M. MALEK & Ch. BASKIOTIS

PROLOG

EISTI, 2002-03

PROLOG(UE)

`Prolog` est un langage de programmation issu de la logique computationnelle. Son nom vient de « PROgrammation LOgique », son fondateur est Alain Colmerauer de l'université de Marseille et son année de naissance 1973. Comme langage de programmation, `Prolog` utilise un formalisme différent des autres langages de programmation pour l'écriture des programmes et la définition des spécifications, car il est de nature différente.

En effet on considère, en général, que le travail de l'informaticien est de construire un programme qui représente l'organisation et le fonctionnement d'un système complexe d'informations. Pour ce faire il dispose de plusieurs langages de programmation qu'on peut classer en trois catégories :

(i) LANGAGES PROCÉDURAUX OU IMPÉRATIFS : Le programme écrit dans un tel langage doit indiquer à l'ordinateur les actions à faire pour obtenir le résultat voulu. Il s'agit donc des langages de bas niveau, la programmation se limitant à passer des ordres à l'ordinateur afin que ce dernier puisse effectuer la suite des actions désirées. Ainsi il y a, de la part de l'informaticien, un travail considérable à fournir pour traduire les spécifications d'un système complexe d'informations en un programme qui marche. Des langages de telle nature sont tous les langages de 3e génération (`Fortran, Pascal, C, Ada`, etc).

(ii) LANGAGES FONCTIONNELS : Le programme écrit dans un tel langage comporte une suite d'appels à des fonctions. Comme on le sait, le retour d'une fonction appelée fournit au programme qui fait cet ap-

pel, une valeur. La suite de ces valeurs doit nous permettre d'obtenir le réultat voulu. Ces langages sont des langages de niveau intermédiaire car la programmation est pensée en termes des valeurs calculées et non pas, comme précédement, en termes de comportement. Il y a donc pour l'informaticien moins de travail à effectuer, car les valeurs font partie des spécifications d'un système complexe d'informations. `Lisp`, `Simula`, `ML` et `Caml` sont des langages de cette nature.

(iii) LANGAGES RELATIONNELS OU DÉCLARATIFS : Le programme écrit dans un tel langage comporte des définitions des relations entre différentes entités du système d'information. Nous pouvons donc le considérer comme une déclaration de l'existence de ces relations. Le résultat voulu est obtenu par la mise en fonctionnement de ces relations. Il s'agit d'un langage de haut niveau, dans la mesure où les spécifications d'un système complexe d'informations sont faites selon une approche relationnelle. Le travail donc, que doit fournir l'informaticien est minime. Par conséquent ces programmes sont faciles à construire, faciles à comprendre, faciles à tester, faciles à maintenir et faciles à adapter pour satisfaire à d'autres buts. `Prolog` est un langage de cette catégorie.

Que l'industrie du logiciel soit quasi monopolisée par l'utilisation des langages procéduraux est la conséquence du fait que les ordinateurs ont une architecture de type von Neumann[1]. machine. En effet un langage de programmation est un intermédiaire – on aurait pu dire une interface – entre l'homme et l'ordinateur. Il permet à l'homme d'expliquer à la machine ce qu'il souhaiterait qu'elle fasse. Donc avec un langage de programmation on doit pouvoir faire l'une de deux choses :

— soit établir la structure de l'ordinateur. Ce que nous faisons avec tous les langages procéduraux. Dans ce cas le programmeur doit obtenir une correspondance entre le modèle de l'ordinateur et le modèle du problème. Ce travail n'est pas intrinsèque au langage de programmation. Il s'agit d'une tâche qui, pour le même problème, doit se faire quasiment de la même façon pour tout langage procédural, à l'extérieur du travail de la programmation – et même avant celui-ci. Ainsi avec un langage procédural il est difficile d'écrire un programme et encore plus difficile

[1] Les ordinateurs à architecture von Neumann ont un procédé de traitement de l'information fondé sur le cycle « fetch-execute », à savoir une boucle qui passe successivement par les trois états suivants :

— activer l'instruction prochaine;
— décoder l'instruction;
— exécuter l'instruction.

de le maintenir, ce qui explique assez bien la raison pour laquelle on a créé ce qu'on appelle *industrie du logiciel*.

— soit établir la structure du problème. Pour ce faire nous devons avoir un modèle du monde ou, de façon plus modeste, de la partie de l'univers (du micromonde, comme on disait jadis) dans lequel ce situe notre problème. C'est l'approche utilisée les langages de deux autres catégories. Pour le `Lisp`, par exemple, toute action peut s'exprimer à l'aide d'une fonction, toute donnée peut être codée à l'aide d'une liste. Donc pour le `Lisp` tous les problèmes sont réductibles à des fonctions et des listes. La manière dont les fonctions et les listes sont prises en compte par l'ordinateur n'est pas une préoccupation pour le programmeur. De même pour `Prolog` tout problème peut se ramener à un calcul de la valeur de vérité d'une suite des prédicats.

De ce qui précède découle qu'un avantage des langages déclaratifs est le fait qu'ils obligent le programmeur d'établir une démarche algorithmique indépendante des caractéristiques technologiques et de l'architecture matérielle de l'ordinateur sur lequel le programme s'exécutera. `Prolog` en particulier utilise comme élément de base de la programmation la notion du prédicat. Sa définition inclut les arguments d'entrée et de sortie et laisse donc au programmeur seulement le soin de déterminer le résultat souhaité – la sortie – et les paramètres – les entrées – qui permettent d'obtenir ce résultat, sans qu'il soit préoccupé d'écrire des instructions détaillées concernant la démarche pour arriver au résultat.

Les langages de programmation fondés sur la logique computationnelle ont en commun :

— l'utilisation des faits du domaine de connaissance comme des données;

— l'utilisation des règles pour exprimer les relations entre les faits;

— l'utilisation de la déductions pour répondre à des questions concernant le domaine de connaissance particulier.

Leurs différences par rapport aux langages procéduraux ce concentrent essentiellement à l'absence de contrôle du flux des instructions à exécuter. Ainsi il n'y a pas dans la programmation logique les différents blocs conditionnels qu'on trouve dans les autre langages. Par exemple il n'existe pas le bloc `if - then - else`, ni des boucles `while` ou `repeat`. De plus il n'y a pas des variables globales ni des états qui se modifient globalement.

`Prolog` permet d'utiliser la logique pour traiter des informations avec un ordinateur, i.e. d'utiliser la logique comme un langage de programma-

tion. L'idée qui était dominante à l'époque de la première apparition de `Prolog` peut se résumer à la fameuse formule de N.Wirth[2]

Algorithmes + Structure de données = Programmes

qui fournissent aux programmes une comportement dynamique en ce sens que la conséquence d'un programme est le résultat de.l'intervention d'un algorithme dans un ensemble de données doté d'une structure. En même temps cette formule établit la séparation entre algorithmes et structure de données.

P.Hayes, à la même époque, pensait que la computation était de la déduction contrôlée, tandis que E.F.Codd envisageait les systèmes de bases de données comme étants composés de deux éléments : un premier élément relationnel qui déterminait la structure logique des données et un second élément de contrôle qui permettait le stockage et le traitement des données. R.Kowalski, en partant de ces idées, a établi[3] une formule analogue à celle de Wirth et qui constitue aujourd'hui la thèse principale de la programmation logique, à savoir

Algorithme = Logique + Contrôle

Cette formule établit la séparation entre la logique, qui contient les spécifications des données, et le contrôle, qui établit l'ordre dans lequel les opérations logiques doivent s'effectuer. Si donc on enlève des programmeurs la tâche de spécifier la composante « contrôle » de la formule, les programmes logiques acquièrent une structure statique car il contiennent la connaissance la plus appropriée à utiliser mais ils n'explicitent pas les méthodes qui permettraient de l'explorer. Par exemple `Prolog` exécute la partie « Contrôle » de l'algorithme automatiquement. Il ne laisse donc à la charge du programmeur que la partie « Logique ».

L'utilisation industrielle de `Prolog` est en dents de scie. Il a été le langage choisi par les japonais dans leur tentative de construire l'ordinateur de la 5e génération. Le projet a échoué – pour des raisons financières et politiques – mais `Prolog` n'est pour rien. Récemment il a été utilisé pour la programmation de l'énorme base de données du projet du génome humain avec beaucoup de succès. `Prolog` souffre essentiellement du fait qu'il n'est pas connu. Il y avait un espoir de rendre `Prolog` moins « ovni-esque » avec l'introduction dans le primaire à la fin des années 80 du langage `Logo`, qui à bien des égards ressemble beaucoup au `Prolog`, mais quelques années après,

[2] « inventeur » des langages de programmation à forte dominante algorithmique, `Algol` et `Pascal`.
[3] in Communications ACM, vol.22, no 7, july 1979, pp.424-436

des esprits supérieurs du ministère de l'Éducation Nationale ont troqué le `Logo` contre le `Basic` ! Ainsi l'abrutissement des élèves du primaire était garanti et la méconnaissance de Prolog assurée.

Le cours de `Prolog`, en deuxième année de l'EISTI, se place en même temps que le cours de la Logique Computationnelle qui permet aux élèves d'étudier les fondements théoriques de la programmation logique. L'objectif du cours est de permettre aux élèves d'apprendre à écrire des programmes en `Prolog`, ce qui, d'une part, leur permettra de maîtriser un langage très puissant de mise en œuvre des maquettes des programmes et, d'autre part, leur sera utile et nécessaire pour aborder les deux cours qui vont suivre et qui font partie du cycle des cours d'IA, à savoir Intelligence Artificielle et Systèmes Experts.

0.1 RÉFÉRENCES

Pour compléter l'enseignement, les élèves pourraient utiliser soit le livre de
W. F. CLOCKSIN - C. S. MELLISH : *Programmer en Prolog*, Éditions Eyrolles, traduction en français du livre *Programming in Prolog* aux éditions Springer-Verlag,
soit le livre de
J. ELBAZ : *Programmer en Prolog*, Éditions Ellipses, 1991
soit encore le livre de
I. BRATKO : *Prolog, Programming for artificial intelligence*, Addison-Wesley, 1986
dont il existe aussi une traduction française.
Les élèves qui souhaiteraient, en plus, avoir un livre qui regroupe les fondements de la programmation logique et les techniques de base du langage de programmation Prolog, peuvent utiliser le livre de
U. NILSSON - J. MAŁUSZYNSKI : *Logic, Programming and Prolog*, Wiley, 1990.
Les élèves qui voudraient, après la fin de ce cours, approfondir leurs connaissances en Prolog, peuvent consulter le livre de
L. STERLING - E. SHAPIRO : *L'art de Prolog*, Éditions InterÉditions,
et dont tous les exemples de programmes sont sur Internet, en libre accés à l'adresse du MIT : `ftp://mitpress.mit.edu`.
On peut aussi citer trois autres livres introductifs qui contiennent quelques applications interéssantes :
J. MALPAS : *Prolog : a relational language and its applications*, Prentice-

Hall, 1987
C. MARCUS : *Prolog programming*, Addison-Wesley, 1986
J. STOBO : *Problem solving with Prolog*, Pitman, 1989

ainsi qu'un livre plus orienté ingénierie logicielle

T. AMBLE : *Logic programming and knowledge engineering*, Addison-Wesley, 1987.

Signalons encore un livre récent qui pourrait être utile au ptogrammeur :

W. F. CLOCKSIN : *Clause and Effect : Prolog programming for the working programmer*, Springer, 199 7.

Enfin le manuel de référence du langage peut se trouver dans les livres :

P. DERANSART, A. ED-DBALI, L. CERVONI : *Prolog : The Standard*, Springer-Verlag, 1986.
R. O'KEEFE : *The draft of Prolog*, MIT Press, 1990.

1

LA LOGIQUE DES PROPOSITIONS COMME LANGAGE DE PROGRAMMATION

1.1 Les faits 8
1.2 Les règles 9
1.2.1 Les questions 10
1.3 Les données 10
1.3.1 Les données simples 11
1.3.2 Les données structurées 11
1.4 Exécution 13
1.5 Un exemple 13
1.6 Exercices 14

COMME nous venons de voir, le langage de programmation `Prolog` est fondé sur la logique mathématique afin de stocker et traiter des informations sous forme symbolique. Ces informations symboliques sont issues de la modélisation des connaissances que possède un être intelligent. Afin d'effectuer cette modélisation, la connaissance est décomposée en différentes parties qui sont considérées comme étant autonomes et indépendantes, bien que c'est rarement le cas. Chacune de ces parties, qui, en règle générale, est relative à un environnement donné, constitue ce qu'on appelle un univers du discours et chaque fois, pour résoudre un problème, on travaille à l'intérieur d'un univers du discours spécifique que l'on notera $\mathcal{U}$.

Une manière répandue et commode pour modéliser les connaissances d'un univers du discours est d'utiliser le triplet *objet – attribut – valeur*. Par *objet* on entend les objets ou entités physiques ou conceptuelles de l'univers du discours. On utilise un *attribut* pour décrire les caractéristiques et/ou les propriétés des objets et aussi leurs relations. La *valeur*, enfin, est obtenue par la spécification d'un attribut pour un objet particulier.

La programmation en `Prolog` se fait à l'intérieur d'un univers du discours. Comme `Prolog` est un langage relationnel, ses objets sont essentiellement des relations entre les objets de l'univers du discours. Ainsi la programmation en `Prolog` consiste

— en la spécification des *faits* concernant les objets,
— en la construction des *règles* concernant les relations entre objets,
— en la réponse à des *questions* posées concernant l'existence des objets et/ou les relations entre objets.

Les faits, règles et questions sont des *expressions logiques*. D'autre part une *constante* – qui est le nom d'un objet ou d'une relation entre objets – est un *terme logique*. Le nom d'une constante en `Prolog` doit commencer toujours par une lettre minuscule. Un terme logique est aussi une *variable* qui peut représenter n'importe quel objet. Le nom d'une variable en `Prolog` doit toujours commencer par une lettre majuscule.

1.1 Les faits

Les faits en `Prolog`

— soit affirment l'existence d'un objet particulier avec des caractéristiques particulières. Exemple : toto est un cube de couleur bleue, ce qui en `Prolog` donne : `cube(toto, bleue).`, ce qui a comme conséquence que le fait en question a la valeur de vérité 1 - vrai,
— soit fournissent la valeur – numérique ou symbolique – de la caractéristique d'un fait. Exemple : le volume du cube toto est 100 ce qui en `Prolog` donne : `volumeCube(toto, 100).`.

La forme générale d'un fait est la suivante:

$$\texttt{nomFait(objet}_1\texttt{, objet}_2\texttt{, ... , objet}_n\texttt{).}$$

Le point « . » avec lequel termine le fait, indique la fin du fait.

Les différents objets qui interviennent à la déclaration d'un fait ce sont les arguments du fait. Remarquons que deux faits de même nom peuvent ne pas faire référence à la même relation. Cela dépend du nombre d'arguments. Si le nombre d'arguments est le même, les deux faits font référence à la même relation éventuellement avec des objets différents. Si par contre le nombre d'arguments n'est pas le même, alors les deux faits se rapportent à deux relations différentes.

Nous pouvons composer les faits entre eux en utilisant les connecteurs logiques « $\wedge$ - *et* » et « $\vee$ - *ou* ». La conjonction « *et* » est notée en `Prolog` par la virgule « , » et la disjonction « *ou* » par le point-virgule « `;` »[1].

Avec beaucoup de précautions, nous pouvons considérer que les faits en `Prolog` jouent le même rôle que les données pour un langage de 3e génération. Cette analogie permet aussi de comprendre qu'un fait, dans la mesure ou il est une donnée, il ne peut pas être inconnu et, donc, il ne peut pas être représenté par une variable, c'est-à-dire avoir un nom qui commence avec une lettre majuscule. Ainsi `cube(toto, bleue, 10).` est un fait legal en `Prolog`, tandis que `X(toto, bleue, 10).` qui exprime une relation inconnue pour le cube `toto` est incomprehensible pour `Prolog`.

De ce qui vient d'être dit, on conçoit aisément qu'en Logique Computationnelle `nomFait` était appelé soit *prédicat*, soit *foncteur*. En `Prolog`, `nomFait` est toujours un prédicat qui est considéré comme le nom d'une base de données et si les valeurs des arguments se trouvent dans cette base de données, alors la valeur du prédicat `nomFait` est égale à vraie.

1.2 Les règles

Une règle en `Prolog` exprime la manière dont un fait est relié ou découle d'autres faits, c'est-à-dire une règle est ce que, en Logique Computationnelle, on a appelé axiome ou théorème logique.

La forme générale d'une règle est la suivante :

$$\texttt{A:-}\ \texttt{B}_1\texttt{,}\ \texttt{B}_2\texttt{,}\ \ldots\ \texttt{,}\ \texttt{B}_n$$

où `A` et $\texttt{B}_k$ sont des faits. `A` est *l'entête* de la règle et il doit être un atome positif et $\texttt{B}_1\texttt{,}\ \texttt{B}_2\texttt{,}\ \ldots\ \texttt{,}\ \texttt{B}_n$ constituent *le corps* de la règle et ce sont des littéraux.

Cette forme de la règle est la forme qu'en programmation logique nous l'avons vu sous le nom de la *clause de Horn*. Nous remarquons qu'un fait est aussi une clause de Horn dont le corps est vide.

[1] Remarquons que le connecteur « $\vee$ - *ou* » ne fait pas partie du dispositif nécessaire pour décrire une clause de Horn. Ainsi une fbf $a \vee b \rightarrow c$ peut se décomposer en deux clauses de Horn $a \rightarrow c$ et $b \rightarrow c$.

Les faits et les règles d'un univers du discours, constituent *la base de données* de cet univers du discours ou, encore, la base de connaissances de l'environnement.

Remarquons pour finir qu'avec beaucoup plus de précautions que ci-dessus, nous pouvons considérer que les règles en `Prolog` jouent le même rôle que les sous-programmes ou les fonctions pour un langage de 3e génération. Si, donc, on tient compte des équivalences établies – avec des precautions – entre `Prolog` et les langages de 3e génération, on peut dire qu'en `Prolog`, programmes et données sont melangées et ne font qu'une entité – la base de données. Cet aspect des choses, i.e. le melange programmes et données, constitue une des grandes particularités des langages de 5e génération. Comme on verra plus tard nous pouvons modifier par programme la base de données. Donc un programme qui est en train de se dérouler, pourrait s'auto-modifier, c'est-à-dire nous pouvons avoir des modifications dynamiques des parties d'un programme qui ne sont pas prévues par son code mais sont dues à la nature des données.

1.2.1 Les questions

La dernière forme de l'expression logique en `Prolog` ce sont les questions. Une question est en réalité posée par l'utilisateur. `Prolog` parcourt sa base de données – i.e. les faits et les règles – afin de vérifier si la question est filtrée soit par les faits qu'il possède (i.e. le fait de la question est filtré par les faits de la base de données), soit par un fait issu par application des règles qu'il possède sur les faits de sa base de données. Ainsi la réponse à une question est conditionnée par la possibilité que la question est ou n'est pas une conséquence logique de la base de données.

Une question a la forme suivante :

$$\texttt{nomRelation(objet}_1\texttt{, objet}_2\texttt{, ... , objet}_n\texttt{)?}$$

On dit aussi qu'une question est un BUT. Remarquons qu'une question est une clause avec une entête vide.

1.3 Les données

Si on se réfère à la Logique Computationnelle, toutes les données en `Prolog` sont des termes. Les différents types de termes que nous avons déjà vus en Logique Computationnelle on les retrouve tout naturellement en `Prolog`, saupoudrés en plus avec des épices propres à ce langage. Ainsi une distinction fondamentale pour les données en `Prolog` s'opère entre les données simples et les données structurées.

1.3.1 Les données simples

`Prolog` est un langage absolument, ou mieux, viscéralement non typé. L'avantage de ce fait est que nul part on ne déclare qu'un élément est un tableau ou un réel ou une chaîne de caractères. On écrit un programme comme on construit un discours sans avoir à établir d'avance quels seront les adjectifs, les verbes ou les noms communs que nous utiliserons. Mais malgré tout, il faut au moins pouvoir faire la distinction entre constantes et variables. En `Prolog` cette distinction se fait de manière implicite. Le nom d'une variable doit toujours commencer par une lettre majuscule, par exemple `Variable`, tandis que celui d'une constante par une lettre minuscule, par exemple `cONSTANTE`.

On distingue les constantes en nombres et atomes. Les nombres peuvent être des entiers ou des réels. Tout ce qui n'est pas nombre et qui commence par une lettre miniscule, peut être considéré comme un atome. Par exemple `toto` ou `av_du_Parc_95000_Cergy` sont des atomes. De même une chaîne de caractères alphanumériques entourée de simple quote « ' » est aussi un atome, e.g. `'Toto', '1,av. du Parc, 95000 Cergy'` ou `'3a'` sont des atomes.

Les variables commencent toujours par une lettre majuscule ou par le caractère « _ ». Il y a aussi la variable anonyme, représentée par « _ » et qui peut être unifiée à n'importe quelle variable ou constante pour laquelle il n'y aura pas dans le programme un usage explicite. Par exemple considérons l'ensemble de règles :

$$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$$
$$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$$

dont la première ne fait pas appel à $\texttt{objet}_3$ et la seconde n'utilise pas $\texttt{objet}_2$. On peut donc les écrire sous la forme :

$$\texttt{nomRelation}(\texttt{objet}_1, \texttt{objet}_2, \_) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$$
$$\texttt{nomFait}(\texttt{objet}_1, \_, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$$

1.3.2 Les données structurées

Les constantes et les variables sont des données brutes, sans aucune structuration. On peut aussi envisager des données structurées, e.g. des données contenues dans des bases de données. Ainsi on peut envisager une base de données contenant des adresses des personnes selon l'exemple suivant : `adresse('Toto', 1, av, 'du Parc', 95000, 'Cergy')` et dont l'explication est évidente. En se référant au cours de la Logique Computationnelle, on constate que `adresse` représente un foncteur. Bien évidem-

ment, dans la mesure où `Prolog` est un langage de logique d'ordre 1, nous pouvons envisager d'avoir comme arguments d'un foncteur d'autres foncteurs. Ainsi on peut aussi écrire `adresse(nom('Toto'), numero(1), rue(av, 'du Parc'), ville(95000, 'Cergy'))` à la place du foncteur précédent et où `nom, numero, rue, ville` sont aussi des foncteurs.

Un autre type des données structurées, sont les prédicats qui, comme nous le savons, expriment des valeurs de vérité concernant des relations. Par exemple le prédicat `couleurRouge (titreLivre, rouge)` est un prédicat qui a la valeur 1 si la couleur du livre `titreLivre` est rouge. Il est possible aussi, selon la remarque précédente, que les arguments d'un prédicat soient des foncteurs. Ainsi la valeur du prédicat `couleurRouge(titre(TitreLivre), couleur(Couleur))` est égale à 1 si la variable `Couleur` est unifiée à la couleur `rouge` et à 0 sinon. Ici `titre` et `couleur` sont des foncteurs.

La distinction, en programmation `Prolog`, entre foncteurs et prédicats est parfois assez subtile et, de toute façon, elle est toujours laissée à la charge du programmeur, i.e. en clair `Prolog` n'est pas capable de distinguer entre prédicats et foncteurs et, pour s'en sortir, il applique à la lettre les règles établies par la logique des prédicats. Ainsi, si on écrit en `Prolog`, la règle

```
couleurRouge(X) :- livre(titreLivre(X),couleur(rouge)).
```

accompagnée des faits :

```
livre(titreLivre(petitLivre), couleur(rouge)).
livre(titreLivre(vert), couleur(verte)).
```

on obtient pour la question `couleurRouge(petitLivre).` la réponse `oui` et pour la question `couleurRouge(vert).` la réponse `non`. Mais, grâce à la particularité de `Prolog` de pouvoir répondre à des questions symétriques, on peut aussi poser la question `couleurRouge(X).` et avoir comme réponse non pas une réponse affirmative, à laquelle il fallait s'attendre du fait que dans la base de données il y a un livre de couleur rouge, mais carrément son titre, à savoir `X=petitLivre`, comme si le prédicat `couleurRouge` était un foncteur. Ainsi si on écrit dans le programme, la ligne supplémentaire :

```
bibliothequeRouge(couleurRouge(X)).
```

le compilateur (ou l'interpréteur) de `Prolog` ne s'aperçoît pas que `couleurRouge(X)` soit un prédicat et en tant que tel ne doit pas apparaître comme un argument. La surprise viendra si on veut connaître tous les livres rouges d'une bibliothèque et, tout naturellement, on pose la question : `bibliothequeRouge(couleurRouge(X)).` Dans ce cas la seule réponse qu'on reçoit est `X` égale au nom de la variable interne avec laquelle `Prolog` a unifié `X`.

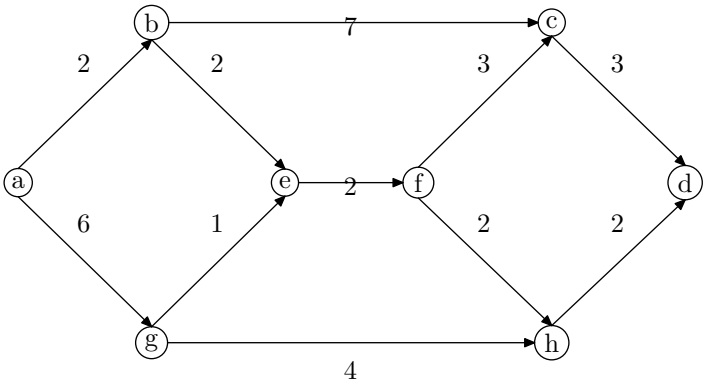
1.4 Exécution

Pour exécuter un programme `Prolog` il faut d'abord le charger dans le fenêtre de travail à l'aide de la commande `?-consult ('nomProgramme.pl').`. Ensuite il faut poser la question.

Afin de répondre à une question posée, `Prolog` est toujours capable de construire, à partir de sa base de données, une arborescence dont la racine est la question posée. Ensuite `Prolog` parcourt cette arborescence en appliquant l'algorithme de résolution SLD, que nous avons vu en Logique Computationnelle, sans toutefois faire la vérification des occurrences. Si, en appliquant cet algorithme, il arrive à « filtrer » la question, alors il affiche le message `yes` et, en fonction de la question posée, le contenu des variables de la question exemplifiées (instanciées) à des constantes. `Prolog` est en mesure de fournir toutes les réponses contenues dans la base de données. Pour les avoir, il suffit, après chaque réponse affichée, de taper la touche «`;` » qui, rappelons-le, en `Prolog` signifie « `ou` ». Si, par contre on veut s'arrêter, alors il faut taper la touche « Retour ».

1.5 Un exemple

Considérons le graphe ci-après :



Si nous utilisons la logique des propositions nous pouvons représenter, en `Prolog` ce graphe par la base de données suivante :

Programme 1.5.1

```
gr(a,b,2).
gr(a,g,6).
gr(b,e,2).
gr(b,c,7).
gr(g,e,1).
gr(g,h,4).
gr(e,f,2).
gr(f,c,3).
gr(f,h,2).
gr(c,d,3).
gr(h,d,2).
```

Chacune de ces clauses représente une proposition qui a une valeur de vérité – vrai ou faux. Si on pose la question

`?-gr(b,c,7)`

alors la réponse est `oui`, c'est-à-dire vrai, tandis que la réponse à la question

`?-gr(b,f,10)`

est bien évidemment `non`, c'est-à-dire faux. Si donc on représente par E l'ensemble de clauses : $E = \{gr(a,b,2), gr(a,g,6), \ \ldots, gr(h,d,2)\}$, alors on obtient une réponse positive à une question composée de la clause q si et seulement si $E \models q$. Il est évident que cette condition est remplie si $q \in \widetilde{I}_E$ qui est le modèle minimal d'Herbrand du programme E. Dans la mesure où le programme E est, dans notre cas, composé des clauses filtrées, nous avons comme modèle minimal d'Herbrand, les prédicats qui font partie des clauses du programme, ce qui en absence des règles donne: $\widetilde{I}_E = E$. Par conséquent les seules fbf qui peuvent être satisfiables par E sont les clauses du programme. On voit ainsi que la logique des propositions est une logique « pauvre » qui ne permet pas d'exprimer toute la richesse du raisonnement humain.

1.6 Exercices

Exercice 1.1 *Examiner la base de données de l'exemple* **??**. *Établir la notion d'un sommet successeur d'un autre. Par exemple le sommet d est successeur du sommet a.*

Exercice 1.2 *Considérons le texte suivant :*

Si Cyclope est un personnage mythologique, alors il est immortel mais s'il n'est pas un personnage mythologique, alors il est mortel. Si Cyclope est

soit immortel, soit mortel, alors il a un seul œil. Cyclope est magique s'il a un seul œil.

(1) Représenter ce texte à l'aide d'une base de connaissances avec des propositions de la logique propsitionnelle.

(2) Donner les modèles pour que Cyclope soit magique si nous faisons l'hypothèse qu'il soit un personnage mythologique.

(3) Écrire en `Prolog` *un programme qui permet de répondre aux questions suivantes :*

(a) Cyclope est-il mythique?

(b) Cyclope est-il magique?

(c) Cyclope a-t-il un œil?

Exercice 1.3 *Soit le problème suivant :*

Trois coureurs Toto, Koko et Lolo portent des maillots de couleur différente et courent ensemble lors d'une course. Nous allons essayer d'établir l'ordre de l'arrivée de ces coureurs. Pour cela nous disposons des informations suivantes :

Toto dit qu'il est arrivé avant le coureur qui porte le maillot rouge. Koko, qui porte le maillot jaune, dit qu'il est arrivé avant le coureur au maillot vert.

Utiliser `Prolog` *pour trouver l'ordre d'arrivée des coureurs.*

2

LA LOGIQUE DES PRÉDICATS COMME LANGAGE DE PROGRAMMATION

2.1 Les prédicats de base de `Prolog` 18
2.1.1 Entrées-sorties 18
2.1.2 Opérations en `Prolog` 18
2.1.3 Prédicats extralogiques 19
2.2 Un exemple 19
2.3 Sémantique de `Prolog` 21
2.3.1 Ordre des clauses 22
2.3.2 Ordre des buts 24

COMME en Logique Computationnelle un programme en `Prolog` qui n'a pas des variables est un programme dont sa sémantique [1] se limite au programme lui-même. Il n'y a donc pas de nouvelles connaissances. Si par contre nous introduisons des variables, alors il est possible d'en déduire des connaissances nouvelles. Ce point peut être vérifier facilement en se reportant à l'exemple du graphe du chapitre précédent. Si notre programme se limite à la base de données qui représente le graphe, nos connaissances se limitent aux successeurs et prédécesseurs immédiats de chaque sommet. Si nous introduisons des variables qui permettent de définir des nouveaux prédicats, alors nous pouvons avoir des connaissances supplémentaires concernant e.g. les successeurs au sens large d'un sommet donné.

La programmation en `Prolog` consiste essentiellement à construire

[1] La sémantique d'un programme *P* sont toutes les formules atomiques closes qui peuvent être induites par *P*. En d'autre termes c'est toutes les connaissances que nous pouvons obtenir en utilisant les connaissances du programme *P*.

des prédicats qui permettront d'inférer des connaissances nouvelles à partir des bases de données existantes. On voit ainsi que la Logique Computationnelle est un outil pour ce que, en langage branché, on appelle *forage de données – data mining* et qui est en réalité une activité aussi vieille que le monde.

2.1 Les prédicats de base de Prolog

Pour construire nos propres prédicats nous pouvons utiliser des prédicats de Prolog. En effet SWI-Prolog a une impressionnante collection des prédicats dont vous trouverez la liste complète et leur utilisation dans le guide de référence du langage. Nous présentons dans ce paragraphe quelques prédicats très utiles avec une explication sommaire de leur signification.

2.1.1 Entrées-sorties

print(X), print('toto')	afficher le contenu de X ou le mot « toto »
tab(5)	afficher 5 espaces blancs
nl	saut d'une ligne
read(X)	lecture d'une valeur et stockage dans X

2.1.2 Opérations en Prolog

Les opérations sont en notation infixe. Nous avons :

:-	si
?	opérateur de requête
X = Y	égalité par unification
X == Y	identité sans unification
X =/= Y	X est non identique à Y
X < Y	
X =< Y	
X >= Y	
X > Y	
X =.. Y	opérateur \textsl{div} X est le nom d'un foncteur et Y contient, sous forme de liste, ses éléments

En dehors des opérateurs établis par Prolog le programmeur peut définir ses propres opérateurs en utilisant la requête ?. La forme générale d'un opérateur est
:- op(priorité, type, nom) avec

— Priorité: une valeur entre 1 et 32000 qui indique la priorité de l'opérateur (1 est le plus prioritaire, 32000 le moins prioritaire).

— Type :
  – infixe : `xfx, xfy, yfx, yfy`
  – préfixe : `fx, fy`
  – postfixe : `xf, yf`

— Nom : le nom de l'opérateur.

En ce qui concerne le type de l'opérateur, le symbole « `x` » interdit les associations tandis que le symbole « `y` » les autorise. Ainsi les quatre opérations numériques sont du type `yfx` et par conséquent une expression comme

```
a + b + c + d
```

sera représentée en interne comme suit

```
(((a + b) + c) + d).
```

La virgule est un opérateur du type `xfy` et donc l'expression

```
a, b, c, d
```

sera interprétée comme suit:

```
(a, (b, (c, d)))
```

2.1.3 Prédicats extralogiques

Les prédicats extralogiques sont des prédicats qui soit testent le statut d'un terme, soit induisent une action.

Dans la première catégorie on trouve

`var(X)`	`X est une variable`
`atom(X)`	`X est un nom d'une constante`
`number(X)`	`X est un nombre`

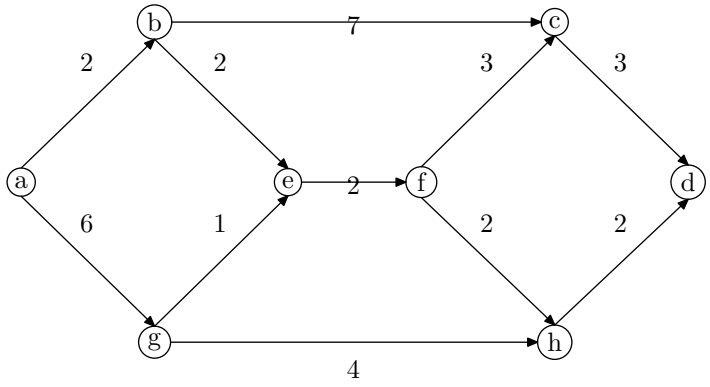
Dans la deuxième catégorie on trouve d'une part

`assert(X)`	`Ajouter à la base de données le fait X`
`asserta(X)`	`Ajouter au début de la base de données le fait X`
`assertz(X)`	`Ajouter à la fin de la base de données le fait X`
`retract(X)`	`Supprimer de la base de données toutes les occurrences de X`

et, d'autre part, le prédicat `fail` et le coup-choix (cut) `!`.

2.2 Un exemple

Considérons de nouveau l'exemple du graphe du chapitre précédent dont voici de nouveau sa représentation graphique :



Nous avons vu que la base de données du graphe permet de répondre à de questions du type : `?gr(a,b,_)`. Pour améliorer nos connaissances on doit utiliser la logique des prédicats qui, grâce à l'existence des quantificateurs, permet de poser des questions du type

`Existe-t-il des sommets X tels que gr(a,X,_)?`

et qui sous forme clausale s'écrit

`?-gr(a,X,_).` [2]

Dans le cadre de la logique des prédicats, les clauses du programme sont considérées comme des prédicats et, par conséquent, leur valeur de vérité dépend de la valeur des arguments. La résolution SLD permet de voir qu'il y aura deux réponses positives à cette question, car il y a deux substitutions qui filtrent la clause du but, à savoir:

```
X = b
X = g.
```

Notons que les clauses qui satisfont à la question posée font toujours partie de l'ensemble minimal d'Herbrand.

La logique des prédicats nous permet aussi d'écrire et d'utiliser des règles. Ainsi nous pouvons ajouter au programme précédent la clause :

Programme 2.2.1

```
successeurImmediat(X,Y) :- gr(X,Y,_).
```

[2] Remarquons que la clause écrite en `Prolog` respecte la convention qui veut que les noms des constantes commencent par une lettre minuscule – ici le sommet `a` – et les noms des variables par une lettre majuscule – ici `X` qui représente les sommets reliés directement à `a` et dont `a` est l'extrémité de départ. De plus on utilise aussi le symbole « _ » qui représente la variable anonyme, c'est-à-dire une variable dont la valeur précise ne nous intéresse pas. Pour plus de détails concernant la variable anonyme, voir dans la suite.

Ascèse 2.1 *Examiner le résultat des questions:*
`successeurImmediat(c,Y).`
`successeurImmediat(X,Y).`
`successeurImmediat(X,c).`
Pourriez-vous faire la liaison avec le modèle minimal d'Herbrand?

Pourriez-vous anticiper la réponse de `Prolog` *aux questions:*
`successeurImmediat(X,a).`
`successeurImmediat(d,Y).`

En fonction des résultats obtenus est-il possible d'établir deux nouveaux prédicats qui décrivent les propriétés des sommets `a` *et* `b`*?*

Nous pouvons maintenant envisager de définir la notion du successeur au sens large, à savoir un sommet qu'on peut atteindre d'un sommet fixé en passant par plusieurs sommets intermédiaires. Cette notion peut introduite au programme en y ajoutant les clauses suivantes:

Programme 2.2.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
```

Notons que ce programme est sous forme recursive parce que la deuxième clause contient un appel à cette même clause. Nous examinerons par la suite les méthodes d'écriture des programmes récursifs.

Ascèse 2.2 *Examiner le résultat des questions:*
`successeur(c,Y).`
`successeur(X,c).`
Pourriez-vous faire la liaison avec le modèle minimal d'Herbrand?
Établir l'arbre de dérivation SLD pour les questions précédentes.

Nous allons par la suite revenir plusieurs fois sur cet exemple du graphe.

2.3 Sémantique de Prolog

Considérons un programme défini E écrit en Prolog. Sa sémantique est l'ensemble des fbf closes que nous pouvons en déduire des clauses du programme E en appliquant les règles de la logique des prédicats. Ainsi une question – qui, en réalité, est une clause – que nous posons, constitue un but à vérifier par E et si ce but est dans la sémantique de E, alors la réponse du programme sera positive. Pour anticiper la réponse du programme on

peut utiliser le modèle minimal d'Herbrand du programme. En effet notons par $\mathcal{U}(E)$ et$\mathcal{B}(E)$ l'univers et la base d'Herbrand du programme E respectivement. Une interprétation I de E est un sous-ensemble de la base d'Herbrand. Une interprétation I est un modèle d'Herbrand pour E si

— pour chaque fait du programme du type `p.` on a que `p` est dans I;
— pour chaque règle du programme du type `p :- q1, q2, ..., qN p` est dans I si `q1, q2, ..., qN` sont dans I.

Donc un but est dans la sémantique du programme E s'il est dans chaque modèle de E et, par conséquent, à l'intersection de tous les modèles de E qui forme le modèle minimal d'Herbrand $\widetilde{I}_E$ pour E.

De ce qui vient d'être dit, on conçoit aisément que les buts qu'un programmeur se fixe quand il écrit un programme et que l'on notera par $\mathcal{G}(E)$, doivent être contenus dans $\widetilde{I}_E$. Dans ce cas on dit que le programme E est complet. Si les buts dépassent le modèle minimal, i.e. si $\widetilde{I}_E \subset \mathcal{G}(E)$, alors on dit que le programme est correct.

Un programme peut être complet et correct et pourtant, en réponse à un but précis qui est dans $\mathcal{G}(E)$, ne pas pouvoir aboutir en un nombre fini d'étapes à cause du caractère non déterministe de `Prolog`. De façon formelle nous pouvons dire qu'un programme en `Prolog` relatif à un but se termine toujours si l'arbre de dérivation du but est fini. C'est le cas par exemple pour tous les programmes qui ne contiennent pas de clauses récursives. Dans le cas contraire le programme ne s'arrête pas. En fait le comportement d'un programme `Prolog` depend de l'ordre de clauses – faits et règles – et de l'ordre des prédicats, à l'intérieur de chaque clause [3].

2.3.1 Ordre des clauses

L'ordre dans lequel sont écrites les clauses d'un programme `Prolog` détermine l'ordre dans lequel seront trouvées les différentes solutions. En effet `Prolog` pour trouver toutes les réponses à un but, parcourt l'arbre de dérivation de la résolution SLD selon l'algorithme *en profondeur d'abord*. Chaque clause du programme est placée sur une branche de cet arbre en fonction de la place qu'elle occupe dans l'ordre des clauses du programme. Si donc nous avons deux programmes qui sont identiques mais dans lesquels l'ordre des clauses n'est pas le même, le parcours de l'arbre ne se fera pas de la même façon mais les deux graphes seront isomorphes et par conséquent si

[3] Les prédicats qui forment les prémisses d'une règle seront considérés comme des buts à satisfaire lors d'une dérivation SLD. C'est la raison pour laquelle dans la bibliographie l'ordre des prédicats porte le nom d'*ordre des buts*, nom que nous utiliserons par la suite.

l'un de deux n'a pas une branche infinie, l'autre non plus n'a pas de branche infinie.

Il n'y a pas une règle générale pour l'ordre des clauses dans un programme Prolog. L'usage cependant veut que, dans le cas d'un programme récursif, on ait d'abord le(s) fait(s), c'est-à-dire le(s) test(s) d'arrêt, et ensuite les règles récursives.

Ascèse 2.3 *Vérifier l'ordre des solutions trouvées pour le but*
`?-successeur(f,X).`
si à la place du programme 1.3 on utilise le programme

Programme 2.3.1

```
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
successeur(X,Y) :- successeurImmediat(X,Y).
```

Pourriez-vous expliquer les différences?

2.3.2 Ordre des buts

L'ordre dans lequel sont présentés les premisses ou (sous-)buts dans une règle d'un programme Prolog conditionne l'exécution du programme et, donc, il détermine les solutions qui seront trouvées. Car, contrairement à la permutation des clauses qui aboutit à des arbres de dérivation toujours isomorphes entre eux, la permutation des buts donne naissance à des arbres de dérivation différents. De plus en fonction de la place qu'il occupe un appel récursif dans une règle, le programme peut nous fournir des solutions avant d'aboutir à une branche infinie ou même aboutir à cette branche infinie avant de donner la moindre solution.

Ascèse 2.4 *Vérifier les solutions trouvées pour le but*
`?-successeur(f,X).`
si à la place du programme 1.3 on utilise le programme

Programme 2.3.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeur(Z,Y), successeurImmediat(X,Z).
```

Pourriez-vous expliquer les différences?
Pourriez-vous anticiper la réponse aux questions
`?-successeur(d,X).` *et*
`?-successeur(X,a).`

Comme précédemment avec les clauses, il n'y a pas non plus une méthode pour établir l'ordre des buts dans une règle. L'ascèse ci-dessus suggère d'utiliser un appel récursif à la fin des prémisses d'une règle (récursivité droite) plutôt qu'un appel récursif au début des prémisses (récursivité gauche) mais il ne s'agit pas d'une méthode générale.

Ascèse 2.5 *On dira que deux sommets sont au même niveau s'ils sont successeurs immédiats du même sommet.*

(1) Écrire le programme `memeNiveau(X,Y)`.

(2) Vérifier en posant la question `?memeNiveau(b,X)` *que deux variables différentes n'ont pas nécessairement des valeurs différentes. Apporter une solution.*

(3) Compléter ce programme pour tenir compte du fait que la relation `memeNiveau(X,Y)` *est symétrique, c'est-à-dire que si, par exemple, on a* `memeNiveau(bmg)`*, alors on a aussi* `memeNiveau(g,b)`.

3

LISTES ET RECURSIVITÉ

3.1 Les listes et leur représentation 26
3.2 La recursivité 27
3.3 Techniques de recursivité 30
3.3.1 Recursivité pour les fonctions numériques 30
3.3.2 Recursivité simple 31
3.3.2.1 Arrêt lorsque la liste est vide 31
3.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé 32
3.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte 33
3.3.3 Recursivité multiple 33
3.4 Exercice 35

NOUS avons vu jusqu'ici le stockage des données à l'aide des base de données. Mais le traitement des données à partir de ces bases n'est pas toujours très facile à effectuer. On a envie d'avoir des données en mémoire dynamique facilement manipulables. À vrai dire `Prolog` est très chichement doté en types des données. Comme son grand ancêtre, le `Lisp`, `Prolog` ne dispose comme type des données, en tout et pour tout, que les listes. Bien sûr tout programmeur experimenté sait que la profusion des types de données qui sont l'apanage des plusieurs langages de programmation, en commençant par le plus illustre, le `Fortran`, sont très souvent source de confusions. Il est donc important pour l'élève–ingénieur de comprendre que l'esprit humain doit dompter la machine, qui par construction et par essence est bête, et arriver à faire des choses merveilleuses en utilisant très peu des matériaux. `Prolog` constitue un excellent exercice pour cet objectif.

3.1 Les listes et leur représentation

La seule structure des données que `Prolog` reconnaît ce sont les *listes*. Ce qui veut dire qu'en `Prolog` il n'y a pas des tableaux et surtout il n'y a pas des indices de tableau ou des pointeurs.

Une liste est une suite des termes de n'importe quelle nature, separés par des virgules, entourée par deux crochets, [et] . Par exemple `[toto, 1, av_du_parc, cergy, la_logique_est_super]`. Bien évidemment une liste peut avoir des sous-listes, une sous-liste peut avoir des sous-sous-listes et ainsi de suite à la manière des poupées russes mais généralisées en ce sens qu'une poupée peut contenir plusieurs sous-poupées de même taille et qui, à leur tour, puissent avoir plusieurs sous-sous-poupées de même taille. Par exemple `[toto,[ 1,[ av_du_parc], [cergy]], la_logique_est_super]`.

Il faut peut être le repéter: on ne peut pas accéder directement à un élément quelconque d'une liste. (Par contre on peut accéder à un élément dont on connaît effectivement son rang dans la liste.) On peut seulement séparer ce qu'on appelle la *tête* d'une liste du *reste* de la liste, en utilisant le symbole du séparateur « | ». Ainsi, si une liste est représentée par la variable `L`, on peut écrire `L=[Tete | Reste]` où `Tete` représente le premier élément de `L` et `Reste` est la liste composée des autres élément de `L`. Par exemple si `L=[toto, 1, av_du_parc, cergy, la_logique_est_super]`, alors on a `Tete =toto` et `Reste = [1, av_du_parc, cergy, la_logique_est_super]`.

De ce qui précède on peut en conclure qu'on peut accéder au premier élément d'une liste. Considérons maintenant une liste ayant n éléments `L=[`$x_1, x_2, \ldots, x_n$`]`. Si on veut accéder au k-ième terme, où k a une valeur précise et connue, nous avons deux possibilités:

— soit directement en posant pour `L :[ T`$_1$`,T`$_2$`,...,T`$_k$ `| Reste ]` avec `T`$_k \longleftarrow x_k$.

— soit d'une façon séquentielle, à la manière de la lecture des enregistrements d'un fichier en accés séquentiel.On construit la représentation `L`$_1$ `= [Tete | Reste]`, où `Tete` $\longleftarrow x_1$. On recupère le `Reste` dans une liste notée L_2 et on recommence: L_2 =`[Tete | Reste]` avec maintenant `Tete` $\longleftarrow x_2$. En continuant ainsi on arrive, au bout de k itérations, à accéder au k-ième élément de la liste.

Même si la première possibilité vous paraît plus facile, rappelez-vous ce qu'on vous a toujours dit concernant les apparences et concentrez-vous sur la deuxième possibilité. C'est celle qui est utilisée en `Prolog` mais dans sa version recursive.

3.2 La recursivité

Pour appliquer la recursivité sur les listes il faut savoir que si on introduit une liste, e.g. `[toto, 1, av_du_parc, cergy, la_logique_est_super]`, `Prolog` introduit toujours à la fin de la liste, comme un élément supplémentaire, une liste vide, de sorte qu'on ait `[toto,1,av_du_parc,cergy, la_logique_est_super, []]`. Ainsi quand on progresse à l'intérieur d'une liste, élément par élément, on peut comprendre qu'on est arrivé à la fin de la liste en comparant la liste qui reste chaque fois avec la liste vide. Le test de la liste vide constituera pour beaucoup de programmes recursifs, le test d'arrêt.

En règle générale, un programme est composé de deux parties :

— Une partie concernant le ou les tests d'arrêt.

— Une partie concernant les appels recursifs du prédicat à lui-même.

La programmation recursive est un type particulier de programmation au même titre que la programmation fonctionnelle ou la programmation orienté objet. Il faut savoir que la mise en œuvre d'un programme est plus facile qu'avec les autres types de programmation et ses résultats sont beaucoup beaucoup plus spectaculaires parce qu'on utilise la puissance de la recursivité. En effet les programmes en `Prolog` expriment seulement ce que le programmeur souhaite réaliser et non pas la manière de faire pour le réaliser comme c'est le cas avec les langages imératifs.

Nous allons examiner en detail le mécanisme des appels recursifs en utilisant la liste `L=[toto, 1, av_du_parc, cergy, la_logique_est_super]`. On cherche à calculer la longueur de cette liste, c'est-à-dire le nombre d'éléments qui la composent. On va donc procéder élément par élément et chaque fois on prendra en compte le premier élément de la liste. Il faut donc pouvoir accéder au premier élément de la liste. Pour accéder au second élément, il faut supprimer de la liste le premier élément et appeler le programme de façon recursive. On a donc le programme:

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

L'appel `longueur(Y,N1)` est un appel recursif. Ce qui signifie qu'avant la réalisation du test d'arrêt: `longueur([], 0)`, les demandes de `N is N1+1` sont empilées et ne sont pas exécutées. Elles vont commencer à être exécutées, et dans l'ordre inverse de leur empilement, après l'exécution du test d'arrêt. Concrètement si on applique ce programme à la liste `L` nous aurons le déroulement suivant :

1er appel : `longueur([toto | 1, av_du_parc, cergy, la_logique_est_super],N)`

— État du programme :
`longueur([ 1, av_du_parc, cergy, la_logique_est_super ],N1)`
— État de la pile :

N is N1 + 1.

2e appel : `longueur([ 1 | av_du_parc, cergy, la_logique_est_super ],N1)`

— État du programme :
`longueur([ av_du_parc, cergy, la_logique_est_super],N2)`
— État de la pile :

N1 is N2 + 1.
N is N1 + 1.

3e appel : `longueur([ av_du_parc | cergy, la_logique_est_super],N2)`

— État du programme :
`longueur([ cergy, la_logique_est_super],N3)`
— État de la pile :

N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

4e appel : `longueur([ cergy | la_logique_est_super],N3)`

— État du programme :
`longueur([la_logique_est_super],N4)`
— État de la pile :

N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

5e appel : `longueur([la_logique_est_super], | []], N4)`

— État du programme :
`longueur([], N5)`
— État de la pile :

N4 is N5 + 1.
N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

6e appel : `longueur([],N5)`

— État du programme :
`N5 ← 0`

— État de la pile :

N4 is N5 + 1. N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 +1.

Le test d'arrêt sert ici comme initialisation de la valeur de la longueur à 0. Les ordres successifs d'addition de la valeur 1 aux différents valeurs de `N` n'ont pas été exécutés mais seulement stockés dans la pile. Dès que le programme a rencontré un test d'arrêt, les ordres stockés dans la pile commencent à être exécutés. Ainsi la suite du programme se fera par de depilement successifs et exécution des commandes depilées. Nous avons donc:

7e étape: `N5 = 0`

— État du programme :

`N4 ⟵ N5 +1 = 0+1 = 1`

— État de la pile :

N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

8e étape: `N4 = 1`

— État du programme :

`N3 ⟵ N4 +1 = 1+1 = 2`

— État de la pile :

N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

9e étape: `N3 = 2`

— État du programme :

`N2 ⟵ N3 +1 = 2+1 = 3`

— État de la pile :

N1 is N2 + 1.
N is N1 + 1.

10e étape: `N2 = 3`

— État du programme :

`N1 ⟵ N2 +1 = 3+1 = 4`

— État de la pile :

N is N1 + 1.

11e étape: `N1 = 4`

— État du programme :

`N ⟵ N1 +1 = 4+1 = 5`

— État de la pile : `< pile vide >`

En examinant le deroulement du programme, on constate que le test d'arrêt est le dernier a être évoqué lors des appels recursifs mais le premier a être exécuté lors du depilement des ordres empilés. Par conséquent il faut considérer les tests d'arrêt comme la partie du programme qui initialisent les valeurs des variables utilisées.

3.3 Techniques de recursivité

Nous présentons ci-après les techniques de base qui permettent de réaliser des programmes recursifs en `Prolog`.

3.3.1 Recursivité pour les fonctions numériques

Bien qu'en `Prolog` nous avons seulement des prédicats, nous pouvons envisager d'avoir des fonctions si leur résultat est stocké dans une variable qui fait partie des arguments de la fonction. Ainsi, par exemple, on peut envisager d'écrire un prédicat qui sera en réalité une fonction qui calcule la somme de N premiers nombres naturels. Ce prédicat peut avoir la forme suivante :

Programme 3.3.1

```
somme(0,0).

somme(N,Somme) :- N > 0, N1 is N - 1, somme(N1, Somme1),
                          Somme is Somme1+1.
```

On constate donc que pour programmer une fonction numérique sous forme récursive, il faut

(1) *Déterminer le(s) test(s) d'arrêt,* c'est-à-dire décider quand la fonction retourne une valeur prédéterminée, sans appel recursif à elle-même.
Le test d'arrêt pour une fonction numérique se fait en comparant la valeur d'une variable, dont son contenu évolue en fonction des appels recursifs avec une valeur fixée par le programme. La valeur prédéterminée que le programme retourne est la valeur d'initialisation de la variable dont nous venons de parler.

(2) *Déterminer le(s) cas recursif(s).* Un appel recursif d'une fonction s'effectue avec un argument plus simple et on utilise le résultat pour calculer la réponse de l'argument courant. Un argument plus simple en

recursivité numérique est un argument qui est plus proche de la valeur utilisée pour l'initialisation par le test d'arrêt.

Ascèse 3.1 *Calcul du factoriel n!.*

Ascèse 3.2 *Calcul de la puissance d'un nombre x^n.*

Ascèse 3.3 *Calcul des nombres de Fibonacci.*

Ascèse 3.4 *Calcul du plus grand diviseur commun et du plus petit commun multiplicateur de deux entiers.*

3.3.2 Recursivité simple

Ce cas concerne les listes qui n'ont pas des sous-listes ou même si elles en ont, on ne les prendra pas en considération.

Si nous avons à faire un traitement sur un élément quelconque, il faut l'avoir soit stocké au préalable dans une base de données, soit introduit dans une liste. Dans ce dernier cas et étant donné que nous ne pouvons pas indexer les listes par des pointeurs, on est obligé de parcourir la liste jusqu'à arriver à atteindre l'élément voulu. Ce parcours se fera par des appels recursifs où un des arguments sera la liste qui, à chaque appel, sera systématiquement debarassée de son premier élément. La technique donc de la programmation recursive pour les listes avec arrêt simple est identique à celle pour les fonctions numériques, mais les tests d'arrêt se font sur les listes et leurs éléments et non pas sur la valeur d'une variable. On distingue trois types de tests d'arrêt que nous présentons séparement ci-après.

3.3.2.1 Arrêt lorsque la liste est vide

Le programme s'arrête lorsque la liste que nous sommes en train de traiter devient vide. Les programmes que nous pouvons mettre sous ce type sont

— soit des programmes d'énumération: nombre d'éléments qu'une liste contient, nombre d'occurences d'un élément dans une liste, etc.

— soit des programmes d'affichage du contenu d'une liste ou de copie d'une liste dans une autre liste ou, encore, la concaténation de deux listes.

Nous donnons comme exemple le calcul de la longueur d'une liste

Programme 3.3.2

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

```
longueur([Tete | Reste], Longueur) :- longueur(Reste,Longueur1),
                                      Longueur is Longueur1 + 1.
```

Ascèse 3.5 *Afficher le contenu d'une liste.*

Ascèse 3.6 *Copier une liste dans une nouvelle liste.*

Ascèse 3.7 *Concatener deux listes en créant une troisième.*

3.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé

On s'arrête dès qu'un élément spécifique, fixé au préalable a été retrouvé. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur le fait qu'élément particulier appartient à une liste comme, par exemple, le programme `membre`.

Programme 3.3.3

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

membre(X, [Tete | Reste]) :- membre(X, Reste).
```

Bien sûr un tel programme pose le problème de sa fin dans le cas où l'élément spécifique ne fait pas partie de la liste. Normalement dans ce cas le programme s'arrête en échec. On peut éviter cette sortie en échec, en ajoutant la clause :

```
membre(_, []).
```

Mais en procédant ainsi, on ignore si on a trouvé ou non l'élément spécifique. Si on s'inspirait de la programmation impérative, on serait tenté ici d'ajouter un indicateur qui nous informererait sur le résultat effectif. Mais la bonne solution en Prolog est de distinguer le traitement de deux situations en utilisant l'opérateur " `ou` " comme suit :

```
toto :- (membre(a, [b,a,c]),
          write('L élément a fait partie de la liste'));
        (write ('L élément a ne fait pas partie de la liste')), nl.
```

Si le traitement particulier dans chaque cas est long, on peut envisager d'utiliser deux clauses qui s'excluent mutuellement:

```
toto :- membre(a, [b,a,c]),
        write('L élément a fait partie de la liste')), nl.

toto :- not(membre(a, [b,a,c])),
        write ('L élément a ne fait pas partie de la liste')), nl.
```

Ascèse 3.8 *Donner la position d'un élément dans une liste.*

Ascèse 3.9 *Supprimer un élément d'une liste et obtenir une nouvelle liste sans cet élément.*

Ascèse 3.10 *Remplacer dans une liste un élément par un autre et obtenir une nouvelle liste.*

Ascèse 3.11 *Insérer dans une liste, après un élément spécifique, un autre élément et obtenir une nouvelle liste.*

3.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte

On s'arrête dès qu'une position spécifique, fixée au préalable a été retrouvée. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur l'élément qui occupe une place particulière dans une liste comme par exemple le programme suivant qui affiche le *n*-ième élément d'une liste

Programme 3.3.4

```
afficheNth(1, [X|_]) :- write(X), nl.

afficheNth(N, [Tete | Reste]) :- N > 1, N1 is N - 1,
                                 afficheNth(N1, Reste).
```

Ascèse 3.12 *Supprimer le n-ième élément d'une liste et obtenir une nouvelle liste.*

Ascèse 3.13 *Remplacer dans une liste un élément dans une position donnée par un autre et obtenir une nouvelle liste.*

Ascèse 3.14 *Insérer dans une liste, avant un élément qui se trouve à une position spécifique, un autre élément et obtenir une nouvelle liste.*

3.3.3 Recursivité multiple

On doit travailler avec des récursivités multiples si la liste que nous sommes en train de traiter contient des sous-listes. Si e.g. on cherche à savoir si un élément donné fait partie d'une liste et si cette liste contient des sous-listes, on doit examiner séparement ses sous-listes.

La programmation des tests d'arrêt pour la recursivité multiple est identique à celle de la programmation simple. Par contre la programmation des cas recursifs est différente. On utilisera la technique de *recursivité Tête – Reste* dont nous présentons ci-après un exposé succint.

Programmation selon la technique de récursivité Tête – Reste. On suppose que nous avons une fonction qui a comme argument une liste représentée par `[Tete | Reste]` et que nous allons appeler cette fonction de façon recursive en modifiant l'argument représenté par la liste.

(1) Déterminer le(s) cas de reste – recursivité.
Ce sont les cas où la tête `Tete` de la liste est un atome et nous appelons recursivement la fonction avec comme argument pour la liste son `Reste`.
Il y a deux type de reste – recursivité.
   (a) On retourne simplement les résultats de la fonction appelée recursivement sur le reste de la liste.
   (b) On associe les résultats de reste – recursivité avec une valeur dérivée de la tête de la liste.

(2) Déterminer les cas de tête – reste recursivité.
Ce sont les cas où la tête `Tete` de la liste est une liste. Dans ce cas il faut appeler recursivement la fonction avec comme argument la tête de la liste et, ensuite, appeler recursivement la fonction avec comme argument le reste de la liste. À la fin il faut associer les résultats de deux appels, pour obtenir le résultat correct pour la liste entière.

On présente comme exemple d'application de la recursivité Tête – Reste la recherche d'un élément dans une liste contenant des sous-listes.

Programme 3.3.5

```
membreT(X, [X | _]).   % Test d'arrêt

membreT(X, [Tete | Reste]) :- atom(Tete),
                              membreT(X, Reste). % Reste - recursivité

membreT(X, [Tete | Reste]) :- not(atom(Tete)),
                             (membreT(X,Tete);
                              membreT(X, Reste)).% Test - reste recursivité
```

On utilise aussi la recursivité multiple dans des cas où on doit traiter en même temps plusieurs listes. Examinons, à titre d'exemple, le programme qui opère un tri d'une liste numérique selon ses valeurs ascendantes .

Programme 3.3.6

```
tri([X|Xs],Y) :- partition(Xs,X,Petits,Grands),
                 tri(Petits,Ps),
                 tri(Grands,Gs),
                 conc(Ps,[X|Gs],Y).

tri([],[]).

partition([X|Xs],Y,[X|Petits],Grands) :- X < Y,
                                  partition(Xs,Y,Petits,Grands).

partition([X|Xs],Y,Petits,[X|Grands]) :- X >= Y,
                                  partition(Xs,Y,Petits,Grands).

partition([],Y,[],[]).
```

Bien évidement on n'utilise pas la technique de tête – reste recursivité mais le programme `tri` est appelé deux fois, comme en recursivité simple, avec deux listes différentes.

Ascèse 3.15 *Étant donnée une liste qui contient des sous-listes, obtenir un nouvelle liste qui a les mêmes éléments que la première liste mais sans sous-listes.*

Ascèse 3.16 *Faire un tri selon l'ordre alphabétique d'une liste qui contient des noms.*

3.4 Exercice

Exercice 3.1 *Nous avons les données suivantes :*

(1) Il y a 5 maisons.

(2) Dans chaque maison habite une personne de nationalité différente.

(3) Dans chaque maison habite une personne qui boit une boisson différente.

(4) Chaque maison a une couleur différente.

(5) La personne qui boit du lait habite la maison du milieu.

(6) Le norvégien habite à la première maison.

(7) L'anglais boit de la bière.

(8) La personne qui boit du café habite à la maison rouge.

(9) La maison blanche est juste après la maison verte.

(10) Le norvégien habite à gauche de la maison bleue.

(11) L'allemand habite juste avant de la maison jaune.

(12) L'anglais habite à la maison blanche.

(13) Le suédois habite deux maisons après la personne qui boit de l'eau.

(14) La personne qui habite à la maison verte boit du thé.

Trouver la liste complète des maisons, de leurs occupants et la boisson qu'ils boivent, en sachant qu'il y ait aussi un italien.

4

LES BASES DE DONNÉES EN PROLOG

4.1 Définir les relations 37
4.2 Définir les requêtes 38
4.2.1 Les requêtes recursives 40
4.3 La programmation logique et le modèle relationnel des bases de données 40
4.3.1 L'opération d'union 40
4.3.2 L'opération de différence 40
4.3.3 L'opération du produit cartésien 41
4.3.4 L'opération de projection 41
4.3.5 L'opération de sélection 41
4.4 Gestion de la base de données 43
4.5 Introduction aux bases de données déductives 45

NOUS montrons dans ce chapitre comment représenter les bases de données relationnelles en `Prolog`. Les notions de base comme les relations et les requêtes sont détaillées. Nous présentons ensuite, une brève introduction à la base de données déductives et la relation avec le modèle minimal de Herbrand. Les exercices montreront commet traiter les requêtes universelles en se servant de l'opérateur négation proposé par `Prolog`. Ensuite, la notion du monde fermé est introduit.

4.1 Définir les relations

En base de données relationnelle, une relation est exprimée par une table. Chaque colonne désigne un attribut particulier de la relation. Une ligne correspond à une enregistrement. Le schéma relationnel est décrit par le nom de la table et la désignation de chaque colonne. En `Prolog` une table est définie par un prédicat, le nombre de colonnes correspond à l'arité de ce prédicat, l'argument numéro *i* dans un tel prédicat correspondra à la colonne

numéro *i*. Un enregistrement sera décrit par un fait représenté à l'aide du prédicat-relation et des termes filtrés. Prenons par exemple la relation *père-fils*, nous pouvons exprimer deux enregistrements (*toto-titi* et *toto-koko*) de la table *père* en Prolog comme suit :

```
pere(toto, titi).
pere(toto, koko).
```

Le prédicat *père* est un prédicat d'arité 2, décrivant une relation entre deux données. Le schéma relationnel associé à *père* est *père (Pere,Fils)*. [1]

Remarquons bien que la base de faits en Prolog peut exprimer plusieurs relations appartenant à des schémas relationnels différents.

```
pere(toto, titi).
pere(toto, koko).
mere(lola, titi).
mere(lola, koko).
```

4.2 Définir les requêtes

Nous avons vu que nous pouvons constituer notre base de données avec un ensemble de faits. Le nombre de prédicats exprime le nombre de schémas relationnels constituant la base de données [2]. Nous classons les requêtes en deux catégories : *requêtes simples* et *requêtes composées*.

Les requêtes simples sont exprimées par un des prédicats constituant la base de faits. Ce sont des questions que l'on pose, par exemple :

```
?pere(toto,X). donne tous les enfants de toto.
?pere(X,koko). donne le père de koko.
```

Les requêtes composées sont exprimées par un ensemble de requêtes simples avec des relations entre eux (conjonction ou disjonction de requêtes). Pour effectuer ce type de requêtes en Prolog, nous définissons des règles permettant de simplifier la question ultérieurement. Le corps d'une clause

[1] Ceci est équivalent à la définition d'une table en Oracle appelée père et contenant deux colonnes, la première colonne représente le père et la deuxième représente le fils. Dans l'exemple précédent, cette table contenait deux lignes.

[2] Le nombre de tables en Oracle

exprime la conjonction des requêtes à poser. Sa tête est la définition simplifiée de la requête.
Par exemple, supposons que nous avons une base de données exprimant trois relations : *parent, homme et femme*, et que nous aimerions chercher les *pères* et les *mères*.

```
parent(toto, titi).
parent(toto, koko).
parent(lola, titi).
parent(lola, koko).
homme(toto).
homme(koko).
homme(titi).
femme(lola).
```

Pour cela nous définissons ces deux nouvelles relations sous forme de règles que nous ajoutons au programme :

```
pere(X,Y):-parent(X,Y), homme(X).
mere(X,Y):-parent(X,Y), femme(X).
```

À partir de cela, nous pouvons poser les requêtes :

```
?pere(X,koko).  qui est équivalente à la requête composée:
                trouver le parent de koko X et X est homme.
?mere(X,koko).  qui est équivalente à la requête composée:
                trouver le parent de koko X et X est femme.
```

Remarquons que dans ce cas, la définition de chacune des relations *père* et *mère* a permis d'exprimer une relation de *conjonction* entre des requêtes simples.

D'un autre côté, si nous partons de l'ensemble de faits composés des schémas relationnels *père et mère* seulement, nous pouvons constituer la relation *parent* qui sera donnée par les règles suivantes :

```
parent(X,Y):-pere(X,Y).
parent(X,Y):-mere(X,Y).
```

Dans ce cas, la définition de la relation *parent* a permis d'exprimer une relation de *disjonction* entre requêtes simples.

4.2.1 Les requêtes recursives

Un cas particulier des requêtes composées sont les requêtes recursives. Une requête recursive est définie par une relation qui est appelée dans le corps d'une clause. Un exemple de ce type de requêtes est la recherche des *ancêtres* en utilisant seulement la relation *parent*

```
ancestre(X,Y):-parent(X,Z),parent(Z,Y).
ancestre(X,Y):-parent(X,Z), ancestre(Z,Y).
```

4.3 La programmation logique et le modèle relationnel des bases de données

La programmation logique permet d'exprimer plusieurs aspects du modèle relationnel des bases de données. Nous avons vu que les relations peuvent être décrites par des faits et que les expressions des requêtes sont souvent exprimées par des règles. L'arité d'une relation est exprimée par le nombre d'arguments du prédicat utilisé dans la définition du fait correspondant.

En général, cinq principaux types d'opérations définissent une algèbre relationnelle : l'union, la différence ensembliste, le produit cartésien, la projection et la sélection. Nous montrons comment implémenter chacune de ces opérations avec `Prolog` :

4.3.1 L'opération d'union

L'opération d'union permet de créer une relation d'arité n à partir de deux relations r et s d'arité n. La nouvelle relation est l'union des deux autres relations et est exprimée par les deux règles suivantes :

```
r_union_s(X1,...,Xn) :-r(X1,...,Xn).
r_union_s(X1,...,Xn) :-s(X1,...,Xn).
```

La relation *parent*, défini à partir des relations *père* et *mère*, est un exemple de ce type de relation.

4.3.2 L'opération de différence

La différence ensembliste est exprimée à l'aide de la négation :

```
r_diff_s(X1,...,Xn) :-r(X1,...,Xn), not s(X1,...,Xn).
r_diff_s(X1,...,Xn) :-s(X1,...,Xn), not r(X1,...,Xn).
```

Et ceci en supposant que r et s sont d'arité n.

Ascèse 4.1 *Soit les deux relations :* `enseignantPere` *et* `enseignantMere` *exprimées par deux prédicats d'arité trois chacune, définies par les shémas relationnels :* `enseignantPere(Enfant,Pere,Mere)`, `enseignantMere(Enfant,Pere,Mere)`.
Donner la relation `enseignantUnSeulParent(Enfant,Pere,Mere)`.

4.3.3 L'opération du produit cartésien

Cette opération permet de définir, à partir d'une relation r d'arité m et une relation s d'arité n, une nouvelle relation d'arité $k = m+n$ de la façon suivante :

```
r_prod_s(X1,...Xm,...,Xk) :-r(X1,...,Xm),s(Xm+1,...,Xm+n).
```

Ascèse 4.2 *Soit les deux relations :* `pere` *et* `mere` *exprimées par deux prédicats d'arité deux chacune, définies par les schémas relationnels :* `pere(Enfant,Pere)`, `mere(Enfant,Mere)`.
Donner la relation `parents(Enfant,Pere, Mere)`.

4.3.4 L'opération de projection

La projection permet de définir une nouvelle opération contenant seulement un sous-ensemble des attributs composant la relation de départ. Par exemple la relation $r13$ est une projection de r selon le premier et le troisième argument :

```
r13(X1,X3) :-r(X1,X2,X3)
```

Ascèse 4.3 *Soit la relation :* `mere` *exprimée par un prédicat d'arité 2, avec le schéma relationnel suivant :* `mere(Enfant,Mere)`. *Donner la relation* `femmeayantEnfant(Mere)`. *Que remarquez-vous quand il s'agit d'une femme ayant plusieurs enfants ?*

4.3.5 L'opération de sélection

L'opération de sélection consiste à définir à partir d'une relation de départ, une autre relation dérivée et ceci en posant certaines contraintes sur certaines données. Ces contraintes peuvent être des contraintes d'ordre logique (exemple $X_1 < X_2$) ou bien des contraintes exprimées en relations définies antérieurement. Les deux exemples suivants illustrent ces deux cas de figure.

```
r1(X1,X3) :-r(X1,X2,X3), X2 > X3.
```

```
r2(X1,X3) :-r(X1,X2,X3), r3(X1).
r3(const1).
r3(const2).
```

La relation *r*2 permet de sélectionner l'ensemble des valeurs vérifiant la relation *r* à condition que $X1$ soit unifiée avec *const1* ou avec *const2*.

Ascèse 4.4 *Soient les relations :* `parent, fille, garcon`*, définies par les schémas relationnels :* `parent(Parent,Enfant)`, `fille(Enfant)`, `garcon(Enfant)`. *Donner la relation* `parentFille(Parent)` *qui exprime le fait que* `Parent` *a une fille. Que remarquez-vous quand il s'agit d'un parent ayant plusieurs filles ?*

Notons bien que dans ce paragraphe, nous avons présenté les opérations de base, nous pouvons bien entendu « fabriquer » nos propres opérations à partir des operations de base.

Exemple 4.3.1 *Soit la base de données suivante, contenant les relations* `personne` *et* `voiture`.

```
voiture(123, fiat, toto, marron).
voiture(321, volvo, tata, rouge).
voiture(111, mazerati, cathy, blanche).
voiture(222, renault, loulou, rouge).
voiture(314, citroen, alma, verte).
personne(jojo, m, 16, toto, cathy).
personne(toto, m, 45, loulou, louloute).
personne(cathy, f, 40, koko, tata).
personne(anne, f, 14, toto, cathy).
personne(lolo, m, 30, koko,tata).
personne(louloute, f, 38, olive, alma).
personne(alma, f, 65, momo, mimi).
```

Pour répondre à la question : quelles sont les marques conduites par des femmes ? nous composons la requête suivante :

```
marque(X):- voiture(_,X,P,_), personne(P,f,_,_,_).
```

Remarquer bien que dans la composition de cette requête les opérations de base suivantes ont été appliquées :

— *Dans l'appel de* `voiture(_,X,P,_)` *une projection selon les arguments* `marque` *et* `personne` *est réalisée.*

— *Dans l'appel de* `personne (P,f,_,_,_)` *une selection selon les deux arguments* `personne` *et* `sexe` *est effectuée, les personnes trouvées via l'appel précédent et qui sont du sexe féminin sont choisies.*

— *Ensuite un produit cartésien et une projection selon le deuxième attribut du schéma relationnel* `voiture` *sont respectivement effectués.*

Ascèse 4.5 *Écrire les requêtes qui permettent de répondre aux questions suivantes :*

(1) Quels sont les propriétaires d'une volvo rouge?

(2) Donner les couples `(prénom, âge)` *des femmes qui conduisent des voitures.*

4.4 Gestion de la base de données

Nous avons vu dans les sections précédentes qu'il est possible de définir les relations et les opérations sur les relations en `Prolog`, en utilisant les faits et les règles. Nous montrons dans cette section une technique permettant de gérer l'ensemble des relations et permettant de les traiter toutes en utilisant une structure unique.

Supposons que notre base de données contient les relations : r_1, ..., r_n. Supposons pour simplifier que la clé de chaque relation r_i est donnée par un seul attribut cle_i et que la valeur de cet attribut est donnée par le premier argument de la relation. Supposons que $nomAtt_{ij}$ désigne le nom de l'attribut numéro j concernant la relation r_i.

Nous définissons le prédicat `bddVirt` d'arité 4. Le premier argument contient le nom de la relation, le deuxième la valeur de la clé, le troisième le nom de l'attribut j et le quatrième sa valeur.

```
bddVirt(Ri, ValCLEi, NomATTij, ValATTij) :-
                              Ri(ValATTi1, ..., ValATTij, ...)
```

Exemple 4.4.1 *Reprenons la base de données contenant les personnes et les voitures.*

La relation `bddVirt` *permet de définir une relation unique :*

```
bddVirt(voiture,Nu,numero,Nu) :- voiture(Nu,_,_,_).
bddVirt(voiture,Nu,marque,Ma) :- voiture(Nu,Ma,_,_).
bddVirt(voiture,Nu,proprietaire,Pr) :- voiture(Nu,_,Pr,_).
bddVirt(voiture,Nu,couleur,Co) :- voiture(Nu,_,_,Co).
bddVirt(personne,Nom,nom,Nom) :- personne(Nom,_,_,_,_).
bddVirt(personne,Nom,sexe,Se) :- personne(Nom,Se,_,_,_).
bddVirt(personne,Nom,age,Ag) :- personne(Nom,_,Ag,_,_).
bddVirt(personne,Nom,pere,Pe) :- personne(Nom,_,_,Pe,_).
bddVirt(personne,Nom,mere,Me) :- personne(Nom,_,_,_,Me).
```

Remarquez bien que la définition de cette relation permet de préciser explicitement le schéma relationnel de chaque relation dans la base de données, ainsi que sa clé. Remarquez aussi qu'il est possible de généraliser cette définition au cas où nous avons plusieurs clés et cela en augmentant l'arité du prédicat `bddVirt`.

Reprenons la question : quelles sont les marques conduites par des femmes ? Nous composons la requête suivante :

```
marque(X):- bddVirt(voiture,Nu,proprietaire,P),
            bddVirt(voiture,Nu,marque,X),
            bddVirt(personne,P,sexe,f).
```

Remarquez bien que chaque appel à `bddVirt` *permet d'effectuer une opération parmi les deux suivantes :*

(1) Une operation de projection selon le troisième argument, dans le cas où le quatrième argument est une variable libre.

(2) Une opération de selection selon le troisième argument dans le cas où le quatrième argument correspond à ensemble de valeurs donné.

Plus concrètement, dans l'exemple du calcul de `marque(X)` *l'appel* `bddVirt(voiture,Nu,propriétaire,P)` *permet de projeter les données de la base de données* `voiture` *selon l'argument* `propriétaire`. *Le deuxième appel permet de projeter ces même données selon l'argument* `marque`. *Le troisième appel* `bddVirt(proprietaire,P,sexe,f)` *effectue une selection sur l'ensemble des propriétaires trouvés par le premier appel selon l'argument* `sexe` *et avec valeur* `féminin`. *Finalement, le calcul de* `marque(X)` *est effectué par le produit cartésien des trois requêtes et ensuite par la projection sur la marque.*

Ascèse 4.6 *Réécrire les requêtes qui permettent de répondre aux questions suivantes en utilisant* `bddVirt` *:*

(1) Quels sont les propriétaires d'une volvo rouge ?

(2) Donner les couples `(prénom, age)` *des femmes qui conduisent des voitures.*

4.5 Introduction aux bases de données déductives

Nous avons vu comment exprimer une base de données relationnelle en utilisant des relations sous forme de faits. Nous avons vu comment utiliser les règles pour formuler des requêtes composées ou bien des requêtes recursives. En fait, nous pouvons utiliser les règles pour enrichir les bases de données avec des nouveaux faits. Ceci peut se faire soit en créant (en même temps) des nouveaux schémas relationnels (c'est-à-dire des nouveaux faits), soit en créant des nouvelles règles qui portent sur des faits existants. Par exemple nous pouvons ajouter le schéma relationnel `ancetreR` avec les données associées. Les faits `ancetreR` sont appelés des faits déduits car ce sont des faits qui ont été ajoutés à la base de faits et qui ont été déduits des faits existants et de la règle de déduction.

```
parent(toto, titi).
parent(toto, koko).
parent(lola, titi).
parent(lola, koko).
parent(koko,tintin)
parent(tintin,tino).
parent(tino,tony).
homme(toto).
homme(koko).
homme(titi).
homme(tintin).
homme(tino).
homme(tony).
femme(lola).
ancetre(X,Y):-parent(X,Z),parent(Z,Y).
ancetre(X,Y):-parent(X,Z), ancetre(Z,Y).
q2(X,Y):-ancetre(X,Y),assert(ancetreR(X,Y)).
```

A la fin de l'execution de ce programme, les faits ajoutés à la base de données sont : `ancetreR(toto, tintin). ancetreR(lola, tintin). ancetreR(koko, tino). ancetreR(tintin, tony). ancetreR(toto,`

tino). ancetreR(toto, tony). ancetreR(lola, tino). ancetreR(lola, tony). ancetreR(koko, tony). Remarquer bien que chacun de ces faits est deductible à partir du programme, en utilisant en particulier le modus-ponens et l'instantiation des variables. L'ensemble des faits obtenus correspond au modèle minimal de Herbrand car l'ensemble de départ est constitué d'un ensemble de faits filtrés.

Ascèse 4.7 *Soit le programme suivant :*

```
gr(a,b,2).
gr(a,g,6).
gr(b,e,2).
gr(b,c,7).
gr(g,e,1).
gr(g,h,4).
succImm(X,Y):-gr(X,Y,_).
suc(X,Y):-succImm(X,Y).
suc(X,Y):-succImm(X,Z),suc(Z,Y).
```

(1) Trouver le modèle minimal de Herbrand.

(2) Quelle sera la réponse à la question `?suc(X,Y)` *en* `Prolog`*.*

Ascèse 4.8 *Soit le programme suivant :*

```
acouleur(ciel).
acouleur(nuage).
acouleur(mer).
couleur(ciel,bleu).
couleur(nuage,blanc).
couleur(arbre,vert).
objetCouleur(X,Y):-acouleur(X),couleur(X,Y).
```

(1) Poser les questions : `?objetCouleur(X,Y)`, `?objetCouleur(mer,Y)`, `?objetCouleur(arbre,Y)`.

(2) Ajouter deux clauses qui traitent le problème de l'absence de relations.

Ascèse 4.9 *Soit la base de données relationnelle contenant les trois tables suivantes :*

Écrire un programme `Prolog` *qui répond aux requêtes suivantes :*

(1) Donner la liste des noms et couleurs de tous les produits.

(2) Donner les noms et les quantités des produits de couleur rouge.

NPRO	NOMP	QTES	COULEUR
100	Bille	100	Verte
200	Poupée	50	Rouge
300	Voiture	70	Jaune
400	Carte	350	Bleu

Table 4.1: Table des produits

NVEN	NOMC	NPRV	QTEV	DATE
1	Dupont	100	30	08-03-1999
2	Martin	200	10	07-01-1999
3	Charles	100	50	01-01-2000
4	Charles	300	50	01-01-2000

Table 4.2: Table des ventes - clients

NACH	NOMF	NPRA	QTEA	DATE
1	Fournier	100	70	01-03-1999
2	Fournier	200	100	01-03-1999
3	Dubois	100	50	01-09-1999
4	Dubois	300	50	01-09-1999

Table 4.3: Table des achats - fournisseurs

(3) *Donner pour chaque produit en stock, le nom du fournisseur associé.*

(4) *Donner pour chaque produit en stock en quantité supérieure à 10 et de couleur rouge, les triplets nom de fournisseurs ayant vendu ce type de produit et nom de client ayant acheté ce type de produite et nom du produit.*

(5) *Donner les noms de clients ayant acheté au moins un produit de couleur verte.*

(6) *Donner les noms des clients ayant acheté tous les produits stockés.*

(7) *Donner les produits fournis par tous les fournisseurs et achetés par au moins un client.*