverbe --> [vite].
adverbe --> [lentement].
```

```
?- phrase([zoe,marche,lentement],[]).
yes
?- phrase([zoe,ecrit,parfois],[]).
no
?- phrase([S,ecrit,vite],[]).
S = zoe
S = tom
?- phrase(P,[]).
P = [zoe,marche,vite]
P = [zoe,marche,lentement]
…
P = [tom,ecrit,lentement]
```

Au chargement du programme, l'interpréteur Prolog traduit les règles de la grammaire dans la syntaxe usuelle.

Syntaxe DCG	**Syntaxe Prolog**
phrase --> nompropre, verbe.	phrase(A,B) :- nompropre (A,C), verbe(C,B).
nompropre --> [zoe].	nompropre ([zoe|A],[A]).
verbe --> [marche].	verbe ([marche|A],[A]).

Le formalisme des DCG est donc une simple couche de syntaxe qui facilite l'écriture d'un analyseur en Prolog, en évitant au programmeur de préciser tous les arguments.

La grammaire ci-dessus ne permet pas de vérifier les accords en genre et en nombre. Par exemple, si le vocabulaire contient le verbe « marchent », la phrase « zoé marchent » sera acceptée alors qu’elle est grammaticalement incorrecte en français.

Pour vérifier ces accords, on ajoute des paramètres au vocabulaire et aux règles de grammaire afin d’expliciter les règles d’accord en genre (G) et en nombre (N).

```
phrase --> sujet(G,N), verbe(N).
sujet(G,N) --> article(G,N), nom(G,N).
article(fem,sing) --> [la].
article(masc,sing) --> [le].
article(_,plur) --> [les].
nom(masc,sing) --> [chat].
nom(fem,sing) --> [fille].
nom(_,plur) --> [oiseaux].
verbe(sing) --> [marche].
verbe(sing) --> [dort].
verbe(plur) --> [volent].
```

```
?- phrase([la,fille,marche],[]).
yes
?- phrase([les,chat,dort],[]).
no
?- phrase([le,chat,V],[]).
V = marche
V = dort
?- phrase(P,[]).
P = [la,fille,marche]
P = [la,fille,dort]
P = [le,chat,marche]
P = [le,chat,dort]
P = [les,oiseaux,volent]
```

De la même manière, on peut ajouter à la grammaire des arguments représentants les *traits sémantiques* des différents éléments : par exemple, on impose que le verbe dormir ne peut prendre comme sujet qu’un animal ou un humain.

Les mots du lexique peuvent être regroupés dans un fait *lexique* dont le premier argument est le mot et les suivants ses traits syntaxiques et sémantiques. Les éléments atomiques de la grammaire se définissent alors selon une syntaxe différente.

```
article(G,N) --> [Mot], {lexique(Mot, article, G, N)}.
nom(G,N) --> [Mot], {lexique(Mot, nom, G, N)}.

lexique(le, article, masculin, singulier).
lexique(la, article, feminin, singulier).
lexique(les, article, _, pluriel).
lexique(chat, nom, masculin, singulier).
lexique(table, nom, feminin, singulier).
```

Table des matières