LABORATOIRE
EISTI
LAπPI
EN PROCESSUS INTELLIGENTS

Chrysostome BASKIOTIS – Maria MALEK

PROLOG

EISTI, 2009-10

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 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 appel, 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 machine de type von Neumann [1]. 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 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

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.

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, c'est-à-dire d'utiliser la logique comme un langage de programmation. 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

2. « inventeur » des langages de programmation à forte dominante algorithmique, `Algol` et `Pascal`.

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, 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.

3. in Communications ACM, vol.22, no 7, july 1979, pp.424-436

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 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

LOGIQUE DES PROPOSITIONS ET PROGRAMMATION

1.1 Syntaxe de Prolog 9
1.1.1 Les faits 9
1.1.2 Les règles 10
1.1.3 Les questions 10
1.1.4 Présentation en Prolog 11
1.2 Les données 11
1.2.1 Les données simples 11
1.2.2 Les données structurées 12
1.2.3 Types des données 13
1.3 Fonctionnement (simplifié) de Prolog 13
1.4 Repésentation des nombres naturels 14
1.5 Les opérateurs de base de Prolog 16
1.6 Un exemple 17
1.7 Exercices 18

COMME nous venons de voir à l'introduction, 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.

Nous donnons ci-après un exemple de fichier source de `Prolog` :

```
porteParapluie :- ilPleut.
ilPleut.

mortel(X) :- humain(X).
existeHumain :- humain(X).
humain(socrate).
humain(aristote).

animal(X) :- lion(X).
```

Exemple des questions :

```
?- ilPleut.
yes

?- porteParapluie.
yes

?- animal(socrate).
no

?- mortel(X).
X = socrate;
X = aristote;
```

```
no

?- existeHumain.
yes
```

1.1 Syntaxe de Prolog

Dans cette section nous passons en revue les éléments de base du langage, ainsi que leur syntaxe.

1.1.1 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 arguments 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 « *;* ».

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.

1.1.2 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 :

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

où `A` et $\texttt{B}_k$ sont des faits. `A` est *l'entête* de la règle et il doit être un atome positif (c'est-à-dire n'ayant aps de négations) et $\texttt{B}_1\texttt{, } \texttt{B}_2\texttt{, ... , } \texttt{B}_n$ constituent *le corps* de la règle et ce sont des littéraux.

Cette forme de la règle est la forme qu'en logique computationnelle 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.

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 que, 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, c'est-à-dire 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.

1.1.3 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 – à savoir les faits et les règles – afin de vérifier si la question est filtrée soit par les faits qu'il possède (c'est-à-dire 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 :

`nomRelation(objet`$_1$`, objet`$_2$`, ... , objet`$_n$`) ?`

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

1.1.4 Présentation en `Prolog`

Prolog fonctionne avec des clauses de Horn. Leur syntaxe est la suivante :

- Faits
  `humain(socrate).`
  On affirme le fait que Socrate est un être humain.
- Règles
  `porteParapluie :- ilPleut.`
  On lira ce morceau de programme "je porte un parapluie SI il pleut". Ici la notation " `:-`" se lit "SI".
- Variables
  Le nom d'une variable commence toujours par une lettre majuscule. On distingue des variables
  - universelles
    `mortel(X) :- humain(X).`
    `X` ici est une variable universelle : tout être humain est mortel.
  - existentielles
    `existeHumain :- humain(X).`
    `X` est ici une variable existentielle. On cherche à savoir s'il existe un être humain.

1.2 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.

1.2.1 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 :
$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$
$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$
où 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 :
$\texttt{nomRelation}(\texttt{objet}_1, \texttt{objet}_2, \_) \leftarrow \texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$
$\texttt{nomFait}(\texttt{objet}_1, \_, \texttt{objet}_3) \leftarrow \texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$

1.2.2 Les données structurées

Les constantes et les variables sont des données brutes, sans aucune structuration. On peut aussi envisager des données structurées, par exemple 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 évidemment, 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, c'est-à-dire, 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`.

1.2.3 Types des données

Les principaux types des données en Prolog, sont les suivants :

- Des entiers. Ex. `factoriel(6).`
- Des réels. Ex. `pi(3.1415).`
- Des constantes. Ex. `socrate, toto, ... .` Notons que les nombres entiers ou flottants sont considérés comme des constantes.
- Prédicats avec termes : Exemple. Description des branches gauche et droite d'un arbre binaire.
  ```
  brancheGauche(arbre(L,R), L).
  brancheDroite(arbre(L,R),R).
  ```
  Ici `brancheGauche` et `brancheDroite` sont des prédicats, `arbre` est un foncteur.

1.3 Fonctionnement (simplifié) de Prolog

Prolog est principalement un langage interprété, ce qui signifie que l'ordinateur exécute immédiatement les commandes saisies par l'utilisateur : le code source

est immédiatement traduit en code machine. Concrètement, `Prolog` consiste en un interpréteur où on peut saisir des expressions après le prompt (?-), et un moteur d'inférence qui teste si ces expressions sont vraies ou fausses (par unification) en parcourant la base de faits (par backtracking). Le moteur d'inférence fonctionne selon une approche *top-down*, c'est-à-dire il prend le premier élément de la base, il essaie de le prouver et il continue avec le suivant.

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 ».

Pour illustrer le fonctionnement de `Prolog` considérons le programme E suivant :

```
p.
q.
```

Le programme s'écrit sous forme ensembliste `E = {p, q}`. Soit maintenant la question.

`p ∧ q`

Pour répondre à la question,Prolog applique la résolution par réfutation. Donc dans E est placée aussi la négation de la question

`E = {p, q, (¬p ∧ q)}` $= \{p, q, \neg p \vee \neg q\}$

qui nous donne la clause absurde $\perp$. En conséquence le programme induit la fbf `p ∧ q`, c'est-à-dire `E` $\models$ `p ∧ q`.

1.4 Repésentation des nombres naturels

Considérons l'univers de discours $\mathcal{U}$ composé de l'ensemble de nombres naturels et de l'opération de l'addition, munie de son élément neutre 0. Si on veut construire un langage $\mathcal{L}_0$ dont on chercherait une interprétation dans $\mathcal{U}$, on doit avoir un foncteur équivalent à l'opération de l'addition, ainsi que l'élément neutre.

Plaçons-nous dans le cadre d'un Prolog pur, c'est-à-dire d'un Prolog qui ne contient pas des formes arithmétiques. Convenons d'appeler `add/3` le prédicat qui représente l'opération d'addition dans $\mathcal{U}$[1]. Notons aussi par `zero` l'élément neutre de `add` qui est, par ailleurs, une des constantes du langage $\mathcal{L}_0$. Il nous faut aussi un prédicat, que nous appellerons `nat/1` pour caractériser les nombres naturels. On pourra ainsi écrire :

```
nat(zero).
```

pour indiquer que zero est un nombre naturel. La question qui se pose maintenant concerne les autres nombres naturels, à savoir de quelle manière nous allons représenter ces nombres. Depuis Peano, au moins, on sait que nous pouvons construire l'ensemble des nombres naturels à partir de 0 et de l'opérateur de succession $s(X) = X + 1$, où X est un nombre naturel. On peut utiliser cette même technique pour le langage $\mathcal{L}_0$. On se dote donc d'un foncteur `s/1` qui représente l'opérateur de succession dans $\mathcal{L}_0$ et, par conséquent, le programme complet de caractérisation des nombres naturels s'écrit :

```
nat(zero).
nat(s(X)) :- nat(X).
```

On peut, par exemple, poser la question

```
?- nat(s(s(s(s(zero))))).
```

et on aura comme réponse `yes`. Mais le plus rigolo c'est quand on pose la question

```
?- nat(X).
```

Dans ce cas on recupère comme réponse

```
X = zero ;
X = s(zero) ;
X = s(s(zero)) ;
X = s(s(s(zero))) ;
X = s(s(s(s(zero)))) ;
X = s(s(s(s(s(zero)))));
. . . . . .
```

c'est-à-dire l'ensemble des nombres naturels.

Avant d'avancer en programmation, essayons de voir, sous l'aspect logique des prédicats, ce que nous venons de construire. Nous avons une logique du premier ordre $\mathcal{L}_0$ dont les termes sont $\mathbb{T} = \{\texttt{zero}, \texttt{s/1}, \texttt{X}, \texttt{Y}\}$ et nous avons aussi les prédicats $\{\texttt{nat/1}, \texttt{add/3}\}$. L'interprétation I que nous avons adoptée conduit à la signification ϕ_I suivante des termes : (cf. Définition 4.3.3, p.52 du poly de logique) :

- $\texttt{zero} \rightarrow \phi_I(\texttt{zero}) = \texttt{zero}_I = 0$
- $\texttt{s(X)} \rightarrow \phi_I\left(\texttt{s}(\overline{\phi}_I(\texttt{X}))\right) = \texttt{s}_I(\overline{\phi}_I(\texttt{X})) = \overline{\phi}_I(\texttt{X}) + 1$

1. `add/3` signifie que le foncteur `add` est d'arité 3, c'est-à-dire que le nombre de ses arguments est 3.

– $\texttt{add}(X,Y,Z) \rightarrow \phi_I\left(\texttt{add}(\overline{\phi}_I(X),\overline{\phi}_I(Y),\overline{\phi}_I(Z))\right) = \texttt{add}_I\left(\texttt{add}(\overline{\phi}_I(X),\overline{\phi}_I(Y),\overline{\phi}_I(Z)\right)$ d'où $\overline{\phi}_I(X) + \overline{\phi}_I(Y) = \overline{\phi}_I(Z)$.

Pour écrire le programme de l'addition de deux naturels en Prolog nous allons utiliser les deux remarques suivantes :

(1) Si on ajoute 0 à une valeur X, le résultat ets X. En Prolog on a

```
add(zero,X,X).
```

(2) Si on a X+Y=Z, alors (X+1) + Y = (Z+1). En Prolog on a

```
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

On peut, par exemple, faire l'addition

```
?- add(s(s(s(s(s(zero))))), s(s(zero)), Z).
```

et avoir comme réponse

```
Z = s(s(s(s(s(s(s(zero)))))))
```

Mais on peut aussi faire la soustraction entre le troisième terme et le premier

```
?- add(s(s(s(s(s(zero))))), Y, s(s(s(s(s(s(s(zero))))))).
```

avec réponse

```
Y = s(s(zero))
```

et aussi entre le troisième terme et le deuxième

```
?- add(X, s(s(zero)), s(s(s(s(s(s(s(zero))))))).
```

avec réponse

```
X = s(s(s(s(s(zero)))))
```

1.5 Les opérateurs de base de `Prolog`

Toutes les clauses se terminent par un point ".". Les atomes et les symboles de fonctions commencent par une lettre minuscule et les variables par une lettre majuscule. Le tiré bas (ou souligné) représente une variable anonyme : on l'utilise pour indiquer que la variable existe mais on n'a pas besoin de conna
c citre sa valeur. Le ET est représenté par une virgule, le OU par un point-virgule.

Les opérateurs d'égalité et de comparaison sont les suivnats :

– **=** Unifie deux termes : `X = Y` (`X` et `Y` sont des termes quelconques). Si `X` et `Y` sont non instanciés, `X = Y = _`(variable anonyme). Si `X` instanciée à un atome et `Y` est non instancié, `Y` s'instancie à l'instance de `X`. Les variables instanciées sont toujours égales à elle-mêmes, par exemple `2=2` est vrai.

– **==** Vérifie si 2 termes sont identiques

– **<** Vérifie si un terme est plus petit qu'un autre

– **=<** Vérifie si un terme est plus petit ou égal à un autre

– **>** Vérifie si un terme est plus grand qu'un autre
– **>=** Vérifie si un terme est plus grand ou égal à un autre
– \= Teste la non-unification de 2 termes
– \== Teste si 2 termes ne sont pas identiques

Les opérateurs arithmétiques :

– **<** Plus petit
– **=<** Plus petit ou égal
– **>** Plus grand
– **>=** Plus grand ou égal
– **= :=** Égal
– =\= Différent
– **is** Evaluation d'une expression et assignation à une variable

1.6 Un exemple

Considérons le graphe de la figure 1.1 :

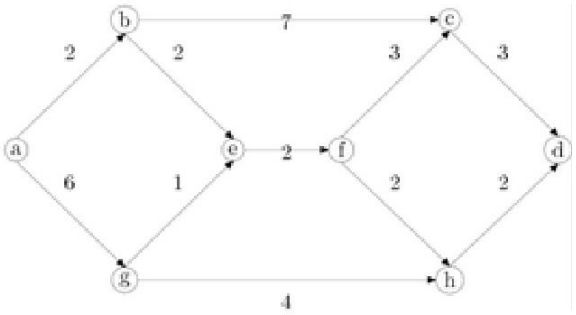


FIGURE 1.1 – Graphe pour un réseau

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

PROGRAMME 1.6.1

```
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.

1.7 Exercices

EXERCICE 1.1 *Écrire le programme Prolog* `mult(X,Y,Z)` *qui calcule le produit de deux naturels* `X`, `Y` *et stocke le résultat dans* `Z`.

Exemple : `mult(s(s(zero)), s(s(zero))), Z)` *donnera comme résultat* `Z = s(s(s(s(s(zero)))))`.

EXERCICE 1.2 *En utilisant le programme précédent, calculer le résultat de la division de deux naturels* `X`, `Y`. *On fait l'hypothèse que* `X` *divise* `Y`, *c'est-à-dire que Y/X est un entier.*

EXERCICE 1.3 *Écrire le programme Prolog* `egal(X,Y)` *qui teste si les variables* `X`, `Y` *ont la même valeur.*

EXERCICE 1.4 *Écrire le programme Prolog* `pair(X)` *qui teste si la valeur de* `X` *est paire. On fera l'hypothèse que* `zero` *est pair.*

EXERCICE 1.5 *Même chose pour le prédicat* `impair(X)`.

EXERCICE 1.6 *Examiner la base de données de l'exemple 1.6. Établir la notion d'un sommet successeur d'un autre. Par exemple le sommet d est successeur du sommet a.*

EXERCICE 1.7 *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 propositionnelle.

(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 1.8 « A moins que nous continuons la politique de soutien des prix, nous perdrons les voix des agriculteurs. Si nous continuons cette politique, la surproduction continuera, sauf si nous contingentons la production. Sans les voix des agriculteurs, nous ne serons pas réélus. Donc si nous sommes réélus et si nous ne contingentons la production, il continuera d'y avoir surproduction. »

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

(2) Écrire un programme en `Prolog` *qui teste la validité logique de la conclusion.*

EXERCICE 1.9 *La stratégie de recherche en profondeur d'abord de* `Prolog` *est incomplète, c'est-à-dire elle ne garantit pas de trouver une solution si elle existe car ce type de recherche peut conduire à une branche infinie de l'arbre de résolution.*

Pour vérifier cette incomplétude on utilise la base de données suivante Pour examiner d'abord la branche infinie, il faut permuter les clause (2) et (3).

(1) a :- b , c.

(2) b :- c , d.

(3) b :- a , d.

(4) c :- e.

(5) d.

(6) e.

Supposons que le but à démontrer est b.

Établir l'arbre de résolution pour ce programme et aussi quand on échange la place des clauses 2 et 3.

EXERCICE 1.10 *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

LOGIQUE DES PRÉDICATS ET PROGRAMMATION

2.1 Les prédicats de base de Prolog 22
2.1.1 Entrées-sorties 22
2.1.2 Opérations en Prolog 22
2.1.3 Prédicats extralogiques 23
2.2 Un exemple 23
2.3 Sémantique de Prolog 25
2.3.1 Ordre des clauses 26
2.3.2 Ordre des buts 26

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 par exemple les successeurs au sens large d'un sommet donné.

La programmation en Prolog consiste essentiellement à construire 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.

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*.

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
    ?           op\'erateur␣de␣requ\^ete
␣␣␣␣X␣=␣Y␣␣␣␣␣\'egalit\'e␣par␣unification
␣␣␣␣X␣==␣Y␣␣␣␣egalite␣sans␣unification
␣␣␣␣X␣=/=␣Y␣␣␣X␣est␣non␣identique␣\'a␣Y
␣␣␣␣X␣<␣Y
␣␣␣␣X␣=<␣Y
␣␣␣␣X␣>=␣Y
␣␣␣␣X␣>␣Y
␣␣␣␣X␣=..␣Y␣␣␣op\'erateur \textsl{div} X est le nom d'un␣foncteur␣et␣Y
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣contient ,␣sous␣forme␣de␣liste ,␣ses␣\'el\'ements
```

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 \'a la base de donn\'ees␣le␣fait␣X
␣␣␣␣asserta(X)␣␣Ajouter␣au␣debut␣de␣la␣base␣de␣donnees␣le␣fait␣X
␣␣␣␣assertz(X)␣␣Ajouter␣\'a␣la␣fin␣de␣la␣base␣de␣donnees␣le␣fait␣X
␣␣␣␣retract(X)␣␣Supprimer␣de␣la␣base␣de␣donn\'ees 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 :

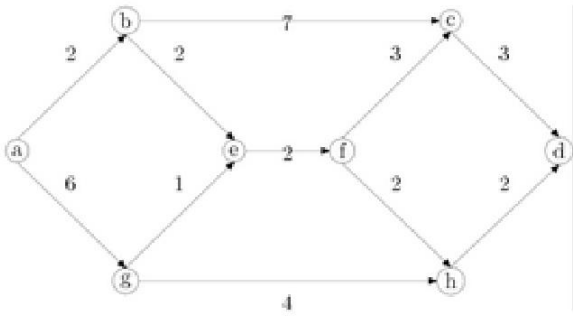


FIGURE 2.1 – Graphe pour un réseau

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 de 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,_).
```

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 de 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.

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 2.2 *Examiner le résultat des questions :*
`successeur(c,Y).`
`successeur(X,c).`
Pourriez-vous faire la liaison avec le modèle minimal de 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 de Herbrand du programme. En effet notons par $\mathcal{U}(E)$ et$\mathcal{B}(E)$ l'univers et la base de Herbrand du programme E respectivement. Une interprétation I de E est un sous-ensemble de la base de Herbrand. Une interprétation I est un modèle de 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 de 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].

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.

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 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 30
3.2 La recursivité 31
3.3 Techniques de recursivité 33
3.3.1 Recursivité pour les fonctions numériques 34
3.3.2 Recursivité simple 34
3.3.2.1 Arrêt lorsque la liste est vide . . . 35
3.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé 35
3.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte 36
3.3.3 Recursivité multiple 37
3.4 Exercice 38

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 récursif est composé de deux parties :

- Une partie concernant le ou les tests d'arrêt.
- Une partie concernant les appels récursifs 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 øeuvre 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 récursifs 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 + N.
```

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.
```

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␣\'el\'ement␣a␣fait␣partie␣de␣la␣liste'));
        (write ('L␣\'el\'ement␣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␣\'el\'ement␣a␣fait␣partie␣de␣la␣liste')), nl.
toto :- not(membre(a, [b,a,c])),
        write ('L␣\'el\'ement␣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 récursivement 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'arr\^et

␣␣␣␣membreT(X,␣[Tete␣|␣Reste])␣:-␣atom(Tete),
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣membreT(X,␣Reste).␣%␣Reste␣-␣recursivit\'e

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

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 *Écrire en Prolog :*

(1) Un programme qui permet de tester si un nombre donné est un naturel.

(2) Un programme `s(X)` *qui permet d'obtenir le successeur symbolique de* `X`.

(3) Un programme qui fournit le plus grand entre deux naturels.

(4) Un programme qui permet de tester la parité d'un naturel.

(5) Un programme qui permet d'écrire le prédicat `plus` *déjà étudié en Logique Computationnelle.*

(6) Un programme qui `fact` *permet d'écrire de façon symbolique, le factorielle d'un naturel.*

EXERCICE 3.2 *Structurer une base de connaissances de sorte que si nous avons comme faits qu'un cheval est plus rapide qu'un chien qui, à son tour, est plus rapide qu'un lapin, alors on peut répondre, pour des animaux particuliers, lequel est le plus rapide.*

Étude de la récursivité simple sur des listes.-
Il existe trois types de recursivité simple suivant le critère d'arrêt.

(1) *Arrêt lorsque la liste est vide.*

EXERCICE 3.3 *Disséquer une liste.*

Application : `disseq([b,a,c]) ?; disseq([]) ?; disseq([b, [a, c]]) ?`

EXERCICE 3.4 *Compter le nombre d'éléments d'une liste.*

Application : `longueur([b,a,c],N) ?; longueur([],N) ?; longueur([b, [a, c]]) ?`

EXERCICE 3.5 *Concatener deux listes en créant une troisième.*
Application : `concat([b,ac],[d,e,f],L) ?; concat([],[bac],L) ?; concat([],[b,a,c],[d,e,f]) ?;concat(L,[b,a,c],L1) ?; concat([],[],L) ?; concat (L,[],L1) ?`

(2) *Arrêt lorsqu'un élément spécifique a été retrouvé.*

EXERCICE 3.6 *Vérifier si un élément est membre d'une liste.*

Application : `membre (a,[b,a,c]) ?; membre (a,[a]) ?; membre(a,[b,c,d]) ?; membre(a,[]) ?`

EXERCICE 3.7 *Vérifier si un élément est membre d'une liste et donner sa position dans la liste.*
Application : `membre (a,[b,a,c],Pos) ?; membre (a,[a],Pos) ?; membre(a,[b,c,d],Pos) ?; membre(a,[],Pos) ?`

EXERCICE 3.8 *Supprimer un élément de la liste et obtenir une nouvelle liste sans l'élément supprimé.*

Application : `suppres(a,[b,a,c],L) ?; suppres(a,[b,c,d],L) ?; suppres(a,[a],L) ?; suppress(a,[],L) ?`

EXERCICE 3.9 *Remplacer dans une liste un élément par un autre et obtenir une nouvelle liste.*
Application : `remplace(a,[b,a,c],z,L) ?; remplace(a,[b,c,d],z,L) ?; remplace(a,[],z,L) ?; remplace(a,L,z,[x,y,z]) ?`

EXERCICE 3.10 *Inserer dans une liste, après un élément donné, un autre élément et obtenir une nouvelle liste.*
Application : `insereApres(a,[b,a,c],z,L) ?; insereApres(c,[b,a,c],z,L) ?; insereApres(a,[b,c,d],z,L) ?; insereApres(a,[],z,L) ?; insereApres(a,L,z,[x,y,a,z]) ?`

(3) *Arrêt lorsqu'une position spécifiée est atteinte.*

EXERCICE 3.11 *Récupération du n-ième élémnt d'une liste.*

Application : `position(0,[b,a,c],X) ?; position(1,[b,a,c],X) ?;`
`position(2,[b,a,c],X) ?; position(3,[b,a,c],X) ?;`
`position(4,[b,a,c],X) ? position(2,[],X) ?`

EXERCICE 3.12 *Supprimer le n-ième élément d'une liste.*

Application : `suppressN(0,[b,a,c],X) ?; suppressN(1,[b,a,c],X) ?;`
`suppressN(2,[b,a,c],X) ?; suppressN(3,[b,a,c],X) ?;`
`suppressN(4,[b,a,c],X) ? suppressN(2,[],X) ?`

EXERCICE 3.13 *Inserer dans une liste, avant une position donnée, un autre élément et obtenir une nouvelle liste.*

Application : `insereAvant(0,z,[b,a,c],X) ?; insereAvant(1,z,[b,a,c],X) ?;`
`insereAvant(2,z,[b,a,c],X) ?; insereAvant(3,z,[b,a,c],X) ?;`
`insereAvant(4,z,[b,a,c],X) ? insereAvant(2,z,[],X) ?;`
`insereAvant(3,c,L,[b,a,c,x]) ?`

4

TECHNIQUES DE PROGRAMMATION EN PROLOG

4.1 Fonctionnement de la coupure 42
4.1.1 Coupure rouge et coupure verte 43
4.1.2 Arrêt après la première solution 45
4.1.3 Coupure et clause conditionnelle 46
4.2 La négation comme échec 47
4.3 Opérations numériques avec les listes 50
4.4 Opérations avec les matrices 51
4.5 Prédicats ensemblistes 53
4.5.1 `setof` 54
4.5.2 `bagof` 55
4.5.3 `findall` 55
4.6 Traitements des foncteurs 56

NOUS étudions dans ce chapitre, en premier lieu, un élément particulier de la programmation en `Prolog` et qui est utilisé pour le contrôle dans un programme. La coupure, notée par « ! » en `Prolog`, sert à réduire l'espace de recherche de solutions par la stratégie utilisée par `Prolog`. Nous rappelons que la stratégie de résolution utilisée consiste à développer, pour un but donné, un arbre de résolution SLD dont la racine est le but. Cet arbre est exploré selon la méthode « en profondeur d'abord ». L'utilisation de la coupure dans une clause sert à élaguer une partie des branches de l'arbre dont la clause est la racine.

Le deuxième objectif de ce chapitre est l'étude de l'utilisation de la négation – `not` en `Prolog` – dans le corps d'une clause. Cette utilisation permet de décrire des situations de succés qui seront conditionnés par des situations d'échec d'autres prédicats.

Le troisième objectif est d'effectuer des opérations avec des listes numériques et, ensuite, d'utiliser les listes pour représenter des matrices.

On termine en présentant des prédicats concernant les ensembles, la programmation logique sous contraintes et le traitement des foncteurs.

4.1 Fonctionnement de la coupure

Pour l'enchaînement des buts et des clauses, Prolog dispose d'une facilité en ce qui concerne le contrôle de la circulation de l'information. Il s'agit du symbole cut, noté par !. Il peut être insérer dans le programme comme un but mais ses effets sont à retardement. L'exemple suivant présente le comportement du cut. Soit une primitive exprimée à l'aide de deux clauses suivantes :

```
B :- C1, C2, . . . ,Ci, !, C(i+1), . . ., Cn .
B :- D1, D2 . . . , Dm .
```

Si une de conditions C_1, C_2, . . . ,C_i n'est pas satisfaite, alors le contrôle du programme passe à la clause suivante. Si par contre toutes les conditions C_1, C_2, . . . ,C_i sont satisfaites, alors le cut est atteint et, en tant que but, est réalisé. Ce qui aura comme conséquence l'interdiction partielle du backtracking sur cette primitive. Ainsi le backtracking sur les conditions C_{i+1}, . . . , C_n est permis mais il n'est plus possible sur les conditions C_1, C_2, . . . ,C_i, ni d'ailleurs sur les conditions des clauses qui s'en suivent, c'est-à-dire sur les conditions D_1, D_2, . . . , D_m.

La coupure modifie, en fait, le comportement du programme et elle peut même le rendre logiquement erroné. L'exemple suivant fournit l'équivalent logique d'un programme sans ou avec copure.

```
p :- a, b.
p :- c.
```

Ce programme est équivalent à la fbf $(a \wedge b) \vee c$. Si on place une coupure à la première clause, on aura le programme

```
p :- a, !, b.
p :- c.
```

La fbf équivalente est maintenant $(a \wedge b) \vee (\neg a \wedge c)$. Si on change l'ordre des clauses, on a

```
p :- c.
p :- a, !, b.
```

qui est équivalent à la fbf $(c \vee a \wedge b)$, c'est-à-dire est équivalent au programme initial.

L'exemple suivant fournit un traitement exhaustif de cut.

PROGRAMME 4.1.1

```
(1) p(a).

(2) p(X):- q(X),r(X).
(3) p(X):-u(X).

(4) q(a)
(5) q(b).
(6) q(c).

(7) r(a).
(8) r(b).
(9) r(d).

(10) u(d).
```

Nous savons que dans le cas normal la réponse à la requête `?-p( X)` donne `X = a, X = a, X = b, X = d`. Si nous changeons la première clause par la clause : `p(a) :-!.`
la seule réponse fournie sera `X = a`, car la seule branche développée par l'arbre de résolution serait la branche (1). Remplaçons maintenant la deuxième clause par : `p(X) :-!, q(X), r(X).`
Les réponses obtenues dans ce cas seront `X = a, X = a, X = b`, car les branches (3) et (10) de l'arbre de résolution serraient élaguées. Si nous déplaçons la coupure vers la droite : `p(X) :- q(X),!, r(X).`
nous obtenons les réponses `X = a, X = a`, car dans ce cas ce sont les branches (3) et (5) à (10) qui seront élaguées.

Les règles qu'il faut suivre pour l'écriture des programmes avec cut sont les suivantes :

- Chaque clause doit refléter une règle de logique qui détermine la vérité des faits qui filtre le but. Dans ce cas, nous pouvons ajouter des cuts pour éviter de parcourir, dans l'arbre de la résolution, des branches de calculs inutiles ou superflus.
- Placer le cut le plus proche possible du début de la clause.
- Il ne faut pas utiliser des cuts à la dernière clause d'une primitive.

4.1.1 Coupure rouge et coupure verte

Il y a en réalité deux sortes de cut : le *cut vert* et le *cut rouge*. Le premier sert à élaguer des branches sur lesquelles on sait qu'il n'y ait pas de solutions mais il ne modifie pas la sémantique du programme. Par exemple soit le programme qui trouve le maximum de deux nombres.

PROGRAMME 4.1.2

```
minimum(X,Y,X) :- X=<Y.
minimum(X,Y,Y) :- X>Y.
```

Étant donné que les deux clauses sont exclusives, on sait que si la première réussit, la seconde échouera. Par conséquent nous pouvons éviter son examen par Prolog en utilisant une coupure comme ci-après.

PROGRAMME 4.1.3

```
minimum(X,Y,X) :- X=<Y, !.
minimum(X,Y,Y) :- X>Y.
```

Cette coupure ne change en rien le sens du programme, elle accélère seulement son exécution.

Par contre le cut rouge peut élaguer des branches sur lesquelles il y a des solutions et, par conséquent, modifie le comportement du programme, c'est-à-dire change la semantique du programme.

Reprenons l'exemple précédent. Du fait de l'introduction de la coupure à la première clause, on sait que si on arrive à la seconde clause, on a forcement `X > Y`. Donc, nous pouvons penser, à tort comme on le verra par la suite, que le test `X > Y` effectué par cette clause est superflu. Ainsi on peut avoir le programme

PROGRAMME 4.1.4

```
minimum(X,Y,X) :- X=<Y, !.
minimum(X,Y,Y) .
```

Ici le sens du programme est modifié et à tel point que sa réponse n'est pas toujours correcte comme, par exemple, à la question `?minimum(2,5,5)`. Ceci est dû au fait que la première clause n'est pas satisfaite, mais la deuxième par contre est satisfaite.

ASCÈSE 4.1 *Compléter la 1e clause du programme précédent afin qu'il devient correct sans modifier la seconde clause.*

On verra mieux l'usage du cut rouge avec le programme suivant qui détermine une liste qui est l'intersection, au sens ensembliste du terme, de deux autres listes.

PROGRAMME 4.1.5

```
intersection([],_,[]).
intersection([H|T],L,[H|T2]) :- membre(H,L),
                                intersection(T,L,T2).
intersection([H|T],L,T2) :-     intersection(T,L,T2).
```

À la question `?- intersection([a,b,c], [a,b,c], L).`, nous avons les réponses :

```
L = [a, b, c] ;
L = [a, b] ;
L = [a, c] ;
L = [a] ;
L = [b, c] ;
L = [b] ;
L = [c] ;
L = []
```

ce qui, bien entendu, n'est pas souhaitable.

L'utilisation d'un cut rouge peut rectifier la situation, comme ci-après :

PROGRAMME 4.1.6

```
intersection([],_,[]) :- !.

intersection([H|T],L,[H|T2]) :- membre(H,L), !,
                                intersection(T,L,T2).

intersection([_H|T],L,T2) :- intersection(T,L,T2).
```

Le cut au test d'arrêt est obligatoire si on ne veut pas que l'appel avec des listes qui contiennent aussi des variables dégénère.

Ici il s'agit d'un cut rouge car il modifie les résultats, c'est-à-dire la sémantique du programme.

4.1.2 Arrêt après la première solution

Le coupure peut être utilisée pour que `Prolog` s'arrête dès qu'il a trouvé une solution.

ASCÈSE 4.2 *Écrire le programme `membre` qui s'arrête dès qu'il trouve le premier membre de la liste.*

La première solution d'un but peut être expressément construit à l'aide du prédicat `premiereSolution/1` comme suit :

PROGRAMME 4.1.7

```
premiereSolution(But) :- call(But), !.
```

Ainsi pour le programme de l'ascèse précédente on a `premiereSolution(membre(X,[a, b, c])).`

4.1.3 Coupure et clause conditionnelle

Nous avons utilisé dans le programme 5.2.2 la coupure pour construire l'intersection ensembliste de deux listes. Néanmoins ce programme, malgré les lettres de noblesse qu'il dispose, n'est pas dépourvu des faiblesses. Par exemple la réponse à la question

`?-intersection([a, b ,c],[d, b, a],[a]).`

est `oui` !

Le problème vient du fait que l'intersection est construite de faimplicite à la deuxième clause. Une construction explicite nous permettra d'éviter ce problème. On aura donc

PROGRAMME 4.1.8

```
intersection1([],_,[]) :- !.

intersection1([H|T],L,T2) :- membre(H,L), !,
                             T2 = [H|Z],
                             intersection1(T,L,Z).

intersection1([_H|T],L,T2) :- intersection1(T,L,T2).
```

En regardant bien le programme, on constate que nous avons une construction conditionnelle du type `SI ... ALORS ... SINON` que nous pouvons exprimer à l'aide du programme suivant

PROGRAMME 4.1.9

```
siAlorsSinon(X,Y,Z) :- call(X), !, call(B).
siAlorsSinon(X,Y,Z) :- call(Z).
```

ASCÈSE 4.3 *Écrire le programme `intersection2` en utilisant le prédicat `siAlorsSinon`.*

ASCÈSE 4.4 Étude du cut rouge.

Soit la base de données :

```
ferie(mercredi, premierMai).
temps(mercredi,beau).
temps(samedi,beau).
temps(dimanche,beau).
weekend(samedi).
weekend(dimanche).
promenade(Jour) :- temps(Jour, beau),
                   weekend(Jour).
promenade(Jour) :- ferie(Jour,premierMai).
```

Pourriez-vous expliquer le comportement de `Prolog` dans chacune des situations suivantes :

(1) Si on pose la question `?-promenade(Quand).` on obtient comme réponses

```
Quand = samedi ;
Quand = dimanche ;
Quand= mercredi ;
```

Explications.

(2) Si maintenant on change la primitive `promenade` *comme suit*

```
promenade1(Jour) :- temps(Jour, beau),
                    weekend(Jour),!.
promenade1(Jour) :- ferie(Jour,premierMai).
```

on a comme réponse

```
Quand = samedi;
```

Explications.

(3) Si on utilise

```
promenade2(Jour) :- temps(Jour, beau), !,
                    weekend(Jour).
promenade2(Jour) :- ferie(Jour,premierMai).
```

on n'obtient aucune réponse. Pourquoi ?

(4) Si on écrit

```
promenade3(Jour) :- !, temps(Jour, beau),
                    weekend(Jour).
promenade3(Jour) :- ferie(Jour,premierMai).
```

on obtient la réponse

```
Quand = samedi;
Quand = dimanche;
```

Explications.

4.2 La négation comme échec

Le coupe-choix permet d'introduire la négation. Par exemple le programme suivant introduit le prédicat `different` :

```
dif(X, X) :-!, fail.
dif(X,Y).
```

Nous pouvons aussi introduire la négation non :

```
non(But) :- call(But),!, fail.
non(But).
```

En réalité la négation introduite de cette façon n'est pas une vraie négation logique car elle est fondée sur le succés de l'échec du but ou, en d'autres termes,

un but est faux si nous pouvons démontrer que nous ne pouvons pas le démontrer. Le problème a déjà été évoqué en Logique Computationnelle, chapitre 7, où nous avons notamment écrit :

« Considérons un programme défini E et soit $\mathcal{B}_E$ la base de Herbrand de E. [...] Comme E est un programme défini, c'est-à-dire qui contient des connaissances positives, la base de Herbrand contient seulement des faits positifs. Par conséquent on n'a pas la possibilité d'induire des connaissances négatives. Malgré tout nous avons besoin de savoir, par des méthodes d'induction logique, qu'un fait n'existe pas. Il s'agit essentiellement d'un problème d'interprétation de la négation, c'est-à-dire trouver une méthode qui permet d'obtenir de l'information négative. Il existe plusieurs méthodes mais pour ce cours introductif nous allons nous restreindre aux trois plus anciennes parues simultanément dans le livre *Logic and Data Bases*, édité par H. Gallaire et J. Minker en 1978.

– *Hypothèse du monde fermé* (CWA – Closed World Assumption) qui fut introduite par R. Reiter. Selon cette hypothèse – qui est, en fait, une règle – toute connaissance qui ne fait pas partie explicitement d'une base de données ou qu'il ne peut pas être induite logiquement de cette même base de données, n'existe pas et donc c'est sa négation qui a la valeur de vérité « vraie ». Il s'agit d'une hypothèse très naturelle dans les bases de données relationnelles. Mais pour les bases de données logiques, que sont tous les programmes logiques, il en va tout autrement. En effet, formellement l'hypothèse du monde fermé introduit la règle d'inférence suivante :

(R-MF) $$\frac{\vdash \neg (E \vDash A)}{\vdash \neg A}$$

En toute rigueur on aurait dû conclure que $E \vDash \neg A$ si on avait la preuve que A n'est pas une conséquence logique du programme E. Or cette affirmation n'est licite que si on fait fi de la base de Herbrand $\mathcal{B}_E$ de E et qu'on tient compte seulement de la base de données du programme.En effet, en général la base de Herbrand est un ensemble infini et de ce fait le problème de savoir si une fbf est une conséquence logique d'un programme est indécidable ([1]). Nous pouvons donc en conclure que l'hypothèse du monde fermé n'est pas, en général, applicable.

– *La négation considérée comme un échec*. Il s'agit d'une idée introduite par K. L. Clark. Nous pouvons la présenter comme étant une restriction de la règle (R-MF) dans le cas fini. En effet la règle (R-MF) stipule en substance que

$\neg A$ réussit ssi A ne peut pas être prouvé

Clark a suggeré que, dans cette proposition, la négation soit conšiďéřée comme le résultat d'un échec fini, c'est-à-dire d'un échec qu'on peut obtenir, à l'aide de l'arbre de dérivation SLD, en un nombre fini d'étapes. Dans ce cas nous avons l'interprétation affaiblie suivante :

$\neg A$ réussit ssi A échoue de façon finie

De façon formelle nous avons la définition suivante :

DÉFINITION 4.2.1 *Soit E un programme défini et B un but défini. Un arbre de dérivation SLD d'échecs fini pour $E \cup \{B\}$ est un arbre qui est fini et ne contient pas de branches de succés.*

Si nous voulons établir que le programme E induit $\neg A$, il faut montrer que l'arbre de dérivation pour le but $B : \leftarrow \neg A$ est un arbre d'échecs finis. Nous introduisons ainsi un nouveau

1. ce qui signifie qu'il n' y a pas un algorithme qui, pour un programme donné E et un but B, peut répondre si $E \vDash B$ dans un temps fini. Notons que l'élève cultivé peut, avec profit, relier ce problème avec celui de l'arrêt dans les machines de Turing en se reportant à la bibliographie spécialisée.

type de résolution, la résolution SLDNF qui est une résolution SLD avec « Negation » comme « Failure ». Elle complète la résolution SLD lorsque le but est sous forme d'un atome négatif, i.e. $B :\leftarrow \neg A$, en introduisant les règles suivantes :

$\neg A$ réussit ssi A donne un arbre SLD d'échecs fini
$\neg A$ donne un arbre SLD d'échecs fini ssi A réussi

Ainsi si $\neg A$ réussit, alors il est supprimé de la question et on passe à l'examen de la suite de la question. Si par contre il échoue de façon finie, alors la question est considérée comme aboutissant à un échec. Nous avons donc, en utilisant la résolution SLDNF, les résultats suivants :

– $E \vdash B \circ \theta$ s'il existe un arbre de dérivation SLDNF de $E \cup \{B\}$ avec réponse calculée θ.
– $E \vdash \neg B$ s'il existe un arbre de dérivation SLDNF d'échecs fini pour $E \cup \{B\}$.

Telle quelle la résolution SLDNF pose le problème de sa justesse et aussi de l'équivalence entre implication sémantique et implication syntaxique. [...] »

Ainsi le fonctionnement de ce type de négation ne va pas sans quelques inconvénients. Par exemple, soit la base de données suivante :

```
innocent(toto).
occupation(jojo, ailleurs).
coupable(koko).
innocent(X) :- occupation(X, ailleurs).
coupable(X) :- occupation(X, ici).
```

Si nous posons la question `innocent(fourier).` la réponse sera `non`. Pour améliorer le programme on pense ajouter la clause suivante :

```
coupable(X) :- non(innocent(X)).
```

Alors à la question `coupable(fourier).` nous aurons comme réponse `oui`.

Avec la base de données suivante, nous avons des résultats encore plus extravagants :

```
voiture(rouge).
voiture(verte).
voiture(bleue).
decapotable(rouge).
decapotable(bleue).
berline(X) :- non(decapotable(X)).
```

À la question `?-voiture(X), berline(X).` nous avons comme réponse `oui` et à la question `?-berline(X), voiture(X).` la réponse est `non`.

La raison de ce comportement est simple. Dans la première question, lorsque le prédicat `berline(X)` est activé, la variable `X` est déjà unifiée avec une valeur, e.g. `rouge` et donc le `non` s'applique sur un prédicat dont l'argument est une constante. Dans la deuxième question le `non` s'applique sur une variable non unifiée. Prolog cherchera dans sa base de données à trouver une clause `non(decapotable(_))`, mais il n'en trouvera pas, car d'habitude dans une base des données nous n'indiquons pas les propriétés dont il est dépourvu un objet. Ainsi la recherche échoue et la réponse est non.

Nous pouvons donc en déduire une règle pour l'utilisation du `non` : Il faut utiliser le `non` uniquement avec des prédicats dont les arguments sont des constantes ou des variables déjà instaciées (exemplifiées).

4.3 Opérations numériques avec les listes

Supposons que nous voulons, pour une liste numérique, obtenir une nouvelle liste qui contient le double des éléments de la première liste. Le programme est

PROGRAMME 4.3.1

```
double([],[]).
double([X|T],[Y|L]) :- Y is 2 * X, double(T,L).
```

Si nous voulons avoir le carré des éléments de la liste, nous aurons le programme :

PROGRAMME 4.3.2

```
carre([],[]).
carre([X | T],[Y | L]) :- Y is X*X, carre(T,L).
```

Cette démarche n'est pas très performante car pour chaque type d'opération on est obligé d'inventer un programme qui n'est pas très différent du précédent. Nous pouvons envisager de faire un programme général, appelé par exemple `autofonction`, qui applique une opération quelconque aux éléments d'une liste. Il restera après d'écrire le programme qui correspond à chaque opération que nous voulons implémenter.

Le programme `autofonction` est le suivant :

PROGRAMME 4.3.3

```
autofonction([],[], _).
autofonction([T|R],[T2|Rn],Oper) :- oper(Oper,T,T,T2),
                                    autofonction(R,Rn,Oper).
```

où `Oper` est le nom de l'opération à effectuer. Nous avons par exemple comme opérations arithmétiques :

```
oper(add, X, Y, R) :- R is X + Y.
oper(soust, X, Y, R) :- R is X-Y.
oper(mult, X, Y, R) :- R is X * Y.
oper(div, X, Y, R) :- R is X / Y.
oper(max, X, Y, R) :- (X > Y, R is X); R is Y.
```

Ainsi le double d'une liste est obtenu par le programme suivant :

```
double(L, R) :- autofonction(L, R, add).
```

et le carré par

```
carre(L, R) :- autofonction(L, R, mult).
```

Nous pouvons aussi envisager un programme qui effectue une opération sur les éléments d'une liste. Nous avons pour le programme `opListe` :

```
opListe(Liste, Resultat, Oper) :- opListeAccu(Liste, 0,
                                    Resultat, Oper).
```

avec

```
opListeAccu([],N,N, _).
opListeAccu([T|R],A,N,Oper) :- oper(Oper,A,T,A1),
                               opListeAccu(R,A1,N,Oper).}
```

Ainsi la somme de tous les éléments d'une liste est donnée par le programme :

```
somme(Liste, Resultat) :- opListe(Liste, Resultat, add).
```

Nous pouvons ne pas se limiter à une liste mais envisager d'utiliser la même démarche pour effectuer une opération entre deux listes différentes. Le programme fonction serait :

```
fonction([],[],[],_).
fonction([T1|R1],[T2|R2],[X|Y],Oper) :- oper(Oper,T1,T2,X),
                                        fonction(R1,R2,Y,Oper).
```

Le programme suivant permet le calcul du produit intérieur de deux vecteurs en utilisant la primitive `fonction` :

```
produitInt(X, Y, R) :- length(X, N1),
                       length(Y,N2),
                       N1 == N2,
                       fonction(X, Y, Z, mult),
                       opListe(Z, 0, R, add).
```

4.4 Opérations avec les matrices

Nous pouvons utiliser les dévéloppements précédent pour faire des opérations avec des matrices. Premièrement pour représenter une matrice nous envisageons d'utiliser des listes composées des sous-listes. Chaque sous-liste serait une

colonne de la matrice. Ainsi la liste `[[2,1,1], [4,2,4], [7,3,5]]` représente la matrice $\begin{bmatrix} 2 & 4 & 7 \\ 1 & 2 & 3 \\ 1 & 4 & 5 \end{bmatrix}$.

Occupons-nous d'abord de la transposée d'une matrice. Nous avons le programme :

PROGRAMME 4.4.1

```
transpose([[] | _],[]).
transpose(R,[T | Rs]) :- premiereCol(R,T), suiteCol(R,H),
                         transpose(H,Rs).

premiereCol([],[]).}
premiereCol([[T | R] | Rs],[T | Ls]) :- premiereCol(Rs,Ls).

suiteCol([],[]).
suiteCol([[T | R] | Rs],[R | Ls]) :- suiteCol(Rs,Ls).
```

Pour construire le programme de multiplication de deux matrices, nous procéderons en trois étapes.

1° Multiplication d'un vecteur avec une matrice. Le vecteur se présente sous la forme d'une liste simple. Nous avons :

```
prodVecMat(A,[],[]).
prodVecMat(A,[B | R],[Res | Rs]) :- produitInt(A,B,Res),
                                    prodVecMat(A,R,Rs).
```

2° Multiplication de deux matrices qui se présentent sous forme des listes contenant des listes mais qu'on suppose qu'elles ont été préalablement arrangées pour que la multiplication soit correcte. Nous avons :

```
prodMatMat([],B,[]).
prodMatMat([A | R],B,[Res | Rs]) :- prodVecMat(A,B,Res),
                                    prodMatMat(R,B,Rs).
```

3° Enfin la multiplication des matrices :

```
multMat(A,B,C) :- transpose(A,AT), prodMatMat(AT,B,C).
```

Remarquons qu'il n'est pas nécessaire de faire le test de la conformité des dimensions des matrices, parce qu'il sera fait par le programme `prodInt`. Nous donnons ci-après un programme de calcul du nombre de lignes et des colonnes d'une matrice.

```
dimMat([T | R],NbLigne,NbCol) :- not(atom(T)), length(R,Nb),
                                 NbCol is Nb+1,
                                 length(T,NbLigne).
```

```
dimMat([T | R],NbLigne,NbCol) :- (atom(T); number(T)),
                                 NbCol is 1, length(R,Nb),
                                 NbLigne is Nb+1.
```

4.5 Prédicats ensemblistes

Un ensemble peut être vu comme une liste non ordonné où un élément n'apparaît qu'une seule fois au plus. Nous pouvons simuler un ensemble par une liste qui ne contient pas plusieurs occurrences d'un même élément. Mais cette liste, contrairement à un ensemble, elle est ordonnée.

ASCÈSE 4.5 *Écrire le programme* `list2set(L,S)` *qui transforme la liste* `L` *en un ensemble* `S`.
Exemple : `?- list2set([b,a,c,b,d,c,f,a,d], S).` *donne comme réponse* `S=[b,c,f,a,d].`

Si, à la place d'une liste, nous avons une base de données, Prolog fournit trois prédicats extra-logiques qui permettent de construire soit des ensembles, soit des multi-sets [2], en utilisant les éléments de la base.

Considérons d'abord la bases de données suivante :

```
nom(toto).
nom(koko).
nom(jojo).
nom(toto).

boisson(toto, the).
boisson(toto, lait).
boisson(toto, biere).
boisson(koko, lait).
boisson(koko, the).
boisson(koko, vin).
boisson(jojo, the).
boisson(jojo, vin).
boisson(pierre, lait).

boisson(toto, the, chaud).
boisson(toto, lait, chaud).
boisson(toto, biere, fraiche).
boisson(toto, vin, froid).
boisson(koko, lait, froid).
boisson(koko, the, chaud).
boisson(koko, vin, chaud).
boisson(jojo, the, chaud).
boisson(jojo, vin, chambre).
boisson(pierre, lait, froid).
```

Les prédicats qui fabriquent des ensembles à partir des éléments de cette base sont les suivants :

2. c'est-à-dire des ensembles où des éléments peuvent apparaître plusieurs fois

4.5.1 setof

C'est un prédicat d'arité 3 : `setof(X, nomBdD(..., X, ...),S)` où `X` est un élément de la base de données `nomBdD` et `S` est l'ensemble des éléments `X` de la base, regroupés en fonction des valeurs des autres éléments de la base et triés selon l'ordre alphabétique.

EXEMPLE 4.5.1 `?- setof(X,nom(X),S).`
fournit comme réponse
```
S = [jojo, koko, toto].
```

EXEMPLE 4.5.2 `?- setof(X,boisson(X,Y),L).`
donne les réponses suivantes
```
X = biere,
S = [toto];
X = lait,
S = [koko, pierre, toto];
X = the,
S = [jojo, koko, toto];
X = vin,
S = [jojo, koko]
```
c'est-à-dire pour chaque valeur du second élément de la base, S contient l'ensemble des personnes qui boivent cette boisson.

Si on veut obtenir l'ensemble de personnes qui boivent du lait, c'est-à-dire l'ensemble des éléments qui sont en première position dans la base de données `boisson/2`*, on doit poser la question*
```
?- setof(X,boisson(X,lait),S).
```
qui donnera comme réponse
```
S = [koko, pierre, toto]
```
Enfin si on veut avoir l'ensemble de toutes le personnes de la base boisson/2, on doit poser la question
```
?- setof(X,Y^boisson(X,Y),S).
```
qui donnera comme réponse
```
S = [jojo, koko, pierre, toto]
```
Ici l'opérateur ^ dans la notation `Y^boisson(X,Y)` *teste s'il existe un* `Y` *tel que* `boisson(X,Y)` *est vrai.*

ASCÈSE 4.6 *Écrire un programme qui fournit l'ensemble de toutes les boissons de la base de données* `boisson/2`.

ASCÈSE 4.7 *Écrire un programme qui fournit l'ensemble de toutes les personnes de la base de données* `boisson/3` *qui boivent du lait froid.*

ASCÈSE 4.8 *Écrire un programme qui fournit l'ensemble de toutes les personnes de la base de données* `boisson/3` *qui boivent une boisson froide.*

Il est à noté que s'il n'est pas possible de former un ensemble, le prédicat `setof` échoue. Tester, par exemple la clause `setof(X,boisson(X,champagne),S).`

ASCÈSE 4.9 *En utilisant `setof/3` et `member/2` écrire les programmes Prolog qui permettent :*

(1) de tester si un ensemble est sous-ensemble d'un autre ensemble.

(2) de tester l'égalité entre deux ensembles.

(3) de faire l'union de deux ensembles.

(4) de faire l'intersection de deux ensembles.

4.5.2 bagof

Prédicat d'arité 3 qui a le même comportement que `setof` mais il retourne des multi-sets au lieu des ensembles. Ces multi-sets ne sont pas ordonnés.

EXEMPLE 4.5.3 `?- bagof(X,nom(X),S).`
fournit comme réponse
`S = [toto, koko, jojo, toto].`

ASCÈSE 4.10 *Poser les questions suivantes et comparer les réponses avec celles obtenues avec `setof`.*

(1) `?- bagof(X,boisson(X,Y),L).`

(2) `?- bagof(X,boisson(X,lait),S).`

(3) `?- bagof(X,Y^boisson(X,Y),S).`

ASCÈSE 4.11 *Écrire un programme qui fournit le multi-set de toutes les boissons de la base de données `boisson/2`.*

ASCÈSE 4.12 *Écrire un programme qui fournit le multi-set de toutes les personnes de la base de données `boisson/3` qui boivent du lait froid.*

ASCÈSE 4.13 *Écrire un programme qui fournit le multi-set de toutes les personnes de la base de données `boisson/3` qui boivent une boisson froide.*

Comme avec `setof`, s'il n'est pas possible de former un ensemble, le prédicat `bagof` échoue. Tester, par exemple la clause `bagof(X,boisson(X,champagne),S).`

4.5.3 findall

Prédicat d'arité 3 qui fournit la liste de tous les éléments d'une base de données qui se trouvent à une position fixée.

EXEMPLE 4.5.4 `?- findall(X,boisson(X,Y),S).`
fournit comme réponse tous les éléments qui sont en première position dans boisson/2 :
`S = [toto, toto, toto, koko, koko, koko, jojo, jojo, pierre]`

L'opérateur ^ n'est pas pris en considération par `findall`

EXEMPLE 4.5.5 `?- findall(X,lait^boisson(X,Y),S).`
fournit la même réponse que précédemment
`S = [toto, toto, toto, koko, koko, koko, jojo, jojo, pierre]`

Contrairement aux deux prédicats précédents, `findall` s'il n'est pas possible de former un ensemble, retourne la liste vide []. Tester, par exemple la clause `findall(X,boisson(X,champagne),S).`

ASCÈSE 4.14 *Écrire un programme pour la négation `not` en utilisant le prédicat `findall`.*

Pouvons-nous écrire le même programme en utilisant le prédicat `setof`? Si oui, écrivez et tester le programme. Si non, expliquer les raisons.

4.6 Traitements des foncteurs

Le prédicat `univ`, noté `=..`, sert à composer ou décomposer les termes d'un foncteur. Ainsi
`?- somme(X,Y,R) =.. L.`
fournit comme résultat
`L = [somme, X, Y, R]`
et
`?- T =.. [somme,X,Y,R].`
a comme réponse `T = somme(X, Y, R)`

L'exemple suivant montre deux utilisations de ce prédicat.

EXEMPLE 4.6.1 `?- a*b+c =..[F,X,Y].` *a comme réponse*

```
F = +,
X = a*b,
Y = c
```

et `?- a*(b+c) =..[F,X,Y].` *a comme réponse*

```
F = *,
X = a,
Y = b+c
```

Ce prédicat est surtout utilisé pour l'analyse lexicale

Le prédicat `foncteur` appliqué à un prédicat détermine le nom du prédicat et son arité. Ainsi

```
?- functor(somme(X,Y,R), NomFoncteur, Arite).
```

donne comme réponse
`NomFoncteur = somme,`
`Arite = 3`

5

PROGRAMMATION LOGIQUE SOUS CONTRAINTES (PLC)

5.1 PLC sur des domaines finis 59
5.2 PLC sur les nombres rationnels et réels 64
5.3 CHR : Constraint Handling Rules 66
5.4 Références 67
5.5 Exercices 68

La programmation logique sous contraintes (PLC) est une généralisation de la programmation logique qui cherche à doter la programmation logique de l'efficacité des méthodes de la résolution des contraintes, de sorte que la résolution d'un problème soit aussi rapide qu'avec les langages impératifs et, au contraire, le temps de développement beaucoup plus court. Pour ce faire on rajoute, à côté du mécanisme d'inférence de Prolog, un solveur des contraintes qui peut examiner les contraintes non seulement dans le programme, mais aussi dans la question. La PLC prend en considération d'autres structures mathématiques en plus de l'univers de Herbrand.

La PLC a comme objectif de calculer des valeurs des variables $\mathcal{X} = \{X_1, \ldots, X_n\}$ qui appartiennent aux domaines $\mathcal{D} = \{D_1, \ldots, D_n\}$ respectivement et qui satisfont à un ensemble des contraintes $\mathcal{C} = \{C_1(X_1, \ldots, X_n), \ldots, C_m(X_1, \ldots, X_n)\}$. Ce que nous appelons contraintes sont des relations logiques entre les variables, à savoir une contrainte est une fonction

$$C_i : D_1, \ldots, D_n \rightarrow \{vrai, faux\} = \{1, 0\}$$

Un tuple $S = \{x_1, \ldots x_n\}$ est une *solution* pour le système $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ ssi $\forall c_i \in \mathcal{C}$: $c_j(S) = 1$.

En Prolog pour résoudre des problèmes de PLC on cherche de manière itérative, à trouver une distribution des valeurs des variables dans leur domaine res-

pectif, de sorte que les contraintes soient satisfaites.

Il est important de ne pas confondre programmation logique sous contraintes et programmation mathématique. La PLC est un programme qui code une méthode pour résoudre un problème particulier. En tant que programme, le code contient

- un ensemble des variables ;
- des contraintes entre ces variables, et
- des programmes qui indiquent la manière selon laquelle les variables doivent se modifier afin de pouvoir trouver des valeurs pour ces variables qui satisfont aux contraintes.

La difficulté de l'approche Prolog "pure" vient du fait que les objets sémantiques manipulés par Prolog sont déclarés de manière explicite. Or les contraintes expriment de manière implicite les relations entre ces objets sémantiques. Par exemple écrivons en Prolog, le programme qui calcule le minimum entre deux valeurs numériques. On a

PROGRAMME 5.0.1

```
mini(X,Y,Z) :- X =< Y, X = Z.
mini(X,Y,Z) :- Y =< X, Y = Z.
```

À la question

```
?- mini(1,2,Z).
```

on obtient la réponse

```
Z = 1
```

Mais à la question

```
?- mini(X,2,1).
```

on obtient la réponse

```
ERROR: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, 2, 1) ? Exception details
Exception term: error(instantiation_error, context(system: (=<)/2, _G353))
       Message: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, 2, 1) ? Can't ignore goal at this port
```

et aussi à la question

```
?- mini(X,X,Z).
```

on obtient la réponse

```
ERROR: =</2: Arguments are not sufficiently instantiated
   Exception: (6) mini(_G286, _G286, _G288) ? abort
% Execution Aborted
```

Manifestement Prolog ne sait pas interpréter les relations qu'elles existent entre `X`, `Y` et `Z` et s'expriment à travers les conditions des règles et qui lui auraient permis d'établir que `X = 1` à la deuxième question et que `Z = X` à la troisième.

Si on utilise la bibliothèque des contraintes sur les nombres rationnels, et on écrit le programme

```
:- use_module(library(clpq)).

miniC(X,Y,Z) :- {X =< Y, X =:= Z}.
miniC(X,Y,Z) :- {Y =< X, Y =:= Z}.
```

dont la syntaxe s'éclaircira par la suite, on a

```
?- miniC(X,2,1).
X = 1

?- miniC(X,2,1).
X = 1

9 ?- miniC(X,X,Z).
{Z=X}
```

c'est-à-dire Prolog arrive maintenant à interpréter les relations de manière implicite aussi.

Une expression (par exemple une expression arithmétique) est un terme formé en utilisant l'alphabet du domaine (les nombres dans le cas des expressions arithmétiques). Une contrainte est une formule du type $s \otimes t$ où par $\otimes$ on désigne un opérateur du domaine (par exemple pour les contraintes arithmétiques les opérateurs sont <, =<, =>, >,$\neq$).

Pour faire de la programmation avec contraintes en Prolog, il faut utiliser des librairies spécialisées, qui sont les suivantes :

- `clp/bounds` : Contraintes portant sur des domaines des nombres entiers.
- `clp/clp_distinct` : Contraintes portant sur des nombres entiers. Elle n'est pas très différente de `clp/bounds`
- `clp/clpfq` : Contraintes sur des nombres rationnels.
- `clp/clpqr` : Contraintes sur des nombres réels.

Pour pouvoir utiliser la librairie adéquate, il faut la charger, à l'aide de la requête `:- use_module`. On a, par exemple pour la librairie `clp/bounds` :

```
:- use_module(library('clp/bounds')).
```

Nous présentons ci-après les principales librairies.

5.1 PLC sur des domaines finis

Les domaines finis sont particulièrement utiles pour la résolution des problèmes combinatoires et de problèmes de distribution, de planification, etc.

L'appel au solveur se fait par la requête :

```
:- use_module(library(clpfd)).
```

Nous donnons ci-après la liste des principaux prédicats.

- `Expr op Expr`, où `op` est un des opérateurs `#=, #\=, #<, #=<, #>, #>=`
  Exemples :

```
X in 1..2, Y in 3..5, X #=<Y, Y #=< B, X+Y #= T.
X in 1..2,
X+Y #=T,
Y in 3..5,
B #>=Y,
B in 3..sup,
T in 4..7.
```

- `domain(+Var, +Min, +Max)` Les variables dans la liste `Var` sont dans `[Min, Max]`.
- Contraintes propositionnelles. Ce sont
  - `#\Q` .- Vraie si la contrainte `Q` est fausse.
  - `P #/\ Q` .- Vraie si les contraintes `P` et `Q` sont vraies toutes les deux.
  - `P #\/ Q` .- Vraie si au moins une de contraintes `P, Q` est vraie.
  - `P #==> Q, Q #<== P` .- Vraie si `P` implique `Q`.
  - `P #<==> Q` .- Vraie si les contraintes `P` et `Q` sont simultanément vraies ou fausses.
- `labeling(+Options, +Vars)` Indique à Prolog de tester systématiquement toutes les valeurs des domaines finis des variables qui sont dans `Vars`, afin de trouver des solutions compatibles avec les contraintes des ces variables. `Options` est une liste d'options dont le détail se trouve dans le manuel de SWI-Prolog, annexe A.7
- `label(+Vars)` Équivalent à `labeling([], +Vars)`.
- `all_different(+Vars)` Les variables sont distinctes deux à deux.
- `sum(+Vars, +Rel, ?Expr)` La somme des valeurs des variables de la liste `Vars` est en relation `Rel` avec l'expression `Expr`.
- `tuples_in(+Tuples, +List)` `List` est une liste de tuples d'entiers. `Tuples` est une liste de variables dont le nombre est égal au nombre d'éléments d'un tuple de `List`. Ces variables sont contraintes par les éléments correspondants des tuples de `List`.
  Exemple :
  PROGRAMME 5.1.1

```
tuples_in([[X,Y]], [[1,2],[1,5],[4,0],[4,3]]).
X in 1\/4,
tuples_in([[X, Y]], [[1, 2], [1, 5], [4, 0], [4, 3]]),
Y in 0\/2..3\/5.
```

La conception d'un programme avec contraintes sur des bornes finis se fait selon les quatre étapes suivantes :

(1) *Définition des variables*.- On précise les variables qui seront sous contraintes. Habituellement ces variables sont stockées dans une liste, par exemple `ListeVar`.

(2) *Définition des domaines*.- Il s'agit de définir les domaines de variation des variables déterminées précédemment.

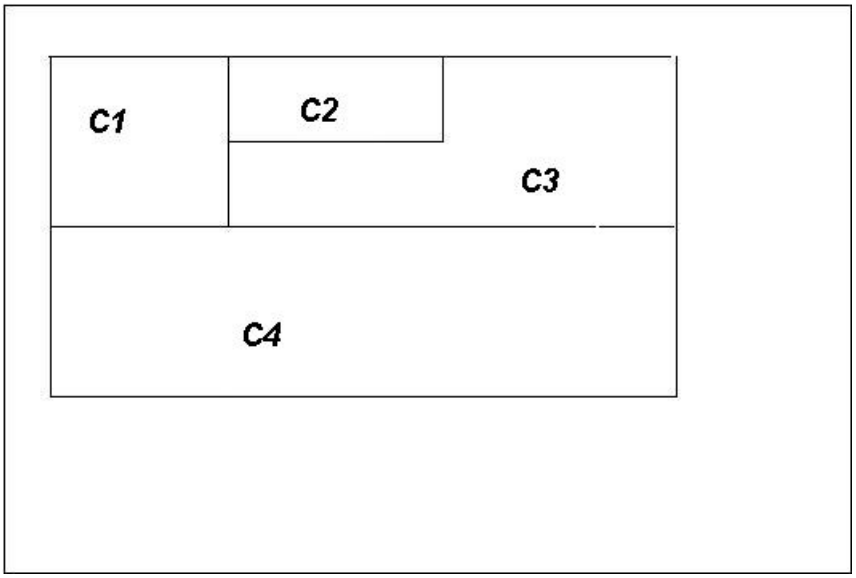
(3) *Spécification des contraintes*.- On spécifie les relations (contraintes) que les variables ont entre elles. Pour les spécifications on utilise le prédicat `all_different/1` avec comme argument une liste dont les éléments doivent être tous différents et les opérateurs
`#=, # \=, #<, #=<, #>, #>=`
qui ont la même signification que les opérateurs habituels. La spécification des contraintes sert à restreindre l'espace de variation des contraintes qui est l'espace de recherche de solutions.

(4) *Recherche de solutions*.- Prolog utilise la technique de la force brute, à savoir :

- Choix d'un variable.
- Choix, pour cette variable, d'une valeur issue de son domaine de variation.
- Réduction des domaines de variation des autres variables par propagation des contraintes.
  S'il n'y a pas de solution, retour en arrière et sélection d'une autre valeur pour la variable.
  Sinon, choisir une autre variable et recommencer.
- Si on a assigné des valeurs à toutes les variables, retourner ces valeurs (la solution).
- Recommencer l'algorithme jusqu'à épuisement des toutes les affectations possibles des variables.

L'exemple suivant fournit un modèle pour l'utilisation de cette librairie dans le cas d'un problème combinatoire.

Soit la carte suivante :



Nous voulons colorier cette carte en utilisant trois couleurs – bleue, jaune et rouge – de sorte que deux régions adjacentes n'aient pas la même couleur.

Le programme suivant fournit une réponse.

PROGRAMME 5.1.2

```
  %  Chargement de la librairie clp/bounds
  :- use_module(library('clp/bounds')).

  %  Association num\'ero - couleur
  couleur(1, bleu).
  couleur(2, jaune).
  couleur(3, rouge).

  colorier :-
          ListeVar = [C1, C2, C3, C4],   % Definition des variables
          ListeVar in 1..3,              % Definition des domaines
          label(ListeVar),               % Lancement de l algorithme
          all_different([C1,C2]),        % Contraintes sur les variables
          all_different([C1, C3]),       % Il faut que toutes soient differentes
          all_different([C1, C4]),
          all_different([C2, C3]),
          all_different([C3, C4]),
          affiche(ListeVar).             % Affichage du r\'esultat.

  affiche([]):- nl.
  affiche([H|T]) :- couleur(H,C), write(C), tab(2), affiche(T).
```

En lançant le programme, on obtient les réponses

```
bleu  jaune  rouge  jaune
bleu  rouge  jaune  rouge
jaune  bleu  rouge  bleu;
jaune  rouge  bleu  rouge
rouge  bleu  jaune  bleu;
```

```
rouge  jaune  bleu  jaune
```

Une façon de programmer avec moins de lignes est de grouper les contraintes selon la notion de voisinage. On a ainsi le programme ci-après :

PROGRAMME 5.1.3

```
  %  Chargement de la librairie clp/bounds
  :- use_module(library('clp/bounds')).

  %  Association num\'ero - couleur
  couleur(1, bleu).
  couleur(2, jaune).
  couleur(3, rouge).

  colorier :-
        ListeVar = [C1, C2, C3, C4],  % Definition des variables
        ListeVar in 1..3,             % Definition des domaines
        label(ListeVar),              % Lancement de l algorithme, c-a-d recherche
                                      % d une affectation de valeurs a la liste
                                      % des variables ListeVar, qui satisfait
                                      % aux contraintes
        all_different([C1, C2, C3]),  % Contraintes sur les variables
        all_different([C1, C3, C4]),  % Il faut que toutes soient differentes
        affiche(ListeVar).            % Affichage du r\'esultat.

  affiche([]):- nl.
  affiche([H|T]) :- couleur(H,C),write(C),tab(2), affiche(T).
```

où `[C1, C2, C3]` et `[C1, C3, C4]` sont des listes qui regroupent des régions voisines.

Nous terminons ce paragraphe par l'examen d'un problème de planification d'un trajet par train. On suppose qu'on a une liste de quadruples qui indiquent, pour chaque train, les gares de départ et d'arrivée ainsi que les horaires de départ et d'arrivée. On représente les gares et les horaires par des nombres entiers. On a le programme suivant :

PROGRAMME 5.1.4

```
:- use_module(library(clpfd)).
trains([[1,2,0,1],[2,3,4,5],[2,3,0,1],[3,4,5,6],[3,4,2,3],[3,4,8,9]]).
itineraire(A, D, Ps) :- Ps = [[A,B,_T0,T1],[B,C,T2,T3],[C,D,T4,_T5]],
                         T2 #> T1,
                         T4 #> T3,
                         trains(Ts),
                         tuples_in(Ps, Ts).
```

On cherche un itinéraire réalisable qui mène de la gare 1 à la gare 4. Nous avons

```
?- itineraire(1, 4, Ps).
Ps = [[1, 2, 0, 1], [2, 3, 4, 5], [3, 4, 8, 9]].
```

5.2 PLC sur les nombres rationnels et réels

L'appel au solveur de la librairie sur les nombres rationnels se fait par

```
:- use_module(library(clpq)).
```

et sur les nombres réels

```
:- use_module(library(clpr)).
```

Les principaux prédicats de cette librairie sont

- {⟨contraintes⟩} Exemple : `{ {A+B = 10, A = B}.`
- `entailed(+Contrainte)` Le prédicat se vérifie si la contrainte est vérifiée.
  Exemple : `{ A =< 4}, entailed(A= \=5).`
- `inf(+Expr, -Inf)` et `sup(+Expr, -Sup)` Unification de `Inf` (resp. `Sup`) avec l'infinimum (resp. supremum) de l'expression `Expr`.
  Exemple :

```
?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y}, sup(Z, Sup).

?- { 2*X+Y >= 16, X+2*Y >= 11, X+3*Y >= 15, Z = 30*X+50*Y}, inf(Z, Inf).
```

- `minimize(+Expr)` et `maximize(+Expr)` Calcul du minimum (resp. maximum) de l'expression `Expr` et unifie les variables aux valeurs correspondantes.
  Exemple :

```
?- { 2*X+Y =< 16, X+2*Y =< 11, X+3*Y =< 15, Z = 30*X+50*Y}, maximize(Z).
X = 6.6,
Y = 2.8,
Z = 338.0
?- { 2*X+Y >= 16, X+2*Y >= 11, X+3*Y >= 15, Z = 30*X+50*Y}, minimize(Z).
X = 7.0,
Y = 2.0,
Z = 310.0 ;
```

- `bb_inf(+Ints, +Expr, -Inf)` Unifie `Int` avec le minimum de l'expression `Expr` sous la condition que les variables de la liste `Ints` soient entières.
- `bb_inf(+Ints, +Expr, -Inf, -Vertex, +Eps)` Même chose que précédemment mais en donnant une tolérance de `Eps` aux valeurs entières. Ainsi `X` est considéré comme entier si `abs(round(X)-X) < Eps`. Vertex contient la liste des valeurs de la solution pour les variables entières.

En fonction du problème considéré nous pouvons soit, dans le cas où le problème n'a pas de contraintes, appliquer l'algorithme, soit, dans le cas contraire, suivre la démarche présentée dans la section précédente.

Ainsi pour calculer la racine carrée d'un nombre avec l'algorithme de Newton, on établit l'algorithme et on obtient le programme suivant :

PROGRAMME 5.2.1

```
:- use_module(library(clpr)).
% Calcul racine carr\'{e}e
racine(N, R) : - racine(N, 1, R).
racine(0, S, R) : - !, S=R.
racine(N, S, R) : - N1 is N-1, \{ S1 = S/2 + 1/S \}, racine(N1, S1, R).
```

qui à la question

```
racine(5,R).
```

donne la réponse

```
R = 1.41421
```

Pour la programmation linéaire en nombres entiers, nous avons le programme

PROGRAMME 5.2.2

```
% Probl\'{e}me d'optimisation lin\'{e}aire en nombres entiers.

:- use_module(library(clpr)).

exemple(pl, Obj, Vs, Ints, []) :-       % D\'{e}finition des variables
                                    Vs = [ X1,X2,X3,X4,X5,X6,
                                           Y1,Y2,Y3,Y4,Y5,Y6,
                                           Z1,Z2,Z3,Z4,Z5,Z6],

Ints = [Y6, Y5, Y4, Y3, Y2,             %D\'{e}termination des variables enti\'{e}res
        X6, X5, X4, X3, X2, X1],

Obj =   2700*Y1 + 1500*X1 + 30*Z1       % La fonction \'{a} minimiser
      + 2700*Y2 + 1500*X2 + 30*Z2
      + 2700*Y3 + 1500*X3 + 30*Z3
      + 2700*Y4 + 1500*X4 + 30*Z4
      + 2700*Y5 + 1500*X5 + 30*Z5
      + 2700*Y6 + 1500*X6 + 30*Z6,
      allpos(Vs),                       % Les variables doivent \^{e}tre positives
     { Y1 = 60, 0.9*Y1 +1*X1 -1*Y2 = 0, % Les contraintes lin\'{e}aires
       0.9*Y2 +1*X2 -1*Y3 = 0, 0.9*Y3 +1*X3 -1*Y4 = 0,
       0.9*Y4 +1*X4 -1*Y5 = 0, 0.9*Y5 +1*X5 -1*Y6 = 0,
       150*Y1 -100*X1 +1*Z1 >= 8000,
       150*Y2 -100*X2 +1*Z2 >= 9000,
       150*Y3 -100*X3 +1*Z3 >= 8000,
       150*Y4 -100*X4 +1*Z4 >= 10000,
       150*Y5 -100*X5 +1*Z5 >= 9000,
       150*Y6 -100*X6 +1*Z6 >= 12000,
       -20*Y1 +1*Z1 =< 0, -20*Y2 +1*Z2 =< 0,
       -20*Y3 +1*Z3 =< 0,
```

```
        -20*Y4 +1*Z4 =< 0, -20*Y5 +1*Z5 =< 0,
        -20*Y6 +1*Z6 =< 0,
        X1 =< 18, 57 =< Y2, Y2 =< 75, X2 =< 18,
        57 =< Y3, Y3 =< 75, X3 =< 18, 57 =< Y4,
        Y4 =< 75, X4 =< 18, 57 =< Y5, Y5 =< 75,
        X5 =< 18, 57 =< Y6, Y6 =< 75, X6 =< 18}.

allpos([]).

allpos([X | Xs]) :- {X >= 0}, allpos(Xs).

% Minimum relax\'{e} (sans la contrainte de solution enti\'{e}re

optR(Inf) :- exemple(pl, Obj, _, _, _), inf(Obj, Inf).

% Minimum avec variables enti\'{e}res

opt(Sommets, Inf) :- exemple(pl, Obj, _, Ints, _),
                     bb_inf(Ints, Obj, Inf,Sommets, 0.001).
```

avec réponses

```
?- optR(Inf).
Inf = 1.16719e+006

?- opt(S,Inf).
S = [75, 70, 70, 60, 60, 0, 12, 7, 16|...],
Inf = 1.2015e+006
```

5.3 CHR : Constraint Handling Rules

SWI-Prolog à l'aide de CHR fournit la possibilité d'écrire ses propres solveurs pour des contraintes spécifiques. Il peut être utilisé à plusieurs types d'applications, comme planification, distribution, test de type, etc.

Pour utiliser la technique du CHR dans un programme Prolog, il faut au préalable

- inclure les contraintes et les règles dans un module à l'aide de la requête

  `:- module(nomModule, [nomContrainte1/arité, ...,nomContrainteN/arité]).`
- faire appel au solveur CHR avec la requête

  `:- use_module(library(chr)).`
- et enfin il faut déclarer chaque contrainte avec la requête

  `:- chr_constraint(nomContrainte1/arit\U{e9}, ...,nomContrainteN/arit\U{e9}).`

Ensuite il faut écrire les containtes, sous la forme

```
etiquette @ contrainte.
```

L'exemple suivant reprend le programme pour l'addition de deux naturels dans le cadre de satisfaction des contraintes.

PROGRAMME 5.3.1

```
:- module(addition,[add/3]).
:- use_module(library(chr)).
:- chr_constraint add/3.

zero1 @ add(0,Y,Z) <=> Y = Z.
zero2 @ add(X,0,Z) <=> X = Z.
zero3 @ add(X,Y,0) <=> X = 0, Y = 0.

meme1 @ add(X,E,E) <=> X = 0.
meme2 @ add(E,Y,E) <=> Y = 0.

succ1 @ add(s(X),Y,Z) <=> Z = s(W), add(X,Y,W).
succ2 @ add(X,s(Y),Z) <=> Z = s(W), add(X,Y,W).
succ3 @ add(X,X,s(Z)) <=> Z = s(W), X = s(Y), add(Y,Y,W).

cherche @ add(X,Y,s(Z)) <=> true | add(X1,Y1,Z),
                              (X = s(X1),Y = Y1 ; X = X1,Y = s(Y1)).
```

Utilisation

```
?- add(s(s(0)),s(s(s(0))),s(s(s(s(s(0))))).

yes.

?- add(s(s(0)), s(0), Z).

Z = s(s(s(0))).
```

5.4 Références

Pour la rédaction de ce chapitre, les références suivantes ont été utilisées :

[1] K. R. APT, P. ZOETEWEIJ : An Analysis of Arithmetic Constraints on Integer Intervals, Constraints, v. 12, nu. 4, 2007, pp. 429-468

[2] T. FRÜHWIRTH, S. ABDENNADHER : *Essentials of Constraint Programming*, Springer, 2003

[3] J. Jaffar, M. J. Maher : Constraint logic programming : a survey, Journal of Logic Programming 19 & 20, 503-581, 1994

[4] J. Jaffar et al. : The semantics of constraint logic programs, Journal of Logic Programming, 37, 1-46, 1998.

[5] M. WALLACE : Survey : Practical Applications of Constraint Programming, Imperial College, September 1995

5.5 Exercices

EXERCICE 5.1 *Soient les inégalités*

$$\begin{aligned} x-y &\leq 3 \\ x+y &\leq 1 \\ -x+z &\leq -3 \\ -x-z &\leq -1 \\ -20 \leq x &\leq 20 \\ -20 \leq y &\leq 20 \\ -20 \leq z &\leq 20 \end{aligned}$$

Calculer les valeurs maximales de x, y et z en utilisant la librairie `clpr`*, avec*

- `sup`
- `maximize`

Faire le même programme en utilisant la librairie `clpfd`*. Que constatez-vous*

EXERCICE 5.2 *En utilisant la librairie* `chr`*, écrire un solveur qui permet le calcul de nombres premiers de 1 jusqu'à N.*

EXERCICE 5.3 *Pour un casse, Toto doit ouvrir un coffre. Les informations dont il dispose sont les suivantes :*

- *Le code est composé de 9 chiffres au total, de 1 à 9.*
- *Chaque chiffre apparaît une seule fois dans le code.*
- *Le 1er chiffre est différent de 1, le 2ème différent de 2 et ainsi de suite.*
- *La différence entre le 4ème et le 6ème chiffres*[1] *est égale au 7ème chiffre.*
- *Le produit de trois premiers chiffres est égal à la somme de deux derniers.*
- *La somme du 2ème, 3ème et 6ème chiffre est inférieure au 8ème chiffre.*
- *Le dernier chiffre est inférieur au 8ème chiffre.*

Pour aider Toto il faut choisir la librairie à utiliser, modéliser le problème et calculer la solution.

EXERCICE 5.4 *En utilisant le programme* `add/3` *du listing 5.3.1 écrire le programme* `mult(X,Y,R)` *qui calcule le résultat symbolique* `R` *de la multiplication de deux naturels* `X` *et* `Y`*.*

EXERCICE 5.5 *Une usine fabrique deux produits P_1 et P_2.*
Pour fabriquer une unité du produit P_1 il faut une unité de chacune de deux matières premières m_1 et m_2.
Pour fabriquer une unité du produit P_2, il faut deux unités de la matière première m_1 et une unité de chacune de deux matières premières m_2 et m_3.
Les stocks des matières premières sont 1500 unités pour m_1, 1200 unités pour m_2 et 500 unités pour m_3.
La vente d'une unité de P_1 procure un gain de 15 euros et de P_2 10 euros.

1. Attention : il n'est pas écrit la différence du 4ème chiffre moins le 6ème

En utilisant la PLC calculer les quantités à produire pour chaque produit, de sorte que le gain soit maximal.

6

LES BASES DE DONNÉES EN PROLOG

6.1 Définir les relations 71
6.2 Définir les requêtes 72
6.2.1 Les requêtes recursives 73
6.3 La programmation logique et le modèle relationnel des bases de données 74
6.3.1 L'opération d'union 74
6.3.2 L'opération de différence 74
6.3.3 L'opération du produit cartésien 74
6.3.4 L'opération de projection 75
6.3.5 L'opération de sélection 75
6.4 Gestion de la base de données 76
6.5 Introduction aux bases de données déductives . . 78
6.6 L'itérativité en Prolog 80
6.7 Utilisation de l'itérativité aux bases de données . . 81

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.

6.1 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).
```

6.2 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 p\'ere 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 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*.

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

```
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 \'equivalente␣\'a␣la␣requ\^ete␣compos\'ee:
                 trouver le parent de koko X et X est homme.
?mere(X,koko).   qui est \'equivalente␣\'a␣la␣requ\^ete␣compos\'ee:
                 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.

6.2.1 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).
```

6.3 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 :

6.3.1 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.

6.3.2 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 6.1 *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).`

6.3.3 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 faon suivante :

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

ASCÈSE 6.2 *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).`

6.3.4 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 6.3 *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 ?*

6.3.5 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 $r2$ permet de sélectionner l'ensemble des valeurs vérifiant la relation r à condition que $X1$ soit unifiée avec *const1* ou avec *const2*.

ASCÈSE 6.4 *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 6.3.1 *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 6.5 *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.*

6.4 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 6.4.1 *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 6.6 *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.*

6.5 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 6.7 *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 6.8 *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 6.9 *Soit la base de données relationnelle contenant les trois tables suivantes :*

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

TABLE 6.1 – Table des produits

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.

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

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 6.2 – 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 6.3 – Table des achats - fournisseurs

(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.*

6.6 L'itérativité en Prolog

La programmation en `Prolog` nécessite l'utilisation de la récursivité. Mais il y a des situations où la récursivité est imposée de facomplètement artificielle et n'porte strictement rien à la cohérence du programme, comme par exemple le parcours des enregistrements d'un fichier séquentiel. Ce paragraphe est consacré à la programmation en `Prolog` de la structure itérative `tant que`.

Il s'agit de créer un opérateur de la forme suivante :

```
tantQue X alorsFaire Y
```

dont la programmation en `Prolog` est donnée par

PROGRAMME 6.6.1

```
    :- op(100, fy, tantQue).
    :- op(900, xfx, alorsFaire).

tantQue X alorsFaire Y :- (X,Y,fail); true.
```

On peut associer cet opérateur avec un autre qui teste si un indice numérique est entre une valeur minimale et une valeur maximale :

PROGRAMME 6.6.2

```
:- op(100, xfx, estEntre).
:- op(900, xfy, et).

Indice estEntre Lmin et Lmax :- entre(Indice,Lmin,Lmax).

entre(Indice,Indice,Lmax) :- Lmax >= Indice.

entre(Indice,Lmin,Lmax) :- Lmin < Lmax, L is Lmin+1,
                           entre(Indice,L,Lmax).
```

On constate ici que la définition de l'opérateur `estEntre` nécessite un appel récursif à lui-même qui, s'agissant d'un opérateur, est impossible. D'où le recours au prédicat `entre` qui lui, en tant que prédicat, peut faire un appel récursif à lui-même.

En utilisant ce deux opérateurs on peut par exemple afficher la table de multiplication entre deux valeurs numériques `L1` et `L2` :

PROGRAMME 6.6.3

```
tableMultiplication(L1,L2 :- tantQue (I estEntre L1 et L2) alorsFaire
                             (tantQue (J estEntre L1 et L2) alorsFaire
                             (K is I*J, write(K), write(', '),write(nl))).
```

6.7 Utilisation de l'itérativité aux bases de données

Nous reprenons les bases de données `voiture` et `personne` de l'exemple 4.3.1. Nous allons aussi utiliser la base de données virtuelles `bddVirt` de l'exemple 4.4.1.

Nous pouvons envisager d'afficher un élément particulier d'une base de données ou, encore, toute la base de données. On introduit les deux opérateurs :

PROGRAMME 6.7.1

```
:- op(1100, fx, afficheListe).
:- op(1100, fx, afficheTout).

afficheListe [X,Y] :- nl,
                      for(Y, (write(X), nl)),
                      fail.

afficheTout X :- afficheListe [X, X].
for(X,Y) :-  X, Y, fail.
for(X,Y).
```

Nous pouvons en utilisant le premier opérateur de répondre à des questions du type « donner les marques de voitures dont les propriétaires sont des femmes et aussi les noms de ces femmes » comme suit :

PROGRAMME 6.7.2

```
question :- afficheListe [(Marque, Prop),
                  (personne(Prop, f, _,  _, _),
                   voiture(_, Marque, Prop, _))].
```

Nous pouvons aussi afficher toute une base de données en utilisant le deuxième opérateur :

PROGRAMME 6.7.3

```
question :- afficheTout voiture(X,Y,Z,W).
```

Pour améliorer l'interactivité lors de l'interrogation de bases de données, nous pouvons envisager d'introduire un nouvel opérateur qui permet de s'approcher du langage naturel lorsque nous posons une question.

PROGRAMME 6.7.4

```
:- op(900, xfx, avecCle).
:- op(100, xfx, 'aComme').

BdD avecCle Cle aComme (Attribut = Valeur) :-
                        bddVirt(BdD, Cle, Attribut, Valeur),
                       not(Valeur = nil).
```

qui permet de poser la question précédente comme suit :

PROGRAMME 6.7.5

```
question :- afficheListe [(Ma, Pr), (voiture avecCle N aComme (marque = Ma),
                                     voiture avecCle N aComme (proprietaire = Pr),
                                     personne avecCle Pr aComme (sexe = f))].
```

7

TROIS APPLICATIONS

7.1 Algorithme de routage 83
7.2 Planification 87
7.3 Programmation pour Sudoku 89

LE dernier chapitre de ce support de cours est consacré à la présentation de trois applications issues des domaines de télécommunications, de la robotique et du génie logiciel. L'objectif est de montrer de façon concrète la souplesse et l'adaptivité de `Prolog`.

7.1 Algorithme de routage

Un problème des télécomunications qui concerne la couche réseau et qui est relatif à l'envoi des paquets de la source à tous les déstinataires, est le routage de ces paquets à travers le réseau([1]). Pour ce faire on considère le réseau comme un graphe dans lequel les sommets représentent des nœuds des « packets switch » ou IMP (Interface Message Protocol) et les arcs une ligne de communications. Un des algorithmes utilisés est celui du plus court chemin dû à Dijkstra. Pour faire fonctionner cet algorithme il faut à chaque arc qui relie deux nœuds associer un coût. Ce coût est calculé en fonction de la distance, de la bande passante, de la moyenne du trafic, du coût de communication, de la moyenne du délai d'attente, etc.

On suppose que la base de données est constituée par le graphe du réseau `gr(sommet, prédecesseur, poids(sommet, prédecesseur))`. L'algorithme que nous

1. Les idées de base pour ce paragraphe nous ont été fournies par notre collègue Bernard Glonneau. La référence est le livre de A.S.Tanenbaum : Computer Networks,2nd edition, Prentice Hall, 1988, pp.289-292.

allons utiliser est une adaptation de l'algorithme connu en Intelligence Artificielle sous le nom « le meilleur d'abord ». On utilise les ensembles OUVERT et FERME dont les éléments sont les triplets (x,y,d(x)) où x est un sommet, y est le sommet prédecesseur de x et d(x) est le coût pour arriver de la racine à x. De plus l'ensemble OUVERT est constamment trié selon les valeurs ascendantes de la distance d(x).

L'algorithme, adapté pour le Prolog, de Dijkstra est le suivant :

```
Initialisation :
OUVERT ⟵  [x1,x1,0], où x1 le sommet initial.
FERME ⟵  []
tableRoutage ⟵  []

Tant que ( OUVERT ≠  [] ) faire
Début
    Si  OUVERT = [(x,y,d(x) | Reste], alors
    Début
        OUVERT ⟵  Reste,
        FERME ⟵  [(x,y,d(x) | FERME]
        tableRoutage ⟵  [(x,y,d(x) |  tableRoutage]
    Fin

    Si ( EXPANSION(x) ≠  [] ), alors
    Début
        Pour tout w ∈  EXPANSION(x) faire
        Début
            Si ( (w,_,_) ∉  OUVERT U FERME ), alors
            Début
                d(w) ⟵  d(x) + p(x,w)
                OUVERT ⟵  [(w,x,d(w)) | OUVERT]
            Fin
            Si ( (w,_,d(w)) ∈  OUVERT ), alors
            Début
                Si ( d(w) > d(x) + p(x,w) ), alors
                Début
                    OUVERT ⟵  OUVERT -{(w,_,d(w))}
                    d(w) ⟵  d(x) + p(x,w)
                    OUVERT  ⟵  [(w,x,d(w)) | OUVERT]
                Fin
            Fin
            Si ( (w,_,d(w)) ∈  FERME ), alors
            Début
                Si ( d(w) > d(x) + p(x,w) ), alors
                Début
                    FERME ⟵  FERME -{(w,_,d(w))}
                    d(w) ⟵ d(x) + p(x,w)
                    OUVERT  ⟵  [(w,x,d(w)) | OUVERT]
```

```
                Fin
            Fin

        Fin
    Fin

    Trier OUVERT selon les valeurs ascendantes de d(x).

Fin

Utiliser la liste tableRoutage pour reconstruire le chemin optimal.
```

Le programme complet est le suivant :

PROGRAMME 7.1.1

```
/* Base de données */

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).
sommetInit(a).

gr(X,Y,D) :- not sommetInit(X), gr(Y,X,D).


/* Programmes utilitaires */

membre((X,Y,Z),[(X,Y,Z) | Ouvert]).
membre((X,Y,Z),[(X1,Y1,Z1) | Ouvert]):- membre((X,Y,Z),Ouvert).

suppres((X,Y,Z),[(X,Y1,Z1) | R],R).
suppres((X,Y,Z),[(F,G,W) | R],[(F,G,W) | R1]) :-
suppres((X,Y,Z),R,R1).

conc([],L2,L2).
conc([T | R],L2,[T | Rs]) :- conc(R,L2,Rs).

for(X,Y) :- X, Y, fail.
for(X,Y).

/* Affichage du routage */

decort2([]).
decort2([(X,Y) | Ouvert]) :- write(X), tab(2), write(Y),nl,
decort2(Ouvert).

decort3([]).
decort3([(X,Y,D) | Ouvert]) :- write(X), tab(2), write(Y), tab(2),
```

```
                                        write(D),nl, decort3(Ouvert).

    /* Inverse d'une liste */

    reverse([T | R],Acc,Res) :- reverse(R,[T | Acc],Res).
    reverse([],Res,Res).

    /* Remplacement d'un élément d'une liste par un autre élément */

    remplace((X,Y),[(X,Y) | R],(Z,W),[(Z,W)\(\mid\)R]).
    remplace((X,Y),[(F,G) | R],(Z,W),[(F,G) | R1]) :- remplace((X,Y),R,(Z,W),R1).

    /* Remplacement d'une liste par une autre liste */

    transfert([],[]).
    transfert([X | R],[X | R1]) :- transfert(R,R1).

    /*  Tri d'une liste  */

    tri([(X,Y,Val) | R],L) :- partition(R,(X,Y,Val),P,G),
                              tri(P,Ps), tri(G,Gs),
                              conc(Ps,[(X,Y,Val) | Gs],L).
    tri([],[]).

    partition([(Z,W,V) | Xs],(X,Y,Val),[(Z,W,V) | Ps],G) :-
                              V < Val,
                              partition(Xs,(X,Y,Val),Ps,G).
    partition([(Z,W,V) | Xs],(X,Y,Val),P,[(Z,W,V) | Gs]) :-
                              V >= Val,
                              partition(Xs,(X,Y,Val),P,Gs).
    partition([],(X,Y,Val),[],[]).


    /* MAJ de l' OUVERT */

    majOuvert([],S,Dist,Ferme,Ferme,Ouvert,Ouvert).
    majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme1,Ouvert,[(Ss,S,Dist1) | Ouvert1]) :-
                              not membre((Ss,Pred,Val),Ouvert),
                              not membre((Ss,Pred,Val),Ferme),
                              Dist1 is Dist+Pds,
                              majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).
    majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme1,Ouvert,[(Ss,S,Dist1) | Ouvert2]) :-
                              membre((Ss,Pred,Val),Ouvert),
                              Dist1 is Dist+Pds,
                              Dist1 < Val,
                              suppres((Ss,Pred,Val),Ouvert,Ouvert1),
                              majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert1,Ouvert2).
    majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme1,Ouvert,Ouvert1) :-
                              membre((Ss,Pred,Val),Ouvert),
                              Dist1 is Dist+Pds,
                              Dist1 $\geq$ Val,
                              majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).
    majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme2,Ouvert,[(Ss,S,Dist1) | Ouvert1]) :-
                              membre((Ss,Pred,Val),Ferme),
                              Dist1 is Dist+Pds,
                              Dist1 < Val,
                              suppres((Ss,Pred,Val),Ferme,Ferme1),
```

```
                        majOuvert(Expansion,S,Dist,Ferme1,Ferme2,Ouvert,Ouvert1).
majOuvert([(Ss,Pds) | Expansion],S,Dist,Ferme,Ferme1,Ouvert,Ouvert1) :-
                        membre((Ss,Pred,Val),Ferme),
                        Dist1 is Dist+Pds,
                        Dist1 $\geq$ Val,
                        majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1).

/* Calcul du routage pour un réseau donné.  Algorithme de Dijsktra */

routage([],Ferme,[]) :- write('Fin algorithme - Routage '), nl.
routage([(S,Pr,Dist)$\mid$Ouvert],Ferme,[(S,Pr,Dist)$\mid$ Lr]) :-
                        trouveTout(S,Expansion),
                        majOuvert(Expansion,S,Dist,Ferme,Ferme1,Ouvert,Ouvert1),
                        tri(Ouvert1,Ouvert2),
                        routage(Ouvert2,[(S,Pr,Dist) | Ferme1],Lr).

trouveTout(X,R) :- asserta(grlst([])),
                   for(gr(X,Y,D),(write(Y), tab(2),
                   write(D),nl,
                   grlst(L), retract(grlst(V)),
                   asserta(grlst([(Y,D) | L])))),
                   grlst(R1),retract(grlst(V)),reverse(R1,R).
```

Exemple de question à poser pour ce programme :

```
? question :- sommetInit(S), routage([(S,S,0)],[],L), decort3(L).
```

7.2 Planification

Un problème de la robotique est la construction d'une suite d'actions – planification – qui permet au robot d'agir sur l'environnement. Bien sûr cette suite d'actions doit être cohérente et minimale, en ce sens que les actions ne remettent pas en cause les résultats des actions antérieures. Afin de réaliser la planification on doit pouvoir décrire le monde et son évolution en utilisant un ensemble des concepts. Essentiellement nous avons besoin de décrire les objets du monde et les actions qui permettent son évolution. À chaque instant le monde se trouve à un état donné. On peut passer d'un état à un autre par l'intermédiaire d'une action. En appliquant donc une suite d'actions on peut passer d'un état initial à un état final (but). La construction de la suite d'actions se fait à l'aide d'un générateur de plans d'actions.

Il existe des générateurs universels de plans d'actions, c'est-à-dire des générateurs qui peuvent s'appliquer à différents types de problèmes, comme par exemple STRIPS ou WARPLAN qui d'ailleurs a été écrit en `Prolog`. Le générateur que nous présentons ici est un simple générateur qui est universel. Son principal défaut est qu'il ne protège pas – contrairement à WARPLAN – la destruction d'un but déjà réalisé par la réalisation d'un autre but.

À titre d'exemple nous présentons ce générateur dans le cas du monde des blocs. Ce monde contient des blocs (cubes) qu'il s'agit de déplacer. La situation

d'un cube se décrit à l'aide de deux prédicats : `libre(X)` qui indique si le bloc `X` est libre et `sur(X,Y)` qui indique si le bloc `X` se trouve sur le bloc `Y` et où `Y` peut aussi être la `table`. L'évolution du monde se fait à l'aide d'une seule action qui est le déplacement d'un cube A du cube B sur lequel se trouve au cube C : `deplacer(A,B,C)`. Les contraintes pour la réalisation de cette action sont `libre(A)`, `libre(C)`, `different(B,C)`. L'évolution du monde due à cette action se traduit par le retrait des certains faits et par l'ajout d'autres faits. Ainsi les faits qu'on supprime sont `libre(C)` et `sur(A,B)`. Les faits qu'on rajoute sont `libre(B)` et `sur(A,C)`.

Le programme `Prolog` de ce générateur est le suivant :

PROGRAMME 7.2.1

```
    /* Le générateur de plans */

   genplan(G,S) :- planif(G,S,S1), ecrire(S1).

    /*  Le programme qui effectue la planification  */

    planif([B | R],S,S2) :-  realiseFait(B,S,S1), planif(R,S1,S2).
    planif([],S,S).

    /*  Le programme qui réalise un fait   */

    realiseFait(B,S,S) :- B.
    realiseFait(B,S,S) :- sitSuc(B,S).
    realiseFait(B,S,[Act | S1]) :-ajout(B,Act),action(Act,Cond),
                                    planif(Cond,S,S1).

    /*  Teste si un fait peut être obtenu par un état qui
        succède imm\édiatement à l'état␣présent.␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*/

␣␣␣␣sitSuc(F,[Act␣|␣S])␣:-␣ajout(F,Act).
␣␣␣␣sitSuc(F,[Act␣|␣S])␣:-␣sitSuc(F,S),␣non(supp(F,Act)).
␣␣␣␣sitSuc(F,[init])␣:-␣F.

␣␣␣␣/*␣␣Définition␣de␣l'action et de ses contraintes pour le déclechement  */

    action(deplace(A,B,table),[sur(A,B),libre(A),ne(B,table)]).
    action(deplace(A,B,C),[libre(C),sur(A,B),libre(A),ne(A,C)]).

    /*  Les effets d'une␣action␣␣␣␣*/

␣␣␣␣ajout(sur(A,C),deplace(A,B,C)).
␣␣␣␣ajout(libre(B),deplace(A,B,C)).
␣␣␣␣supp(sur(A,B),deplace(A,B,C)).
␣␣␣␣supp(libre(C),deplace(A,B,C)).

␣␣␣␣/*␣␣␣Programmes␣utilitaires␣␣␣␣*/

␣␣␣␣ecrire([S1␣|␣S2])␣:-␣ecrire(S2),␣nl,␣write(S1),␣nl.
␣␣␣␣ecrire([]).

␣␣␣␣non(X)␣:-␣X,␣!,␣fail.
␣␣␣␣non(X).
```

```
    /* Base de données    */

    sur(c,table).
    sur(d,table).
    sur(a,c).
    sur(e,d).
    sur(b,e).
    libre(a).
    libre(b).
    libre(table).
```

Une question à poser est par exemple la suivante

```
?-question :- genplan([sur(a,table),sur(c,a),libre(c),
                       sur(b,table),sur(e,b),sur(d,e),libre(d)],[init]).
```

7.3 Programmation pour Sudoku

La programmation en Prolog de la résolution d'une grille de Sudoku est extremement facile à mettre en œuvre ; Nous allons principalement s'appuyer sur les contraintes que nous avons vu au chapitre 4.

Pour écrire le programme il faut partir des règles du jeu qui peuvent se resumer comme suit :

- Le sudoku est un plateau de 81 cases disposées sur un carré de 9 cases de côté.
- Le plateau est divisé en 9 sous-plateaux, appelés voisinages, de 9 cases chacun, disposés en des carrés de 3 cases de côté.
- Chaque case peut recevoir un nombre entier entre 1 et 9.
- Toutes les lignes du plateau doivent avoir des nombres différents.
- Toutes lescolonnes du plateau doivent avoir des nombres différents.
- Tous les voisinages du plateau doivent avoir des nombres différents.

On reprend ces règles et on écrit le programme.

Le programme est le suivant :

PROGRAMME 7.3.1

```
:- use_module(library('clp/bounds')).

/*
   test lit le fichier grille.pl dans lequel est stockée la grille du jeu
   et lance le programme de résolution.
*/
sudoku :- consult('c:/tmp/grille.pl'), go.

sudoku(Grille) :-
%  Affichage de la grille d'entrée
        nl, writeln('Grille avant resolution :'), nl,
        affiche1(Grille), nl, writeln('Solution'), nl,
```

```
%␣␣Suppression␣des␣sous-listes␣de␣la␣grille␣et␣stockage␣de␣grille␣sans␣sous-liste␣dans␣Gril
␣␣␣␣␣␣␣␣␣flatten(Grille,GrilleF),
%␣␣Tous␣les␣éléments␣de␣GrilleF␣sont␣des␣nombres␣entiers␣dans␣l'intervalle [1, 9]
        GrilleF in 1..9,
%  Calul de la transposée de la grille et stockage dans TGrille
        transpMat(Grille,TGrille),
%  Définition des lignes de Grille
        [Ligne1,Ligne2,Ligne3,Ligne4,Ligne5,Ligne6,Ligne7,Ligne8,Ligne9] = Grille,
%  Définition des colonnes de Grille (= lignes de TGrille)
        [Colonne1,Colonne2,Colonne3,Colonne4,Colonne5,Colonne6,Colonne7,Colonne8,Colonne9]
%  Définition des noms des cases de chaque ligne
        [X11,X12,X13,X14,X15,X16,X17,X18,X19] = Ligne1,
        [X21,X22,X23,X24,X25,X26,X27,X28,X29] = Ligne2,
        [X31,X32,X33,X34,X35,X36,X37,X38,X39] = Ligne3,
        [X41,X42,X43,X44,X45,X46,X47,X48,X49] = Ligne4,
        [X51,X52,X53,X54,X55,X56,X57,X58,X59] = Ligne5,
        [X61,X62,X63,X64,X65,X66,X67,X68,X69] = Ligne6,
        [X71,X72,X73,X74,X75,X76,X77,X78,X79] = Ligne7,
        [X81,X82,X83,X84,X85,X86,X87,X88,X89] = Ligne8,
        [X91,X92,X93,X94,X95,X96,X97,X98,X99] = Ligne9,
%  Définition des voisinages qui sont des sous-grilles de dimension (3x3)
        [X11,X12,X13,X21,X22,X23,X31,X32,X33] = Vois11,
        [X14,X15,X16,X24,X25,X26,X34,X35,X36] = Vois12,
        [X17,X18,X19,X27,X28,X29,X37,X38,X39] = Vois13,
        [X41,X42,X43,X51,X52,X53,X61,X62,X63] = Vois21,
        [X44,X45,X46,X54,X55,X56,X64,X65,X66] = Vois22,
        [X47,X48,X49,X57,X58,X59,X67,X68,X69] = Vois23,
        [X71,X72,X73,X81,X82,X83,X91,X92,X93] = Vois31,
        [X47,X48,X49,X57,X58,X59,X67,X68,X69] = Vois32,
        [X77,X78,X79,X87,X88,X89,X97,X98,X99] = Vois33,
%  Les éléments des lignes, colonnes et voisinages ont des valeurs différentes
        all_different(Ligne1), all_different(Ligne2), all_different(Ligne3),
        all_different(Ligne4), all_different(Ligne5), all_different(Ligne6),
        all_different(Ligne7), all_different(Ligne8), all_different(Ligne9),
        all_different(Colonne1), all_different(Colonne2), all_different(Colonne3),
        all_different(Colonne4), all_different(Colonne5), all_different(Colonne6),
        all_different(Colonne7), all_different(Colonne8), all_different(Colonne9),
        all_different(Vois11), all_different(Vois12), all_different(Vois13),
        all_different(Vois21), all_different(Vois22), all_different(Vois23),
        all_different(Vois31), all_different(Vois32), all_different(Vois33),
%  Chaque variable de GrilleF esta ssignée dans son domaine de variation
        label(GrilleF),
%  Afficage du résultat
        affiche(Grille).

affiche([]).
affiche([H|T]) :- write(H),nl, affiche(T).

affiche1([]).
affiche1([H|T]) :- write('['),afficheVect(H), affiche1(T).

afficheVect([]) :- nl.
afficheVect([T|R]) :- ((number(T), write(T));(not(number(T))), write('_')),
                      ((R\==[],write(',␣'));(R==[],write(']'))),
                      afficheVect(R).

premiereCol([],[]).
```

```
premiereCol([[T|R]|Rs],[T|Ls]) :- premiereCol(Rs,Ls).

suiteCol([],[]).
suiteCol([[T|R]|Rs],[R|Ls]) :- suiteCol(Rs,Ls).

transpMat([[]|_],[]).
transpMat(R,[T|Rs]) :- premiereCol(R,T),suiteCol(R,H), transpMat(H,Rs).
```

Le programme grille.pl qui sert à écrire la grille à résoudre est le suivant :

PROGRAMME 7.3.2

```
go :-
  Grille = [
 [3,_,_,_,6,8,_,_,_],
 [_,7,_,_,_,_,_,_,_],
 [_,1,6,_,_,_,_,2,9],
 [_,_,_,_,_,_,9,_,8],
 [7,_,_,6,_,_,_,_,_],
 [1,8,_,_,_,5,_,_,4],
 [6,3,_,4,_,_,_,_,_],
 [_,_,_,_,3,_,2,_,_],
 [_,_,4,_,5,_,1,_,_]
 ],
 sudoku(Grille).

 /*  Pour écrire une autre grille
    utilisez la grille vide suivante :

[
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_],
 [_,_,_,_,_,_,_,_,_]
 ]
 */
```

Pour appeler ce programme on entre la question `?- sudoku`.

Table des matières

1 LOGIQUE DES PROPOSITIONS ET PROGRAMMATION **7**

1.1 Syntaxe de `Prolog` . . . 9

1.1.1 Les faits . . . 9

1.1.2 Les règles . . . 10

1.1.3 Les questions . . . 10

1.1.4 Présentation en `Prolog` . . . 11

1.2 Les données . . . 11

1.2.1 Les données simples . . . 11

1.2.2 Les données structurées . . . 12

1.2.3 Types des données . . . 13

1.3 Fonctionnement (simplifié) de `Prolog` . . . 13

1.4 Repésentation des nombres naturels . . . 14

1.5 Les opérateurs de base de `Prolog` . . . 16

1.6 Un exemple . . . 17

1.7 Exercices . . . 18

2 LOGIQUE DES PRÉDICATS ET PROGRAMMATION **21**

2.1 Les prédicats de base de `Prolog` . . . 22

2.1.1 Entrées-sorties . . . 22

2.1.2 Opérations en `Prolog` . . . 22

2.1.3 Prédicats extralogiques . . . 23

2.2 Un exemple . . . 23

2.3 Sémantique de `Prolog` . . . 25

2.3.1 Ordre des clauses . . . 26

2.3.2 Ordre des buts . . . 26

3 LISTES ET RECURSIVITÉ **29**

3.1 Les listes et leur représentation . . . 30

3.2 La recursivité . . . 31

3.3 Techniques de recursivité . . . 33

3.3.1 Recursivité pour les fonctions numériques . . . 34

3.3.2 Recursivité simple . . . 34

3.3.2.1 Arrêt lorsque la liste est vide . . . 35

3.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé . . . 35

3.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte . . . 36

3.3.3 Recursivité multiple . . . 37

3.4 Exercice . . . 38

4 TECHNIQUES DE PROGRAMMATION EN PROLOG **41**
4.1 Fonctionnement de la coupure . . . 42
4.1.1 Coupure rouge et coupure verte . . . 43
4.1.2 Arrêt après la première solution . . . 45
4.1.3 Coupure et clause conditionnelle . . . 46
4.2 La négation comme échec . . . 47
4.3 Opérations numériques avec les listes . . . 50
4.4 Opérations avec les matrices . . . 51
4.5 Prédicats ensemblistes . . . 53
4.5.1 `setof` . . . 54
4.5.2 `bagof` . . . 55
4.5.3 `findall` . . . 55
4.6 Traitements des foncteurs . . . 56

5 PROGRAMMATION LOGIQUE SOUS CONTRAINTES (PLC) **57**
5.1 PLC sur des domaines finis . . . 59
5.2 PLC sur les nombres rationnels et réels . . . 64
5.3 CHR : Constraint Handling Rules . . . 66
5.4 Références . . . 67
5.5 Exercices . . . 68

6 LES BASES DE DONNÉES EN PROLOG **71**
6.1 Définir les relations . . . 71
6.2 Définir les requêtes . . . 72
6.2.1 Les requêtes recursives . . . 73
6.3 La programmation logique et le modèle relationnel des bases de données . . . 74
6.3.1 L'opération d'union . . . 74
6.3.2 L'opération de différence . . . 74
6.3.3 L'opération du produit cartésien . . . 74
6.3.4 L'opération de projection . . . 75
6.3.5 L'opération de sélection . . . 75
6.4 Gestion de la base de données . . . 76
6.5 Introduction aux bases de données déductives . . . 78
6.6 L'itérativité en Prolog . . . 80
6.7 Utilisation de l'itérativité aux bases de données . . . 81

7 TROIS APPLICATIONS **83**
7.1 Algorithme de routage . . . 83
7.2 Planification . . . 87
7.3 Programmation pour Sudoku . . . 89