Chrysostome Baskiotis

LOGIQUE COMPUTATIONNELLE & PROLOG

Fascicule 2

Introduction au Prolog

Année 2013

PROLOGUE AU PROLOG

`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 résulte 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 proéduraux. 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 presque 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 par les langages de deux autres catégories. Pour le `Lisp`, par exemple, toute action peut s'exprimer à l'aide d'une fonction, toute donnée peut être codée à l'aide d'une liste. Donc pour le `Lisp` tous les problèmes sont réductibles à des fonctions et des listes. La manière dont les fonctions et les listes sont prises en compte par l'ordinateur n'est pas une préoccupation pour le programmeur. De même pour `Prolog` tout problème peut se ramener à un calcul de la valeur de vérité d'une suite des prédicats.

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

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

---

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.

- l'utilisation des faits du domaine de connaissance comme des données
- l'utilisation des règles pour exprimer les relations entre les faits ;
- l'utilisation des 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 autres 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 (« inventeur » des langages de programmation à forte dominante algorithmique, `Algol` et `Pascal`.)

Algorithmes + Structure de données = Programmes

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

P. Hayes, à la même époque, pensait que la computation était de la déduction contrôlée, tandis que E. F. Codd envisageait les systèmes de bases de données comme étants composés de deux éléments : un premier élément relationnel qui déterminait la structure logique des données et un second élément de contrôle qui permettait le stockage et le traitement des données. R. Kowalski en partant de ces idées, a établi (in Communications ACM, vol.22, no 7, july 1979, pp.424-436) 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'exploiter. 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 l'école 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` 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 de 2e année et qui font partie du cycle des cours d'IA, à savoir Intelligence Artificielle symbolique et l'intelligence artificielle compurationnelle.

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.
Dans le même esprit mais nettement plus orienté Prolog, il y a le livre
M. SPIVEY : An introduction to logic programming through Prolog, Prentice-Hall, 1995.
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*, Sprin-

ger, 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 craft of Prolog*, MIT Press, 1990.

4

LOGIQUE DES PROPOSITIONS ET PROGRAMMATION

4.1 Syntaxe de `Prolog` 48
  4.1.1 Les faits 48
  4.1.2 Les règles 49
  4.1.3 Les questions 50
  4.1.4 Les faits, règles et questions en `Prolog` 50
4.2 Les données 52
  4.2.1 Les données simples 52
  4.2.2 Les données structurées 53
  4.2.3 Types des données 54
4.3 Fonctionnement (simplifié) de `Prolog` 54
4.4 Repésentation des nombres naturels 55
4.5 Les opérateurs de base de `Prolog` 57
4.6 Un exemple 57
4.7 Exercices 58

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 une caractéristique et/ou une propriété des objets et aussi de 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` :

```
ilPleut.                        // Un fait
porteParapluie :- ilPleut.      // Une règle

humain(socrate).                // Un fait
humain(aristote).               // Un autre fait
mortel(X) :- humain(X).         // Une règle
existeHumain :- humain(X).      // Une autre règle

animal(X) :- lion(X).           // Un fait

?- mortel(socrate).             // Une question
?- humain(X).                   // Une autre question
```

Exemple des questions avec les réponses de `Prolog` :

```
?- ilPleut.
yes

?- porteParapluie.
yes

?- animal(socrate).
no

?- mortel(X).
X = socrate;
X = aristote;
no

?- existeHumain.
yes
```

4.1 Syntaxe de `Prolog`

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

4.1.1 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 différentes composantes 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 avec, éventuellement, 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` est 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.

4.1.2 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*.

La forme générale d'une règle est la suivante :

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

où A et 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 pas de négations) et $B_1, B_2, \ldots, B_n$ 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'univers.

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.

4.1.3 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 :

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

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

4.1.4 Les faits, règles et questions 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.

MISE EN PRATIQUE

La manière la plus simple pour exprimer une relation est de disposer d'une liste de faits, c'est–à–dire avoir une base de données avec des faits.
Par exemple construisons une base de données qui contient les noms de différentes personnes :

```
homme(socrate).
homme(aristote).
```

Remarquons que même les noms propres doivent commencer par une lettre minuscule, car dans le cas contraire Prolog considérera qu'il s'agit d'une variable.

Nous pouvons maintenant poser des questions. Par exemple

```
?- homme(socrate).
   ==> true
```

mais

```
?- homme(toto).
  ==> false
```

Cette réponse peut paraître surprenante car Toto est bien un homme.
On doit comprendre que si quelque chose n'est pas déclaré dans la base de données, Prolog, comme d'ailleurs tous les langages de programmation, considère qu'il n'existe pas.
Pour s'en convaincre, voici la réponse de Prolog à la négation de la dernière question~:

```
?- not(homme(toto)).
   ==> true
```

Nous pouvons aussi voir défiler toute la base de données en utilisant les variables comme suit :

```
?- homme(X).
   ==> X = socrate ;
       X = aristote.
```

Cette réponse est obtenue parce que Prolog a unifié la variable X avec chaque élément de la base de données homme.

Nous pouvons ajouter une nouvelle base de données concernant les hommes qui sont mortels (Bien sûr nous savons que tout homme est mortel. Mais Prolog ne le sait pas et nous ferons en sorte qu'il puisse l'envisager et, donc, de nous le dire.) :

```
mortel(socrate).
mortel(aristote).
```

Nous pouvons maintenant poser des questions composées, comme par exemple :

```
?- mortel(socrate), mortel(aristote).
   ==> true
```

Mais, comme nous avons dit, notre objectif est d'arriver à stocker une connaissance non pas factuelle et véhiculée par les bases de données, mais une connaissance générale, une loi ou, comme on dit en Prolog et en IA, une règle applicable sur des faits particuliers.

Il faut pour cela réfléchir.
Nous avons deux bases de données "homme" et "mortel" que nous voulons mettre en association.
Ceci est a priori envisageable parce que ces deux bases ont quelques (ici tous les) éléments en commun.
Une relation peut être, entre autre, une relation de causalité

```
— s'il pleut, je prends mon parapluie
ou d'antériorité
— j'initialise d'abord somme à 0 et ensuite je rajoute des éléments dans somme.
Dans notre cas nous avons une relation de causalité.
Il faut maintenant trouver l'élément qui est la cause et l'élément qui est l'effet.
Si on considère que mortel est l'effet dont la cause est l'homme,
on peut se rendre compte que ce n'est pas vrai en général car même le soleil est mortel
et le soleil n'est pas un homme ! Par contre un homme est toujours mortel.
Par conséquent nous avons la formule bien formée ""~homme \/ mortel" ce qui en Prolog s'écrit

mortel(X) :- homme(X).

qui est une règle. En utilisant cette règle, on a

mortel(socrate).
   ==> true

ce qu'on savait déjà.
Ce qui est intéressant ici ce que nous pouvons avoir des nouvelles connaissances
sous forme explicite. Par exemple si on ajoute le fait homme(toto)., alors on a

{?- mortel(toto).
   ==> true

c'est-à-dire nous obtenons une nouvelle connaissance qui, certes, était latente
dans la base de données mais l'ordinateur n'y avait pas accès et par conséquent
il ne pouvait pas l'utiliser.
```

4.2 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.

4.2.1 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 :
$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) : -\texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$
$\texttt{nomFait}(\texttt{objet}_1, \texttt{objet}_2, \texttt{objet}_3) : -\texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$
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 :

$$\texttt{nomRelation}(\texttt{objet}_1, \texttt{objet}_2, _) : -\texttt{nomFait}_1(\texttt{objet}_1, \texttt{objet}_2), \texttt{nomFait}_2(\texttt{objet}_1)$$

$$\texttt{nomFait}(\texttt{objet}_1, _, \texttt{objet}_3) : -\texttt{nomFait}_3(\texttt{objet}_1), \texttt{nomFait}_4(\texttt{objet}_3)$$

4.2.2 Les données structurées

Les constantes et les variables sont des données brutes, sans aucune structuration. On peut aussi envisager d'avoir 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'aperçoî t 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`.

4.2.3 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.

4.3 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, ¬p ∨ ¬q}

qui nous donne la clause absurde ⊥. En conséquence le programme induit la fbf p ∧ q, c'est-à-dire E ⊨ p ∧ q.

4.4 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))) ;
```

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

```
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 $\mathbb{T} = \{\mathtt{zero}, \mathtt{s}/1, \mathtt{X}, \mathtt{Y}\}$ et nous avons aussi les prédicats $\{\mathtt{nat}/1, \mathtt{add}/3\}$. 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) :

- $\mathtt{zero} \to \phi_I(\mathtt{zero}) = \mathtt{zero}_I = 0$
- s(X) $\to \phi_I\left(\mathtt{s}(\overline{\phi}_I(\mathtt{X}))\right) = \mathtt{s}_I(\overline{\phi}_I(\mathtt{X})) = \overline{\phi}_I(\mathtt{X}) + 1$
- $\mathtt{add}(\mathtt{X},\mathtt{Y},\mathtt{Z}) \to \phi_I\left(\mathtt{add}(\overline{\phi}_I(\mathtt{X}),\overline{\phi}_I(\mathtt{Y}),\overline{\phi}_I(\mathtt{Z}))\right) = \mathtt{add}_I\left(\mathtt{add}(\overline{\phi}_I(\mathtt{X}),\overline{\phi}_I(\mathtt{Y}),\overline{\phi}_I(\mathtt{Z})\right)$ d'où $\overline{\phi}_I(\mathtt{X}) + \overline{\phi}_I(\mathtt{Y}) = \overline{\phi}_I(\mathtt{Z})$.

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)))))
```

4.5 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 connaître 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

4.6 Un exemple

Considérons le graphe de la figure 4.1 :

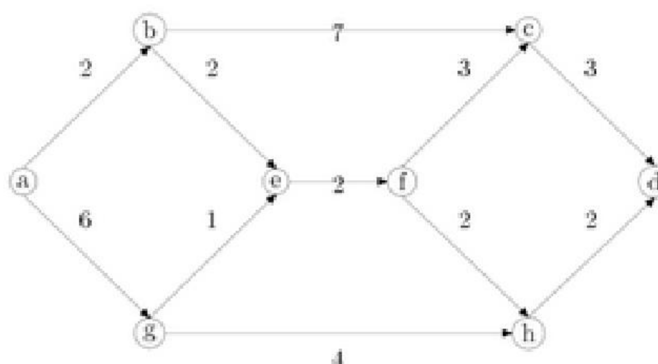


FIGURE 4.1 – Graphe pour un réseau

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

PROGRAMME 4.6.1

```
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 $q \in \widetilde{\mathcal{I}}_E$ 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 : $\widetilde{\mathcal{I}}_E = E$. 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.

4.7 Exercices

EXERCICE 4.1 *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(s(zero))), Z)` *donnera comme résultat* `Z = s(s(s(s(s(s(zero)))))).`

EXERCICE 4.2 *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 4.3 *Écrire le programme Prolog* `egal(X,Y)` *qui teste si les variables* `X`, `Y` *ont la même valeur.*

EXERCICE 4.4 *Écrire le programme Prolog* `pair(X)` *qui teste si la valeur de* `X` *est paire. On fera l'hypothèse que* `zero` *est pair.*

EXERCICE 4.5 *Même chose pour le prédicat* `impair(X)`.

EXERCICE 4.6 *Examiner la base de données de l'exemple 4.6. établir la notion d'un sommet successeur d'un autre. Par exemple le sommet* d *est successeur du sommet* a.

EXERCICE 4.7 *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 4.8 *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.

5

APPLICATIONS DU CALCUL PROPOSITIONNEL AVEC PROLOG

5.1 Substitutions et assignations 61
5.2 Le triplet Hoare 62
5.3 Les règles des connecteurs 63
5.3.1 La négation comme connecteur 63
5.3.2 La disjonction comme opérateur 64
5.3.3 La conjonction comme opérateur 64
5.3.4 Règles pour $\vee$ et $\wedge$ 64
5.3.5 Règles pour $\rightarrow$ 65
5.3.6 Règles pour la substitution 65
5.4 Test logique de l'argumentation 66
5.5 Applications du calcul propositionnel 67
5.5.1 Énigmes 67
5.5.2 Circuits électroniques digitaux 69
5.6 Exercices 71

Nous allons maintenant explorer la manière que la logique d'ordre 0 peut être appliquée dans des situations concrètes. On commence par déterminer la substitution et l'assignation qui sont des opérations fondamentales en programmation. Ensuite on présente les règles des connecteurs du calcul propositionnel et on termine avec deux applications.

5.1 Substitutions et assignations

On note par `f[X]` une fbf qui exprime une propriété de la variable `X`. Si on remplace la variable `X` par le terme `t` (substitution de `X` par `t`) on peut noter cette opération par `f[X:=t]`. Il faut bien noter que X est obligatoirement une variable. En effet la substitution aboutit en une modification de la valeur de X et on ne peut modifier la valeur d'un élément que si celui-ci est une variable

Remarquons que la notation la plus souvent utilisée pour la substitution est `f[t/X]` qui est plus compliquée à manipuler. On rencontre aussi la notation $\mathtt{f}^{t}_{X}$.

On peut envisager des substitutions simultanées de $\mathtt{X}_1, \ldots, \mathtt{X}_n$ par $\mathtt{t}_1 \ldots, \mathtt{t}_n$ que l'on notera

`f[X1:=t1,...,Xn:=tn]`. Il faut faire attention à la différence entre la substitution simultanée, comme la précédente, et les substitutions consécutives `f[X1:=t1], ..., f[Xn:=tn]`.

Effectuer la substitution `f[X:=t]` consiste à faire l'opération de l'assignation `X:=t`. Il va de soi qu'il ne faut confondre l'assignation avec l'égalité, bien que parfois on utilise le même symbole pour les deux opérations. Essayons de préciser les choses.

Considérons l'instruction d'assignation

```
X := t
```

et qui doit se lire, d'après Dijkstra, « `X` devient `t` » et qui

- calcule l'expression `t`, et
- stocke le résultat dans la variable `X`.

Après cette assignation, on pourrait tester l'égalité entre la valeur stockée dans `X` et la valeur calculée de `t`, c-à-d. le test de l'égalité ne peut se faire qu'après avoir effectué l'assignation.

5.2 Le triplet Hoare

Nous allons aborder maintenant un concept fondamental pour la vérification des programmes et que l'on utilisera très souvent dans la suite.

Considérons un programme informatique. Il dispose d'un ensemble d'instructions et aussi d'un ensemble de constantes et des variables. L'ensemble des valeurs des constantes et des variables forme ce qu'on appelle l'*état* du programme. Après l'exécution d'une instruction la valeur d'une ou plusieurs variables est modifiée. Nous avons donc un nouvel état du programme.

Afin qu'une instruction puisse être exécutée, certaines variables doivent avoir des valeurs qui obéissent à des contraintes précises. Par exemple si on fait une division par la variable `X`, il faut que cette dernière soit différente de 0. Il faut donc que l'état du programme, juste avant l'exécution de la division, contienne la condition $X \neq 0$. Le sous-ensemble de l'état qui contient toutes les conditions pour qu'une instruction (ou même un ensemble d'instructions) S puisse être exécutée s'appelle *précondition* et sera noté par {P}. Après l'exécution de l'instruction un certain nombre de variables ont des valeurs modifiées. C'est la conséquence de l'exécution de l'instruction dans l'environnement défini par la précondition {P}. Les sous-ensemble de l'état qui contient ces variables constitue la postcondition {Q} de l'instruction S.

Le triplet précondition – instruction – postcondition

{P} S {Q}

s'appelle *triplet de Hoare* et joue un rôle fonadamental dans la vérification des programmes.

EXEMPLE 5.2.1 *Le triplet {X > 5} X := X+1 {X > 0} est valide. De même le triplet {X >= 0} X := X+1 {X > 0} . On voit ainsi qu'on peut avoir plusieurs préconditions pour la même instruction.*

En utilisant les triplets de Hoare, on peut définir la substituition comme suit :

DÉFINITION 5.2.1 *[Définition de la substitution] Pour la substitution `f[X:=t]` nous avons le triplet de Hoare*

{f [X := t]} X := t {f}

EXEMPLE 5.2.2 *Soit la substitution* `(X > 1) [X:=X+1]`, *alors nous avons le triplet*

`(X > 1) [X:=X+1] X:=X+1`

ou, même le triplet

`(X+1 > 1) [X:=X+1] X:=X+1`

Il faut bien sûr montrer que l'exécution de l'instruction $\texttt{X} := \texttt{X}+1$ *satisfait à la définition. Il faut démontrer que* $(\texttt{x} > 1)[\texttt{X} := \texttt{X}+1]$ *a la même valeur de vérité dans la précondition que* $(\texttt{x} > 1)$ *à la post-condition. En effet c'est le cas, car l'exécution de l'instruction démarre dans l'état où* $(\texttt{x} > 1)[\texttt{X} := \texttt{X}+1]$ *est vraie et se termine dans l'état où* $(\texttt{x} > 1)$ *est vraie.*

En général pour prouver qu'une substitution `f[X:=t]` est vraie, il faut pouvoir montrer que les valeurs de vérité de `f[X:=t]` dans la précondition et de `f` dans la postcondition soient les mêmes. Ceci vient du fait que les expressions `f[X:=t]` et `f` sont les mêmes à l'exception près que `f[X:=t]` a une occurrence de `t` et `f` a une occurrence de `X`. Étant donné la substitution ne modifie que la valeur de `X`, il faut montrer que la valeur de `t` dans la précondition est égale à la valeur de `X` à la postcondition, ce qui est effectivement le cas car l'exécution de la substitution `X:=t` stocke dans `X` la valeur de `t`.

L'utilisation du triplet de Hoare pour le développement formel d'un programme peut se faire de deux façons :

- soit en vérification, c-à-d. on dispose d'un triplet complet et dans ce cas on vérifie par des méthodes de la logique qu'il n'y a pas d'erreur entre la précondition, la postcondition et le programme ;
- soit en calcul quand on dispose d'un triplet incomplet, c-à-d. un triplet composé seulement de deux termes quelconques du triplet et on calcule le troisième terme.

ASCÈSE 5.1 *Déterminer les préconditions pour les triplets suivants dont nous connaissons l'instruction et la postcondition.*

	Instruction	*Postcondition*
1	`X := X + 7`	`X + Y > 20`
2	`X := X - 1`	X^2 `+ 2X = -3`
3	`X := X - 1`	`(X+1)(X-1) = 0`
4	`Y := X + Y`	`Y = X`
5	`Y := X + Y`	`Y = X + Y` *height*

5.3 Les règles des connecteurs

Dans cette section nous rappelons les règles des connecteurs de la logique d'ordre 0 et on rajoute queques unes supplémentaires.

5.3.1 La négation comme connecteur

Nous avons pour la disjonction les règles suivantes :

Double négation : $\neg\neg p \leftrightarrow p$

Distributivité de $\neg$ par rapport à $\leftrightarrow$: $\neg(p \leftrightarrow q) \leftrightarrow (\neg p \leftrightarrow q)$

Cette propriété permet de déterminer la non-équivalence $(p \not\leftrightarrow q)$ comme $(\neg p \leftrightarrow q)$.

5.3.2 La disjonction comme opérateur

Nous avons pour la disjonction les règles suivantes :

Élément nul : $p \vee vrai \leftrightarrow vrai$

Élément identité : $p \vee faux \leftrightarrow p$

Distributivité de $\vee$ par rapport à $\vee$: $p \vee (q \vee r) \leftrightarrow (p \vee q) \vee (p \vee r)$

Monotonicité : $(p \rightarrow q) \rightarrow (p \vee r \rightarrow q \vee r)$

Tautologie : $p \vee \neg p \leftrightarrow vrai$

Idempotence : $p \vee p \leftrightarrow p$

5.3.3 La conjonction comme opérateur

Une règle fondamentale :

$(p \wedge q) \leftrightarrow [(p \leftrightarrow q) \leftrightarrow (p \vee q)]$

Nous avons aussi les règles suivantes :

Élément nul : $p \wedge faux \leftrightarrow faux$

Élément identité : $p \wedge vrai \leftrightarrow p$

Distributivité de $\wedge$ par rapport à $\wedge$: $p \wedge (q \wedge r) \leftrightarrow (p \wedge q) \wedge (p \wedge r)$

Monotonicité : $(p \rightarrow q) \rightarrow (p \wedge r \rightarrow q \wedge r)$

Contradiction : $p \wedge \neg p \leftrightarrow faux$

Idempotence : $p \wedge p \leftrightarrow p$

- $(p \wedge q) \leftrightarrow [(p \wedge \neg q) \leftrightarrow \neg p]$
- $(p \wedge q) \leftrightarrow (p \leftrightarrow q) \wedge p$
- $p \wedge (q \leftrightarrow r) \leftrightarrow [(p \wedge q) \leftrightarrow (p \wedge r) \leftrightarrow p]$

Remplacement : $[(p \leftrightarrow q) \wedge (p \leftrightarrow r)] \leftrightarrow [(p \leftrightarrow q) \wedge (r \leftrightarrow q)]$

5.3.4 Règles pour $\vee$ et $\wedge$

(1) $p \wedge (p \vee q) \leftrightarrow p$

(2) $p \vee (p \wedge q) \leftrightarrow p$

(3) $p \wedge (p \vee q) \leftrightarrow p \wedge q$

(4) $p \vee (\neg p \wedge q) \leftrightarrow p \vee q$

Distributivité de $\vee$ par rappoort à $\wedge$: $p \vee (q \wedge r) \leftrightarrow (p \vee q) \wedge (p \vee r)$

Distributivité de $\wedge$ par rappoort à $\vee$: $p \wedge (q \vee r) \leftrightarrow (p \wedge q) \vee (p \wedge r)$

De Morgan :

(1) $\neg (p \wedge q) \leftrightarrow \neg p \vee \neg q$

(2) $\neg (p \vee q) \leftrightarrow \neg p \wedge \neg q$

5.3.5 Règles pour $\rightarrow$

Pour l'implication nous avons les règles suivantes :

- $(p \rightarrow q) \leftrightarrow [(p \vee q) \leftrightarrow p]$

Distributivité de $\rightarrow$ par rapport à $\rightarrow$: $[p \rightarrow (q \rightarrow r)] \leftrightarrow [(p \rightarrow q) \rightarrow (p \rightarrow r)]$

Distributivité de $\rightarrow$ par rapport à $\leftrightarrow$: $[p \rightarrow (q \leftrightarrow r)] \leftrightarrow [(p \rightarrow q) \leftrightarrow (p \rightarrow r)]$

- $[(p \wedge q) \rightarrow r] \leftrightarrow [p \rightarrow (q \rightarrow r)]$
- $[p \wedge (q \rightarrow p)] \leftrightarrow (p \wedge q)$
- $[p \wedge (q \rightarrow p)] \leftrightarrow p$
- $[p \vee (p \rightarrow q)] \leftrightarrow vrai$
- $[p \vee (q \rightarrow p)] \leftrightarrow (q \rightarrow p)$
- $[(p \vee q) \leftrightarrow (p \wedge q)] \leftrightarrow (p \leftrightarrow q)$

Modus ponens : $[p \wedge (q \rightarrow r)] \rightarrow q$

Monotonicité : $p \rightarrow (q \rightarrow p)$

Transitivité :

(1) $[(p \rightarrow q) \wedge (q \rightarrow r)] \leftrightarrow (p \rightarrow r)$

(2) $[(p \leftrightarrow q) \wedge (q \rightarrow r)] \leftrightarrow (p \rightarrow r)$

(3) $[(p \rightarrow q) \wedge (q \leftrightarrow r)] \leftrightarrow (p \rightarrow r)$

Le bloc "case" :

(1) $(p \vee q \rightarrow r) \leftrightarrow [(p \rightarrow r) \wedge (q \rightarrow r)]$

(2) $r \leftrightarrow [(p \rightarrow r) \wedge (\neg p \rightarrow r)]$

5.3.6 Règles pour la substitution

Pour la substitution nous avons les règles suivantes :

(a) $(X = Y) \wedge f[X := t] \leftrightarrow (X = Y) \wedge f[Y := t]$

(b) $(X = Y) \rightarrow f[X := t] \leftrightarrow (X = Y) \rightarrow f[Y := t]$

(c) $t' \wedge (X = Y) \rightarrow f[X := t] \leftrightarrow t' \wedge (X = Y) \rightarrow f[Y := t]$

ASCÈSE 5.2 *Le dual d'une fbf est une fbf obtenue en interchangeant*

- *vrai et faux*
- $\vee$ *et* $\wedge$
- $\rightarrow$ *et* $\not\leftarrow$
- $\leftrightarrow$ *et* $\not\leftrightarrow$

Soient les fbf suivantes

(1) $p \wedge q$

(2) $p \rightarrow q$

(3) $p \leftrightarrow q$

Si on note chacune de ces fbf par A *et la fbf duale par* A_D*, montrer que nous avons* $A = \neg A_D$*.*

5.4 Test logique de l'argumentation

Nous allons utiliser la logique des propositions pour tester la validité d'une argumentation. Par argumentation on entend un ensemble de faits qui permettent d'établir une conclusion.

EXEMPLE 5.4.1 *Considérons la situation suivante :*

S'il y des nuages et les rideaux ne sont pas tirés, Toto allume la lumière dans sa chambre. Toto n'a pas allumé la lumière. Il y a des nuages. Donc les rideaux de la chambre de Toto sont tirés.

Intutivement on peut admettre que cette dernière proposition est valide. Néanmoins du point de vue logique il faut pouvoir démontrer la conclusion à partir des prémisses (faits). Nous voyons ainsi qu'une argumentation est un ensemble de faits $f_1, f_2, \cdots, f_n$ et une conclusion c et le schéma formel logique est le suivant

Argumentation : $f_1 \wedge f_2 \wedge \cdots \wedge f_n \rightarrow c$

EXEMPLE 5.4.2 (SUITE) *Établissons le schéma formel de cette situation. On note par*

- p = *il y a des nuages ;*
- q = *les rideaux sont tirés ;*
- r = *lumière allumée.*

Le schéma formel de cette situation est donc le suivant :

$$\begin{aligned} f_1 &: \textit{Si} \quad p \textit{ et } \neg q, \quad \textit{alors} \quad r. \\ f_2 &: \neg r. \\ f_3 &: \quad p. \\ c &: \textit{Alors} \quad q \end{aligned}$$

ou, sous forme symbolique :

$$\frac{p \wedge \neg q \rightarrow r, \quad \neg r, \; p}{q}$$

En appliquant l'algorithme de résolution entre $p \wedge \neg q \rightarrow r = \neg(p \wedge \neg q) \vee r = \neg p \vee q \vee r$ *et* r *on obient* $\neg p \vee q$ *qui associée ensuite à* p *fournit le résultat* q *et ainsi la conclusion est prouvée.*

On voit ainsi que pour établir en logique computationnelle la validité d'une argumentation

- on traduit les faits et la conclusion en symboles logiques reliés par des connecteurs, c'-à-d. en fbf et donc depourvues de contenu ;
- on établit un schéma logique ;
- on applique les règles de la logique sur le schéma et soit on démontre la conclusion, soit on arrive à une absurdité, et
- si on a établi que le schéma logique est correct, on peut ensuite l'appliquer en donnant aux fbf leur signification (leur contenu), ce qui permet de valider l'argumentation.

Ainsi une argumentation correcte revêt le statut d'un théorème. Elle est donc vraie indépendamment de la valeur de vérité de ces fbf. Par conséquent si on veut établir qu'une argumentation n'est pas correcte, il est parfois plus facile de trouver un contre exemple au lieu de démontrer son absurdité par l'application des règles logiques.

EXEMPLE 5.4.3 *Si X est un nombre naturel, alors si Y est plus grand que 10, alors Z est égal à 10. Y est plus grand que 10. Par conséquent soit X est nombre naturel, soit Z est égal à 10.*

On pose

- *p = X est un nombre naturel ;*
- *q = Y est plus garnd que 10;*
- *r = Z est égal à 10.*

Le schéma formel de l'argumentation est

$$\frac{p \rightarrow (q \rightarrow r)\,,\;\; q}{p \vee z}$$

On applique l'algorithme de résolution sous forme ensembliste $C = \{\neg p \vee \neg q \vee r, \quad q, \quad \neg p, \quad \neg r\}$ d'où successivement on obtient $C = \{\neg q \vee r, \quad q, \quad \neg r\}, \quad C = \{r, \quad \neg r\} \quad C = \{\perp\}$, donc absurde.

EXEMPLE 5.4.4 *On aurrait pu aussi cherché à trouver un contre-exemple. En effet le schéma formel s'écrit aussi*

$$(p \rightarrow (q \rightarrow r)) \wedge\, q \rightarrow p \vee r$$

ou encore

$$\frac{p \rightarrow (q \rightarrow r)\,,\;\; q}{p \vee r}$$

Si donc on pose $p = r = faux$ et $q = vrai$, on a que $(p \rightarrow (q \rightarrow r)) \wedge\, q$ a la valeur vrai et $p \vee r$ a la valeurs faux, donc cette fbf a la valeur de vérité faux et, par conséquent, l'argumentation n'est pas correcte.

5.5 Applications du calcul propositionnel

En appliquant ce qui vient d'être dit nous pouvons résoudre différents problèmes apartenant à plusieurs domaines. La technique de la résolution est la suivante :

- analyse logique du problème afin d'identifier les faits et les règles ;
- constructioon en Prolog, d'une base de données composée de ces règles et faits, et
- formulation en Prolog des questions relatives au problème à résoudre.

Dans la suite nous ptrésentons des applications pour des énigmes et pour des circuits éléctroniques.

5.5.1 Énigmes

Les énigmes constituent un exemple intéressant pour des applications de Prolog, car elles peuvent facilement être etendues à des problèmes de la vie courante de tous les jours. Une énigme commence par des phrases en français, dont chacune enonce un fait et elle est considérée comme vraie. Elle se termine par une question, pour laquell il faut trouver la réponse, ou une conclusion qu'il faudrait vérifier.

Pour résoudre une énigme, nous créons une base de données avec des faits issus les phrases de l'énigme.

EXEMPLE 5.5.1 *Trois enfants courent ensemble une distance donnée.*

A l'arrivée, Toto dit qu'il a terminé avant son camarade qui porte un maillot rouge.

Lolo, qui porte un maillot jaune, dit qu'il est arrivé avant le garçon qui porte un maillot vert.

Établir l'ordre d'arrivé des trois enfants.

Ce qui nous intéresse ici est l'ordre d'arrivée de trois enfants. On créera donc une liste que l'on appellera avec le nom générique `Struct` et qui contiendra trois sous-listes, une par enfant. Chaque sous-liste aura deux éléments : le nom de l'enfant et la couleur de son maillot. La place d'une sous-liste dans la liste `Struct` indiquera l'ordre d'arrivée de l'enfant correspondant. Esuite on enregistre les faits dans cette structure et on demande à Prolog de résoudre l'énigme.

Le programme est le suivant :

```
pb(Struct) :- createListe(Struct, 3,2),
              faits(Struct).

faits(Struct) :- avant3([toto,_],[_,rouge],Struct),
                 avant3([lolo,jaune],[_,vert],Struct).

%  Cr\'eatiion d'une liste L contenant NbLst sous-listes
%  chacune de sous-listes contenant NbElt \'el\'ements
createListe(L,NbLst,NbElt) :- length(L,NbLst), createListe(L,NbElt).
createListe([],NbElt).
createListe([L1|R],NbElt) :- length(L1,NbElt), createListe(R,NbElt).

% Dans une structure contenant trois sous-listes
% X est avant Y
avant3(X,Y,[X,Y,_]).
avant3(X,Y,[X,_,Y]).
avant3(X,Y,[_,X,Y]).
```

Il reste maintenant à formule la question :

```
?- pb(Struct).
```

qui fournit comme réponse

```
Struct = [[lolo, jaune], [toto, vert], [_G353, rouge]]
```

c-à-d. Lolo est arrivé avant Toto qui lui est arrivé avant le troisième enfant dont le nom n'est pas précisé à l'enoncé, ce qui Prolog l'indique en utilisant la variable système `_G353`. On voit ainsi que Prolog peut répondre même si les données sont incomplètes. Si on veut préciser que le nom du troisième enfant est Titi, alors on a pour le prédicat faits le programme

```
faits(Struct) :- avant3([toto,_],[_,rouge],Struct),
                 avant3([lolo,jaune],[_,vert],Struct),
                 member([titi,_],Struct).
```

Dans ce cas la réponse de Prolog est

```
Struct =  = [[lolo, jaune], [toto, vert], [titi, rouge]]
```

En étudiant cet exemple, nous constatons que pour résoudre en Prolog une énigme il faut suivre la démarche ci-après :

(1) Constituer un univers du discours $\mathcal{U}$ dont les éléments sont ceux sur lesquels porte l'énigme. Dans l'exemple, l'énigme porte sur des enfants, donc $\mathcal{U} = \{\texttt{enfants}\}$.

(2) Chaque élément du discours a des attributs, dont certains sont utilisés par les enoncés de l'énigme. Dans l'exemple, les attributs de l'enfant utilisés par l'énigme sont le nom de l'enfant et la couleur de son maillot. On peut donc représenter un enfant par une liste `[nom, couleur]`.

(3) Pour chaque attribut on détermine ses valeurs telles qu'elles sont précisées par l'énigme.De cette façon on se rend compte des valeurs manquantes que nous pouvons éventuellement compléter si leur détermination est indispensable pour le fonctionnement du programme([1]). Dans l'exemple, l'attribut `nom` a comme valeurs `toto` et `lolo`. et nous constatons qu'il manque le nom du troisième enfant, mais cette connaissance n'est pas indispensable pour avoir une réponse de Prolog.

(4) On crée une base de données à l'aide d'une liste composée avec un nombre d'éléments égal au nombre d'éléments du discours. Chaque élément de cette liste contient une sous liste qui a autant d'éléments qu'il y a des attributs. Cette liste contient donc tous les faits de l'énigme.

(5) Écrire éventuellemnt des programmes pour des prédicats qui font référence à des situations particulières de l'énigme. Dans l'énigme on doit écrire le prédicat `avant3`.

Après ces étapes, on doit poser la question à Prolog qui répondra (ce qui ne signifie pas que sa réponse sera celle que nous attendions).

ASCÈSE 5.3 (D'APRÈS ELBAZ, P.86) *Il y a trois maisons alignées de couleurs différentes (bleue, verte et blanche), occupées par des personnes de nationalité différente (français, espagnol et anglais) et pratiquant des sports différents (tennis, natation et foot). On sait que :*

- *l'habitant de la maison verte pratique la natation ;*
- *la maison verte est située avant celle de l'espagnol ;*
- *l'anglais habite la maison blanche ;*
- *la maison blanche est avant celle dont l'occupant joue au foot, et*
- *le tennisman habite la première maison.*

Établir le sport pratiqué par le français et la nationalité de l'occupant de la maison bleue.

5.5.2 Circuits électroniques digitaux

Ce sont des circuits électroniques dont les entrées-sorties sont des valeurs booléennes 0 et 1. On les utilise pour faire des opérations arithmétiques. Dans un tel circuit les valeurs des sorties sont conditionnées par les valeurs des entrées. Un circuit est décrit par un diagramme qui est la donnée de ess portes et de leurs interconnexions. Une porte est un composant qui a une seule sortie dont la valeur est une fonction booléenne de ses entrées.

Pour utiliser Prolog afin de décrire le comportement d'un circuit, il faut d'abord établir des prédicats qui décrivent le fonctionnement de chaque type de porte (par exemple porte ET, porte OU, porte NAND, etc). Ensuite pour chaque circuit donné on doit

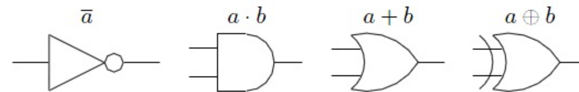
(1) donner des étiquettes à chaque connexion du circuit ;

(2) établir le fonctionnement du circuit en utilisant les étiquettes des connexions et les prédicats relatifs aux portes utilisées par le circuit.

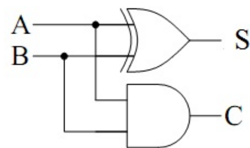
---

1. Il s'agit dans ce cas des connaissances latentes, connues de tous et, de ce fait, non présentées par les enoncés de l'énigme.

Sur la figure suivante sont données les conventions graphiques traditionnelles pour les opérateurs logiques.



EXEMPLE 5.5.2 *Le circuit de la figure 5.5.2 représente un demi additionneur binaire (half-adder) de deux nombres booléens* `a` *et* `b`*.*



La table de vérité de ce composant est

a	b	s (somme)	r (report - carry)
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

De cette table de vérité on en conclut que $s = (\neg a \wedge b) \vee (a \wedge \neg b)$, $r = a \wedge b$.

On remarque que la somme s *est le résultat d'un OU exclusif, c-à-d ? d'un XOR.Si on veut décrire son fonctionnement en Prolog, on doit d'abord écrire des prédicats pour différentes portes :*

```
% Entr\'ee - sortie :  es /2
es(X,X).

% non /2
non(0 ,1). non(1 ,0).

% et /3
et(0 ,0 ,0). et(0 ,1 ,0). et(1 ,0 ,0). et(1 ,1 ,1).

% ou /3
ou(0 ,0 ,0). ou(0 ,1 ,1). ou(1 ,0 ,1). ou(1 ,1 ,1).

% xor /3
```

```
xor(X,Y,Z) :- non(X,U), et(U,Y,T), non(Y,V), et(X,V,W), ou(T,W,Z).

% nor /3
nor(X,Y,Z) :- ou(X,Y,T), non(T,Z).

% nand /3
nand(X,Y,Z) :- et(X,Y,T), non(T,Z).

% implique /3
implique(X,Y,Z) :- non(X,U), ou(U,Y,Z).

% equiv /3
equiv(X,Y,Z) :- non(X,U), non(Y,V), et(X,Y,T), et(U,V,W), ou(T,W,Z).

% nonet /4
nonet(X,Y,Z,T) :- et(X,Y,W), et(W,Z,U), non(U,T).
```

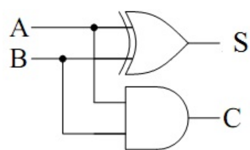
Ensuite on établira le prédicat qui décrit le fonctionnement du demi additionneur :

```
% demi additionneur
halfAdder(A,B,S,C) :- xor(A,B,S),  et(A,B,C).
```

et à la fin nous avons la réponse de Prolog

```
?- halfAdder(A,B,S,C).
A = 0,B = 0,S = 0,C = 0 ;
A = 0,B = 1,S = 1,C = 0 ;
A = 1,B = 0,S = 1,C = 0 ;
A = 1,B = 1,S = 0,C = 1 ;
```

ASCÈSE 5.4 *Reprenons le circuit du chapitre 3 qui est donné par la figure 5.4*



Écrire un programme en Prolog qui simule le comprtement de ce circuit.

5.6 Exercices

EXERCICE 5.1 (D'APRÈS A. TURING) *A. Turing a démontré que le problème d'arrêt d'un programme informatique est indécidable, c'-à-d. qu'il n'existe pas une procédure qui peut déterminer dans un temps fini qu'un programme quelconque s'arrête.*

Montrer la conséquence de ce résultat, à savoir qu'une fonction `halt` *ne peut pas être programmée.*

La démonstration se fera par l'absurde. Il faut donc d'abord écrire un prédicat `halt/1` *qui prend comme argument le nom d'un programme et qui se vérifie si le programme s'arrête. Écrire ensuite un prédicat* `test/1` *avec comme argument le nom d'un programme, qu'il appelle* `halt` *et qui qui se vérifie et continue à s'exécuter si halt se vérifie. Démontrer par la suite que l'association de ces deux programme conduit à une absurdité.*

EXERCICE 5.2 *Cinq maisons consécutives, de couleurs différentes, sont habitées par des hommes de différentes nationalités. Ils possèdent tous un animal différent, ont chacun une boisson préférée différente et fument des cigarettes différentes.*

On sait que :

(1) Le norvégien habite la première maison,

(2) La maison à coté de celle du norvégien est bleue,

(3) L'habitant de la troisième maison boit du lait,

(4) L'anglais habite la maison rouge,

(5) L'habitant de la maison verte boit du café,

(6) L'habitant de la maison jaune fume des kool

(7) La maison blanche se trouve juste après la verte,

(8) L'espagnol a un chien,

(9) L'ukrainien boit du thé,

(10) Le japonais fume des craven

(11) Le fumeur de old gold a un escargot,

(12) Le fumeur de gitane boit du vin,

(13) Un voisin du fumeur de chester eld a un renard,

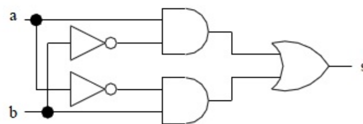
(14) Un voisin du fumeur de kool a un cheval.

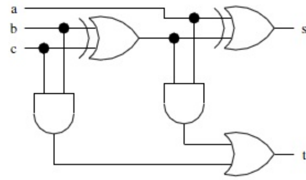
On cherche à savoir qui boit de l'eau et qui possède un zèbre.

Question supplémentaire posée par des esprits malins : Si je ne peux pas résoudre ce problème mais je suis capable d'écrire un programme qui calcule la solution, suis-je intelligent ?

Je pense que la réponse est oui, si le programme est écrit en Prolog.

EXERCICE 5.3 *Pour les circuits suivants, écrire le programme Prolog correspondant.*




a
b
c
s
t

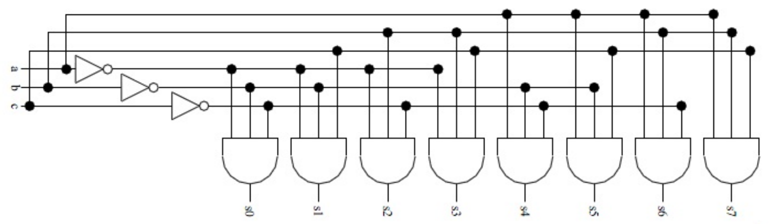

a
b
c

Table des matières

PROLOGUE AU PROLOG **49**

4 LOGIQUE DES PROPOSITIONS ET PROGRAMMATION **47**

4.1 Syntaxe de `Prolog` . . . 48
4.1.1 Les faits . . . 48
4.1.2 Les règles . . . 49
4.1.3 Les questions . . . 50
4.1.4 Les faits, règles et questions en `Prolog` . . . 50
4.2 Les données . . . 52
4.2.1 Les données simples . . . 52
4.2.2 Les données structurées . . . 53
4.2.3 Types des données . . . 54
4.3 Fonctionnement (simplifié) de `Prolog` . . . 54
4.4 Repésentation des nombres naturels . . . 55
4.5 Les opérateurs de base de `Prolog` . . . 57
4.6 Un exemple . . . 57
4.7 Exercices . . . 58

5 APPLICATIONS DU CALCUL PROPOSITIONNEL AVEC PROLOG **61**

5.1 Substitutions et assignations . . . 61
5.2 Le triplet Hoare . . . 62
5.3 Les règles des connecteurs . . . 63
5.3.1 La négation comme connecteur . . . 63
5.3.2 La disjonction comme opérateur . . . 64
5.3.3 La conjonction comme opérateur . . . 64
5.3.4 Règles pour $\vee$ et $\wedge$ . . . 64
5.3.5 Règles pour $\rightarrow$ . . . 65
5.3.6 Règles pour la substitution . . . 65
5.4 Test logique de l'argumentation . . . 66
5.5 Applications du calcul propositionnel . . . 67
5.5.1 Énigmes . . . 67
5.5.2 Circuits électroniques digitaux . . . 69
5.6 Exercices . . . 71