Programmation Logique :

une introduction à partir des clauses de Horn

OBJECTIFS LIES AU CHAPITRE 4 – PROGRAMMATION LOGIQUE

Savoirs : notions théoriques

4.1.1. Lien Prolog / LP1 : clauses de Horn
4.1.2. Syntaxe du langage Prolog
4.1.3. Sémantique opérationnelle du langage Prolog : arbres de résolution
4.1.4. Récursivité et arrêt

Savoirs faire : pratiques

4.2.1. Passer d'un problème représenté en LP1 à un programme Prolog
4.2.2. Savoir programme un problème simple en Prolog et utiliser ce programme
4.2.3. Respecter des règles de base de génie logiciel (documentation et lisibilité du code)
4.2.4. Test et débuggage d'un programme Prolog
4.2.5. Concevoir des programmes récursifs en Prolog

1. INTRODUCTION : PROLOG ET PROGRAMMATION LOGIQUE

Le langage Prolog est le principal langage de support à la programmation logique. Tout en n'étant quasiment plus utilisé dans le monde industriel, la programmation logique garde son importance, particulièrement en recherche, dans des domaines tels que l'Intelligence Artificielle et le Traitement Automatique des Langues Naturelles (TALN). Dans ce paradigme de programmation, on ne dit pas à la machine ce qu'elle doit faire instruction par instruction comme dans un langage procédural. Au contraire, on se contente de modéliser logiquement les connaissances liées à un problème : on parle de programmation déclarative. C'est un algorithme de déduction automatique, issu de la méthode de résolution, qui permet de déduire de nouvelles connaissances en réponse à une question.

Un exemple suffira à montrer la différence entre ces paradigmes de programmation. Supposons que l'on désire implémenter un programme qui calcule la factorielle d'un entier. En programmation procédurale (encore encore programmation impérative), ce programme suivra le pseudo-algorithme suivant :

```
Lire(n) ;
Factorielle := 1 ;
POUR i = 1 A n FAIRE
    Factorielle := Factorielle * i ;
Retourner(Factorielle)
```

Ici, on spécifie exactement à la machine toutes les opérations qu'elle aura à faire : partir d'une valeur de 1, puis multiplier cette valeur par tous les entiers compris entre 1 et n. Cet algorithme traduit donc directement la définition mathématique de la factorielle, à savoir : $n ! = \sum_{i=1}^{i=n} i$

A l'opposé, en programmation logique, on se contente de modéliser le problème à partir de deux propriétés de la factorielle :

- n ! = 1
- pour n > 0, n! = n * (n-1) !

Ce qui ce décrit facilement d'un point de vue logique :

```
Factorielle(0,1).
∀n ( Sup(n,0) ∧ Factorielle(n-1,fn1) ⇒ Factorielle(n,n*fn1))
```

Pour calculer alors la factorielle d'un nombre n donné, un langage de programmation logique va supposer qu'il s'agit de la conclusion d'un raisonnement dont les formules logiques précédentes tiennent lieu de prémisses. On cherche donc à conclure à `Factorielle(n,fn)` à partir de cette modélisation logique. Dans le cas de Prolog, on applique la méthode de résolution pour en déduire que cette factorielle est égale à n fois celle de (n-1), qui est elle-même égale à (n-1) fois celle de (n-2) et ainsi de suite récursivement jusqu'à arriver à `0! = 1`. Au final, on aura fait les mêmes opérations que dans le cas de la programmation impérative, mais sans donner explicitement à l'ordinateur la séquence d'opérations à réaliser. Ceci explique, que pour certains problèmes de haut-niveau, la programmation logique permet de prototypage de système dix fois plus rapide qu'un langage impératif (qu'il soit orienté objet ou non).

Le langage Prolog s'appuie sur les capacités de formalisation de la logique des prédicats. Comme nous l'avons expliqué au chapitre précédent, la semi-décidabilité de la LP1 rend problématique son application au calcul automatique. C'est pourquoi Prolog se limite à un sous-ensemble de la LP1 limité aux clauses de Horn et ne comportant aucun symbole fonctionnel, qui, lui, est décidable.

Nous allons tout d'abord étudier la notion de clauses de Horn. Notons auparavant qu'il existe d'autres langages de programmation logique, dont certains dérivent directement de Prolog : λ-Prolog (qui met en jeu des logiques d'ordre supérieures à la LP1), CLIPS (langage déclaratif et logique utile à la modélisation de systèmes experts et reprenant la syntaxe du langage C) et Oz, qui est un langage qui permet de combiner des paradigmes de programmation logique, fonctionnels, impératifs et objets.

2. CLAUSES DE HORN

2.1. DEFINITION

Que l'on considère la logique des propositions ou la logique des prédicats du 1er ordre, on appelle clause de Horn des clauses comportant au plus un littéral positif. Par rapport à l'ensemble des formules logiques que l'on peut construire, il ne peut donc exister que trois types de clauses de Horn :

- les clauses de Horn négatives, qui ne comportent que des littéraux négatifs : $\neg L1 \vee \ldots \vee \neg Ln$
- les clauses de Horn positives, qui ont un littéral positif unique (il ne peut y en avoir plus) : L
- les clauses de Horn, strictes, qui comportent un littéral positif et un nombre quelconque (supérieur ou égal à 1) de littéraux négatifs : $LC \vee \neg L1 \vee \ldots \vee \neg Ln$

On a par exemple :

- $\neg Paris(x) \vee \neg Lyon(x)$ est une clause de Horn négative,
- $Paris(Tour\_Eiffel)$ est une clause de Horn positive,
- $\neg Paris(x) \vee France(x)$ est une clause de Horn stricte.

Par construction, le sous-ensemble des clauses de Horn est strictement inclus dans la logique des prédicats du 1er ordre. On sait par exemple qu'il existe à Las Vegas une reproduction de la Tour Eiffel. On peut donc écrire en LP1 $Paris(Tour\_Eiffel) \vee Las\_Vegas(Tour\_Eiffel)$. Cet énoncé ne constitue pas une clause de Horn, puisqu'il présente deux littéeraux positifs. Nous allons toutefois voir que cette situation ne limite pas les capacités de représentation des clauses de Horn au regard de nos besoins.

Pour aller plus loin : Alfred Horn (1908-2001)

Horn A. (1951) On sentences which are true of direct unions of algebras, *Journal of Symbolic Logic*, **16**, pages 14-21..

2.2. INTERPRETATION

L'impact limité en termes de formalisation de la restriction de la logique des prédicats du 1er aux clauses de Horn, identifié dès 1951 par Alfred Horn, vient du fait que les trois types de clauses de Horn correspondent aux trois éléments centraux que l'on retrouve dans tout raisonnement. A savoir :

- *les règles d'inférence*, qui permettent de déduire une nouvelle connaissance LC à partir de connaissances connues ($L1, \ldots, Ln$). Celles-ci se représentent sous la forme $L1 \wedge \ldots \wedge Ln \Rightarrow LC$. et conduisent à la forme clausale suivante : $\neg L2 \vee \ldots \vee \neg Ln \vee LC$. On retrouve la définition des clauses de Horn strictes.
- *les faits*, permettent la déclaration de connaissances L unitaires. Elles se représentent comme une fbf de forme L, ce qui correspond directement à la définition des clauses de Horn positives.
- *Les conclusions* correspondent enfin à l'ensemble des connaissances nouvelles ($L1, \ldots, Ln$) que l'on peut déduire de faits initiaux après application d'un raisonnement déductif utilisant également les règles d'inférences. Ces conclusions se représentent par une formule de la forme $L1 \wedge \ldots \wedge Ln$. Lorsque l'on veut démontrer la validité du raisonnement qui a conduit à ces conclusions à l'aide du théorème de réfutation, il nous faut nier cette conclusion. Cela nous conduit à une clause de la forme $\neg L1 \vee \ldots \vee \neg Ln$ qui correspond directement à la définition des clauses de Horn négatives.

Il semble ainsi que tout raisonnement standard puisse être représenté dans le sous-ensemble des clauses de Horn. Afin d'illustrer cette affirmation, prenons un exemple inspiré que l'exemple introductif à la logique des prédicats du 1er ordre que nous avons donné au chapitre précédent :

(P1) *Pour tout x, si x est coupable alors x est condamné*
(P2) *Caïn est coupable*
(P3) *Il existe au moins un x tel que x est coupable et x est condamné*

(C) donc *Caïn est condamné*

Nous avions vu que cet exemple pouvait être représenté de la manière suivante :

(P1) `∀x (Coupable(x) ⇒ Condamne(x) )`
(P2) `Coupable(Cain)`
(P3) `∃x (Coupable(x) ∧ Condamne(x))`

(C) `Condamne(Cain)`

Si on procède à la mise sous forme clausale de la formule de réfutation correspondant à ce raisonnement, il vient immédiatement :

(P1) `¬Coupable(x) ∨ Condamne(x)`
(P2) `Coupable(Cain)`
(P3) `Coupable(A)`
(P3') `Condamne(A)`

(¬C) `¬Condamne(Cain)`

On constate que les clauses que l'on obtient sont bien des clauses de clauses de Horn. La séparation de la prémisse (P3) n'est pas un artifice pour ne pas avoir deux littéraux positifs. Sa mise sous forme clausale conduit en effet bien à deux clauses de Horn positives.

Question de cours : clauses de Horn

On considère ici l'ensemble des clauses suivantes, où P, Q, R et S désignent des formules atomiques

(1) $P \vee Q \vee \neg R$
(2) $P \vee \neg S$
(3) $\forall x\ (Q(x) \vee T(x))$

1 — Quelles sont, parmi ces formules, celles qui ne sont pas des clauses de Horn

2 — Pourriez-vous modéliser dans la LP1 restreinte aux clauses de Horn le raisonnement ci-dessous :

(P1) *Pour tout x, si x est coupable alors x est condamné*
(P2) *Caïn est coupable*

2.3. ABSENCE DE SYMBOLES FONCTIONNELS

Une seconde caractéristique du langage Prolog est qu'il n'autorise pas l'emploi de symboles fonctionnels. En pratique, cette contrainte ne représente pas non plus de limitation en termes de modélisation. En effet, il est toujours possible de remplacer un terme fonctionnel par une formule prédicative.

Remplacement des symboles fonctionnels

Soit f une fonction de n variables internes à un domaine d'interprétation donné. Il est possible de remplacer cette fonction à l'aide d'un prédicat F d'arité (n+1) défini comme suit : pour tous termes `t1, …, tn` *et* `res, F(t1,…,tn, res)` *est vrai si* `res = f(t1, …, tn)`

Exemple – Considérons l'énoncé suivant : *le grand parent de quelqu'un est toujours le parent (père ou mère) d'un parent de ce dernier*. Cette connaissance peut se traduire comme suit en LP1 :

```
∀x Egal(gd_parent(x), parent(parent(x))
```

Il est alors possible de remplacer les termes fonctionnels `gd_parent(x)` et `parent(x)` par des prédicats explicitant le résultat de ces fonctions :

```
∀x ∃g ∃p ∃pp (Gd_Parent(x,g) ∧ Parent(x,p) ∧ Parent(p,pp) ∧ Egal(g,pp))
```

Cette formule peut enfin être simplifiée dans un second temps sous la forme :

$\forall x\ \exists gp\ \exists p$ `(Gd_Parent(x,gp)` $\wedge$ `Parent(x,p)` $\wedge$ `Parent(p,gp))`

Remarque – La différence qui existe entre le terme fonctionnel `f(t1, …, tn)` et la formule prédicative *`F(t1,…,tn, res)`* équivalente est de même nature que celle qui distingue dans certains langages de programmation une fonction, qui retourne explicitement son résultat, et une procédure, qui passe celui-ci en argument de sortie.

A titre illustratif, il n'est pas inutile de revenir sur l'exemple d'indécidabilité que nous avions étudié en exercice de cours dans le paragraphe précédent. Celui-ci concernait le raisonnement suivant :

$\forall x\ (\ P(x) \Rightarrow Q(f(x))\ )\ ,\ \forall x\ (\ Q(x) \Rightarrow P(f(x))\ )\ ,\ P(A) \models^{?} \forall x\ P(x)$

Si l'on remplace les termes fonctionnels pour représenter ce raisonnement en Prolog, il vient dès lors :

(P1) $\forall x\ (\ P(x) \Rightarrow Q(f(x))\ ) \equiv \forall x\ (\ P(x) \Rightarrow F(x,y) \wedge Q(y))$

(P2) $\forall x\ (\ Q(x) \Rightarrow P(f(x))\ ) \equiv \forall x\ (\ Q(x) \Rightarrow F(x,y) \wedge P(y))$

(P3) $P(A) \equiv$

(¬C) $\neg \forall x\ P(x) \equiv \exists x \neg P(x)$

On voit que (P1) et (P2) nous donnent deux clauses $\neg P(x) \vee F(x,y) \vee\ Q(y)$ et $\neg Q(x) \vee F(x,y) \vee\ P(y)$ qui ne sont pas des clauses de Horn, puisqu'elles comportent chacune deux littéraux positifs. L'indécidabilité de cet exemple n'est donc pas réellement une surprise : seul un ensemble de clauses de Horn pourrait nous assurer d'une décidabilité du problème.

2.3. DECIDABILITE

La vérification de la satisfaisabilité d'une formule logique[1] est une question essentielle en informatique. Elle est en effet directement liée au problème de satisfaisabilité booléenne (appelé également problème SAT) qui, à la suite des travaux indépendants de Cook et Levin au début des années 1970, a conduit à la définition de la classe des problèmes NP-complets en théorie de la complexité des algorithmes.

Pour aller plus loin : la classe des problèmes NP-complets et le théorème de Cook-Levin

- problèmes dont on peut vérifier une solution en un temps polynomial, mais dont les temps de recherche d'une solution sont eux de compléxité exponentielle

- Cook et Levin ont montré qu'il était possible de ramener tout problème NP-complexe au problème de la satisfaisabilité d'une formule booléenne (problème SAT) en un temps polynomial.

Stephen Cook (1971) The complexity of theorem proving procedures, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158.

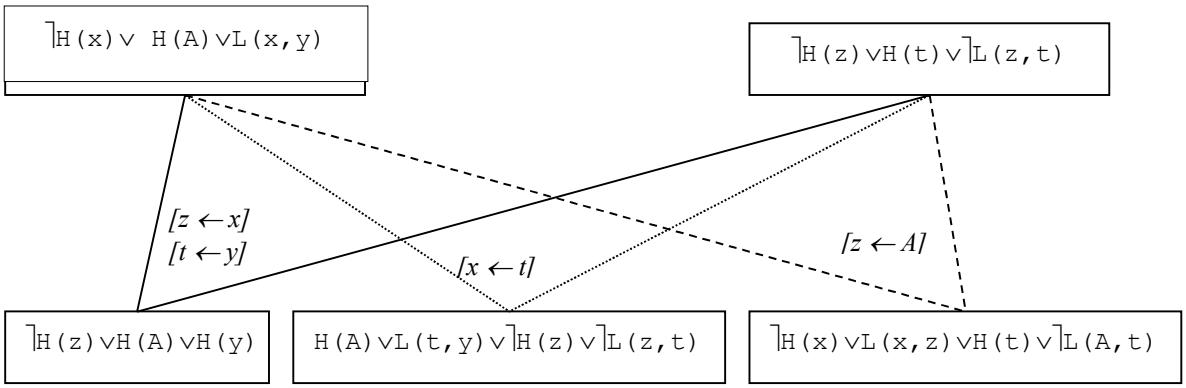
Leonid Levin (1973) Универсальные задачи перебора, Universal'nye perebornye zadachi, *Проблемы передачи информации* (*Problemy Peredachi Informatsii*), **9** (3), pages 115–116

Plus précisément, le problème SAT revient à montrer qu'une formule de la logique des propositions mise sous forme normale conjonctive (donc clausale) est satisfaisable. Nous avons vu que si nous étendions cette question à la logique des prédicats, le problème devenait indécidable : il n'existe pas

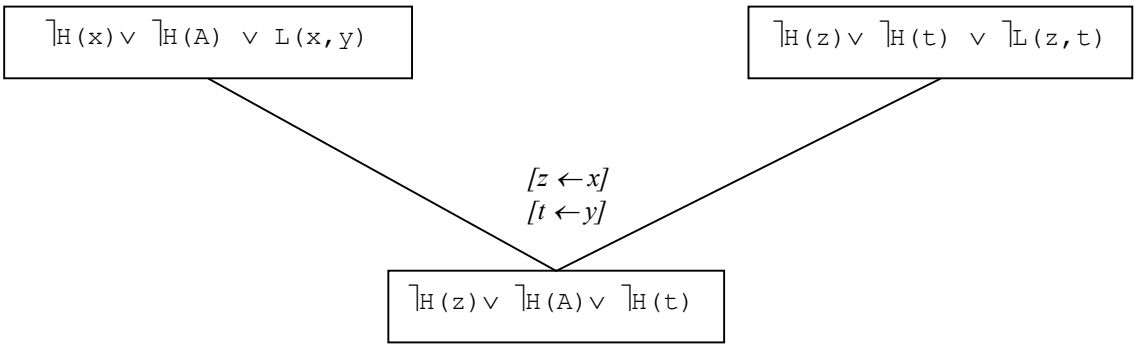
[1] Et donc par conséquence la vérification du caractère contradictoire d'une formule, qui nous intéresse dans le cas de la résolution.

d'algorithme pouvant décider en un temps fini si une forme clausale de la LP1 est ou non satisfaisable. Heureusement, la LP1 réduite aux clauses de Horn devient décidable, ce qui permet en particulier l'utilisation de la résolution dans le langage Prolog.

Nous ne démontrerons pas ce résultat ici, car nous étudierons la terminaison de la résolution Prolog dans la partie consacrée à la sémantique de ce langage. Nous nous contenterons de donner une idée de l'intérêt des clauses de Horn en termes de décidabilité. Lorsque l'on applique la résolution à des clauses quelconques, plusieurs choix de résolution se posent le plus souvent à nous, qui vont conduire à autant de nouvelles résolvantes. Considérons par exemple la paire de clauses $\neg H(x) \vee H(y) \vee L(x,y)$ et $\neg H(z) \vee H(t) \vee \neg L(z,t)$. Il ne s'agit pas d'un ensemble de clauses de Horn, puisque la première d'entre elles présente 2 littéraux positifs. On observe immédiatement sur la figure suivante que ces deux clauses donnent lieu à elles seules plusieurs résolvantes dont la complexité peut aller croissante.



Étudions maintenant l'ensemble de clauses $\neg H(x) \vee \neg H(y) \vee L(x,y)$ et $\neg H(z) \vee H(t) \vee \neg L(z,t)$ formellement très proche du précédent, mais qui ne contient plus que des clauses de Horn. Étant donné que la résolution doit associer un littéral positif à un littéral négatif après unification, et que les clauses de Horn ne disposent au mieux que d'un unique littéral positif, les possibilités de résolution sont bien plus réduites, comme le montre le schéma ci-dessous :



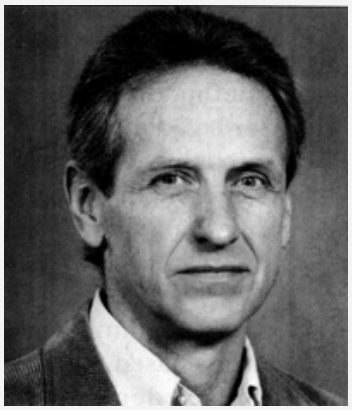
On obtient dès lors moins de résolvantes : on conçoit donc par l'exemple que la résolution limitée aux clauses de Horn se traduit par une combinatoire d'analyse réduite. Cette réduction est en fait telle que la résolution est alors décidable, comme nous le verrons avec Prolog.

3. LANGAGE PROLOG : SYNTAXE

Prolog est un langage de programmation logique développé en 1972 par l'équipe d'Alain Colmerauer et Philippe Roussel à Marseille, en collaboration avec Robert Kowalski de l'Université d'Edinburgh. Philippe Roussel a développé le premier interpréteur Prolog, tandis que l'on doit le premier compilateur Prolog est l'oeuvre Robert Kowalski. Cette origine gémellaire a eu pour effet le développement de plusieurs dialectes Prolog dont la filiation se partageait à ce qu'on appelait alors le Prolog de Marseille ou le Prolog d'Edinburgh. Le Prolog de Marseille ayant une syntaxe moins lisible, plus liée à une formalisation mathématique, s'est toutefois éteint progressivement. Depuis 1995, ce langage est normalisé (ISO Prolog) suivant une syntaxe qui reprend celle du Prolog d'Edinburgh.

Initialement développé pour les besoins du Traitement Automatique des Langues où il reste largement utilisé, Prolog est également utilisé en Intelligence Artificielle (systèmes experts), dans le monde du jeu et en démonstration automatique de preuve. Nous allons maintenant étudier la syntaxe du langage ISO Prolog.

Pour aller plus loin : Les inventeurs de Prolog

rajouter une photographie de Ph. Roussel

3.1. REPRESENTATION DES ELEMENTS DU PROBLEME : TERMES

En logique des prédicats du 1er ordre, il existe trois types de termes : les constantes, les variables et les termes fonctionnels. Comme évoqué au paragraphe 2, les fonctions n'existent pas en Prolog et doivent être remplacées par une formulation prédicative. Il nous reste donc deux types de termes :

Termes Prolog

- *les **constantes** sont soit des termes numériques (entiers, flottants) soit des identificateurs représentant des éléments spécifiques du problème que l'on nommera sous la forme de chaînes de caractères commençant par une **minuscule**. Les chaînes de caractères entre apostrophe, représentant elles mêmes, sont également des constantes.*
- *les **variables** sont représentées par un identificateur sous forme de chaîne de caractères commençant nécessairement par une **majuscule.***

Exemple – Les termes suivants sont des constantes : `jean`, `x`, `2`, `2.8`, `c3PO`, `psg`, `'message a afficher'`. A l'opposé, les termes suivants sont des variables : `Jean`, `X`, `C3PO`, `PSG`. Notons au passage que la chaîne de caractères `2,8` ne représente pas un réel : comme dans tous les langages de programmation, la séparation entre partie entière et décimale doit être marquée par un point et non une virgule (notation anglo-saxonne). De même, l'utilisation des accents est interdite en Prolog.

Remarque – On remarque donc que les conventions de représentation des constantes et variables sont inversées en logique et en Prolog, puisqu'une majuscule initiale caractérise ici une variable et non pas une constante. Il est d'autant plus impérieux de prendre garde à cette inversion que Prolog ne requiert pas de déclaration des variables pas plus que des constantes. Supposez que vous désiriez intégrer dans votre programme une constante et que vous la définissiez par erreur avec une majuscule initiale, Prolog ne pourra pas détecter votre mégarde : il considérera que vous avez utilisé une nouvelle variable et ne vous indiquera aucune erreur de compilation ou interprétation.

3.2. JUGEMENTS DE BASE : FORMULES PREDICATIVES ET NEGATION PROLOG

Les relations de base entre termes du domaine sont représentés, comme en LP1, par des formules atomiques (i.e. prédicatives). Les prédicats se définissent comme en LP1, à l'exception notable de la capitalisation de leur identifiant :

Prédicats Prolog

*Les **prédicats** d'arité n sont représentés sous la forme `id(t1,…,t2)` avec.*

- `id` *identificateur du prédicat : chaîne de caractères commençant par une minuscule.*
- `t1,… tn` *ensemble de termes Prolog.*

Exemple – les exemples suivants correspondent à des formules atomiques du langage Prolog : `mortel(X)`, `mortel(socrate)`, `aime(jean,marie)`.

Remarque – Remarquons qu'en Prolog, les formules atomiques sont toutes positives. En effet, il n'y a donc pas de représentation directe de la négation, puisque nous avons vu dans l'interprétation des clauses de Horn (paragraphe 2.2) que tous les littéraux sont positifs dans l'écriture implicative des règles (`L1`$\wedge$…$\wedge$`Ln` $\Rightarrow$ `LC`) de même que dans les faits. La question de cours du paragrapge 2.2 nous a montré qu'il est généralement possible d'éviter le recours à un littéral négatif par une modification astucieuse de la modélisation.

Il est possible de définir un prédicat de négation en Prolog, à l'aide d'un opérateur appelé coupure, mais cela dépasse le cadre de l'étude de ce que l'on a coutume d'appeler le « Prolog pur », c'est-à-dire la partie de Prolog exactement calquée sur la logique des prédicats. Nous n'étudierons donc pas la négation Prolog dans le cadre de ce livre.

Remarquons enfin que, comme en LP1, un prédicat peut-être sans problème d'arité 0.

3.3. PROGRAMME : CLAUSES PROLOG

Enfin, un programme Prolog est constitué d'un ensemble de clauses Prolog qui sont formellement équivalentes à des clauses de Horn. Réaliser un programme Prolog consiste ainsi à définir un ensemble de connaissances logiques qui modélisent le problème. Chaque nouvelle connaissance est indépendante des autres, à la différence des instructions d'un programme impératif. Elle consistera en l'articulation de relations prédicatives regroupées dans une clause, qui se terminera par un point. Ces connaissances peuvent être de deux types déjà rencontrées dans le paragraphe 2.2 :

Clauses Prolog : programme

- *les faits* consistent à déclarer une connaissance à l'aide d'une formule prédicative. Cette formule atomique est nécessairement unique, puisqu'avec deux prédicats nous ne serions plus en présence d'une clause de Horn (deux littéraux positifs). On a donc :

  `Fait.` avec `Fait` formule atomique unique.

- *.les règles d'inférence* permettent relier une `Conclusion` à un ensemble de prémisses dont elle est la conséquence sémantique. Ces connaissances sont représentées par des formules prédicatives. La règle correspond à la clause de Horn stricte suivante : `Premiise1` $\wedge$ … $\wedge$ `Premissen` $\Rightarrow$ `Conclusion`. En Prolog, on n'adopte toutefois pas cette écriture implicative, mais une écriture procédurale de la forme :

  `Conclusion :- Premisse1, …, Premissen.`
  où `Conclusion` , `Premisse1,…, Premissen` sont des formule atomiques

  Cette écriture se lit comme suit : pour montrer la `Conclusion` (appelée *tête de règle*), il suffit d'avoir montré avant toutes les prémisses `Premisse1 à Premissen` présentes dans ce qu'on appelle le *corps de la règle*

Remarques – Noter l'utilisation des symboles réservés suivant :

`.` (point)	sert à marquer la fin d'une clause
`,` (virgule)	sert à traduire la conjonction
`:-`	marque l'implication en lecture procédurale
`!`	symbole de coupure, qui ne sera pas étudié dans ce cours limité à Prolog pur.

Exemples – On considère un petit programme qui permet de calculer la distance entre plusieurs villes le long d'une autoroute. On veut modéliser les connaissances suivantes :

- *la distance de Tours à Blois est de 56 kilomètres et celle de Blois à Orléans est de 49 kilomètres*
- *Une ville X est plus proche d'une ville S qu'une ville Y si sa distance à S est inférieure à la distance Z à S.*

On peut représenter cet ensemble de connaissances par un programme Prolog. Pour cela, nous

utilisons deux prédicats pour représenter différentes relations exprimées dans le problème :

- `dist/3` (arité 3) tel que `dist(X,Y,D)` est vrai si D est la distance entre X et Y,
- `prox/3` (arité 3) tel que `prox(X,Y,S)` est vrai si X est plus proche de S que Y.

Les connaissances portées par le problème se représentent alors par le programme Prolog suivant :

```
dist(tours, blois,56).
dist(blois, orlans,49).
prox(Pres,S,Loin) :- dist(Pres,S,Dpres), dist(Loin,S,DLoin), sup(DLoin,DPres).
prox(Pres,S,Loin) :- dist(S,Pres,Dpres), dist(Loin,S,DLoin), sup(DLoin,DPres).
prox(Pres,S,Loin) :- dist(Pres,S,Dpres), dist(S,Loin,DLoin), sup(DLoin,DPres).
prox(Pres,S,Loin) :- dist(S,Pres,Dpres), dist(S,Loin,DLoin), sup(DLoin,DPres).
```

On voit que la modélisation Prolog n'est pas toujours directe : ici, nous avons besoin de 4 règles pour modéliser la commutativité de la distance (la distance de A à B et la même que celle de B à A) qui n'était pas explicitée dans la donnée du problème. Une solution plus élégante était envisageable, mais ce simple exemple nous montre que la réalisation d'un programme Prolog se limite rarement à une traduction littérale du problème.

On remarque également que la conjonction de faits énoncés dans le problème (*la distance de Tours à Blois est de 56 kilomètres et celle de Blois à Orléans est de 49 kilomètres*) a donné lieu à deux clauses séparées. On aurait pu imaginer écrire cet énoncé à l'aide du symbole de conjonction :

```
dist(tours, blois,56), dist(blois, orlans,49).
```

Cette écriture est toutefois interdite, puisque nous serions alors en présence d'une conjonction, alors que par définition une clause est faite de disjonctions. Cette impossibilité n'est toutefois par une restriction : les deux faits sont totalement indépendants, il n'y a aucun souci à les représenter par deux clauses séparées. Ainsi, on a la règle générale :

Conjonction de faits

Il est interdit de représenter en Prolog une conjonction de faits telle que `Fait1, Fait 2.` Pour représenter cette connaissance, on utilisera deux clauses séparées :

```
Fait1.
Fait 2.
```

Pour la même raison, si la conjonction de prémisses est autorisée dans le corps des règles Prolog, il n'est pas possible d'avoir une conjonction de conclusions en tête de règle. La clause suivante ne peut donc pas se rencontrer en Prolog :

```
français(X), europeen(X)  :- blesois(X).
```

Cette règle nous dit que *si quelqu'un est blésois, alors il est français et européen*. On a :

$\forall x$ (Blesois(x) $\Rightarrow$ Français(x) $\wedge$ Europeen(x))

La mise sous forme clausale de cette fbf montre que nous ne sommes pas en présence d'une clause si nous conservons la conjonction : $\neg$Blesois(x) $\vee$ (Français(x) $\wedge$ Europeen(x)). Par distributivité, on obtient par contre deux clauses : $\neg$Blesois(x) $\vee$ (Français(x) et $\neg$Blesois(x) $\vee$ Europeen(x). Il s'agit de clauses de Horn puisqu'elles n'ont qu'un littéral positif. Elles correspondent aux implications $\forall$x (Blesois(x) $\Rightarrow$ Français(x)) et $\forall$x (Blesois(x) $\Rightarrow$ Français(x)) donc à deux clauses Prolog indépendantes :

```
français(X) :- blesois(X).
europeen(X) :- blesois(X).
```

On voit bien que l'on ne perd absolument pas en généralité avec ce découpage en deux clauses : les deux conclusions sont en effet totalement indépendantes. Ainsi, il vient de manière générale :

Conjonction de conclusion dans une règle Prolog

Il est interdit de représenter en Prolog une conjonction de conclusions dans une règle telle que `Conclusion1, Conclusion :- Hypothese.` Cette règle peut être rédigée sans perte d'information par deux règles indépendantes :

```
Conclusion1 :- Hypothese.
Conclusion2 :- Hypothese.
```

Question de cours : syntaxe Prolog.

On considère le petit programme Prolog suivant :

```
finistere(Brest);
morbihan(Vannes) , morbihan(Lorient);
bretagne(X) :- finistere(X);
bretagne(X) :- morbihan(X);
france(bretagne(X)).
```

1. Détectez les erreurs de syntaxe que comporte le programme.

2. Quelles sont les faits et les règles dans ce programme ?

3. Quelles sont les constantes et les variables dans ce programme ? Ne comporte-t-il pas d'erreurs de ce point de vue ? S'agit-il d'erreurs de syntaxe ou de sémantique, à savoir que l'interpréteur Prolog acceptera le programme mais aura un comportement erroné.

4. Corrigez le programme pour qu'il soit complètement correct.

3.4. UTILISER PROLOG : QUESTIONS

Le lecteur attentif aura noté que nous n'avons pas utilisé jusqu'ici le troisième type de clause de Horn : les clauses négatives, qui correspondent à la conclusion d'un raisonnement. Ces clauses ne font en effet par partie du programme Prolog, mais servent à interroger celui-ci pour obtenir de nouvelles connaissances (un résultat au sens informatique) à partir du programme.

Ces questions vont être considérées par Prolog comme une conclusion qui doit être démontrée conséquence sémantique des certaines clauses du programme. Nous étudierions au paragraphe 4 la stratégie de résolution utilisée par Prolog. Pour l'heure, notons qu'elle est directement dérivée de la résolution en LP1. Comme nous l'avons vu dans le chapitre précédent, les réponses à l'interrogation du programme, lorsqu'elles existent, seront alors données par un processus d'unification.

Dans l'immédiat, il suffit de noter que l'utilisation d'un programme Prolog consiste à lui poser des questions. Pour ce faire, on définit des questions Prolog suivant la syntaxe suivante :

Questions Prolog : programme

- les questions Prolog sont constituées d'une formule atomique ou de la conjonction de plusieurs formules atomiques, suivant la syntaxe:

`?- Atome.`	avec `Atome` formule atomique unique.
`?- Atome1, …, Atomen.`	avec `Atome1, …, Atomen` formules atomiques.

Exemples – Si nous reprenons l'exemple précédent de calcul de distance, on peut ainsi imaginer de poser les questions suivantes :

```
?- dist(tours,blois,56).
```

Prolog répondra alors : `YES` :

```
?- dist(tours,blois,40).
```

Prolog répondra alors : `NO`

```
?- dist(tours,blois,X).
```

Prolog répondra alors : `YES X = 56`, la réponse précise sur la valeur de X étant obtenue par unification avec le fait correspondant du programme. Bien entendu, la résolution ne se limite pas à aller questionner directement la base de faits du programme, mais peut également concerner les conclusions présentes en tête de règles. On peut ainsi demander :

```
?- prox(Proche,blois,Loin).
```

Prolog répondra alors : `YES ; Proche = 'Orleans', Loin = 'Tours'`

3.5. PORTEE DES TERMES PROLOG

Nous avions vu en LP1 que deux clauses utilisée dans la résolution n'avaient rien avoir entre elles et

que donc leurs variables étaient totalement séparées. De même, en Prolog, toute clause constitue un grain de connaissance qui (du moins en Prolog pur…) doit être considérée comme indépendante des autres. Prenons l'exemple suivant :

```
parent(X,Y) :- pere(X,Y).  équivalent à la fbf LP1 : ∀x∀y (Pere(x,y) ⇒ Parent(x,y)).
parent(X,Y) :- mere(X,Y).  équivalent à la fbf LP1 : ∀x∀y (Mere(x,y) ⇒ Parent(x,y)).
```

Ces deux clauses sont indépendantes, et il est clair que même si elles portent le même identificateur, les variables Y des deux clauses ne représentent pas le même objet du problème. Il en est de même pour les deux variables X : rien ne nous oblige à considérer qu'elles représentent la même personne. On aurait ainsi pu écrire :

```
parent(EnfantP,Pere) :- pere(EnfantP,Pere).
parent(EnfantM,Mere) :- mere(EnfantM,Mere).
```

Ainsi, en Prolog, toute variable est propre à la clause où elle est présente : il n'y a pas de notion de variable globale ou locale dans un programme, dont l'identificateur peut se retrouver dans plusieurs instructions et désigner toujours la même valeur. Le lien entre les valeurs des variables des différentes clauses n'est pas exprimé dans le programme, c'est l'algorithme de résolution qui le fera au cas par cause lors des étapes de résolution. Dès lors, il vient :

Portée d'une variable Prolog

La portée d'une variable Prolog se limite à la clause dans laquelle est figure.

A l'opposé, une variable présente à différents endroits dans la tête et le corps d'une clause donnée réfère bien toujours au même élément du problème : la variable est partagée sur l'ensemble de la clause. Considérons l'exemple suivant :

```
entre(Petit,Milieu,Gd) :- inferieur(Petit,Milieu), inferieur(Milieu,Gd).
entre(Gd,Milieu,Petit) :- inferieur(Petit,Milieu), inferieur(Milieu,Gd).
```

Si Milieu (par exemple) ne réfère pas nécessairement au même élément du problème dans les deux clauses, il doit être par contre clair que l'on est bien en présence de la même valeur lors que la variable est partagée entre la tête de règle `entre(Petit,Milieu,Gd)` et les prémisses `inferieur(Petit,Milieu)` et `inferieur(Milieu,Gd` du corps de cette même règle. Le passage en LP1 de ce mini-programme Prolog confirme la véracité de cette observation : les variables partagée sont bien dans la portée du même quantificateur :

```
∀p ∀milieu ∀g (Inferieur(p,milieu) ∧ Inferieur(milieu,g) ⇒ entre(p,milieu,g))
∀p ∀milieu ∀g (Inferieur(p,milieu) ∧ Inferieur(milieu,g) ⇒ entre(g,milieu,p))
```

A l'opposé, une constante réfère explicitement à une donnée précise du problème. Il serait hors de question que la constante `blois`, par exemple, réfère à deux villes différentes dans deux clauses d'un programme Prolog : si nous utilisons une constante, c'est bien pour référer de manière stable au même élément tout au long du problème. D'où l'affirmation :

Portée d'une constante Prolog

La portée de toute constante est l'ensemble du programme Prolog dans lequel elle se situe.

Ces notions de portée doivent être bien comprises pour éviter les erreurs, puisqu'en Prolog, il n'y a pas de déclaration explicite de variable ou de constantes. Enfin, il n'est peut-être pas inutile de préciser qu'un prédicat donné a toujours le même sens tout au long d'un programme.

Portée de la définition d'un prédicat Prolog

Un prédicat Prolog caractérise la même relation logique tout au long du programme où il se situe.

Tout en étant évidente, cette dernière affirmation est là encore importante car les prédicats ne sont pas eux non plus définis explicitement. Ainsi, si suite à un oubli, nous utilisons dans une clause un prédicat d'arité donnée avec 2 arguments, puis avec un seul (mais le même identificateur) dans une autre clause, l'interpréteur ou le compilateur Prolog considérera que vous avez défini deux prédicats différents. Il ne vous adressera donc aucun message d'alerte à ce sujet.

3.6. ANONYMAT DE VARIABLES

La portée d'une variable étant limitée à sa clause, il semble plus lisible de renommer toutes les

variables d'une clause à une autre pour éviter les confusions. Si cela est recommandé du point de vue génie logiciel (voir l'encart à ce sujet), trouver de nouveaux noms de variables est rapidement usant. Et ce d'autant plus que la variable considérée n'a parfois que guère d'utilité dans le programme ! Considérons ainsi la règle Prolog suivante :

```
est_mere(X) :- mere(X,Y).
```

Ici, la valeur de la variable Y n'a guère d'importance : on veut simplement vérifier lors de la résolution qu' il soit possible d'unifier Y avec quelque chose pour confirmer que X est la mère de quelqu'un, peu importe son nom. De même, supposons que l'on pose la question :

```
?- est_mere(rachida).
```

Visiblement, peu nous chaut ici le nom Y de l'enfant obtenu lors de la résolution, on désire simplement avoir une réponse positive ou négative. Prolog nous donnera pourtant en réponse la valeur de Y, s'il est arrivé à faire une telle résolution. Pour éviter des effets de bord indésirables, il est possible de spécifier à Prolog qu'une variable doit être effectivement unifiée, mais que l'on se moque de sa valeur.

Anonymat d'une variable

Lorsque la valeur précise d'une variable n'est pas importante, il est possible de l'anonymer en remplacant son identifiant par le symbole _ (underscore ou tiret inférieur). Par exemple :

```
est_mere(X) :- mere(X,_).
```

Il faut bien comprendre qu'une variable anonymée garde tout son statut de variable : Prolog va ainsi lui donner un nom pour ses besoins internes d'unification. Par exemple, dans la distribution SWI-Prolog, cet identificateur prend la forme `G_123`, le numéro de variable étant géré par Prolog pour éviter précisément les confusions de variables. En fait, l'interpréteur renomme de la sorte toutes vos variables, qu'elles soient identifiées explicitement ou anonymes.

Cet anonymat doit toutefois être utilisé avec discernement, sous peine d'erreurs. Imaginons par exemple que l'on ait la règle suivante, qui nous dit que `GP` est le grand parent de `PF` si l'on trouve un parent `P` du petit-enfant `PE` tel que `GP` soit lui-même le parent de `P`.

```
gdparent(PE,GP) :- parent(PE,P), parent(P,GP).
```

Dans cette règle, l'identité du parent `P` ne nous intéresse pas : seule son existence revêt de l'importance à nos yeux. On pourrait donc avoir pour idée d'anonymer la variable partagée `P` :

```
gdparent(PE,GP) :- parent(PE,_), parent(_,GP).
```

Cette écriture pose toutefois problème. En effet, l'anonymat de la variable signifie à Prolog que l'on ne prête pas attention à sa valeur. Dans le cas présent, plus rien n'impose alors que la valeur de la variable partagée dans le premier prédicat `parent(PE,_)` soit identique à cette du second prédicat `parent(_,GP)`. La variable n'est plus partagée mais va être au contraire remplacer par deux variables internes différentes : la règle prend alors une signification erronée comme le montre alors sa traduction en LP1 :

$$\forall pe\ \forall g\_1\ \forall g\_2\ \forall gp\ (parent(pe,g\_1) \wedge parent(pe,g\_2) \Rightarrow gdparent(pe,gp))$$

Il est toutefois possible de signifier à Prolog que la valeur de la variable ne nous importe pas en réponse, mais qu'elle doit être considérée lors de la résolution : le semi-anonymat

Semi-anonymat d'une variable

Lorsque la valeur précise d'une variable n'est pas importante pour l'utilisateur, mais que cette variable est partagée entre plusieurs éléments d'une clause Prolog, il est possible de mettre sous forme semi-anonyme en précédent son identifiant par le symbole _ (underscore ou tiret inférieur). Par exemple :

```
gdparent(PE,GP) :- parent(PE,_P), parent(_P,GP).
```

Le semi-anonymat ne répond pas à notre désir de limiter le nombre de variables à nommer dans un programme, mais permet au moins de limiter les affichages inutiles en sortie. Notons au passage qu'il est également possible d'utiliser des variables anonymes ou semi-anonymes dans les questions que l'on pose via l'interface de dialogue Prolog. On peut ainsi écrire :

```
?- mere(rachida,_).
```

Quelques recommandations issues du génie logiciel appliquées à Prolog

Comme nous avons pu l'observer à plusieurs reprises, Prolog fait appel très fréquemment à des déclarations implicites de variables, de constantes ou de prédicats. De ce fait, si l'on n'y prend pas garde, un programme Prolog peut être syntaxiquement et sémantiquement correct tout en étant très peu lisible. Supposons par exemple que vous ayez réalisé le programme suivant :

```
p(j,p).
p(p,z).
g(X,Y) :- p(X,Z), p(Z,Y).
eg(X) :- g(A,X).
```

Ce programme est très peu compréhensible pour un concepteur qui reprendrait votre code … mais également pour vous-même si vous l'avez délaissé un certain temps. Or, un bon programme est également un programme réutilisable.

Le rôle du génie logiciel est précisément de donner aux informaticiens des méthodes propres pour analyser un problème, concevoir conceptuellement et enfin implémenter proprement un programme. Découvrir les méthodes issues du génie logiciel dépasse de très loin l'objet de cet ouvrage, mais il nous semble nécessaire de rappeler quelques recommandations basiques en termes d'implémentation de programme :

1) *documentation du code* — tout programme doit comporter des commentaires qui facilitent sa compréhension. Ainsi, un programme Prolog devrait toujours comporter :
   - un commentaire d'entête de fichier : nom du fichier, du programmeur, date et version du programme, objet et contenu du code présent dans ce fichier,
   - des commentaires de séparation ou de définition des prédicats : on peut ainsi séparer la partie base de faits de la partie base de règles du programme, mais plus globalement marquer des séparations entre parties du programme qui font sens. Par ailleurs, puisqu'il n'y a pas de définition explicite de prédicats, la sémantique de ces derniers doit être clairement expliquée par des commentaires
   - commentaires de clarification, par exemple sur des parties délicates du programme dont la compréhension n'est pas triviale
2) *lisibilité du code* — La lisibilité du code passe par l'utilisation d'identificateurs (variable, constantes, prédicats) explicites et intelligibles, et par un alignement ou une identitation propre du code. Par exemple, dans un programme Prolog, le symbole implicatif séparant la tête et le corps des règles doit toujours être aligné.

En Prolog, les commentaires sont des chaînes de caractères libres commençant par le symbole réservé /* et se terminant par le symbole de clôture */. Ils sont ignorés par l'interpréteur Proolog. Voici ainsi comment aurait dû être réalisé le programme donné en exemple initial.

```
/*****************************************************************/
/* Exemple de programme proprement redige                        */
/* Date : 15/11/2013         Version : 1.0                       */
/* Concepteur : Jean-Yves Antoine                                */
/* Description : programme de genealogie Prolog                  */
/* Contenu : predicats parent/2 et gd_parent/2                   */
/*****************************************************************/

/*****************************************************************/
/* DEFINITION DES PREDICATS                                      */
/* parent(X,Y) est vrai si X a pour parent Y                     */
/* gd_parent(X,Y) est vrai si X a pour grand parent Y            */
/*****************************************************************/

/********************** BASE DE FAITS ************************/
```

```
parent(jean,paul).
parent(paul,zazie).

/********************** BASE DE REGLES ***************************/

gd_parent(PtEnft,GdP) :- parent(PtEnft,Parent), parent(Parent,GdP).
Est_gd_parent(GdP)    :- gd_parent(_,GdP).
```

On remarque alors que la partie commentaire est bien plus importante que le code en lui-même. Sans être une condition suffisante, il s'agit généralement d'un bon indicateur de la qualité d'un code.

3.7. SYNTHESE : PROGRAMME PROLOG ET LOGIQUE DES PREDICATS

Nous savons désormais comment écrire de manière syntaxiquement correcte un programme en langage Prolog. Reste à savoir comment faire fonctionner ce dernier. C'est tout l'objet du paragraphe suivant, qui est consacré à la sémantique opérationnelle de Prolog. Celle-ci consistera à adopter la résolution de la logique des prédicats. Aussi est-il intéressant de revenir sur un exemple en LP1. Considérons le programme Prolog suivant :

```
/********************** BASE DE FAITS ***************************/

parent(jean,paul).
parent(paul,zazie).

/********************** BASE DE REGLES ***************************/

gd_parent(PtEnft,GdP) :- parent(PtEnft,Parent), parent(Parent,GdP).
```

Ce programme peut se transformer en LP1 sous la forme des formules suivantes :

```
Parent(Jean,Paul)
Parent(Paul,Zazie)
∀PE ∀GP ( ∃P(Parent(PE,P)∧Parent(P,GP) ⇒ Gd_parent(PE,GP))
```

On obtient par mise sous forme clausale les clauses suivantes :

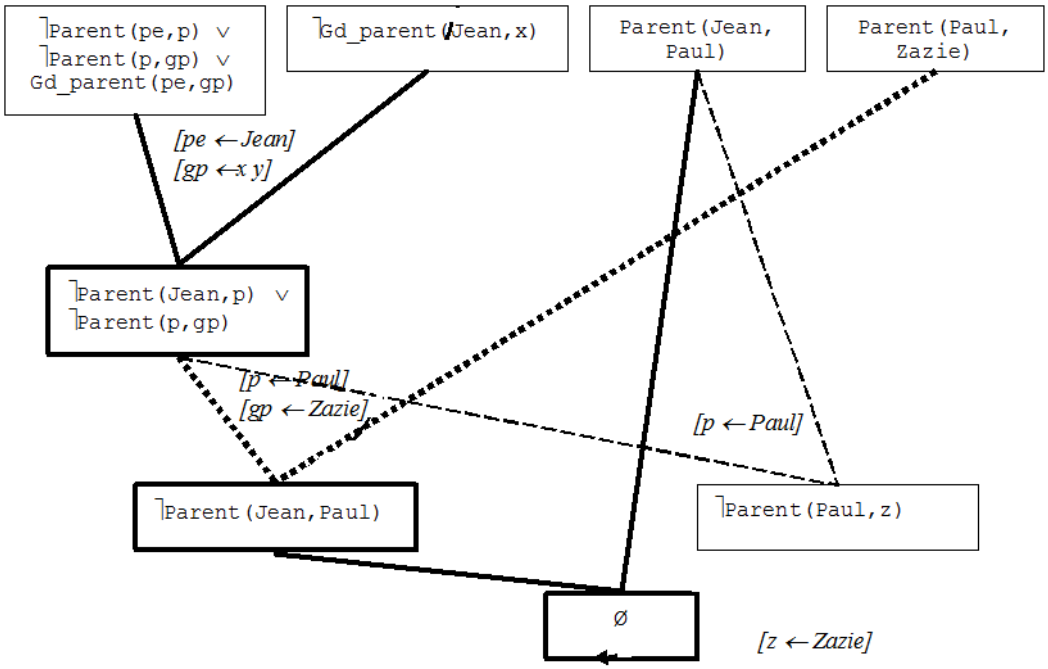
```
Parent(Jean,Paul)
Parent(Paul,Zazie)

∀pe ∀gp ( ⌉∃p(Parent(pe,p)∧Parent(pe,gp) ∨ Gd_parent(pe,gp))
≡ ∀pe ∀gp (∀p (⌉Parent(pe,p) ∨ ⌉Parent(p,gp)) ∨ Gd_parent(pe,gp))
≡ ∀pe ∀gp ∀p (⌉Parent(pe,^p) ∨ ⌉Parent(p,gp) ∨ Gd_parent(pe,gp))
```

D'où la clause : `⌉Parent(pe,^p) ∨ ⌉Parent(p,gp) ∨ Gd_parent(pe,gp)`

Cherchons l'identité du grand_parent de *Jean*. La question Prolog ?-gd`_parent(jean,X)`

Cette question se traduit sous la forme de la conclusion de raisonnement : `∃P Gd_parent(Jean,x)`
Après réfutation, cette conclusion donne la clause : `∀P ⌉Gd_parent(Jean,x)`

On peut alors trouver plusieurs arbres de résolution LP1 pour démontrer la validité du raisonnement correspondant à la question posée au programme Prolog. Nous donnons ci-dessous 2 chemins possibles :



La figure ci-dessus est loin de représenter tous les chemins de résolution possibles pour arriver à démontrer la validité du raisonnement. De même, ne figurent pas les résolvantes qui n'auraient pas pû conduire à la solution. L'interpréteur Prolog doit dès lors faire face à un dilemme équivalent : quelles clauses Prolog apparier en priorité, comment faire si cet appariement ne conduit à rien ? C'est tout l'objet de la mise en place d'une stratégie de résolution systématique, qui est présentée dans le paragraphe ci-dessous.

4. SEMANTIQUE OPERATIONNELLE DE PROLOG

Nous avons vu que Prolog se base sur la résolution LP1 et les connaissances logiques décrites dans le programme pour fournir des réponses aux questions qui lui sont posées. On pourrait dès lors penser qu'une modélisation logique cohérente suffit pour programmer proprement en Prolog. Pourtant, la stratégie de résolution employée par Prolog présente des spécificités qui rendent indispensables l'étude de la sémantique opérationnelle du langage. En effet, si on n'y prend garde, un programme Prolog logiquement correct pourra boucler par méconnaissance du fonctionnement de la résolution Prolog.

Nous allons procéder étape par étape pour décrire la stratégie de résolution utilisée par Prolog. Celle-ci se résume en trois points :

- recherche en chaînage arrière par décomposition de buts
- recherche en profondeur d'abord par tentative avec retour-arrière
- gestion de l'instanciation des variables par unification

4.1. CHAINAGE ARRIERE : RESOLUTION PAR DECOMPOSITION DE BUTS

Étudions tout d'abord un exemple très simple pour donner une idée intuitive de la manière dont Prolog résoud un problème. Pour simplifier les explications, notre programme ne comportera que des prédicats d'arité 0, ce qui est parfaitement autorisé en Prolog (notre programme peut donc se représenter formellement en logique des propositions).

```
pression_atmospherique_elevee.                    (clause 1)
temperature_elevee.                               (clause 2)
anticyclone :- pression_atmospherique_elevee.     (clause 3)
beautemps :- temperature_elevee, anticylcone.     (clause 4)
```

Supposons que l'on désire savoir s'il va faire beau temps. On pose donc la question : `?- beautemps.`

Pour répondre à cette question, Prolog va utiliser la seule règle qui permet de conclure à cette conclusion, à savoir la clause (4). Celle-ci peut se traduire, en lecture procédurale, par : *pour montrer qu'il va faire beau, il faut montrer que les prémisses* `temperature_elevee` *et* `anticylcone` *sont vérifiées.* Ainsi, pour montrer la conclusion, on doit démontrer les prémisses de la règle : deux sous-problèmes remplacent dont le problème originel. C'est la raison pour laquelle la conclusion est appelée le but à résoudre, et les prémisses les nouveaux sous-buts à démontrer. C'est pourquoi on dit que Prolog procède par décomposition de buts en sous-buts. On parle également de chaînage arrière, puisqu'on part du but final qui est à résoudre pour remontrer à des sous-buts, en espérant récursivement en arriver à des connaissances connues du programme (les faits).

L'application de la règle (4) remplace donc le but à résoudre {`beautemps`} par deux sous-buts : {`temperature_elevee, anticyclone`}.

Le premier sous-but est déjà démontré : la clause (2) est en effet un fait qui affirme que la température est élevée. Ce sous-but est donc résolu, et il ne nous reste qu'à résoudre le but {`anticyclone`} en procédant récursivement par décomposition de buts. La règle (3) permet précisément d'avancer dans la résolution du but {`anticyclone`}, puisque la tête de la clause correspond à ce but :

```
anticyclone :- pression_atmospherique_elevee.
```

Par chaînage arrière, le but {`anticyclone`} est donc remplacé par le nouveau sous-but {`pression_atmospherique_elevee`}. Il s'agit d'une connaissance du programme puisqu'il correspond au fait de la clause 1. Ce dernier sous-but est donc démontré, et l'intepréteur Prolog peut répondre positivement à la question. En résumé, on a donc :

> **Stratégie de résolution en chaînage arrière par résolution de buts.**
>
> *Pour résoudre un but Prolog* `B`*, on cherche une clause du programme qui a B comme conclusion. Deux cas de figure peuvent se présenter :*
>
> *- la but est démontré si la clause est un fait :* `B.`
>
> *- si c'est une règle de la forme* `B :- P1, …, Pn.` *il faut résoudre récursivement ces sous-buts. Le but à résoudre* `B` *est donc remplacé par l'ensemble des sous-buts* {`P1, …, Pn`}

Exemple – Dans notre exemple, on peut représenter la résolution du but `beautemps` par la succession des remplacements de sous-buts à résoudre avec le numéro de la clause Prolog ayant permis chaque transformation :

{`beautemps`} —(4)→ {`temperature_elevee, anticyclone`} —(2)→ {`anticyclone`} —(3)→ {`pression_atmospherique_elevee`} —(1)→ ∅

Cette représentation est appelée arbre de résolution Prolog. Nous allons en effet voir que le plus souvent, la résolution n'est pas directe et va laisser la place à l'apparition de branches dans l'arbre.

4.2. RECHERCHE EN PROFONDEUR D'ABORD PAR TENTATIVE : RETOUR-ARRIERE

Dans l'exemple précédent, l'application de la règle (4) nous a donné deux buts à résoudre : {`temperature_elevee, anticyclone`}. Nous avons ensuite appliqué la clause (2), qui était un fait, pour résoudre le sous-but `temperature_elevee`. Toutefois, nous aurions pu choisir de tenter de résoudre en premier lieu le but `anticyclone`. Ce qui, ci nous faisons le parallèle avec la logique des prédicats du 1er ordre, revient à choisir une littéral ou l'autre de la clause ¬`temperature_elevee` ∨ ¬`anticyclone` pour commencer la résolution. Lorsque nous faisons la résolution à la main, nous choisissons bien entendu l'ordre qui nous semble le plus approprié. Dans le cadre d'une résolution automatique telle que celle mise en œuvre par Prolog, il faut bien sûr se définir un ordre de traitement systématique, qu'il soit judicieux ou pas. On a ainsi la règle suivante :

> **Stratégie de résolution en profondeur d'abord.**
>
> *Lorsque la résolution d'un B fait appel à une règle comportant plusieurs sous-buts comme prémisses :*
>
> `B :- P1, …, Pn`

la stratégie résolution Prolog consiste à remplacer le but B par les sous-buts P1, ..,Pn et à chercher à résoudre complètement les sous-buts dans leur ordre d'apparition dans le corps de règle : on résout ainsi en premier le sous-but P1 avant de considérer seulement dans un second temps la résolution du sous-but P2 et ainsi de suite récursivement.

Cette stratégie de résolution est qualifiée de résolution en profondeur d'abord car la résolution de P1 peut entraîner la résolution de nouveau sous-but et ainsi de suite : on s'enfonce donc en profondeur dans la résolution de P1 avant de passer à la suite. Pour nous en persuader, nous allons considérer un exemple légèrement plus complexe, toujours en travaillant avec des prédicats d'arité 0 :

```
pression_atmo_hte.                      (clause 1)
temp_hte.                               (clause 2)
anticyclone :- isobare_hte.             (clause 3)
isobare_hte :- pression_atmo_hte.       (clause 4)
beautemps :- anticyclone, temp_hte.     (clause 5)
```

On observe que dans la clause (5), l'ordre des sous-buts a été changé dans le corps de la règle : le but `temp_hte`, qui se résoud directement, sera donc considéré après le but `anticyclone`. Posons nous une fois de plus la question : `?- beautemps.` On obtient l'arbre de résolution suivant :

```
{beautemps}
   |
   (5)
   ↓
{ anticyclone, temp_hte }          on va chercher à résoudre anticyclone
   |
   (3)
   ↓
{ isobare_hte, temp_hte }          on va chercher à résoudre isobare_hte
   |
   (4)
   ↓
{ pression_atmo_hte, temp_hte }          on va chercher à résoudre pression_atmo_hte
   |
   (1)
   ↓
{temp_hte }                              on cherche enfin à résoudre temp_hte
   |
   (2)
   ↓
   ∅    YES
```

On voit ici que l'on s'est enfoncé dans un premier temps dans la résolution du sous-but `anticyclone`, alors que `temp_hte` aurait pû être résolu de suite : d'où le terme de profondeur d'abord. Cette recherche en profondeur d'abord est bien sûr récursive : imaginons que le sous-but `anticyclone` ait été décomposé lui-même en deux sous-buts, nous nous serions enfoncé en premier lieur dans la résolution de ce sous-sous-but.

Dans les exemples que nous avons étudiés jusqu'ici, une seule règle pouvait permettre à chaque fois d'avancer dans la résolution d'un but. Dans un programme réel, ce cas est très rare : le plus souvent, Prolog doit choisir entre plusieurs règles pour résoudre un but : on dit qu'on est à un point de choix de la résolution. Pour ce faire, il choisit de manière systématique d'utiliser la première règle, quitte à revenir ensuite sur ce choix s'il n'est pas concluant. On parle de régime par tentative avec retour-arrière.

Régime de recherche par retour-arrière

Lorsque plusieurs clauses peuvent permettre a priori de résoudre un but (i.e. présentent ce but comme conclusion ou comme fait), la résolution arrive dans ce qu'on appelle un point de choix. Prolog choisit alors toujours la première clause du programme pour le résoudre en priorité.

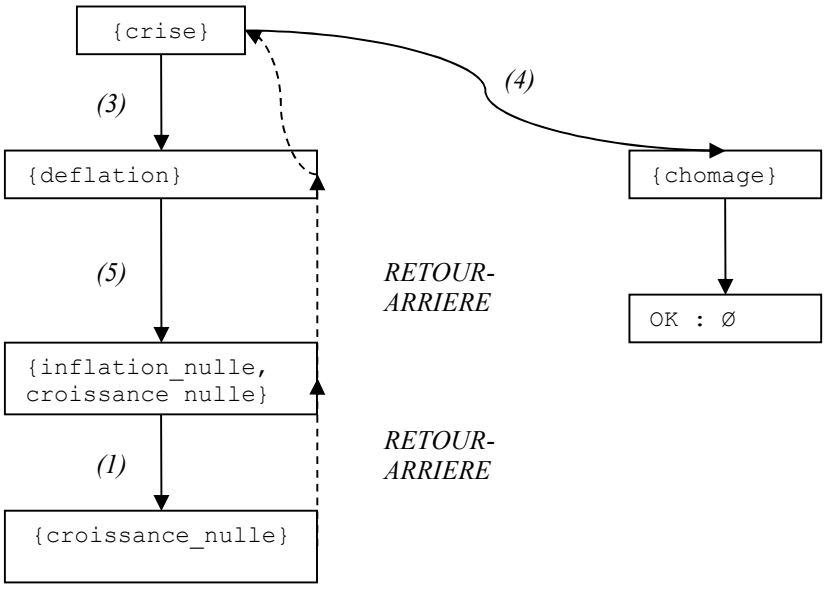
> *En cas d'échec à résoudre complètement le but en question, Prolog effectue un retour-arrière, c'est-à-dire qu'il annule ses dernières tentatives de résolution jusqu'à revenir à un point de choix : il poursuit alors la résolution avec la première règle de ce point de choix non encore essayée.*
>
> *Si le retour-arrière remonte jusqu'à la première tentative de résolution sans rencontrer de point de choix non encore totalement exploré, c'est qu'il n'y a pas de solution : la réponse de l'interprétateur à la question est négative.*

Cette règle nous permet ainsi de nous assurer d'explorer toutes les solutions possibles au problème, dans un ordre bien déterminé, qui n'est pas nécessairement optimal mais a l'avantage de la systématicité. Considérons un petit exemple pour étudier cette recherche par tentative avec retour-arrière. Une fois de plus, notre exemple ne comportera que des prédicats d'arité 0. Il se situe cette fois dans le domaline de la macro-économie :

```
inflation_nulle.                                  (clause 1)
chomage.                                          (clause 2)
crise :- deflation.                               (clause 3)
crise :- chomage.                                 (clause 4)
deflation :- inflation_nulle, croissance_nulle.   (clause 5)
```

On cherche à savoir s'il y a crise, d'où la question : `?- crise.`

On obtient l'arbre de résolution Prolog ci-dessous.



Il y a point de choix dès le début de la résolution : pour résoudre le but `crise`, l'interpréteur Prolog a le choix entre utiliser la règle (3) et la règle (4). Il va d'abord explorer la règle (3), qui remplace le but par le sous-but `deflation`. Celui-ci ne peut être résolu que par la règle (5) qui donne lieu au remplacement du sous-but `deflation` par les buts {`inflation_nulle, croissance_nulle`}. Suivant sa stratégie de résolution en profondeur d'abord, Prolog cherche d'abord à résoudre `inflation_nulle`, ce qui est fait directement à l'aide du fait de la clause (1), mais il ne peut résoudre le but `croissance_nulle` : aucune règle ne l'a comme conclusion ou fait. L'interpréteur Prolog doit donc faire un retour arrière et dépiler tous les sous-buts déjà démontrés. Il remonte jusqu'au dernier point de choix, qui était en fait le premier rencontré et qui correspond à la résolution de `crise`. Une seconde clause peut être explorée : la règle (4), qui va conduire à la solution.

Remarque – Comme nous venons de la voir, le retour-arrière permet à Prolog d'envisager un nouveau chemin de résolution après que la stratégie de recherche par tentative et profondeur d'abord se soit fourvoyée dans une impasse. Il permet donc d'obtenir systématiquement une solution lorsqu'elle existe. Dans ces situations, le retour-arrière est automatique, jusqu'à ce que la résolution produise une réponse. Une fois cette solution obtenue, il est toutefois possible d'obliger Prolog à effectuer un retour-arrière sur commande : dans toutes les implémentations Prolog, il est en effet possible de demander à Prolog, via l'interface d'interrogation de l'interpréteur, de trouver une autre solution :

partant du point de succès où il s'était arrêté, l'interpréteur effectue alors un ou plusieurs retours-arrières pour explorer d'autres chemins de résolution pouvant, le cas échéant, aboutir à une solution alternative.

4.3. SYNTHESE : RETOUR SUR UN EXEMPLE LP1

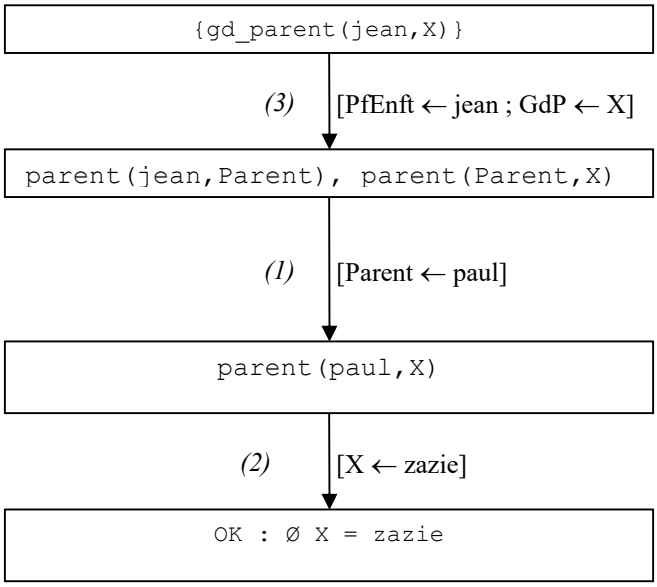
Nous venons d'étudier la stratégie de résolution du langage Prolog. Comment peut-elle comparée à la résolution en LP1 ? Pour répondre à cette question, revenons sur le programme Prolog étudié au paragraphe 3.7 et lequel nous avions donné une traduction et un exemple de résolution en logique des prédicats. Ce sera également l'occasion d'observer la mise en œuvre de la résolution Prolog sur un exemple avec des variables et constantes (prédicats d'arité strictement supérieure à 0). Rappelons le contenu de ce programme :

```
/********************** BASE DE FAITS ***************************/

parent(jean,paul).
parent(paul,zazie).

/********************** BASE DE REGLES ***************************/

gd_parent(PtEnft,GdP) :- parent(PtEnft,Parent), parent(Parent,GdP).
```

La question qui était posée au programme était alors :

?- `gd_parent(jean,X)`

Nous avions vu que la résolution LP1 pouvait prendre plusieurs chemins différents. Au contraire, Prolog impose des contraintes dans le choix des règles à utiliser (première règle non encore utilisée ayant pour tête le premier but à résoudre) et des buts à résoudre (profondeur d'abord sur le premier sous-but d'un corps de règle) qui fait qu'un seul chemin de résolution sera étudié. On obtient ainsi l'arbre de résolution Prolog ci-après, dans lequel les opérations d'unification doivent être précisées : une règle peut en effet servir à plusieurs occasions au cours de la résolution avec des valeurs de variables différentes, de même que ces unifications sont celles qui donneront les réponses à la question.



On observe que l'on retrouve un des chemins de la résolution LP1 étudiée dans le paragraphe 3.7 et figuré en gras sur la figure de la résolution LP1 correspondante. Les autres chemins ne seront pas explorés par Prolog : ils correspondent à chaque fois à l'application, dans un ordre différent de celui imposé par la stratégie de résolution Prolog, de même clauses utiles à cette résolution.

4.4. RESOLUTION PROLOG ET PROBLEME DE TERMINAISON : RECURSIVITE

Hors programme L1

5. CONCLUSION

Dans ce chapitre, nous aurons ainsi vu un premier exemple pratique d'application de la logique au domaine de l'informatique : la programmation logique. A la fin des années 1970, cette forme de programmation déclarative fut promue comme base d'une informatique du futur (on parlait alors d'informatique de IVe génération) qui ferait la part belle à l'Intelligence Artificielle. C'est-à-dire une informatique de la connaissance et du raisonnement où les systèmes informatiques atteindraient une certaine forme d'intelligence à laquelle est reliée le mythe de célèbre test de Türing. L'histoire et les intérêts économiques liés au développement d'Internet, en particulier, en ont décidé autrement.

La programmation logique est ainsi restée confidentielle, même si elle est encore utilisée fréquemment dans les recherches en Intelligence Artificielle et en Traitement Automatiques des Langues. Cela reste également, à la marge, un langage de maquettage apprécié dans certains projets industriels complexes. Il n'en reste pas moins que l'index réalisé par le communauté de programmeurs TIOBE classait Prolog seulement comme le 32e langage de programmation le plus utilisé fin 2013 : Prolog était utilisé dans seulement 0,3% des projets informatiques, qu'ils soient industriels ou de recherche.

Au cours de notre découverte du langage Prolog, nous avons fait quelques exercices et travaux pratiques relevant de ce que l'on appelle en programmation logique la programmation de type « base de données ». Si Prolog n'est pas utilisé pour réaliser et interroger des bases de données relationnelles, il n'en reste pas moins que le modèle relationnel de bases de données s'interprète sans problème d'un point de vue logique. Certains langages de requête déductifs pour bases de données, tels Datalog, font ainsi directement appel à la formalisation logique : Datalog est ainsi un pur sous-langage du langage Prolog. C'est cette seconde application de la logique en informatique que nous allons étudier dans le chapitre suivant.

Programmation logique et Prolog pur - Exercices & Problèmes

EXERCICES

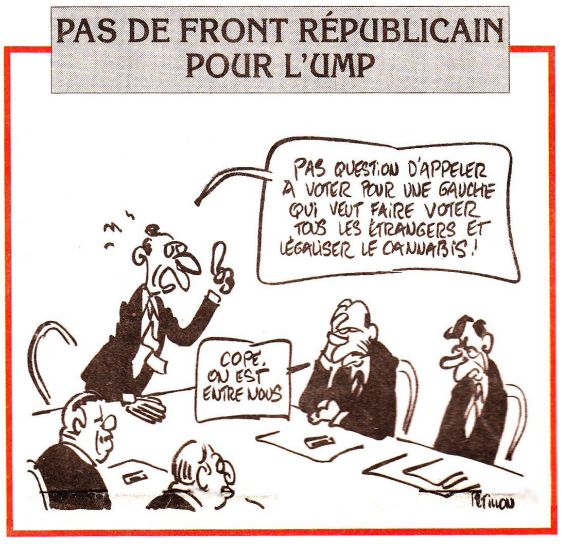
Exercice 4.1. — Bug à tous les étages : syntaxe Prolog (objectif 4.1.2.)

Enseignant moyennement doué, Ali Dentic a placé sur l'ENT un squelette de programme correspondant à son TP sur les cursus d'informatique réalisables à Blois. Or, ce programme ne respecte pas toujours la syntaxe du langage Prolog… Pouvez-vous l'aider à détecter ses erreurs de syntaxe (et non pas de sémantique !) avant que ses étudiants ne les découvrent ? Il est demandé de donner à chaque fois l'erreur correspondante et sa correction.

```
/*************************************************************/
/* FICHIER : di.pl */
/* AUTEUR : Dentic Ali */
/*************************************************************/

/*-- suite(Avt,Apr) vrai si on peut suivre le cursus Apr juste apres Avt -----*/

suite(bac,l1_info1), suite(l1_info,l2_info), suite(l2_info,l3_info).
suite (bac,iut_info1).
suite(iut_info1,dut_info).

/*-- cursus(Deb,Fin) vrai s'il existe un cursus entre Deb et Fin ------------*/

cursus(Deb,Fin) :- suite(Deb,Fin)
cursus(Deb,Fin) :- suite(Deb,Entre),
suite(Entre,Fin).
cursus(Deb,Fin) :- cursus(Deb,Entre,Fin).
```

Exercice 4.2. — Instruction civique européenne : syntaxe Prolog (objectif 4.1.1 & 4.1.2.)

Des pays comme l'Irlande, le Royaume-Uni et, plus récemment, les pays scandinaves ont accordé le droit de vote aux élections locales à leurs résidents étrangers sans qu'aucune catastrophe ne s'ensuive. A l'opposé, la classe politique française rechigne toujours à aborder cette question.



Pourtant, sur cette question, les hommes politiques de tous bords jouent les tartuffes et se moquent de nous. Ils savent en effet que, suite aux accords de Maastricht, les ressortissants de l'Union Européenne dispose du droit de vote aux élections locales en France, s'ils y résident. Et qu'ils ne s'en privent pas !

On considère un petit programme Prolog illustrant cette situation :

```
francais(dupond).
hollandais(gert).
europeen(X) :- francais(X).
europeen(X) :- hollandais(X).
habite(gert,france).
vote(X) :- francais(X).
vote(X) :- europeen(X), habite(X,france).
```

1. Donnez la traduction en logique des prédicats du 1er ordre de la question Prolog : ?- vote(X).

2. Donnez l'ensemble des clauses de Horn correspondant à la traduction en logique des prédicats du 1er ordre du programme et de la question.

3. Donnez **toutes** les réponses obtenues par application de la méthode de résolution sur cet ensemble de clauses de Horn. Pour chaque solution, on donnera le nombre de démonstrations obtenues.

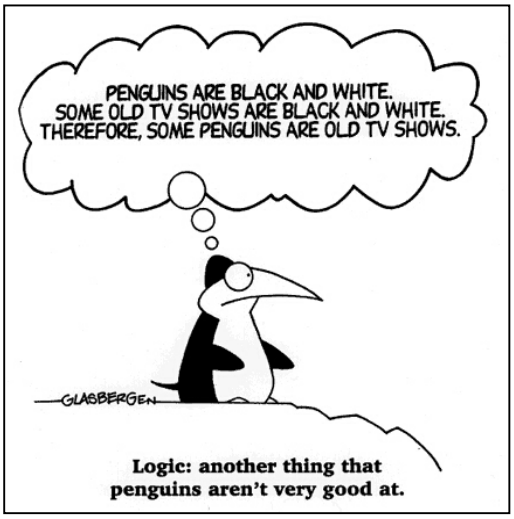
Exercice 4.3. — Le retour d'Aristote : Prolog et Horn (objectifs 4.1.1, 4.1.2. & 3.2.6.)

Au tout début du cours, nous avons étudié quelques exemples de syllogismes, formes de raisonnement étudiés par Aristote, le fondateur de la logique.

Dans cet exercice, nous allons étudier le syllogisme aristotélicien suivant :

Les pingouins sont des oiseaux
Les oiseaux sont des animaux

Donc il existe des pingouins qui sont des animaux



On veut vérifier expérimentalement la validité de ce raisonnement en utilisant un interpréteur Prolog.

1 — A quoi correspondront les hypothèses du raisonnement en Prolog ?

2 — A quoi correspondra la conclusion et comment vérifierons-nous la validité du raisonnement ?

3 — Représenter en Prolog le raisonnement, puis donnez les clauses de Horn correspondantes.

4 — Vérifiez alors la validité de ce raisonnement en appliquant la méthode de résolution.

Complément — Si vous avez déjà étudié la sémantique opérationnelle de Prolog, comparez cette résolution avec la résolution Prolog équivalente.

Exercice 4.4. — Paradis glacés : sémantique opérationnelle (objectifs 4.1.3, 4.2.4.)

Si les îles sont le plus souvent synonymes de soleil, cocotiers et vahinés dans l'esprit des vacanciers occidentaux, il en est qui, balayées par les Quarantièmes Rugissants ou situées aux confins de l'Arctique, n'ont pour elles que leur solitude et leur caractère grandiose. On considère le programme Prolog suivant, correspondant à une mini base de données issue d'un SIG (système d'information géographique) consacré aux îles des régions arctiques :

```
/* clause 1 */   ile(crozet,france,indien).
/* clause 2 */   ile(georgie_sud,royaume_uni,atlantique_sud).
/* clause 3 */   ile(francois_joseph,russie,arctique).
/* clause 4 */   ile(ellesemere,canada,artique).
/* clause 5 */   hemisphere(indien,sud).
/* clause 6 */   hemisphere(atlantique_sud,sud).
/* clause 7 */   hemisphere(arctique,nord).
/* clause 8 */   possede(Pays,Hemis):- ile(_,Pays,Ocean),hemisphere(Ocean,Hemis).
```

1 — Donnez l'arbre de résolution correspondant à la question : `?- possede(france,sud).`
2 — Donnez l'arbre de résolution correspondant à la question : `?- possede(russie,sud).`
3 — Donnez l'arbre de résolution correspondant à la question : `?- possede  (X,nord).`

Exercice 4.5. — Roland Garros : sémantique opérationnelle (objectifs 4.1.3 & 4.2.4.)

On considère le programme Prolog suivant :

```
/* clause 1 */    soleil.
/* clause 2 */    lobes(sharapova).
/* clause 3 */    petit(safina).
```

```
/* clause 4 */     meilleur(williams).
/* clause 5 */     gagnant(X,Y) :- meilleur(X).
/* clause 6 */     gagnant(X,Y) :- plus_malin(X,Y).
/* clause 7 */     gagnant(X,Y) :- plus_chanceux(X,Y).
/* clause 8 */     plus_malin(X,_Y) :- soleil, lobes(X).
/* clause 9 */     plus_malin(X,Y) :- petit(Y), lobes(X).
/* clause 10*/     plus_chanceux(_X,safina).
```

1 — Construire l'arbre de résolution Prolog correspond à la question `?- gagnant(X,_Y).`
2 — Quelle(s) réponse(s) obtient-on ?

Réponse indicative — On obtient 4 solutions possibles avec le même gagnant dans deux cas de figure.

Exercice 4.6. — Dès que le vent soufflera : sémantique opérationnelle (objectifs 4.1.3 & 4.2.4.)

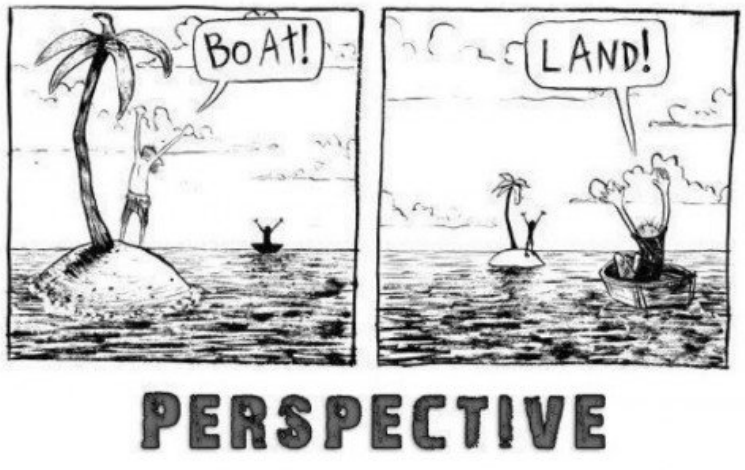
On considère le programme Prolog suivant :

```
/* clause 1 */       navigue(parlier,mardi).
/* clause 2 */       navigue(desjoyeux,mercredi).
/* clause 3 */       navigue(desjoyeux,jeudi).
/* clause 4 */       navigue(ledam,J).
/* clause 5 */       temps(mardi,calme).
/* clause 6 */       temps(mercredi,calme).
/* clause 7 */       temps(jeudi,grostemps).
/* clause 8 */       aime(ledam,grostemps).
/* clause 9 */       aime(X,calme).
/* clause 10*/       chance(desjoyeux,mercredi).
/* clause 11*/       gagne(X,J) :- navigue(X,J), temps(J,T), aime(X,T).
/* clause 12*/       gagne(X,J) :- navigue(X,J), chance(X,J).
```

Question — Donnez **les** réponses données au but :

```
?- gagne(X,mercredi)
```

Donnez l'arbre de résolution correspondant.



Réponse indicative — On obtient 2 gagnants possibles le mercredi ... mais trois démonstrations au total.

Exercice 4.7. — Cluedo logique : programmation de base (objectifs 4.2.2 & 4.2.3.)

Soit c*olonelMoutarde* et *mademoiselleRose* des *suspects*, *chandelier* un *objet contondant,* et le *bureau* et la *bibliothèque* une *salle*. Les empreintes de doigts appartenant au *colonelMoutarde* ont été trouvés dans le *bureau*, et celles de *mademoiselleRose* dans la *bibliothèque*. Un *cadavre* a été découvert dans le *bureau*. Le *chandelier* se trouve dans le *bureau*.

1. En choisissant des prédicats adéquats, représentez ces connaissances par des clauses Prolog.
2. Donner une clause Prolog utilisant les prédicats coupable/1, condamne/1 qui correspondent à l'assertion «*si un suspect est coupable d'un meurtre, alors il doit être condamné* »

3 – Donner une règle Prolog qui correspond à l'assertion « *si un cadavre a été découvert dans une salle et qu'on découvre les empreintes d'un suspect dans cette salle, alors le suspect est coupable* »

4 – Donner une règle Prolog qui correspond à l'assertion « *si un cadavre a été trouvé dans une salle et si un objet (contondant) se trouve dans cette salle, il s'agit indubitablement de l'objet qui a permis de commettre cet acte délictueux* »

5 – Donnez les arbres de dérivation Prolog qui correspondent aux questions « *Mademoiselle Rose est-elle coupable* ?» et «*Le Colonel Moutarde est-il coupable et quel objet a-t-il utilisé pour réaliser son forfait* ?».

Exercice 4.8. — BD documentaire : programmation de base (objectifs 4.2.2.)

Une bibliothèque possède un certain nombre d'ouvrages, tous écrits par des auteurs de l'absurde et classés par catégorie d'œuvre (roman, essai, theâtre) :

Type d'oeuvre	Titre	Auteur
roman	La Peste	Camus
essai	Le mythe de Sisyphe	Camus
théâtre	Rhinocéros	Ionesco
théâtre	En attendant Godot	Beckett
théâtre	Caligula	Camus

On désire définir une base de données qui intègre ces informations, pour ensuite pouvoir interroger la base à l'aide de requêtes automatiques. Par exemple, on veut pouvoir avoir la liste de tous les ouvrages écrits par Albert Camus.

1 —Représenter les informations ci-dessous (par exemple : Godot est une pièce de théâtre dont l'auteur est Beckett) sous la forme de faits Prolog. On cherchera à adopter une représentation la plus générique possible, pour utilisation ensuite dans un programme Prolog.

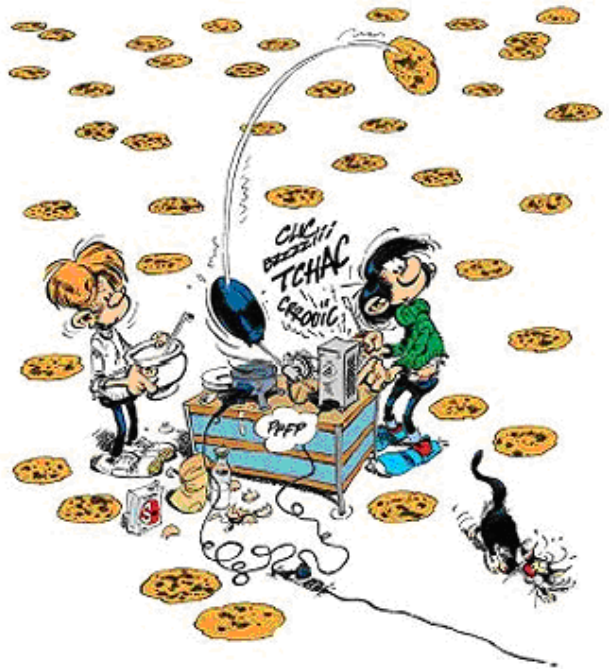
2 — Ecrire un prédicat `partout(Auteur)` qui réussit si la bibliothèque dispose d'œuvres de l'auteur `Auteur` dans tous les types d'ouvrages.

3 — Ecrire un prédicat `plusieurs(Auteur,Type)` qui réussit si la bibliothèque possède plusieurs œuvres de l'auteur `Auteur` dans la même catégorie `Type`. On suppose qu'on dispose d'un prédicat `diff(X,Y)` prédéfini qui réussit si X et Y sont différents.

Exercice 4.9. — Gastronomie logique : programmation de base (objectifs 4.2.2.)

On désire écrire un petit programme Prolog qui aide une personne donnée à choisir un menu au restaurant. Sur la carte de l'établissement, sont précisées les informations suivantes :

Type	Nom	Région
vin	Cidre	Bretagne
vin	Saumur	Val de Loire
plat	Galette andouille	Bretagne
plat	Ravioles	Dauphiné
dessert	Kouign amann	Bretagne
dessert	Pogne	Dauphiné



1. **Représentation des connaissances** — Modéliser les informations ci-dessus par des faits Prolog.
2. Chaque met est typique d'une région. Représenter les informations correspondantes en utilisant un prédicat d'arité 2 `lieu(Elt,Region)` qui réussit si `Elt` provient de `Region`.
3. Ecrire un prédicat d'arité 4 `menu_reg(Vin,Plat,Dessert,Region)` qui réussit si le vin, le plat et le dessert considérés composent un menu typique de la `Region`.
4. Fanch est breton. Traduire cette information à l'aide du prédicat `lieu(Elt,Region)`.

   Ecrire un prédicat d'arité 4 `menu_habituel(Vin,Plat,Dessert,Personne)` qui réussit si le vin, le plat et le dessert composent un menu correspondant à la région d'où est originaire Personne.
5. Par quel but obtiendrez-vous le menu habituel qui sied à Fanch ?

Exercice 4.10. — Tous à Carouf ! programmation de base (objectifs 4.2.2.)

On désire réaliser un programme Prolog qui modélise une base de données permettant de gérer le stock d'un supermarché. Dans cette base de données, chaque produit référencé est décrit par son nom, le rayon où il se trouve, son prix unitaire et la quantité (en nombre d'unités) présente en stock. Le tableau ci-après présente un extrait de cette BD.

Nom	**Rayon**	**Prix (Euros)**	**Stock**
Marronsuisse Danone	Produits laitiers	1,6	0
BN Fraises	Biscuits	0,6	45
Huile olive Puget	Condiments	3,5	20

On suppose que l'on dispose du prédicat prédéfini `between/3` défini comme suit : `between(Min,Val,Max)` réussit si `Val` est un entier ou un réel compris au sens large entre les valeurs `Min` et `Max`. Ses arguments doivent être instanciés par des valeurs lors de l'appel.

1. **Représentation des connaissances –** Comment représenterez-vous ces connaissances dans votre programme Prolog ? La représentation choisie devra vous permettre de répondre aisément aux problèmes ci-dessous.
2. **Gestion de stock –** Donnez l'écriture Prolog du prédicat `rupture/2` défini comme suit : `rupture(Prod,Rayon)` réussit si `Prod` est un produit présent normalement dans le rayon `Rayon`

qui est actuellement en rupture de stock. Ce prédicat devra fonctionner dans tous les modes d'utilisation possibles.

3. **Aide au client –** Donnez l'écriture Prolog du prédicat choix/3 défini comme suit : `choix(Rayon,Px_Max, Prod)` réussit si `Prod` est un produit présent dans le rayon `Rayon` et dont le prix est inférieur à `Px_Max`. Ce prédicat devra fonctionner dans les modes où l'on donne le rayon et le prix maximal recherché.

Exercice 4.11. — Retour au port : récursivité (objectifs 4.1.4, 4.2.3. & 4.2.5.)

On désire réaliser un programme Prolog qui recherche, à partir de la donnée de relations maritimes directes (i.e. sans escales), l'ensemble des voyages possibles que l'on peut effectuer en bateau entre deux ports. Ce programme n'utilise qu'un seul prédicat d'arité 2 : `liaison(dep,arr)` qui réussit s'il existe une liaison maritime entre le port de départ dep et le port d'arrivée arr. Dans l'écriture proposée du programme, les relations directes sont alors représentées par un ensemble de faits, alors que les liaisons avec escale sont obtenues à l'aide d'une règle récursive :

```
/* clause 1 */      liaison(brest,le_conquet).
/* clause 2 */      liaison(le_conquet,molene).
/* clause 3 */      liaison(molene,ouessant).
/* clause 4 */      liaison(santander,brest).
/* clause 5 */      liaison(brest,plymouth).
/* clause 6 */      liaison(Dep,Arr) :- liaision(Dep,Esc), liaison(Esc,Arr).
```

1 — Donnez l'arbre de résolution Prolog correspondant à la question `?-liaison(brest,ouessant).`

2 — Donnez l'arbre de résolution Prolog correspondant à la question `?-liaison(brest,santander).` Quel problème rencontre-t-on ?

3 — Modifiez le programme pour résoudre le problème rencontré.

Exercice 4.12. — Récursivité (objectifs 4.1.4, 4.2.3. & 4.2.5.)

On considère le programme Prolog suivant, définissant une relation d'ordre entre nombres entiers :

```
/* clause 1 */      suivant(1,0).
/* clause 2 */      suivant(2,1).
/* clause 3 */      suivant(3,2).
/* clause 4 */      sup(Sup,Inf) :- suivant(Sup,Inf).
/* clause 5 */      sup(Sup,Inf) :- sup(Sup,Entre), sup(Entre,Inf).
```

1. Donnez l'arbre de résolution correspondant à la question `?- sup(0,3).`
2. Donnez l'arbre de résolution correspondant à la question `?- sup(3,0).`
3. Quel problème observez-vous ? Saurez-vous le résoudre en modifiant le programme ?

Exercice 4.13. — Les amis de mes amis : récursivité (objectifs 4.1.4, 4.2.3. & 4.2.5.)

On considère le programme suivant :

```
/* clause 1 */      ami(paul, pierre).
/* clause 2 */      ami(pierre,jacques).
/* clause 3 */      ami(X,Y) :- ami(X,Z), ami(Z,Y).
```

1 — Comment peut s'interpréter en français la règle (3) ? En quoi cette règle est-elle particulière ?

2 — Donnez les arbres de résolution correspondant à la question `?- ami(paul,jacques)`, puis à la question `?- ami(paul,unautre).` Qu'observez-vous ?

3 — Sauriez-vous modifier le programme pour contourner le problème rencontrer tout en gardant la sémantique attendue du programme.

Problèmes

Problème 4.1 — Vive le roi ! Récursivité en Prolog et LP1 (objectifs 4.2.3. & 4.2.5.)

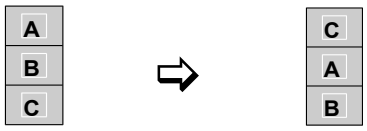
On considère le programme Prolog suivant :

```
avant(louis13,louis14).
avant(louis14,louis15).
avant(louis15,louis16).
ancetre(X,Y) :- ancetre(X,Z), avant(Z,Y).
ancetre(X,Y) :- avant(X,Y).
```

1. Donnez l'arbre de résolution correspondant à la question : `?- ancetre(louis14,louis16).` Qu'observez-vous ?
2. Pouvez-vous modifier le programme pour résoudre le problème observé ? Redonnez l'arbre d'analyse correspondant.
3. Considérons à nouveau le programme initial. Donnez-en une traduction en LP1.
4. Donnez alors l'arbre de résolution correspondant à la même question en LP1. Le problème est-il encore présent ?

Problème 4.2 — Intelligence Artificielle : le monde des blocs (objectif 4.2.5)

Se cherchant de grands défis intellectuels à solutionner, l'Intelligence Artificielle, à ses débuts dans les années 60, s'est passionnée pour ce qu'on appelait le *monde des blocs*. Il s'agissait en fait de résoudre un problème enfantin à la portée de tout nourrisson : déplacer, à l'aide des robots, des cubes empilés dans une configuration donnée pour les replacer dans une autre configuration, ceci en ne prenant à chaque fois qu'un seul cube. Par exemple :



La solution apparut rapidement, et ce grand classique trouva sa place dans tous les enseignements de base en Intelligence Artificielle. Le problème est d'ailleurs assez simple et peut être résolu en Prolog par un étudiant de 1er cycle.

Nous allons étudier ici une toute petite partie du problème. On représente tout empilement à l'aide d'un ensemble de faits `pose_sur(X,Y)`, prédicat qui réussit si X est posé directement Y. Ainsi, la pile à gauche de la figure ci-dessus se décrit comme suit :

```
pose_sur(c,sol).
pose_sur(b,c).
pose_sur(a,b).
```

Pour estimer si une position est correcte ou non, il peut être utile au robot de savoir d'une manière générale si un bloc donné est *au dessus* d'un autre, sans que ceux-ci soient posés l'un sur l'autre. Par exemple, sur la figure de gauche, A est au dessus de B, mais aussi de C et du sol.

Question — On demande d'écrire le prédicat `dessus(X,Y)` qui réussit sur X est au-dessus de Y.

Problème 4.3 — Recherche de diviseur … sans division (objectif 4.2.2 & 4.2.5)

Pour l'ordinateur (et l'être humain !), la division est une opération plus difficile à réaliser qu'une simple addition ou soustraction. C'est ainsi qu'il n'est pas défini en Prolog de prédicat logique réalisant la division (cf. chapitre consacré à l'artithmétique Prolog et ses prédicat extra-logiques). Pour le programmeur désireux de n'utiliser qu'un langage Prolog pur, c'est à dire n'utilisant que des prédicats rigoureusement

logiques, comment répondre alors au problème suivant : réaliser un prédicat d'arité `diviseur(N,Div)` qui réussit si l'entier `Div` est un diviseur de l'entier N.

La solution a ce problème est heureusement trouvée depuis l'antiquité : il suffiser de constater que si B est un diviseur de A, alors B est aussi un diviseur de B-A. Partant de cette propriété, saurez-vous réaliser ce prédicat ?

Indication — On pourra pour utliser le prédicat logique `plus/3` défini comme suit : `plus(X,Y,Z)` réussit si Z est la somme de X et Y entiers relatifs.

Logique et bases de données

OBJECTIFS LIES AU CHAPITRE 5 – LOGIQUE ET BASES DE DONNEES

Savoirs : notions théoriques

5.1.1. Notions de bases de théorie des ensembles : union, intersection, produit cartésien
5.1.2. Modèle relationnel : domaine, relation, attribut, tuple
5.1.3. Opérateurs de base de l'algèbre relationnelle : sélection, projection et jointure
5.1.4. Langage de requête SQL : syntaxe de base
5.1.4. Calcul conjonctif : principes, lien avec la logique des prédicats et propriétés

Savoirs faire : pratiques

5.2.1. Modélisation de base : savoir définir une base de données relationnelle sans redondances
5.2.2. Savoir représenter sous forme logique une base de données relationnelle
5.2.3. Savoir exprimer une requête simple en algèbre relationnelle
5.2.4. Savoir exprimer une requête simple en langage SQL
5.2.5. Savoir exprimer une requête simple à l'aide de requêtes conjonctives
5.2.6. Savoir exprimer une base de données et des requêtes simples en Prolog

1. INTRODUCTION : LOGIQUE ET BASES DE DONNEES

Au cours du chapitre précédent, nous avons étudié une forme de programmation déclarative qui repose directement sur l'utilisation de la logique des prédicats du 1er ordre et de sa sémantique de déduction. Par essence, la logique est adaptée à ce type de programmation qui permet de spécifier le résultat à obtenir sans décrire les opérations que l'ordinateur aura à mener pour y arriver. Les langages assertionnels sont un sous-ensemble de langages déclaratifs, parmi lesquels se trouve SQL, un langage normalisé (norme ISO) qui est utilisé par interroger les bases de données relationnels. Si certains systèmes de gestion de bases de données relationnels (SGBD-R) autorisent la définition graphique de requêtes d'interrogations plus ou moins complexes (voir QBE dans Microsoft Access ou Libre Office Base), tous implémentent le langage de requête SQL.

L'écriture de requêtes en langage SQL ne fait pas appel directement à la logique des prédicats. On peut toutefois lui trouver des équivalents formels reposant sur ce système de représentation : il s'agit du calcul relationnel des tuples et du calcul relationnel des domaines. Nous allons étudier dans ce chapitre ce second calcul, ce qui nous permettra de faire le lien entre bases de données et logique. Nous verrons au passage que les éléments de base du modèle relationnel utilisé pour modéliser les bases de données peuvent être représentés en Prolog.

La logique est également utile dans des modèles plus avancés de base de données tels que les bases de données déductives. Ces bases de données avancées ne se contentent pas de représenter de manière structurée un ensemble d'informations : comme le suggèrent leur nom, elles permettent également de déduire de nouvelles connaissances à partir des connaissances contenues par la base de données. L'interrogation dans les bases de données déductives repose sur le langage Datalog qui, comme son nom le suggère, fait directement appel à la puissance de déduction de Prolog, puisqu'il en constitue un sous-langage. Si les bases de données déductives n'ont pas rencontré l'intérêt commercial que l'on pouvait espérer, les possibilités d'exploration récursive de données du langage semblent lui ouvrir de nouvelles possibilités d'application dans des domaines émergents tels que l'intégration de données ou l'informatique dans les nuages (*cloud computing*)[2].

D'une manière plus générale, le recours à la logique en informatique est de mise dès lors que l'on a à se poser un problème de représentation des connaissances. Ainsi, le développement récent du Web sémantique sous forme d'utilisation massive d'ontologies structurées d'envergure, à l'aide du langage de représentation OWL en particulier, reposent directement sur les logiques de description. OWL définit plusieurs sous-langages plus ou moins expressifs (et donc plus ou moins décidables…) dont la sémantique peut-être exprimée en logique des prédicats du 1er ordre. Dans ce chapitre, nous nous en tiendrons à l'étude d'une formalisation logique des bases de données et de leur interrogation.

2. BASES DE DONNEES RELATIONNELLES : UNE PREMIERE IDEE

Le concept de bases de données est incontournable lorsqu'on s'intéresse à la notion de système d'information permettant de regrouper de manière un ensemble d'informations structurées de manière à faciliter leur interrogation indépendamment des applications qui seront amenées à les utiliser. Avec l'émergence d'Internet et de la mise en ligne de masses de données, elles constituent un support essentiel du développement actuel de l'informatique même si d'autres formes d'organisation des connaissances (voir le Web sémantique dont nous avons déjà causé) y participent également.

La question de l'organisation de grosses masses de données s'est posée dès les années 60, où on ne pouvait stocker l'information que sous la forme de systèmes de gestion de fichiers hiérarchiques équivalents à la façon dont vous structurez vos répertoires et sous-répertoires personnels sur votre ordinateur. Il a fallu attendre 1970 et la proposition par Codd d'un modèle formel reposant sur ce que l'on appelle l'algèbre relationnelle pour disposer d'une solution efficace pour structurer et interroger un masse d'informations complexes. Le modèle relationnel de code a donné rapidement lieu à l'émergence des systèmes de gestion de bases de données relationnels (Oracle, MySQL….) qui sont toujours à l'heure actuelle incontournables en matière de bases de données. Ce chapitre n'a pas pour objectif d'offrir une étude exhaustive du modèle relationnel. Nous allons toutefois tenter d'en donner une idée assez intuitive dans ce paragraphe, à l'aide d'un petit exemple.

[2] Huang S., S. (2011) Datalog and Emerging Applications*: An Interactive Tutorial*.**,** Actes *SIGMOD*'11, Athènes, Grèce

L'emergence des bases de données relationnelles : un point de vue historique

A FAIRE…

1970 : modèle relationnel de données par Codd

1976 : modèle entité-relation pour la modélisation des bases de données

1976 : premiers SGBD-R opérationnels (System-R, INGRES)

1980 : apparition et développement des SGBD-R commerciaux

Repères bibliographiques historiques

- Astrahan M et al. (1976) System-R : a relational approach tu database management, *ACM TODS*, **1**(2).
- Chen P. (1976) The entity-relationship data model : toward a unified view of data. *ACM-TODS*, **1**(1).
- Codd E.F. (1970) A relational model a data for large shared databanks. *CACM*, **13**(6).
- Stonebraker M., Wong E., Kreps P., Held G. (1976) The design and implementation of INGRES. *ACM-TODS*, **1**(3).

Considérons une base de données dans laquelle on souhaite enregistrer les performances de joueurs de basket au fil des matches d'une saison donnée. Dans cette base de données, on souhaite faire figure le nom des joueurs et des informations les concernant (nom équipe, poste préférentiel dans l'équipe, nationalité) et pour chaque match le nombre de points qu'ils ont marqué. Chaque match étant défini par sa date et le nom de l'équipe rencontrée. Une façon très simple d'organiser ces informations est d'utiliser un tableur tel que Microsoft Excell ou Libre Office Calc pour stocker les données sous la forme d'un tableau à 8 colonnes présenté ci-dessous :

Prénom	Nom	Equipe	Poste	Pays	Date	Adversaire	Points
Tony	Parterre	Purs de St Antoine	Meneur	France	09/01/2013	Bouchers de Chicago	17
Tony	Parterre	Purs de St Antoine	Meneur	France	12/01/2013	Bretons de Boston	21
Boris	Babak	Purs de St Antoine	Ailier	France	09/01/2013	Bouchers de Chicago	9
Joakim	Fépeur	Bouchers de Chicago	Pivot	France	09/01/2013	Purs de St Antoine	9
Joakim	Fépeur	Boucher de Chicago	Pivot	France	11/01/2013	Os en gelée	29
Nikola	Tebask	Os en gelée	Ailier	Euskadi	11/01/2013	Bouchers de Chicago	4

Sous un tableur, il est possible d'exprimer des requêtes déjà évoluées sur le contenu de la table considérée. Toutefois, cette modélisation sous la forme d'un tableau unique revêt un grave défaut : l'information y est codée de manière très le nom de son équipe, son poste dans l'équipe et sa nationalité. Il en découle une lourdeur de représentation coûteuse en temps de saisie et en espace mémoire mobilisé, mais également en termes d'évolutivité future de la base de données.

En réfléchissant un peu, on observe que notre base de données stocke en fait deux informations différentes : d'une part, des informations sur les joueurs, qui ne vont pas bouger de toute la saison sportive en cours, et d'autre part sur les matchs qui vont se dérouler au cours de la saison. L'idée des bases de données relationnelles est précisément de fragmenter notre tableau en un ensemble de tables que l'on va mettre en correspondance à l'aide de relations explicites. Dans le cas présent, nous allons ainsi définir :

- une table `Joueur` qui comportera les informations `nom`, `prénom`, `équipe`, `poste`, `pays`,
- une table `Perf` qui comportera les informations sur les performances des joueurs sur chaque match, à savoir : `date`, `adveesaire` et `point`.

Bien entendu, la table `Perf` ainsi définie est incomplète, puisqu'on n'a aucune information sur le joueur qui a réalisé une performance donnée stockée dans le tableau. Pour cela, il nous faut définir un champ informatif supplémentaire, qui permet de manière unique de pointer sur l'élément

correspondant de la table `Joueur`. Ce champ joue le rôle d'identificateur unique, un peu à la manière de votre numéro de sécurité sociale. Nous allons donc ajouter une information `numero` dans la table `Joueur`, qui jouera ce rôle, et c'est cette même information qui sera reprise dans la table `Perf`. Ainsi, le contenu de la base de données est partagé en deux tables comme le montre le schéma ci-dessous.

JOUEUR					
Numéro	**Prénom**	**Nom**	**Equipe**	**Poste**	**Pays**
1	Tony	Parterre	Purs de St Antoine	Meneur	France
2	Joakim	Fépeur	Boucher de Chicago	Pivot	France
3	Nikola	Tebask	Os en gelée	Ailier	Euskadi
4	Boris	Babak	Purs de St Antoine	Ailier	France

PERF			
Joueur	**Date**	**Adversaire**	**Points**
1	09/01/2013	Bouchers de Chicago	17
1	12/01/2013	Bretons de Boston	21
4	09/01/2013	Bouchers de Chicago	9
2	09/01/2013	Purs de St Antoine	9
2	11/01/2013	Os en gelée	29
3	11/01/2013	Bouchers de Chicago	4

On note immédiatement que les redondances d'information observées sur la modélisation avec une seule table ont effectivement disparu. L'idée de séparer l'information entre plusieurs tables, et de les mettre en correspondance à l'aide de champs d'informations, appelés clés, qui permettent de définir de manière univoque un élément du problème (ici : un joueur) est au centre du modèle relationnel de structuration des bases de données.

La modélisation en deux tables telle que nous venons de la proposer n'est pas complètement satisfaisante au regard du modèle relationnel. Elle a simplement pour objectif de montrer l'apport de la séparation de l'information en plusieurs tables reliées entre elles par l'intermédiaire de champs d'information clés. L'objectif de cet ouvrage n'est pas de présenter le modèle relationnel dans son exhaustivité. Nous allons simplement donner quelques définitions formelles essentielles à la compréhension du modèle avant d'en revenir à ses liens avec la logique.

3. MODELE RELATIONNEL : FORMALISATION

3.1. QUELQUES NOTIONS ENSEMBLISTES DE BASE

Il existe plusieurs manières de formaliser la notion de bases de données. Dans la partie 4 de ce chapitre, nous en donnerons ainsi une interprétation logique qui est rigoureusement équivalent à celle que nous allons aborder ici, et qui est même la plus utilisée pour un type de base de données avancées que l'on appelle base de données déductive. Dans l'immédiat, nous allons partir d'une approche la plus naturelle pour définir ce qu'est une base de données, à savoir en donner une vue ensembliste. En effet, une base de données n'est rien d'autres que la réunion d'un ensemble d'éléments d'information. Les notions mathématiques de théorie des ensembles abordées dans l'enseignement secondaire sont désormais réduites à portion congrue. Dans ce paragraphe, nous allons donc aborder quelques notions vraiment basiques sur les ensembles qui vous seront utiles dans la suite de l'ouvrage.

Ensemble – Dans la vision qui nous intéressera ici, un ensemble est, de manière très intuitive une collection d'éléments d'un domaine donné. Les ensembles qui nous intéresserons comporteront un nombre fini d'éléments, alors que la théorie mathématique des ensembles peut envisager l'existence d'ensembles infinis (dénombrables ou pas). Les ensembles finis qui nous intéressent les bases de données peuvent bien sûr prendre leurs éléments dans des domaines infinis.

Relation d'appartenance – Les éléments d'un ensemble sont reliés à ce dernier par la relation d'appartenance $\in$, qui vous est connue depuis l'enseignement secondaire. Le fait qu'un élément x appartienne à l'ensemble S s'écrit ainsi classiquement : $x \in S$.

Définition et notation d'un ensemble – un ensemble peut être défini :

- **en extension** – Dans ce cas, on énumère entre accolades l'ensemble de ses éléments. Pour reprendre notre exemple lié au basket, on peut ainsi définir l'ensemble des prénoms de joueurs par : {Tony, Joakim, Nikola, Boris}. On parle ici de définition en extension de l'ensemble.
- **en intension (ou compréhension)** – Dans ce cas, on ne liste pas les éléments de l'ensemble mais on donne une propriété qu'ils doivent définir de manière nécessaire et suffisante pour appartenir à l'ensemble. Par exemple, l'ensemble $\mathbb{R}^+$ (infini…) des nombres réels positifs pourra être défini comme suit : $\{x \mid x \in \mathbb{R} \wedge x > 0\}$.

Remarque **–** On peut noter que ces deux types de définition peuvent être représentées directement en logique des prédicats du 1er ordre (et donc en Prolog) à l'aide d'un prédicat qui exprime la relation d'appartenance à l'ensemble. Ce prédicat peut lui aussi être défini en intension ou en extension :

- La définition en extension revient à établir une liste de faits introduisant chaque élément de l'ensemble. Par exemple : `Element_de(Joueur, Boris)` en logique des prédicats.
- La définition en intension revient à définir le prédicat par une relation d'équivalence. Pour reprendre l'exemple ci-dessus, on a : $\forall x$ (`Element_de(E, x)` $\Leftrightarrow$ `Reel(x)` $\wedge$ `Sup(x,0))`

Enfin, le symbole $\varnothing$ désigne l'ensemble vide, i.e. ne comportant aucun élément. Sa définition est autant intensionnelle qu'extensionnelle.

Cardinalité d'un ensemble – On appelle cardinalité d'un ensemble le nombre de ses éléments. Elle est notée de la même manière que la valeur absolue. Par exemple :

$|\varnothing| = 0$

$|\{\text{Tony, Joakim, Nikola, Boris}\}| = 4$

Une fois ces objets mathématiques définis, il est possible de travailler sur eux pour les comparer, ou réaliser des opérations ensemblistes.

Comparateurs ensemblistes – Les comparaisons ensemblistes de base sont connues dès le secondaire. Nous nous contentons ici de donner leur traduction logique. Soient deux ensembles E et F, on peut définir :

- *Egalité* $E = F$ deux ensembles sont égaux s'ils ont les mêmes éléments :
  Egal`(E,F)` $\equiv \forall$`x (Element_de(E,x)` $\Leftrightarrow$ `Element_de(F,x)`
- *Inclusion* $E \subset F$ E est inclus dans F si tout élément de E est un élément de F
  Inclus`(E,F)` $\equiv \forall$`x (Element_de(E,e)` $\Rightarrow$ `Element_de(F,f))`

Lorsqu'un ensemble E est inclus dans un ensemble F, on peut dire que E est un **sous-ensemble** de F. Par exemple, E = {1,2} est un sous-ensemble de F = {1,2,3} mais aussi de G = {1,2} puisqu'un ensemble est bien inclus dans lui-même par définition.

Opérations ensemblistes – Les opérateurs ensemblistes de base sont connus de même dès le secondaire. Nous nous contentons ici de donner leur traduction logique. Soient deux ensembles E et F, on peut définir :

- *Intersection* $E \cap F$ L'intersection de deux ensembles E et F est l'ensemble constitué des éléments appartenant à la fois à E et F.
  $\forall$`x (Element_de(`$E \cap F$`,x)` $\Leftrightarrow$ `Element_de(E,x)` $\wedge$ `Element_de(F,x))`
- *Union* $E \cup F$ L'union de deux ensembles E et F est l'ensemble constitué des éléments appartenant à E ou F.
  $\forall$`x (Element_de(`$E \cup F$`,x)` $\Leftrightarrow$ `Element_de(E,x)` $\vee$ `Element_de(F,x))`

Deux autres opérations sont fréquemment utilisées pour l'étude et la modélisation des bases de données. La première est au centre du modèle relationnel : il s'agit du produit cartésien.

> **Produit cartésien –** Soit deux ensembles E et F. Le produit cartésien de E et F, noté $E \times F$, est l'ensemble des couples (e,f) formés à partir de tous les éléments e de l'ensemble E et f de l'ensemble F : $\{ (e, f) \mid (e \in E) \wedge (f \in F) \}$. Soit d'un point de vue logique :
>
> $\forall e \forall$`f (Element_de(`$E \times F$`,(e,f))` $\Leftrightarrow$ `Element_de(E,e)` $\wedge$ `Element_de(F,f))`

Exemple – Soit E = {1, 2,3} et F = {Tony, Boris} alors on a

$$E \times F = \{ (1,Tony), (1, Boris), (2, Tony), (2,Boris), (3, Tony), (3,Boris) \}$$

On définit par ailleurs l'ensemble des parties comme suit :

Ensemble des parties – Soit un ensemble E. L'ensemble des parties de E, noté P(E), est l'ensemble de tous les sous-ensembles de E. $P(E) = \{ S \mid S \subset E\}$

Exemple – Soit E = {1, 2,3} . Alors on a $P(E) = \{\emptyset , \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\} \}$

Partition d'un ensemble – Enfin, nous pouvons définir la partition d'un ensemble E, qui revient à choisir dans l'ensemble des parties de E des sous-ensembles non vides disjoints (aucune inclusion) telle que leur union donne E.

Soit un ensemble E. Une partition de E est un ensemble de sous-ensembles S_i tel que :

- $\cup_i S_i = E$
- $S_i \cap S_j$ pour tout $i \neq j$ (sous-ensembles disjoints)
- $S_i \neq \emptyset$ pour tout i

Exemple – Soit E = {1, 2,3} . Alors { {1}, {2,3} } est par exemple une partition de E.

Question de cours : opérateurs ensemblistes

On considère les ensembles suivants

E = {Tours, Blois, Mer, Amboise, Orléans} de licence informatique	ensemble des villes d'où sont originaires les étudiants
LC = {Blois, Mer, Vendome}	ensemble de villes du Loir et Cher.
IL = {Amboise, Tours, Joué}	ensemble de villes d'Indre et Loire

1. Donnez le contenu des ensembles $E \cap LC$.

2. Quelle est la cardinalité de l'ensemble $LC \cup IL$?

3. Donnez un exemple de partition de l'ensemble E de cardinalité 3.

4. Donnez le contenu du produit cartésien $LC \times IL$

3.2. NOTIONS DE BASE ET TERMINOLOGIE

La définition de la structure d'une base de données relationnelle fait appel à quatre concepts de base : les domaines, les relations, les tuples et les attributs. Nous allons les définir à la suite.

Domaine – Une base de données réunit des informations pouvant prendre un certain nombre de valeurs, qui peuvent être par exemple des entiers (nombres de points inscrits par un joueur, identificateur associué à un joueur), des chaînes de caractères (nom du jouer), ou une donnée au format date. Ces ensembles de valeurs sont regroupés sous la notation de domaine :

Domaine – *On appelle domaine un ensemble de constantes sur laquelle une information donnée peut-être définie)*

Cette définition est bien entendu à rapprocher de la notion de domaine d'interprétation vue en logique des prédicats du 1er ordre. Nous verrons que les deux notions seront à identifier dans la perspective d'une interprétation logique du calcul relationnel.

Remarque – Notons bien que cette notion peut englober des ensembles de valeurs très précis, au sens permis par la théorie des ensembles. Prenons l'exemple des départements métropolitains français. Ceux-ci sont numérotés ordinairement entre 1 (Ain) et 95 (Val d'Oise). Si l'on souhaite représenter une information concernant les départements, le domaine ne sera pas nécessairement celui des entiers mais celui, plus précis, des entiers compris entre 1 et 95. Ceci pour assurer la cohérence de la base de données en évitant, par exemple, les erreurs de saisies.

Relation – Comme son nom le suggère, la notion de relation est consubstantielle au modèle relationnel. C'est elle qui désigne en effet les associations d'information qui font sens pour l'univers du discours étudié. Par exemple, l'association d'informations concernant le numéro, le nom, le prénom, l'équipe, le poste, et le pays d'un joueur va définir une relation appelée `Joueur`. Chacune de ces informations correspond à un domaine donné (entiers ou chaînes de caractères dans cet exemple). C'est pourquoi on définit formellement une relation par la combinaison des domaines qu'elle concerne.

> **Relation –** *Soient D1, D2, …, Dn un ensemble de domaines qui ne sont pas nécessairement distincts. Une relation (ou relation n-aire) est un sous-ensemble du produit cartésien D1 x D2 X … X Dn.*
>
> *L'ensemble des éléments du sous-ensemble effectivement présents dans la relation est appelé* ***extension de la relation****. On parle également d'****instance de la relation*** *puisqu'une relation n'est qu'une instanciation particulière du produit cartésien.*
>
> **Schéma d'une relation** – *Soit une relation R construite sur le produit cartésien D1 X D2 X … X Dn. On appelle schéma de la relation la représentation R(D1,D2, …, Dn).*
>
> **Arité d'une relation** – *Soit une relation R construite sur le produit cartésien D1 X D2 X … X Dn et donc de schéma R(D1,D2, …, Dn). On appelle arité de la relation de nombre n de domaines présents dans son schéma.*

Exemple – La relation `Joueur` est d'arité 6 : elle est en effet construite sur le produit cartésien `Entier X Char X Char X Char X Char X Char` où `Entier` est le domaine des entiers naturels (associé au numéro de joueur) et `Char` le domaine des chaînes de caractères (associé aux autres informations associées aux joueurs). Elle se définit par le schéma de relation suivant :

`JOUEUR(numero,prenom,nom,equipe,poste,pays)` et correspondra au produit cartésien précédent.

Rappelons l'extension de la relation `JOUEUR(numero,prenom,nom,equipe,poste,pays)`.

JOUEUR					
Numéro	**Prénom**	**Nom**	**Equipe**	**Poste**	**Pays**
1	Tony	Parterre	Purs de St Antoine	Meneur	France
2	Joakim	Fépeur	Boucher de Chicago	Pivot	France
3	Nikola	Tebask	Os en gelée	Ailier	Euskadi
4	Boris	Babak	Purs de St Antoine	Ailier	France

Il est important de relever qu'une relation ne correspond pas au produit cartésien des domaines qui le définissent mais uniquement à un sous-ensemble de celui-ci, celui des éléments qui correspondent effectivement à des objets réels de l'univers du discours concerné. Si nous considérions l'ensemble du produit cartésien de la relation, nous pourrions ainsi prendre au hasard une valeur dans chacun des informations `numero,prenom,nom,equipe,poste,pays` pour former un n-uplet aussi improbable que `(1,Joakim,Tebask,Purs de St Antoine,Ailier,France)`.

Remarque – Nous allons étudier ci-après (§ 4.) une représentation logique du modèle relationnel. On peut déjà remarquer qu'une relation n-aire telle que celle étudiée ici se représentera aisément en logique des prédicats du 1er ordre sous forme d'un prédicat d'arité n. Par exemple (en syntaxe LP1 et non pas Prolog) : `Joueur(numero,prenom,nom,equipe,poste,pays)`

Attribut d'une relation – Une relation est formée de l'association de plusieurs domaines qui ont chacun un sens propre dans la relation. Ainsi, même si on utilise cinq fois le domaine `Char` dans la définition de la relation Joueur, chacun des éléments du produit cartésien correspond à une information différente dans l'univers du problème : le prénom, le nom, l'équipe, le poste et le pays. Chacun de ces grains d'information correspond précisément à un attribut de la relation.

> **Attribut –** *Soit une relation définie par le produit cartésien D1 X D2 X … X Dn. On appelle attribut Ai de la relation l'élément associé à un domaine Di.*
>
> **Fonction Sorte** – On appelle `sorte` la fonction qui associe à une relation l'ensemble de ses attributs.

Considérons la relation définie par le schéma : `JOUEUR(numero,prenom,nom,equipe,poste,pays)`
On a alors `Sorte(JOUEUR) = {numero, prenom, nom, equipe, poste, pays}` tandis que `numero`

est par exemple le premier attribut de la relation JOUEUR.

Cette écriture, appelée souvent **schéma nommé de la relation**. Nous reviendrons plus loin sur la notion d'approche nommée ou non nommée pour décrire les bases de données.

Tuple d'une relation – Enfin, l'extension de la relation comporte un ensemble d'éléments qui décrivent séparément un élément de l'univers du discours. Par exemple, tout ce qui a trait au joueur Tony Parterre. Pour une relation n-aire donnée, ces éléments sont constitués de l'association de n valeurs particulières provenant chacun d'un des n attributs de la relation. C'est pourquoi on les appelle n-uplets, ou d'une manière plus fréquente tuples de la relation.

> **Tuples (ou n-uplet) –** *Soit une relation définie par le produit cartésien D1 X D2 X … X Dn. On appelle tuple (ou n-uplet) de la relation toute assignation de n valeurs d'attributs de la relation qui désigne un élément décrit par la base de données.*

Exemple – Rappelons une dernière fois l'extension de la relation JOUEUR.

JOUEUR					
Numéro	**Prénom**	**Nom**	**Equipe**	**Poste**	**Pays**
1	Tony	Parterre	Purs de St Antoine	Meneur	France
2	Joakim	Fépeur	Boucher de Chicago	Pivot	France
3	Nikola	Tebask	Os en gelée	Ailier	Euskadi
4	Boris	Babak	Purs de St Antoine	Ailier	France

Cette instance de relation comprend quatre tuples (ou n-uplets). Le premier d'entre eux est ainsi :

```
(1, Tony, Parterre, Purs de St Antoine, meneur, france)
```

Jeux de mots : terminologie théorique et terminologie grand public.

L'ensemble des termes que nous venons de découvrir dans ce paragraphe relèvent de la théorie des bases de données inhérente au modèle relationnel. De par leur origine ensembliste, ces termes ont une forte connotation mathématiques (relation, n-uplet…) qui ne parle pas toujours à l'informaticien béotien. C'est la raison pour laquelle les éditeurs de systèmes de gestion de bases de données (SGBD) ont souvent préféré utilisé des termes plus parlants, car correspondant plus aux notions rencontrées au quotidien avec les tableurs. Par exemple, là où la théorie parle de relation, la documentation liée à un SGBD préfère parler de table. Il est donc utile de savoir détecter les synonymes le plus fréquemment employés entre terminologie théorique et terminologie destinée au grand public. On a ainsi

Théorie	**SGBD**
Relation	Table
Attribut	Champ, Colonne
Tuple, n-uplets	Enregistrement, Ligne

Base de données – Finalement, nous avons vu sur notre exemple qu'une base de données est constituée de plusieurs relations. L'ensemble de ces relations forment le schéma de la relation.

> **Schéma de base de données –** *On appelle schéma d'une base de données un ensemble fini de relations (et de leur schéma).*

Exemple – Notre base de données liée au basket est ainsi défini par le schéma : BASKET = {JOUEUR, PERF}

3.3. APPROCHE NOMMEE ET APPROCHE NON NOMMEE

Dans la présentation que nous venons de faire des bases de données, nous avons à chaque fois donné un nom aux attributs des relations. Cette **approche nommée** va de soit a priori, puisque le nom associé à un attribut aide à en comprendre le sens. Toutefois, on remarquera que ces attributs sont toujours parfaitement ordonnés. Par exemple, dans la relation JOUEUR, le premier attribut correspondra toujours au numéro du joueur, le second à son prénom et ainsi de suite.

Il est donc possible de caractériser précisément un attribut par sa position dans la relation. C'est pourquoi on peut utiliser une **approche non nommée**, dans laquelle l'attribut n'est plus déterminé par son nom mais uniquement par son numéro.

Du fait de l'ordonnancement des attributs, les deux approches sont rigoureusement équivalentes, on choisira simplement l'approche qui convient le mieux à un problème ou une démarche. Par exemple, lorsque nous étudierons l'interprétation logique des bases de données, nous ferons appel à une approche non nommée (cf. §4). Dans l'immédiat, contentons nous d'illustrer sur un exemple les notations employées dans les deux approches. Le tableau ci-dessous résume la situation.

Notion	Approche nommée	Approche non nommée
Schéma de relation	`Joueur(numero,prenom,nom)` ou `Joueur(numero :Entier,prenom :Char,nom :Char)`	`Entier × Char × Char`

Attribut	`numero`
	`prenom`
	`nom`
		Indiqué par la position :
		`Joueur(1)`
		`Joueur(2`
		`Joueur(3)`

Tuple	`(1,Tony,Parterre)`	`(1,Tony,Parterre)`

On observe que dans l'approche non nommée, le schéma de la relation se limite à la définition du produit cartésien, de même que les attributs sont simplement identifiés par l'indice de leur position. Par contre, lorsqu'on considère les tuples qui composent l'extension de la relation, on retrouve bien sûr le même contenu quelle que soit l'approche.

PERF			
Joueur	**Date**	**Adversaire**	**Points**
1	09/01/2013	Bouchers de Chicago	17
1	12/01/2013	Bretons de Boston	21
4	09/01/2013	Bouchers de Chicago	9
2	09/01/2013	Purs de St Antoine	9
2	11/01/2013	Os en gelée	29
3	11/01/2013	Bouchers de Chicago	4

Question de cours

On considère ci-dessus la relation PERF qui nous a servi à illustrer le cours.

1. Quelle est l'arité de la relation ?

2. Donnez un exemple de tuple de la relation.

3. Donnez le schéma de la relation dans une approche nommée ou non nommée.

4. Quelle réponse donnera la fonction sorte(PERF) dans le cas d'une approche nommée ?

4. INTERPRETATION LOGIQUE DU MODELE RELATIONNEL

4.1. INTERPRETATION LOGIQUE NOMMEE DU MODELE RELATIONNEL & PROLOG

Jusqu'ici, les notions que nous avons présentées faisaient référence à la théorie des ensembles. Cette approche, qualifiée **d'approche conventionnelle**, consiste à considérer la base de données comme un ensemble de tuples ensemblistes. Il est toutefois possible d'adopter une approche logique pour représenter les bases de données. Dans ce cas, on est dans une démarche assertionnelle, au sens où la base de données va être considérée comme un ensemble de faits déclarant (« *assertant* ») à chaque fois un grain de connaissance. Cette **approche logique**, totalement équivalente à la précédente, va faire appel à la logique des prédicats du premier ordre.

Schéma de relations : prédicats – Dans cette interprétation logique, le schéma d'une relation d'arité n peut être représenté par un prédicat de même arité. Rappelons le schéma de la relation JOUEUR qui nous sert d'exemple : `JOUEUR(numero,prenom,nom,equipe,poste,pays)`. Cette relation peut être représentée par un prédicat `Joueur` d'arité 6 dont les arguments correspondent aux attributs de la relation. Les **attributs** de la relation correspondent aux arguments du prédicat.

Comme nous l'avons vu dans la partie de l'ouvrage consacrée à la logique des prédicats, ou encore au langage Prolog, les arguments d'un prédicat ne reçoivent pas de nom, mais sont simplement définis par leur position : nous sommes donc en présence d'une **approche non nommée**.

Tuples et instance de relation : faits – En pratique, le prédicat qui caractérise une relation n'est pas donné par une définition en intension, mais au contraire en extension : on va en effet lister sous la forme d'une liste de faits (au sens des clauses de Horn) l'ensemble des tuples de l'instance de relation. Ainsi, dans le cas de la table JOUEUR, le prédicat `Joueur/6` sera défini par 4 faits successifs :

```
Joueur(1,Tony, Parterre, Purs de St Antoine, Meneur, France)
Joueur(2,Joakim, Fepeur, Bouchers de Chicago, Pivot, France)
Joueur(3,Nikola, Tebask, Os en gelee, Ailier, Euskadi)
Joueur(4,Boris, Babak, Purs de St Antoine, Ailier, France)
```

Ainsi, l'extension de la base est constituée des n-uplets qui sont des modèles du prédicat. On a donc :

$\forall$no $\forall$prenom $\forall$nom $\forall$team $\forall$poste $\forall$pays [(no,prenom,nom,team,poste,pays) $\in$ JOUEUR))

$\Leftrightarrow$ Joueur(no,prenom,nom,team,poste,pays)]

On suppose donc que toutes les autres combinaisons correspondent à des interprétations fausses : on dit que l'on travaille dans l'hypothèse du monde clos.

Traduction en langage Prolog – Les faits correspondant à des clauses de Horn, il est bien sûr possible de traduire directement en langage Prolog le contenu d'une base de données relationnelle, comme le montre le programme ci-dessous qui regroupe l'ensemble des faits nécessaire à décrire les instances des deux relations de la base BASKET = {JOUEUR, PERF}.

```
/* definition en extension de la relation JOUEUR */

joueur(1, tony, parterre, purs_de_st_antoine, meneur, france).
joueur(2, joakim, fepeur, bouchers_de_chicago, pivot, france).
joueur(3, nikola, tebask, os_en_gelee, ailier, euskadi).
joueur(4, boris, babak, purs_de_st_antoine, ailier, france).


/* definition en extension de la relation PERF */

perf(1, 09_01_13, bouchers_de_chicago, 17).
perf(1, 12_01_13, bretons_de_boston, 21).
perf(4, 09_01_13, bouchers_de_chicago, 9).
perf(2, 09_01_13, purs_de_st_antoine, 9).
perf(2, 11_01_13, os_en_gelee, 29).
perf(3, 11_01_13, bouchers_de_chicago, 4).
```

A partir d'une telle modélisation logique, il est possible d'opérer des interrogations dans la base de données. Celles-ci peuvent même donner lieu à des déductions logiques. Dans le paragraphe qui suit, nous nous contenterons d'étudier la recherche d'information dans la base, qui correspond directement aux modes d'interrogation d'une base de données relationnelle.

5. INTERROGATION DES BASES DE DONNEES RELATIONNELLES

5.1. QUELQUES OPERATEURS D'INTERROGATION DES BD RELATIONNELLES

On peut imaginer facilement d'interroger la base de données pour obtenir des informations aussi variées que celles-ci :

- liste des noms et prénoms de joueurs,
- listes des joueurs (numéro, nom, prénom, poste, équipe) qui sont français
- liste des joueurs (nom, prénom) qui ont marqué plus de 20 points dans un match.
- liste des joueurs qui sont meneurs ou pivot
- liste des joueurs qui ont marqué plus de 10 points et qui sont pivot

Toutes ces requêtes peuvent s'exprimer sous la forme de la combinaison d'opérateurs de base de l'algèbre relationnelle, qui est à la base du modèle relationnel. Nous allons en étudier *quelques uns* avant d'en donner dans le paragraphe suivant une interprétation logique.

Projection – La projection est une opération Π qui consiste à sélectionner l'ensemble des tuples d'une instance de relation mais à ne conserver en réponse que certains de ces attributs en réponse.

L'opérateur de projection répond typiquement à ce genre de question : *pouvez-vous me donner la liste des noms et prénom de joueurs.* Elle consiste à faire la projection de la relation `JOUEUR` sur les attributs `nom` et `prenom`. Formellement, on la définit comme suit :

> **Projection –** *Soit d'une relation R et un ensemble d'attributs Ai, … An $\in$ sorte(R). La projection d'une instance I_R de R sur (Ai,…,An) est notée $\Pi_{(Ai,…,An)}$ (I_R) et est définie par :*
>
> $\Pi_{(Ai,…,An)}$ (I_R) = { (t.Ai, .., t.An) | $t \in I$ }.

Exemple – La projection de la relation JOUEUR sur le couple d'attributs `prenom` et `nom` correspond à la nouvelle instance de relation $\Pi_{(prenom,nom)}$ (JOUEUR) où seuls deux attributs sont conservés :

JOUEUR					
Numéro	**Prénom**	**Nom**	**Equipe**	**Poste**	**Pays**
1	Tony	Parterre	Purs de St Antoine	Meneur	France
2	Joakim	Fépeur	Boucher de Chicago	Pivot	France
3	Nikola	Tebask	Os en gelée	Ailier	Euskadi
4	Boris	Babak	Purs de St Antoine	Ailier	France

$\Pi_{(prenom,nom)}$ **(JOUEUR).**	
Prénom	**Nom**
Tony	Parterre
Joakim	Fépeur
Nikola	Tebask
Boris	Babak

Pour aller plus loin : ma première requête SQL

En pratique, l'interrogation de bases de données relationnelles repose sur un standard : le langage de requêtage SQL (*Structured Query Language*) créé en 1974 et normalisé depuis 1986 (la dernière évolution de ce standard ISO date de 2011). Ce langage, très déclaratif dans sa forme, n'est que la traduction des opérations de l'algèbre relationnelle que nous étudions ici, ou leur traduction logique telle que nous allons l'étudier dans le paragraphe suivant. A titre informatif, et en préalable à une étude ultérieure plus poussée des bases de données relationnelles, voici comment se traduirait en SQL notre requête projective $\Pi_{(prenom,nom)}$ (JOUEUR) :

```
SELECT prenom, nom
FROM JOUEUR ;
```

La première ligne introduite par l'opérateur SELECT permet de préciser les attributs de projection, tandis que la seconde ligne introduite par le FROM précise sur quelle relation porte cette opération de projection. C'est aussi simple que cela … mais des requêtes bien plus complexes peuvent être crées en SQL, bien sûr ;

Sélection – La sélection est une opération σ qui consiste à sélectionner un certain nombre de tuples dans une instance de relation pour ne conserver en réponse que ceux qui répondent à une certaine condition de sélection. Ces conditions correspondent en toute généralité à une expression en logique des prédicats du 1er ordre.

L'opérateur de sélection répond typiquement à une genre question : *pouvez-vous me donner la liste des joueurs qui sont français.* Elle consiste à filtrer de la relation `JOUEUR` sur les valeurs de l'attribut `pays`. Les conditions de sélection peuvent bien entendu concerner plusieurs attributs (expression logique complexe). Formellement, on la définit comme suit :

> **Sélection –** *Soit une relation R et un ensemble d'attributs Ai, … An $\in$ sorte(R). La sélection d'une instance I_R de R suivant une contrainte C(Ai…, An) est notée $\sigma_{C(Ai,…,An)}$ (I_R) et est définie par :*
>
> $\sigma_{C(Ai,…,An)}$ (I_R) = { $t \in I$ | C(t.Ai, .., t.An) } *où C(Ai…, An) est une expression logique portant sur les attributs Ai, … An*

Exemple – La question *pouvez-vous me donner la liste des joueurs qui sont français* correspond à une sélection sur l'attribut `pays` suivant la condition logique `C(pays) ≡ pays = 'france'`. Elle correspond à la nouvelle instance notée $\sigma_{pays='France'}$ (JOUEUR).

$\sigma_{pays='france'}$ (JOUEUR).					
Numéro	**Prénom**	**Nom**	**Equipe**	**Poste**	**Pays**
1	Tony	Parterre	Purs de St Antoine	Meneur	France
2	Joakim	Fépeur	Boucher de Chicago	Pivot	France
4	Boris	Babak	Purs de St Antoine	Ailier	France

L'expression logique de sélection peut être bien plus complexe. Elle peut ainsi porter sur plusieurs attributs, comme par exemple la requête qui correspondrait à la sélection des joueurs français de l'équipe de *Purs de St Antoine* : $\sigma_{pays='France' \wedge equipe='Purs\ de\ St\ Antoine'}$ (JOUEUR).

Les conditions peuvent porter sur des inégalités ou tout autre opérateur logique. Elles peuvent concerner également des comparaisons entre valeurs d'attributs différents etc…

Pour aller plus loin : traduction en SQL

Toujours dans le dessein d'exciter votre curiosité, voici maintenant comment se traduirait en SQL la requête de sélection $\sigma_{pays='France'}$ (JOUEUR).:

```
SELECT *
FROM JOUEUR
WHERE pays='France';
```

Ici, l'étoile à côté du champ SELECT précise simplement que l'on va garder toute l'information concernant les tuples sélectionnés. Il n'y a pas de projection, mais on devine immédiatement comment on peut combiner en SQL projection et sélection. La sélection en elle-même est introduite par l'opérateur WHERE suivi de notre critère de recherche.

Pour combiner deux critères dans la requête $\sigma_{pays='France' \wedge equipe='Purs\ de\ St\ Antoine'}$ (JOUEUR), il suffit d'ajouter un nouveau champ AND après le WHERE :

```
SELECT *
FROM JOUEUR
WHERE pays='France'
AND equipe='Purs de St Antoine';
```

La dernière opération de l'algèbre relationnelle que nous allons étudier dans ce chapitre est liée directement au modèle relationnel, puisqu'elle peut-être utilisée pour associer deux relations différentes. Il s'agit de l'opération de jointure naturelle.

Jointure naturelle – La jointure naturelle consiste à associer les tuples de deux relations associées par une relation référentielle clé primaire / clé étrangère afin de reconstruire l'ensemble de l'information portée séparément par les deux tables. Par exemple, sur notre exemple, la jointure naturelle entre les relations JOUEURS et PERF va nous redonner le tableau originel qui nous a servi d'exemple introductif, en associant les tuples de chaque relation à l'aide de l'attribut commun `numero` les joueurs et leurs performances. Considérons par exemple le tuple `(1,09/01/13, Bouchers de Chicago,17)` de la relation PERF. En suivant la valeur 1 de la clé étrangère du tuple, on peut faire la jointure avec le tuple de la relation JOUEURS (`1,Tony,Parterre, Purs de St_Antoine, Meneur, France`). On obtient alors le tuple suivant, qui appartient à la jointure naturelle des deux relations :

`(1,Tony,Parterre,Purs de St_Antoine,Meneur,France,09/01/13,Bouchers de Chicago,17)`

En pratique, la jointure revient donc à faire le produit cartésien des deux tables, puis à faire la sélection dans ce produit cartésien des seuls tuples pour lesquels il y a identité de valeur entre la clé primaire de la table référencée et la clé étrangère de la clé étrangère. Formellement, on définit l'opération comme suit :

> **Jointure naturelle –** *Soit deux relations A et B, un ensemble d'attributs Ai, … An ∈ sorte(A) et un ensemble d'attributs Bi, … Bn ∈ sorte(B) . La jointure naturelle d'une instance I_A de A et d'une instance I_B de B suivant les ensembles d'attributs considérés est notée $I_A \boxtimes_{t(A1 = B1 \dots An = Bn)} I_B$ et est définie par :*
>
> $I_A \boxtimes_{t(A1 = B1 \dots An = Bn)} I_B = \{ t \in A \times B \mid t.A_1 = t.B_1 \dots t.A_n = t.B_n \}$

Par exemple, la jointure naturelle entre les relations PERF ou JOUEURS s'exprimera ainsi :

PERF ⊠ *PERF.numero = JOUEURS.numero* JOUEURS

La notion de jointure peut même être étendue non pas à une relation d'égalité mais à toute relation logique entre les valeurs d'attributs (relation d'ordre comme <, >, ou tout autre relation logique). Notons enfin que l'on peut même joindre une relation avec elle-même : on parle alors d'auto-jointure.

Pour aller plus loin : traduction en SQL

En SQL, la traduction d'une jointure ne fait pas apparaître de nouvel opérateur : on se contente de préciser dans le champ FROM l'ensemble des relations dont il faudra faire le produit cartésien. Le reste des opérations nécessaires à la mise en œuvre de la jointure consiste en des sélections sur les tuples issus du produit cartésien, comme nous l'avons vu. Ceci se traduit directement par le champ FROM. D'où par exemple la traduction de la requête PERF ⊠ *PERF.numero = JOUEURS.numero* JOUEURS

```
SELECT *
FROM PERF, JOUEUR
WHERE PERF.numero=JOUEUR.numero;
```

On remarquera la notation pointée RELATION.attribut dans le champ WHERE, pour bien faire la distinction des relations concernées.

Requête quelconque – Une requête quelconque dans une base de données s'exprimera comme la composition de plusieurs opérateurs de l'algèbre relationnel. Supposons que l'on veuille avoir la liste des prénoms et noms des joueurs français et de leur équipe. On va procéder à deux opérations successives :

- sélection des joueurs français $\sigma_{pays='france'}(JOUEUR)$
- projection sur les attributs prenom, nom, equipe $\Pi_{(prenom,nom)}(\sigma_{pays='france'}(JOUEUR))$

Cette composition aurait pu aussi bien concerner la jointure naturelle (cf. questions de cours). Notons enfin que l'algèbre relationnelle dispose par ailleurs d'autres opérateurs ensemblistes entre relations (union, différence, intersection, division). Leur étude dépasse le propos de ce chapitre, qui vise simplement à donner une idée des relations entre bases de données et modélisation logique.

5.2. INTERPRETATION LOGIQUE DES OPERATEURS DE L'ALGEBRE RELATIONNELLE

Pour terminer ce chapitre consacré aux bases de données, nous allons précisément donner une interprétation logique des opérateurs de l'algèbre que nous venons d'étudier. En particulier, nous allons voir comment représenter en Prolog certaines requêtes étudiées

Projection – Les attributs du modèle relationnel correspondent aux arguments des prédicats de définition des tuples dans une approche logique. La projection revient donc à ignorer certains arguments, comme on le fait avec les variables anonymées en Prolog.

Considérons par exemple une relation R définie par un prédicat d'arité p et un ensemble d'arguments $A_1, \ldots An \in$ sorte(R), avec n < p. La projection d'une instance I_R de R sur (Ai,…,An) peut être définie comme suit en logique des prédicats du 1er ordre :

$\forall A_1 \ldots \forall An \; [(A_1, \ldots, An) \in PROJECTION(R) \Leftrightarrow R(\_,\ldots,\_,A_{1,\ldots}, An,\_,\ldots,\_)]$

Reprenons l'exemple de projection étudié du § 3.4 : *liste des noms et prénom de joueurs* :

$\forall Prenom \, \forall Nom \; [(Prenom, Nom) \in \Pi_{prenom,nom}(JOUEUR) \Leftrightarrow JOUEURS(\_, Prenom, Nom, \_, \_, \_)]$

Cette relation peut s'exprimer sous la forme d'une règle en Prolog, en introduisant un nouveau prédicat qui est le prédicat d'interrogation correspondant à la requête :

```
nom_Joueur(Prenom, Nom) :- joueur(_,_,Prenom,Nom,_,_,_).
```

Sélection – La sélection revient elle à ajouter une condition logique de filtrage sur les attributs des tuples de la relation. Cette contrainte s'exprime directement sous forme logique. Considérons une relation R définire par un prédicat d'arité p et un ensemble d'arguments $A_1, \ldots, An \in$ sorte(R) avec $n \leq p$ sur lesquels est définie une expression logique de sélection $C(A_1, \ldots, An)$, c'est-à-dire qu'on ne souhaite conserver que les tuples qui vérifient cette condition (modèles de la contrainte). La sélection

d'une instance I_R de R suivant la contrainte sur les attributs (Ai,…,An) peut être définie comme suit en logique des prédicats du 1er ordre :

$\forall T \ (T \in SELECT(T) \Leftrightarrow R(T) \wedge C(T.Ai, .., T.An))$

En Prolog, cette contrainte va se traduire par un but à résoudre dans une règle qui définit un nouveau prédicat d'interrogation. Reprenons l'exemple que nous avons étudié, à savoir la requête qui donne *la liste des joueurs qui sont français*. Ici, la contrainte s'exprime par l'équation atomique `pays= France`.

```
francais(No,Prenom,Nom,Equipe,Poste) :- Joueur(No,Prenom,Nom,Equipe,Poste,Pays),
                                        Pays = france.
```

Cette condition peut être directement intégrée au prédicat Joueur en jouant sur l'unification de l'argument `Pays` avec la constante `france`. Profitons-en pour donner une nouvelle écriture du prédicat `francais` qui combine la sélection recherchée et une projection sur uniquement trois attributs : on retrouve la possibilité de composition d'opérateurs déjà évoquée.

```
francais(Prenom,Nom) :- Joueur(_,Prenom,Nom,Equipe,_,france).
```

Jointure naturelle – Enfin, la jointure va consister combiner un produit cartésien avec une sélection, qui s'exprime elle aussi directement en logique des prédicats du 1er, en faisant appel cette fois aux deux prédicats qui définissent les deux relations jointes et en ajoutant les conditions de sélection propres à la jointure.

Soit deux prédicats A et B caractérisant les relations du même nom, un ensemble d'arguments Ai, … An $\in$ sorte(A) et un ensemble d'arguments Bi, … Bn $\in$ sorte(B) . La jointure naturelle d'une instance I_A de A et d'une instance I_B de B suivant les ensembles d'attributs considérés peut-être définie par

$\forall T \ (T \in JOIN(T) \Leftrightarrow \exists TA \ \exists TB \ [T=(TA,TB) \wedge A(TA) \wedge B(TB) \wedge (TA.A_1=TA.B_1) \wedge \ldots \wedge (TA.A_n=TB.B_n)]$

De même, en Prolog, on utilisera les deux prédicats qui définissent les deux relations pour exprimer la jointure à l'aide d'un nouveau prédicat d'interrogation qui reprendra l'union des arguments des arguments des prédicats initiaux. La jointure naturelle entre les deux relations JOUEUR et PERF peut ainsi être résumée par la règle suivante, la condition de jointure étant là encore gérée directement par unification entre variables du même nom :

```
Basket(No,Prenom,Nom,Equipe,Poste,Pays,Date,Adversaire,Points) :-
    Joueur(No,Prenom,Nom,Equipe,Poste,Pays),
    Perf(No,Date,Adversaire,Points).
```

Question de cours

On considère toujours notre exemple de base de données basketball. Représentez à l'aide des opérateurs de l'algèbre relationnelle, puis en Prolog, les requêtes suivantes :

1. Liste des joueurs (prenom, nom, équipe) qui sont des ailiers

2. Liste des joueurs (prenom, nom) des joueurs qui ont joué le 09/01/2013.

3. Liste des matchs sous la forme : data, nom de l'équipe, nom de l'adversaire

5.3. INTERROGATION DES BD RELATIONNELLES : UNE VUE PLUS GENERALE

Dans les paragraphes précédents, nous avons vu comment le modèle de l'algèbre relationnelle permet l'interrogation de bases de données à l'aide d'opérateur de base (projection, sélection, jointure). Puis nous avons vu que ces opérateurs pouvaient se traduire de manière purement logique. Les opérateurs de l'algèbre relationnelle sont plus nombreux que les trois cas (certes les plus fréquemment rencontrés) que nous venons d'étudier. Surtout, le paradigme logique permet d'envisager tout un ensemble de requêtes possibles sans en passer par ces opérateurs. Nous allons étudier dans ce paragraphe quelques exemples qui nous montrent toute la généricité de la formalisation logique pour l'interrogation des bases de données relationnelles.

A faire : vus également dans le TP Prolog « Bases de données »

6. POUR ALLER PLUS LOIN : CONTRAINTES D'INTEGRITE

(partie hors programme : L2)

Nous venons d'étudier dans ce chapitre une approche purement ensembliste puis logique de la notion de base de données. Un souci que l'on a lorsqu'on mémorise ainsi une grande masse d'information dans une base et de s'assurer de la cohérence des données qui y sont stockées. Par exemple, on veut pouvoir s'assurer que le nombre de points marqué par un joueur est toujours positif (les erreurs de saisie sont toujours possibles), ou bien que le numéro d'un joueur donné dans la relation PERF correspond bien à un numéro de joueur défini dans la relation JOUEUR. Intégrer ce genre d'informations dans la conception d'une base de données permet d'éviter bien des incohérences.

Dans un SGBD relationnel, cette cohérence peut-être partiellement assurée par la définition d'assertions de contrôle appelées contraintes d'intégrités, qui empêchent la saisie ou l'importation de données qui ne respecteraient pas ces dernières : ainsi, si par erreur on entrait un nombre de points négatifs, le système refuserait cette insertion d'information dans la base et le ferait savoir par un message d'erreur. Ces contraintes d'intégrité sont de différents types que nous allons maintenant présenter rapidement.

Contraintes de type et de domaine – Ces contraintes sont associées aux attributs. Elles permettent de s'assurer que les valeurs que prennent les attributs dans les instances de relation correspondent bien au domaine de ces attributs. Une contrainte de type spécifiera un type de données particulier (dans notre exemple : `Entier` ou `Char`) tandis que les contraintes de domaine ajouteront des contraintes sur les valeurs que peuvent prendre les attributs pour un type donné. Par exemple, pour reprendre notre exemple sur les départements : `numero_dept` $\in$ `[1,95]`

Les tuples qui ne respectent par ces contraintes sont considérés comme incohérents : ils ne peuvent être chargés dans la base de données. Formellement, les contraintes de type et de domaine ne sont simplement qu'un moyen de définir les domaines sur lesquels sont construites les relations.

Contraintes d'unicité et clé primaire – Nous avons vu sur notre exemple que l'attribut `numero` de la relation JOUEUR permettait de désigner de manière unique chaque tuple de la relation. Dans ce cas, on dit que l'attribut `numero` joue le rôle de clé primaire de la relation. Ses valeurs sont obligatoirement toutes distinctes, puisqu'elles doivent être différentes pour chaque tuple. Pour s'assurer de cette propriété, on dit que l'on définit une contrainte d'unicité sur cet attribut.

On peut noter par ailleurs que le couple d'attributs `(nom, prenom)` peut lui aussi caractériser de manière unique les tuples de la relation. Ainsi, un n-uplet d'attributs peut jouer un rôle de clé primaire et se voir imposer une contrainte d'unicité.

> **Contrainte d'unicité –** *Soit Ai, …, An des attributs d'une relation R, soit donc Ai, … An* $\in$ *sorte(R). Imposer une contrainte d'unicité sur le n-uplet d'attributs (Ai,…,An) signifie qu'il ne peut pas exister dans l'instance de R deux tuples pour lesquels les valuations des attributs (Ai,…,An) sont identiques.*
>
> **Clé primaire –** *Soit Ai, …, An des attributs d'une relation R, soit donc Ai, … An* $\in$ *sorte(R). Le n-uplet d'attributs (Ai,…,An) est une clé primaire de la relation s'il n'existe toujours qu'une seule occurrence de tuple pour une valuation donnée des attributs (Ai,…,An).*

Exemple – Dans la relation `PERF`, l'association d'attribut `(numero,date,adversaire)` peut jouer le rôle de clé primaire. On vérifie aisément sur l'instance de la relation qu'aucun attribut ne peut constituer isolément une clé primaire : aucun de respecte déjà la contrainte d'unicité liée à la définition d'une clé primaire.

Contraintes référentielle – Enfin, nous avions vu en introduction que l'intérêt du modèle relationnel était de permette d'exprimer des associations sémantiques entre relations par l'intermédiaire d'attributs de même rôle. Dans notre exemple, c'est le cas de l'attribut `numero` des deux relations. Cette duplication de l'attribut nous assure de la cohérence de l'information partagée entre différentes relations. Il est par exemple essentiel qu'à chaque valeur de `numero` dans la relation PERF corresponde une instance de tuple unique de la relation JOUEUR, cette unicité étant assurée par le fait que `numero` est la clé primaire de la relation JOUEUR. On dit que la relation PERF référence la relation JOUEUR par l'intermédiaire de l'attribut `numero`.

D'une manière générale, la référenciation d'une relation par une relation référencée nécessite :

- la définition d'une clé primaire dans la relation référencée,
- la reprise de cette clé dans la relation référençante : on parle alors de clé étrangère
- la définition d'une contrainte, appelée contrainte d'intégrité référentielle, qui assure que chaque valeur de la clé étrangère correspond à une valeur de la clé primaire dans l'instance de relation référencée.

Remarquons une fois encore que clé primaire comme clé référentielle peuvent correspondre à un n-uplet d'attributs.

Exemple – Dans notre exemple :

- la relation référencée est JOUEUR qui a pour clé primaire l'attribut `numero`.
- la relation référençante est PERF, relié à JOUEUR par la clé étrangère `numero`.

Remarquons que la clé étrangère de la relation référençante n'est pas la clé primaire de cette dernière. Par ailleurs, rien n'impose que clé étrangère et clé primaire aient le même nom. Ces liens de référenciation permettent de reconstituer l'ensemble de l'information présente dans la base de données à l'aide de requêtes, comme nous l'avons vu dans le paragraphe précédent avec l'opération de jointure. Même s'il existe une contrainte d'intégrité référentielle, il faudra toujours spécifier lors d'une requête, ce type de lien de référenciation, qu'une contrainte d'intégrité référentielle ait été définie ou non lors de la conception de la base de données.

Remarque – On notera que la définition de la jointure naturelle ne fait nulle part référence à la notion de contrainte d'intégrité référentielle. En effet, la jointure peut en toute généralité associer tout couple de relations, que celles-ci soient reliées ou pas par une relation référentielle. Il suffit d'identifier des couples d'attributs entre les deux relations.

Logique et bases de données - Exercices & Problèmes

EXERCICES

Exercice 5.1. — Opérateurs ensemblistes : rappels (objectif 5.XXX)

On considère les deux ensembles suivants A = { Cover, Harry, Lea, Ali } et B = { Gator, Lea}. Donnez les ensembles résultats des opérations suivantes :

1. $A \cap B$

2. $A \cup B$

3. $A \times B$

4. Trouvez un ensemble F qui vérifie $F \subset (A \cap B)$

5. Quelle est la cardinalité de votre ensemble F.

Exercice 5.2. — Bases de données et calcul ensembliste (objectif 5.XXX)

Nous avons vu en cours qu'une base de données pouvait être décrite d'un point de vue logique comme d'un point de vue ensembliste. Cet exercice va vérifier vos compétences de base en théorie des ensembles.

On considère une relation E dont le schéma nommé est donné par : R(Nom, Prenom). Il est très fréquent qu'un prénom puisse jouer le rôle d'un nom de famille et inversement. Par exemple :

Patrick Marcel	mais également	*Marcel Duchamp*
Jean-Yves Antoine	mais également	*Antoine de Saint-Exupéry*
Xavier Martin	mais également	*Martin Luther-King*

Ainsi, le domaine sur lequle seront construits les instances de R n'est pas celui des chaînes de caractères en général, mais celles qui peuvent jouer le rôle de prénom comme de patronymes. Considérant les exemples ci-dessus, on définit ainsi l'ensemble dom = {antoine, marcel, martin}.

1. Donnez alors le schéma de la relation R dans une approche non nommée

2. Partant de la définition du schéma de la relation R, donnez des exemples d'instances A, B et C de R qui vérifient respectivement les propriétés suivantes :

a. Tuples disjoints $A \cap B = \varnothing$
b. Intersection $A \cap B = C$
c. Union $A \cup B = C$
d. Inclusion $A \subset B$
e. Partition A, B et C forment une partition de dom × dom

Exercice 5.3. — Composition d'opérations ensembliste (objectif 5.XX ; d'après P. Marcel)

On considère dans cet exercice les ensembles A = {a,b,d} et B = {b,c,d,e}. La différence ensembliste de A par B, notée A – B, correspond à l'ensemble des éléments de A qui ne sont pas dans B.

1. Donnez la définition en extension puis en intension de l'ensemble correspondant à l'opération A – B.

2. Pourriez-vous donner une expression logique en intension de A-B ? On supposera pour cela que l'ensemble A (resp. B) a été défini en intension par un prédicat : `∀x (Elt_de(A, x) ⇔ A(x))`. Donner en extension les ensembles résultats des opérations ensemblistes suivante définies ci-après :

a. $B - (A \cup B)$

b. $A - (A \cap B)$

c. $(A - B) \cup (B - A)$

d. $(A \cup B) \cap (A \times B)$

e. $(A \times A) - (A \times B)$

Exercice 5.4. — Familiarisation avec les notations (objectif 5.XX ; d'après P. Marcel)

On considère une base de données D composée de deux relations I et J dont voici les deux instances :

I	A	B
	a	b
	c	b
	a	a

J	A
	d

1. Donnez, pour chaque attribut, le domaine associé à chaque attribut.
2. Donnez le schéma des relations de la base de données dans le cas des représentations suivantes :
a) nommée **b)** non nommée **c)** Logique
3. Que donne l'appel à la fonction *Sorte* sur les deux relations dans le cas d'une représentation nommée puis non nommée ?

4. Donnez le résultat de P(J) ou P est la fonction qui donne une partition d'un ensemble.

Exercice 5.5. — Cinéma et exil fiscal (objectif 5.XX)

Dans *Cyrano de Bergerac*, de Jean-Paul Rappeneau, Gérard Depardieu jouait magistralement le rôle du vibrant capitaine de mousquetaires du Roi. Hors cadre, les déboires de l'acteur avec le fisc et sa domiciliation hors de France ont quelque peu écorné cette généreuse image.

On considère une base de données formée de la seule relation suivante, qui recense les différentes résidences qu'a occupées Gérard Depardieu. Cette relation sera appelée `Residence`.

Ville	**Pays**	**Arrivée**	**Départ**
Trouville	France	1998	2010
Néchin	Belgique	2011	2013
Saransk	Mordavie	2014	2015

1. Donnez le schéma de la relation dans une représentation nommée ou non.

2. Que donne le résultat de la fonction `sorte(Residences)` ?

3. Donnez un exemple de tuple de la relation.

4. Pouvez-vous décrire cette base de données sous la forme d'un programme Prolog ?

5. Interrogeons la base de données : définissez un prédicat Pays(X) qui est vrai si Gérard Depardieu a résidé dans ce pays à un moment donné.

Exercice 5.6. — Un peu de modélisation : fusion de régions (objectif 5.XX)

La réforme territoriale de 2015, qui a entrainé des fusions de régions, ne va pas avoir que des conséquences politques. Elle va en effet entraîner obligatoirement de multiples modifications dans les bases de données des systèmes d'information géographique.

Dans cet exercice, nous supposons que nous nous intéressons à la base de données de l'INAO, l'Institut National des Appellations d'Origine, gardien de l'ancrage territorial des produits agrico-alimentaires que nous consommons régulièrement. L'INAO a des problèmes : comment par exemple appeler désormais le beurre de Charentes-Poitou, maintenant qu'il sera produit en … Aquitaine.

Pour ce faire, l'INAO a développé une base de données qui précise pour chaque produit son département de production, la région à laquelle appartenaient ce département, et sa nouvelle région de rattachement. Cette base de donnée à une relation dont on donne ici un extrait d'instance.

Produit AOC	Type	Département	Ancienne Région	Nouvelle Région
Riesling	Vin	Haut-Rhin	Alsace	Alace-Lorraine-Champagne-Ardennes
Gewurztraminer	Vin	Haut-Rhin	Alsace	Alace-Lorraine-Champagne-Ardennes
Riesling	Vin	Bas-Rhin	Alsace	Alace-Lorraine-Champagne-Ardennes
Cancoillotte	Fromage	Doubs	Franche-Comté	Bourgogne
Comté	Fromage	Jura	Franche-Comté	Bourgogne
Sainte-Maure	Fromage	Indre-et-Loire	Centre	Centre
Saint-Nectaire	Fromage	Puy-de-Dôme	Auvergne	Rhône-Alpes-Auvergne

1. Donnez le schéma de la relation dans une représentation nommée ou non.

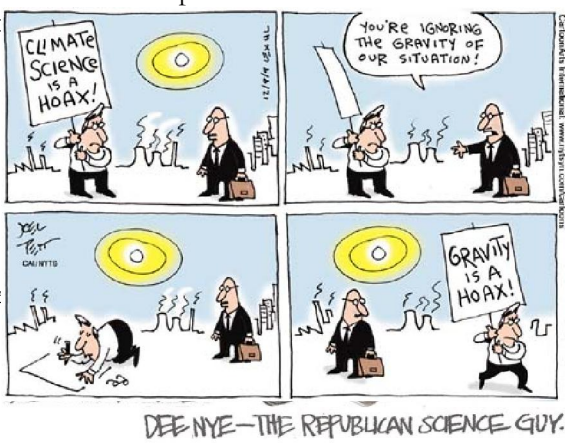
2. Pouvez-vous détecter des redondances inutiles dans la base de données ? A quels ensembles d'attributs correspondent-ils ?

3. Proposez un découpage de la base de données en plusieurs relations dont on donnera le schéma nommé pour limiter au maximum les relations.

4. Ce découpage permet-il de recréer la base de données originelle par produit cartésien ?

Exercice 5.7. — COP21 : Modélisation de base de données (objectif 5.XX)

Anne Hidalgo, la maire de Paris, a réuni récemment, en marge de la conférence de la COP21, les maires des plus grandes métropoles du monde, qui se sont engagés eux-aussi à réduire les émissions de CO2 de leurs villes. On considère une base de données appelée Cop21 qui décrit l'état de la lutte contre le réchauffement climatique dans leur métropole ainsi que leurs engagements de réduction pour les années à venir. La base de données n'est pour l'instant représentée que dans un t
unique. Celle-ci contient les attributs suivants :



- Nom Nom de la métropole concernée (chaîne de caractères)
- Pays Pays où la ville est située (chaîne de caractères)
- Cap Booléen qui est vrai si la ville est capitale du pays.
- Cont Continent où est située la ville (chaîne de caractères)
- An Année de référence pour la valeur d'émission de CO2 (entier)
- Hab Millions d'habitants pour l'année considérée (réel)
- Co2 Nombre de kg de C02 émis par habitant pour cette année (entie

Par exemple, les données pour quelques années sur Paris pourraient être :

- 2015 6105 kg/hab, 10500000 habitants (An=2015, Co2=6105, Hab =
- 2025 5167 kg/hab, 11500000 habitants (An=2025, Co2=5167, Hab =

Ces deux informations correspondent à deux tuples différents.

Cette relation comporte des redondances. Saurez-vous les éliminer en donnant un schéma de bases de données plus adapté. On donnera pour cela le schéma (en notation nommée) de toutes les relations de la base de données, puis l'ensemble des tuples correspondant à la description de la situation parisienne en 2015 et en 2025 dans votre nouvelle base de données. Ces tuples seront définis en extension en Prolog.

Exercice 5.8. — Interrogation des BDs : requêtes ensemblistes (objectif 5.XX ; d'après P. Marcel)

On considère dans cet exercice les ensembles A = {a,b,d} et B = {b,c,d,e}. Donnez les réponses aux requêtes ensemblistes suivantes

1. L'ensemble des x tels que (x,c) ∈ A×B.

2. L'ensemble des (x,y,z) tels que (x,y) ∈ A×B et (y,z) ∈ B×A et z ∈A∩B

3. L'ensemble des (x,y) tels que y = a et (y,x) ∈ B×A

Exercice 5.9. — Ensemble nous vaincrons : requêtes ensemblistes (objectif 5.XX)

Entre les deux tours d'élections comme les régionales ou les municipales, les différents partis peuvent tenter de fusionner leurs listes pour préparer le round final de dimanche prochain. Une fusion de liste, c'est l'union ensembliste de sous-ensembles de listes ! On considère les listes suivantes, définies comme des ensembles : PS = {a,b}, EELV = {c,d}, LR = {e,f}, FN = {f,h}. Vous remarquerez que l'élément f est commun à deux ensembles. Impossible ? Non : lors des dernières élections départementales, deux personnes âgées se sont retrouvées inscrites sur des listes Front National malgré elles : la signature portée sur les candidatures n'était pas la leur !

a — Binôme titulaire suppléant `BINOME = PS × EELV`

b — Duel ou triangulaire ? `GLOUPS = (PS × LR) ∩ (PS × FN)`

c — Fusion ou séparation `TI_AMO_MOI_NON_PLUS = PARTITION(PS ∪ EELV)`

Exercice 5.10. — Interrogation des BDs : requêtes logiques (objectif 5.XX)

On considère une relation de schéma R(prenom :CHAR, nom :CHAR, taille : ENTIER) et on considère une instance I particulière de cette relation. Celle-ci est donnée en extension sous forme logique :

```
R(patrick, marcel,178).
```

```
R(jean-yves, antoine, 186).
R(xavier, martin, 185).
```

1. **Exécution de requête** - Dire en français ce que font les requêtes suivantes puis donnez l'ensemble des instances qui correspondent à l'expression de la requête sur l'instance I :

   a) ∀T (REQUETE(T) ⇔ R(T) ∧ T.taille > 180)
   b) ∀T (REQUETE(T) ⇔ R(T.prenom,T.nom,_)]

2. **Reformulation de requêtes** – Traduisez ces requêtes en algèbre relationnelle puis en Prolog.

Exercice 5.11. — BD universitaire : création de requêtes conjonctives (objectif 5.XX)

Le système éducatif universitaire est devenu au fil des réformes un maquis incompréhensible pour les étudiants. Ainsi, l'étudiant en informatique qui veut faire se spécialiser dans la sécurité informatique peut faire aussi bien une école d'ingénieur après une classe préparatoire, un master professionnel après une licence générale, mais aussi rejoindre cette école après deux années de licence, ou bien faire un DUT puis une licence professionnelle en sécurité des réseaux avant de reprendre un master en VAE après quelques années d'expérience. Au final, il est plus facile de hacker un serveur que de réussir son orientation : comment voulez-vous dès lors lutter contre la cybercriminalité ? Pour aider les étudiants, supposons que nous avons à notre disposition une base de données qui décrit l'ensemble des formations en informatique et les poursuites d'études qu'elles permettent.

Cette base de données est constituée des deux relations suivantes dont on donne le schéma nommé :
· `Formation(niveau :D1 ; mention :varchar ; univ :varchar ;code :varchar)`
· `Poursuite(code :varchar ;niveau :D1 ;mention :varchar)`
Avec `D1 = {dut,bts,licence,lpro,master,ecole}`

1. Instance et tuples – La licence informatique de l'université de Tours (code : FRTOURS1_INFLIC) peut permettre une poursuite d'étude dans des masters de mention décisionnel (c'est le cas de notre master) ou génie logiciel. On veut rentrer ces informations dans une nouvelle instance de la base de données : donnez les tuples correspondant à cette instance, puis leur déclaration si cette base de données était implémentée en Prolog.

2. Question de modélisation – La relation `Poursuite` donne les mentions de formations qui peuvent suivre, mais par leur localisation exacte (université), information qui se trouve dans la relation `Formation` ? Expliquez pourquoi cette découpe en deux relation n'est pas un problème mais au contraire un avantage.

Requêtage de la base de données – Nous allons maintenant interroger la base de données. A chaque fois, nous considérerons trois façons (équivalentes, rappelons-le) de l'interroger :

- sous la forme d'une composition des opérateurs connus de l'algèbre relationnelle,
- sous la forme d'une règle conjonctive logique exprimée en Prolog
- dans la traduction en SQL de ces deux premières formes de requête

Donner donc l'expression des requêtes donnant les résultats suivants :

3. Liste de toutes les informations sur l'ensemble des formations de l'université de Tours.

4. Liste de toutes les mentions de formations existant dans le système universitaire français (supposé intégralement décrit dans notre instance de base de données.

5. Liste cette fois uniquement des mentions correspondant à des formations de niveau master.

6. Donner la mention des formations qui peuvent suivre une licence informatique.

7. Donner la liste des universités qui peuvent suivre une licence informatique.

Exercice 5.12. — Sécurité aérienne : interrogation (objectif 5.XX)

Vol MA370 disparu aux confins de l'Océan Indien comme dans la série Lost (un bout d'empenage vient d'être retrouvé fin 2015 sur l'ile de la Réunion), vol MA17 détruit dans le ciel ukrainien, l'année 2014 fut une année noire pour Malaysian Airlines, compagnie pourtant connue pour son niveau de sécurité. Certes l'enquête menée aux Pays-Bas a montré que la seconde catastrophe fut plus due au zèle aviné de mercenaires pro-russes qu'à une erreur de pilotage ou de maintenance… Il n'en reste pas moins qu'il serait intéressant d'avoir des informations fiables sur la sureté des compagnies, les avions et les pilotes qui nous transportent.

L'Union Européenne a d'ailleurs établi une liste noire (assez réduite) de compagnies inquiétantes par leur manque de fiabilité. On considère précisément une base de données recensant ce type d'information. Elle est composée de trois relations dont on donne le schéma :

- `Pilote(numero:int, nom:string, grade:string, hrs_vol: int)`
- `Avion(numero: int, type:string, capacite: int, hrds_vol: int, indice_surete: int)`
- `Vol(no: int, airline:string, no_p: int, no_a: int, v_dep:string, v_arr:string, date_vol:date)`

Exprimez les requêtes définies ci-après sous les trois formes suivantes :

- requête conjonctive (on donnera la règle et la formule conjonctive)
- algèbre relationnelle
- SQL

1) Quel est le numéro et le type des avions dont la capacité dépasse les 500 passagers.

2) Quel est le numéro des vols au départ de Paris le jeudi 13/1/2015.

3) Quel est le niveau de sureté de chaque avion de la compagnie Malaysian Airlines (airline : MH). On donnera en réponse le numéro de l'avion, son type et son indice de sûreté

4) Quels sont les trajets (ville de départ et d'arrivée) effectués par les Airbus A380 (type d'avion).

5) Quels sont les pilotes qui ont la chance d'aller à Honolulu ?

6) A quels pilotes ont été affectés les vols sur des avions de plus de 600 passagers : on donnera le nom des pilotes, leur numéro et le nombre d'heures de vol.

Exercice 5.13. — SQL spoken, Prolog spoken : interrogation (objectif 5.XX)

Nous avons vu en cours 4 manières différentes d'interroger une base de données : algèbre relationnelle, requête SQL, règle conjonctive en Prolog et formule conjonctive. Autant dire que vous êtes devenus polyglottes en bases de données. Dans cet exercice, nous considérerons une fois encore la base de données COP21 avec une table unique vue précédemment. Donnez à chaque fois la traduction dans les langages d'interrogation manquants des requêtes ci-dessous :

a — Requête SQL SELECT Nom, Pays, Continent FROM Cop21

b — Algèbre relationnelle $\sigma_{\text{Continent = 'Afrique'}}(\text{COP21})$

c — Formule conjonctive

$$\exists nb\ \exists cap\ \exists co2\ \text{Cop21}(\text{Paris},\text{France}, cap,\text{Europe},an,nb,co2) \wedge (co2 < 6000)$$

Exercice 5.14. — BD notes : interrogation (objectif 5.XX)

On considère une base de données qui définit les groupes d'étudiants qui ont rendu un projet, ainsi de la note qu'ils ont reçue pour ce travail. Cette base de données est définie par les deux relations suivantes :

- Etudiant(Prenom,Nom,Age,Pays,Bac,Id_etud) qui décrit les informations concernant chaque étudiant (Id_etud > 0).
- Groupe(Id_etud1,Id_etud2,Note) qui décrit les groupes de projet Prolog et la note obtenue. Dans le cas où un projet a été rendu par un(e) étudiant(e) unique, l'identificateur Id_etud2 sera mis à 0. Les étudiants qui n'ont pas rendu de projet ne sont dans aucun tuple de cette seconde relation.

On demande de donner l'expression des requêtes sur la base de données ci-dessous, dans le langage de requête considéré :

a — SQL *Donner la liste des étudiants(Nom,Prenom,Age) encore mineurs*

b — SQL *Donner la liste (Nom, Prénom) des étudiant-e-s qui ont eu le courage de travailler seul(e)s*

c — Algèbre *Donner la liste des mentions de bacs des étudiants français*

d — Algèbre *Donner la note de Luky Luke, l'étudiant solitaire qui a composé seul*

e — Règle conjonctive *Donner la liste (prénom, nom, note) des résultats de tous les étudiants qui ont rendu un projet*

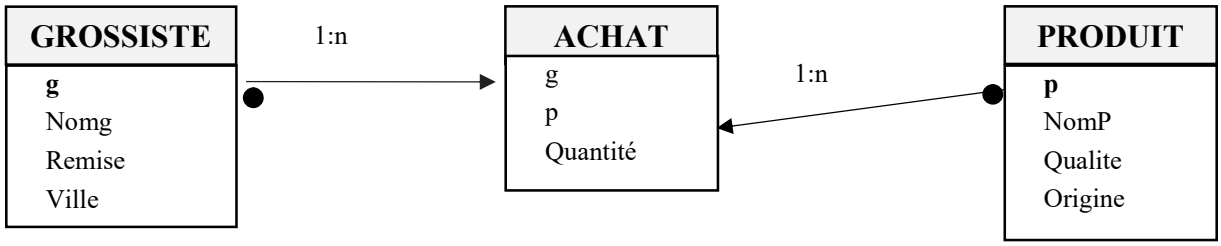
Problèmes

Problème 5.1 — BD Produits en Bretagne : interrogation (objectifs XXX)

Dans l'imaginaire collectif, la Bretagne est le plus souvent reliée à la forêt de Brocéliande et aux pures forêts magiques peuplées d'enchanteurs, de féees et de korrigans. Au-delà de cette doxa Arthurienne, la Bretagne, c'est parfois aussi la région de l'agriculture intensive première productrice de porcs dont le lisier sature les rivières des nitrates qui sont à l'origine de véritables marées d'algues vertes sur la façade septentrionale du littoral breton. Une réalité, que défendent sans le dire les bonnets rouges anti-ecotaxe et qu'a épinglée Pétillon dans sa dernière BD « Palmer en Bretagne » et qui va faire le cadre de cet exercice.

Propriétaire de la seule épicerie de l'ile Molène, dans l'archipel d'Ouessant, Katel Avendre tient fermement à ne proposer dans son établissement que des produits locaux. Fidèle adhérente de l'association « Produit en Bretagne », elle milite en effet pour la relocalisation de l'économie en dépit de l'image parfois déplorable qu'on les produits bretons en matière environnementale. Pour s'assurer de la provenance des produits qu'elle commercialise, elle a décidé de concevoir une base de données où seront mémorisés tous ses achats.

Son épicerie, et donc sa base de données, stocke don des produits ayant chacun un nom, une marque de qualité (bio, label rouge,…) et une ville d'origine. Chaque produit est identifié par un numéro de code. Ces produits sont achetés auprès de grossistes ayant chacun un nom, résidant dans une ville et pratiquant une certaine remise. Chaque grossiste est identifié par un numéro de code. L'achat d'un produit donné se traduit, par l'existence d'une quantité, non nulle, en stock. On donne graphiquement le schéma relationnel de la base de données :



Chaque relation présente dans le schéma ci-dessus peut être définie comme suit :

Grossiste Relation regroupant les informations sur chaque grossiste. Structure de la relation :

Attribut	Commentaires	Type de données
G	Code représentant chaque grossiste	chaîne de caractères
Nomg	Nom du grossiste	chaîne de caractères
Remise	% de remise accordé par le grossiste sur vente en gros	numérique
Ville	Ville d'exercice du grossiste	chaîne de caractères

Produits Relation regroupant les informations sur chaque produit. Structure de la relation :

Attribut	Commentaires	Type de données
p	Code représentant chaque produit - clé primaire	chaîne de caractères
NomP	Nom du produit	chaîne de caractères
Couleur	Couleur du produit	chaîne de caractères
Origine	Lieu de production du produit	chaîne de caractères

Achat Relation regroupant les informations sur chaque achat de produits par chaque grossiste. Structure de la relation :

Attribut	Commentaires	Type de données
p	Code du produit acheté	chaîne de caractères
g	Code du grossiste acheteur	chaîne de caractères
Quantite	Quantité (en kg) de produit acheté	numérique

Le contenu de la base de données est donné ci-dessous :

GROSSISTE	***g***	***Nomg***	***Remise***	***Ville***
	g1	Marchand	6	Plouhinec
	g2	Bihan	8	Erdeven
	g3	Martelot	5	Caudan
	g4	Coz	4	Erdeven
	g5	Pêcheur	9	Paimpol
	g6	Frilouz	0	Redon

PRODUIT	***p***	***NomP***	***Couleur***	***Origine***
	p1	carottes	rouge	Plouhinec
	p2	cocos	blanc	Paimpol
	p3	oignons	brun	Erdeven
	p4	tomates	rouge	Paimpol
	p5	artichauts	vert	St Pol de Léon
	p6	kiwis	vert	Caudan
	p7	marrons	brun	Redon

ACHAT	***g***	***P***	***quantité***
	g1	p1	1
	g1	p4	1
	g1	p5	8
	g1	p6	2
	g2	p2	1
	g2	p4	1
	g3	p2	5
	g4	p4	2
	g5	p3	10
	g5	p2	4
	g5	p4	8

Question 1 — Donnez le schéma nommé de chaque relation, sous forme de produit cartésien de domaines associés à des noms d'attributs.

Question 2 — Quelles sont les attributs qui jouent (ou peuvent jouer) le rôle de clé primaire sur ces relations. Quelles sont les clés étrangères dans la base de données.

Question 3 — Définissez la base de données sous forme logique à l'aide de faits Prolog. On se contentera de modéliser un ou deux tuples par relation.

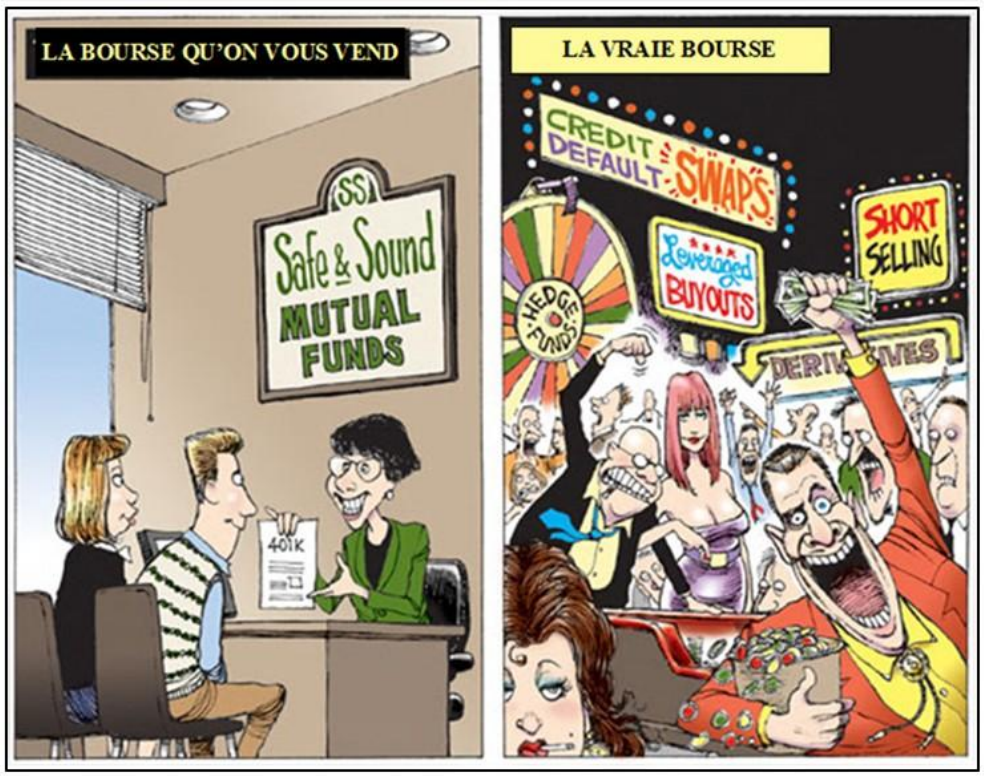
Question 4 — Donnez l'occurrence de relation correspondant au résultat des requêtes suivantes. A chaque fois, on précisera à quel opérateur de l'algèbre relationnelle correspond la requête :

a) liste des villes d'où sont originaires les produits
b) ensemble des informations sur grossistes de Erdeven
c) nom (uniquement) des produits originaires de Paimpol
d) informations sur les produits (nom, couleur, origine, quantité) achetés au grossiste Martelot.

Question 5 — Donnez la traduction en algèbre relationnelle, puis sous forme logique (Prolog) des requêtes précédentes.

Problème 5.2 — Base de Données *Bourse* : modélisation (objectifs XXX)

L'informatisation des différentes places boursières mondiales, ainsi que leur mise en réseau, ont entraîné la constitution de gigantesques bases de données fourmillant d'informations et consultables à tout moment sur Internet. C'est grâce à ces bases de données que se développe le trading haute fréquence, activité purement automatique ou des opérateurs de marché virtuels peuvent exécuter des opérations d'achat et vente d'une durée que quelques microsecondes. En déplaçant son data center de quelques kilomètres, la bourse de Londres a ainsi gagné quelques microsecondes, et donc des parts de marchés sur ce domaine de transactions virtuelles totalement déconnectées de la réalité économiques des entreprises concernées…



Dans le cadre de cet exercice, nous allons considérer une petite base de données consacrée précisément au domaine boursier. Cette base de données concerne l'ensemble des places boursières mondiales et donne plusieurs types d'informations sur les indices, transactions et valeurs qui y sont cotées. Du fait du trading haute fréquence, ces informations peuvent être renouvelées plusieurs millions de fois par seconde. Pour une valeur donnée, il y aura donc des millions d'enregistrement dans la base pour une seconde donnée de transactions…

Attribut	**Commentaires**	**Type de données**
Date	Date de cotation	format date jj-mm-aaaa
Horaire	Heure de cotation	entier : microsecondes depuis 00:00:00
Valeur	Code de la valeur (nom de la société)	chaîne de caractères
Pays	Pays de la société cotée	chaîne de caractères
Resultat	Résultat financier de la société cotée au cours de l'exercice de l'an passé	réel
Marche	Code du marché correspondant (exemple : NYSE pour Wall Street)	chaîne de caractères
Surete	Solidité du marché correspondant (exemple : AAA+)	chaîne de caractères
Varia_M	Variation du marché depuis le début de l'année	pourcentage
Cote	Cours courant de la valeur	réel
Volume	Volume d'actions de la valeur considérée échangées depuis l'ouverture de la journée	réel
Varia_jr	Variation par rapport au jour précédent	pourcentage
Varia_an	Variation depuis le début de l'année	pourcentage

Question 1 — Pour le moment, cette base de données est présentée sous la forme d'un immense tableau unique. Or, celui-ci présente d'indéniables redondances qu'une décomposition relationnelle pourrait faire disparaître. Donnez donc un schéma de base de données qui élimine toutes les redondances. On donnera pour cela le schéma nommé de chaque relation, sous forme de produit cartésien de domaines associés à des noms d'attributs.

Question 2 — Quelles sont les attributs qui jouent le rôle de clé primaire et de clés étrangères dans votre base de données ?

Question 3 — Définissez la base de données sous forme logique à l'aide de faits Prolog. On se contentera d'inventer un ou deux tuples par relation.

Question 4 — Donnez l'occurrence de relation correspondant au résultat des requêtes suivantes. A chaque fois, on précisera à quel opérateur de l'algèbre relationnelle correspond la requête :

a) liste des valeurs (toutes informations) cotées à la bourse de Paris

b) liste des noms de valeurs et de leurs pays

c) liste des valeurs (nom, marché cote, variation annuelle) qui ont connu une progression depuis le début de l'année.

d) Les variations de la bourse sont-elles cohérentes avec le résultat des entreprises ? Pour répondre à cette question, donnez l'expression de la requête qui fournit la liste des valeurs dont la cote courante est en progression depuis le début de l'année tout en ayant eu un résultat positif lors du dernier exercice fiscal.

Question 5 — Donnez la traduction en algèbre relationnelle, puis sous forme logique (Prolog) des requêtes précédentes.

Exercice 5.15. — Evaluation de requêtes conjonctives (objectif 5.XX ; d'après P. Marcel)

On considère une base de données D de (R,S,T,U). On donne ci-dessous une instance I de cette base

$I(R)$

A	B
1	2
4	2

$I(S)$

B	C
2	3
2	5

$I(T)$

A	B
1	2
2	3

$I(U)$

A	D
1	2
3	4

1. On considère maintenant les requêtes conjonctives suivantes

a) `Rep1(y,3) :- R(1,y).`

b) `Rep2(1,2) :- R(x,y).`

c) `Rep3(x,y) :- R(x,2), S(4,5).`

d) `Rep4(x,y,z) :- T(x,y), U(x,y), S(y,z)`

e) `Rep5(z) :- R(x,y), S(y,z), U(z,x).`

f) `Rep6() :- S(x,y), T(y,z).`

Donnez pour chacune d'elle la formule conjonctive qui correspond à la requête, ainsi que l'image de la requête, en précisant bien les interprétations qui mènent au résultat

2. On considère maintenant les requêtes exprimées à l'aide des formules conjonctives suivantes :

a) `∃y U(x,y)`

b) `∃y S(x,y)`

c) `U(x,y)`

d) `∃z (T (y,x) ∧ T(x,z)`

e) `∃y∃z∃w R(x,y) ∧ S(y,z) ∧ T(w,z) ∧ U(x,w)`

f) `∃z (T (x,y) ∧ T(y,z))`

g) $\exists z \exists w \ (U(x,y) \wedge S(z,w))$

Pour chaque formule,
- donnez la liste des variables libres de la formule
- déduisez-en l'expression de la requête conjonctive correspondante
- donnez alors l'image de la requête sur l'instance considérée

Exercices de cours : réponses aux questions

Chapitre 1. INTRODUCTION

Question 1.1 — Peut-on réellement croire un prof de maths ? (objectif 1.1.2)

1 — *Pour être un théorème il suffit qu'un énoncé mathématique soit vrai*

La réponse est fausse : on dit qu'une proposition logique est un théorème si on arrive à la prouver par déduction formelle (approche syntaxique) à partir d'un ensemble d'axiomes, indépendamment de toute notion de vérité. (approche sémantique). Un théorème n'est également une vérité que si le système logique considéré est sain (ou consistant).

2 — *Un théorème peut être faux*

La réponse est vraie : si un système logique n'est pas sain, on peut très bien construire des théorèmes qui ne sont pas des tautologies. Nous avons donné dans le cours l'exemple des géométries non euclidiennes.

Question 2 — La vérité est toujours relative (objectif 1.1.5)

On considère un système logique qui est sain (consistant) mais pas complet**.** Que pensez-vous des deux affirmations suivantes, prises globalement :

1. *Dans ce système, tout théorème est vrai*

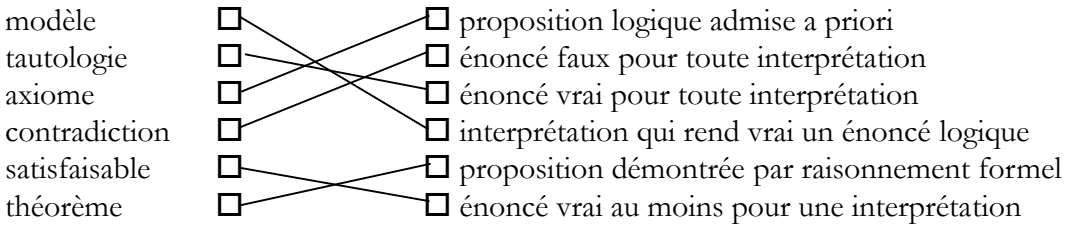
La réponse est vraie : dans un système sain, tout théorème établi formellement correspond également à la conséquence sémantique des axiomes à la base de sa démonstration : les règles d'inférence formelle étant dans cas également des conséquences logiques, on a une propagation de la vérité au fil de la démonstration formelle.

2. *Dans ce système, tout ce qui est vrai n'est pas forcément un théorème*

La réponse est vraie : il faut qu'un système soit complet pour que toute vérité puisse être démontrée formellement comme théorème. Vous avons étudié dans le cours le théorème d'incomplétude de Gödel qui affirme que toute vérité de l'arithmétique ne peut être démontrée comme conséquence formelle (donc comme théorème) des axiomes d'un système logique donné.

Question 1.3 — b.a.ba logique (objectifs 1.1.3. & 1.1.4)

On a les définitions suivantes



Chapitre 2. LOGIQUE DES PROPOSITIONS

Question 2.1 — Formules bien forées (objectif 2.1.2)

a. ⌉⌉P bien formé : partant de P, on nie cet atome deux fois successives

b. ⌉Q ∧ P bien formé : on nie Q et ensuite la fbf obtenue est en conjonction avec P

c. ⌉(Q ∧ P) bien formé : on construit déjà la conjonction de Q et P avant de nier la fbf obtenue : on met les parenthèses avant pour avoir la bonne priorité

d. ⌉QQ ∧ PP n'est pas une fbf. bien formé. On ne peut accoler deux atomes (comme dans QQ sans connecteurs entre eux). Rien ne nous dit ici que l'on a oublié de mettre un connecteur entre des atomes P ou Q. PP et QQ sont simplement deux atomes !

e. Q ∧ ∧ P ce n'est pas une fbf : on ne peur avoir deux connecteurs ainsi accolés

f. P ⌉Q ce n'est pas une fbf : partant de Q, on construit ⌉Q mais la négation étant un connecteur unaire (qui ne s'applique qu'à un terme), elle ne peut relier P.

Question 2.2 — Traduction et parenthèsage (objectifs 2.1.2 & 2.2.1)

a. *Il est riche et* (*mais*) *il n'est pas beau* `Riche ∧ ⌉ Beau`

b. *Il n'est pas beau et riche à la fois.* `⌉(Riche ∧ Beau)`

c. *Il n'est ni riche ni beau* `⌉Riche ∧ ⌉Beau ≡ ⌉(Riche ∨ Beau)`

Question 2.3. – Tables de vérité (objectif 2.2.2.)

1. (P ⇒ (Q ⇒ R)) ⇔ ((P ∧ Q) ⇒ R) est satifaisable. La formule est vraie pour toutes les interprétation, sauf pour le cas P vrai, Q vrai et R faux.

2. Vous devez trouver des tables de vérités identiques pour les deux formules.

3. L'énoncé par Vert ⊕ Rouge suppose que le feu soit au moins rouge ou vert, d'après la table de vérité du ou exclusif. Or, on peut imaginer une situation où le feu est éteint, et dont n'est ni rouge ni vert.

Question 2.4. Système complet de connecteurs (objectif 1.2.3.)

1. Pour montrer que {∨,⌉} est un système complet en partant du système {∨, ∧,⌉}, il faut exprimer le connecteur de conjonction ∧ en fonction des connecteurs {∨,⌉}. En partant de la règle de De Morgan, il vient immédiatement : P ∧ Q ≡ ⌉(⌉P ∨ ⌉Q). La démonstration de la complétude de l'ensemble {∧,⌉} suit le même principe.

2. Pour montrer que {↑} est un système complet en partant du système {∨, ∧,⌉}, il faut exprimer chaque des connecteurs de {∨, ∧,⌉} avec uniquement ↑. L'idée est de partir de la négation en notant que : A ≡ A ∧ A donc ⌉A ≡ ⌉(A ∧ A) ≡ A ↑ A. On utilise ce premier résultat, et la règle de De Morgan pour trouver l'expression de A ∧ B et A ∨ B en fonction de ↑. Il en va de même pour l'ensemble complet {↓}.

Question 2.5. Mise sous forme normale (objectif 2.2.4.)

1. Les deux formules sont équivalentes puisqu'on arrive à la même forme normale disjonctive par équivalence : ⌉P∨⌉Q ∨R. Notons que cette forme normale disjonctive est également la forme conjonctive de la formule. Nous sommes ici dans un cas limite de forme conjonctive comme l'exemple A ∨ ⌉D donné dans le cours.

2. La mise sous forme normale de la formule donne ((⌉P∨Q)∧⌉P)∨P. Il faut voir l'absorption

$(\neg P \vee Q) \wedge \neg P \equiv \neg P$ pour obtenir comme formule $\neg P \vee P \equiv \mathcal{V}$. La formule est donc bien tautologique.

3. Les deux formules sont équivalentes puisqu'on arrive à la même forme normale disjonctive (ou conjonctive) en appliquant les formules d'équivalence.

Question 2.6. Fonctions booléennes (objectif 2.2.4.)

A faire. Exercice hors programme.

Question 2.7. Fonctions booléennes (objectif 2.2.4.)

A faire. Exercice hors programme.

Question 2.8. Fonctions booléennes et méthode de Quine-Mc Cluksey(objectif 2.2.4.)

1. Il y a 3 impliquants premiers $a \wedge b \wedge e$, $a \wedge \neg c \wedge e$ et $b \wedge c \wedge e$.
2. $\mathcal{F} \equiv (a \wedge \neg c \wedge e) \vee (b \wedge c \wedge e)$.

Question 2.9. Résolution (objectif 2.2.6.)

1. Pour montrer qu'une formule est une tautologie par résolution, il faut montrer que sa négation est contradictoire, puisque la résolution ne peut démontrer que le caractère contradictoire d'une formule. On obtient par négation la formule clausale suivante $P \wedge Q \wedge (\neg P \vee \neg R) \wedge R$. L'application de la résolution entre la clause P et la clause $\neg P \vee \neg R$ donne $\neg R$. On obtient la clause vide directement en faisant la résolution entre cette résolvante et la clause R.
2. Ici, on applique directement la résolution pour montrer si la formule est contradictoire ou non. Si elle n'est pas contradictoire, elle sera au minimum satisfaisable. Il n'y a effectivement pas contradiction, on ne peut arriver à la clause vide. On peut aussi démontrer que la formule n'est pas contradictoire en cherchant un modèle de cette dernière. Il suffit par exemple de prendre P vrai et Q faux.
3. Il faut donc vérifier si $P \Rightarrow Q$ est la conséquence logique de $P \Leftrightarrow Q$. On construit donc la formule de réfutation $(P \Leftrightarrow Q) \wedge \neg(P \Rightarrow Q)$. On choisit la bonne formule d'équivalence pour avoir une forme normale conjonctive en transformant $(P \Leftrightarrow Q)$. Soit :

   $(P \Rightarrow Q) \wedge (Q \Rightarrow P) \wedge \neg(P \Rightarrow Q) \equiv (\neg P \vee Q) \wedge (\neg Q \vee P) \wedge P \wedge \neg Q$

   L'application de la règle de résolution entre $\neg P \vee Q$ et $\neg Q$ donne la résolvante $\neg P$, qui va tout de suite donner la clause vide avec la clause P.