COURS
PROGRAMMATION LOGIQUE
PROLOG

Présenté par:
Dr Kamal AMROUN

Sommaire

Introduction

La programmation logique est née de la découverte qu'une partie du calcul des prédicats de la logique mathématique pouvait servir de support théorique à un langage de programmation. Le développement du premier langage logique appelé PROLOG (ce nom a été choisi par Philippe Roussel comme abréviation de **PRO**grammation en **LOG**ique) a commencé en 1972 par l'équipe d'Intelligence Artificielle dirigée par A. Colmerawer à l'Université de Marseille, U.E.R de Luminy, ceci bien sûr en se basant sur les idées développées par Robinson (1966) et Herbrand (1936) en logique mathématique. Prolog est le langage le plus connu du style déclaratif. Il est le résultat d'un mariage réussi entre le raisonnement automatique et le traitement du langage naturel. En 1980, Prolog est reconnu comme un langage de développement en intelligence artificielle. Depuis, plusieurs interpréteurs (il existe aussi des versions compilées) PROLOG ont été proposés et parfois avec des syntaxes différentes.

Un programme logique est très différent des programmes écrits dans un autre style. Il est très évolué permettant d'exprimer de façon naturelle les textes du langage naturel. Il est très adapté au calcul symbolique et au raisonnement. En effet, écrire un programme dans un langage logique consiste simplement à énoncer une suite de faits et de règles de déduction puis à poser des questions (appelées aussi buts). Il

n'y a pas d'instruction, pas d'affectation et pas de boucles implicites, un programme logique est une suite de clauses particulières appelées clauses de Horn qui sont des clauses ayant au plus un littéral positif. On distingue trois types de clauses de Horn :

- faits : clauses ayant un littéral positif et aucun littéral négatif
- règles : clauses ayant un littéral positif et plusieurs littéraux négatifs
- questions (buts) : clauses ayant un ou plusieurs littéraux négatifs et aucun littéral positif.

Cependant, il faut bien comprendre le fonctionnement de Prolog dans sa stratégie de résolution pour éviter la déclaration de faits ou de règles qui ne peuvent pas être exploités par la suite.

Organisation du manuscrit

Ce cours, destiné aux étudiants de la troisième année de la licence informatique, à raison d'un cours et d'un TD/TP par semaine pendant un semestre, est organisé comme suit :

- Le chapitre 1 est consacré au rappel des concepts de base de la logique mathématique nécessaires à la compréhension de la suite du cours,
- Les notations des programmes logiques sont présentées dans le chapitre 2,
- Le langage Prolog est introduit dans le chapitre 3,
- Le chapitre 4 est consacré aux listes et aux arbres en Prolog,
- Le chapitre 5 traite la notion de coupure.

Chapitre 1

Rappels de logique mathématique

Dans ce chapitre, nous rappelons les notions de base de la logique mathématique utiles à la compréhension de la suite du cours. Nous présentons particulièrement le calcul des prédicats (terme, formule, littéral, variables libres, variables liées, etc.) ainsi que les formes normales de Prenexe et de Skolem. Nous terminons ce chapitre par la présentation des clauses et un algorithme permettant de transformer n'importe quelle formule du calcul des prédicats en une ou plusieurs clauses.

1.1 Calcul des prédicats

Nous rappelons les notions suivantes :

1.1.1 Langage du calcul des prédicats

Alphabet

Le langage du calcul des prédicats est défini sur l'alphabet suivant :

un ensemble de constantes : $a_1, \cdots, a_m$

un ensemble de variables : $x_1, \cdots, x_n$

un ensemble de symboles de prédicats : $P_1, \cdots, P_r$

un ensemble de symboles de fonctions : $f_1, \cdots, f_l$

les connecteurs logiques :$\rightarrow, \neg, \wedge, \vee, \leftrightarrow$

les quantificateurs : $\forall, \exists$

les symboles :"(", ")", $\cdots$

Termes

Les termes du calcul des prédicats sont définis inductivement comme suit :

- toute constante est un terme

- toute variable est un terme

- si f est un symbole de fonction et $M_1, \cdots, M_n$ sont des termes alors $f(M_1, \cdots, M_n)$ est aussi un terme.

Exemple 1.1.1. – 5, 6, x, et y sont des termes. $x+1$, $y*z$, $2-3$, $x*y+5+(y-2)$ et $x+(y-2)$ sont aussi des termes car $+$, $*$ et $-$ sont des symboles de fonctions.

Par contre, $x > y$ n'est pas un terme car ">" est un symbole de prédicat.

Atome ou formule atomique

Définition 1.1.1. Si P est un symbole de prédicat d'arité n et $M_1, \cdots, M_n$ sont des termes, alors $P(M_1, \cdots, M_n)$ est un atome ou une formule atomique.

Exemple 1.1.2. Soit le prédicat *plus-grand* et soient les termes 2, 5, 7, x et y. Alors *plus-grand*(7,5), *plus-grand*(7+2,5) et *plus-grand*(7+y*2,5+y*2) sont des formules atomiques. $*$ et $+$ sont des symboles de fonctions.

Signalons que les symboles : $<, >, \leq, \geq, =, \neq, \cdots$ sont des prédicats.

Remarque 1.1.1. Si f est un symbole de fonction et $M_1, \cdots, M_n$ sont des termes, alors $f(M_1, \cdots, M_n)$ n'est pas une formule atomique. Ainsi, $3+5$ ou $+(3,5)$ n'est pas une formule mais un terme.

Littéral

Définition 1.1.2. Un littéral est une formule atomique ou la négation d'une formule atomique.

Exemple 1.1.3. Soit le prédicat P et les termes t_i et t_j. Alors $P(t_i,t_j)$ est un littéral (formule atomique) et $\neg P(t_i,t_j)$ est aussi littéral (négation de la formule atomique).

Formule

Définition 1.1.3. Une formule (parfois appelée formule bien formée) est définie inductivement comme suit :

toute formule atomique est une formule

si A est une formule, alors $\neg A$, $\forall x A$ et $\exists x A$ sont aussi des formules

si A et B sont des formules, alors $A \longrightarrow B$, $A \vee B$, $A \wedge B$ et $A \longleftrightarrow B$ sont aussi des formules

Exemple 1.1.4. Considérons les prédicats A et B. Soient les termes x, y, 7 et 6, alors

$\forall x \exists y(A(x,6) \longrightarrow \neg B(y,7))$ est une formule (bien formée). $5 > 3$, $x = y$, $6 < 7$ et $\forall x \exists y(x = y \longrightarrow x + 50 < y + 60)$ sont aussi des formules.

Ordre des connecteurs

La priorité des connecteurs est définie comme suit (du plus prioritaire au moins prioritaire).

$\neg$ (négation)

$\wedge$

$\vee$

$\forall, \exists$

$\rightarrow$

$\leftrightarrow$

Exemple 1.1.5. $\forall x A(x) \vee \exists B(x,y) \rightarrow A(x) \rightarrow B(x,y)$
Cette formule est équivalente à $((\forall x(A(x) \vee \exists B(x,y)) \rightarrow A(x))) \rightarrow B(x,y)$.

Champ d'un quantificateur

Le champ d'un quantificateur donné est la formule encadrée par la parenthèse ouvrante à droite du quantificateur et la parenthèse fermante correspondante. Dans les formules suivantes, le champ d'un quantificateur est souligné :

$\forall x\underline{\mathbf{A} \wedge \mathbf{B}}$

$\forall x\underline{\mathbf{A}} \rightarrow B$

$\exists x\underline{\mathbf{A} \wedge \mathbf{B}}$

$\exists x\underline{\mathbf{A}} \rightarrow B$

Exemple 1.1.6. $\forall x\underline{P(x)}$

$\forall x\underline{P(x)} \rightarrow Q(x,y)$

$\exists x\underline{P(x)} \rightarrow Q(x,y)$

$$\forall x \underline{P(x) \wedge Q(x,y)}$$
$$\exists x \underline{P(x) \wedge Q(x,y)}$$

Variables liées et variables libres

Définition 1.1.4. Soit x une variable et soit A une formule. Les positions de x dans A sont appelées les *occurrences de* x dans A.

Définition 1.1.5. Soit x une variable et soit A une formule. Une occurrence de x dans A est dite *liée* si elle est dans le champ d'un quantificateur $\forall$ ou $\exists$, sinon elle est dite *libre*. Une formule sans variable libre est dite *close*. Si une formule comporte au moins une variable libre, elle est dite *ouverte*.

Exemple 1.1.7. Soit la formule α suivante :

$$\forall x \exists y (x > 3 \rightarrow x = y + 3).$$

Dans cette formule, les deux occurrences de x et y sont liées. α est donc close.

Remarque 1.1.2. Une variable peut avoir des occurrences libres et des occurrences liées.

Exemple 1.1.8. Soit la formule suivante :

$$(x \neq y) \rightarrow \forall x \exists y (x > 3 \rightarrow x = y + 3).$$

Dans cette formule, les deux premières occurrences de x et y sont libres et elles sont liées dans la suite de la formule.

Définition 1.1.6. Un terme quelconque t est dit *libre* pour la variable x dans une formule A donnée si aucune occurrence libre de x dans A n'appartient à un champ d'un autre quantificateur $\forall y$ et y est une variable utilisée dans t.

Exemple 1.1.9. Soit le terme $t = g(x, m)$. t n'est pas libre pour y dans la formule : $\forall x P(x, y) \longrightarrow \exists z Q(z, x)$ car x du terme t changera de statut de libre vers liée. Par contre, le terme $t' = g(u, z)$ est libre pour y dans la formule $\forall x P(x, y) \longrightarrow \forall z Q(z, x)$.

1.2 Formes normales

Nous allons présenter principalement les formes normales de Prenexe, de Skolem et un algorithme permettant de transformer n'importe quelle formule en forme clausale.

1.2.1 Forme normale Prenexe

Dans cette section, nous allons présenter un moyen de transformer une formule quelconque du calcul des prédicats vers une formule équivalente en forme normale de prenexe. L'intérêt de cette transformation est de montrer plus facilement certaines propriétés.

Définition 1.2.1. Une formule du calcul des prédicats est en forme normale conjonctive de Prenexe si tous les quantificateurs sont en tête de la formule :

$Q_1 x_1 \cdots Q_m x_m \{[A_{11} \vee \cdots \vee A_{1i}] \wedge \cdots \wedge [A_{r1} \vee \cdots \vee A_{rj}]\}$ où

$Q_k \in \{\forall, \exists\}, \forall k \in \{1, \cdots, m\}$ et A_{ij} est une formule atomique ou la négation d'une formule atomique.

Exemple 1.2.1. $\forall x \exists y (P(x, y) \vee \neg Q(x, z))$. Cette formule est en forme normale conjonctive de prenexe. Par contre, la formule $\forall x \exists y P(x, y) \vee Q(x, z)$ n'est pas en forme normale conjonctive de prenexe.

Théorème 1.2.1. *Toute formule du calcul des prédicats A peut être transformée en une autre formule A' équivalente en forme normale conjonctive de prenexe.*

Démonstration. On va construire la formule A' en remplaçant successivement des parties de A par des (sous) formules équivalentes. A cet effet, nous allons appliquer les transformations suivantes.

- Éliminer dans la formule de départ A tous les quantificateurs redondants (éliminer tous les $\forall x$ ou $\exists x$ si aucune occurrence libre de x n'apparaît dans leur champs respectifs).
- Renommer les variables liées comme suit :
  - Si une variable x a n occurrences liées et 0 occurrences libres alors renommer $n-1$ occurrences par différentes variables.
  - Si une variable x a n occurrences liées et au moins une occurrence libre alors renommer les n occurrences liées par n différentes variables.
- Remplacer $A \longrightarrow B$ par $\neg A \vee B$. Remplacer $A \longleftrightarrow B$ par $(\neg A \vee B) \wedge (\neg B \vee A)$
- Appliquer les transformations suivantes :
  - Remplacer $\forall x A \wedge B$ par $\forall x(A \wedge B)$
  - Remplacer $\exists x A \wedge B$ par $\exists x(A \wedge B)$
  - Remplacer $\forall x A \vee B$ par $\forall x(A \vee B)$
  - Remplacer $\exists x A \vee B$ par $\exists x(A \vee B)$
  - Remplacer $A \wedge \forall x B$ par $\forall x(A \wedge B)$
  - Remplacer $A \wedge \exists x B$ par $\exists x(A \wedge B)$
  - Remplacer $A \vee \forall x B$ par $\forall x(A \vee B)$
  - Remplacer $A \vee \exists x B$ par $\exists x(A \vee B)$
- Mettre la formule en forme normale conjonctive en distribuant $\wedge$ sur $\vee$ comme suit :
  - Remplacer $(A \wedge B) \vee C$ par $(A \vee C) \wedge (B \vee C)$

– Remplacer $A \vee (B \wedge C)$ par $(A \vee B) \wedge (A \vee C)$

□

Exemple 1.2.2. Mettre la formule suivante sous forme normale de prenexe conjonctive.

$$\forall x A \longrightarrow \exists y B$$

– On remplace $\forall x A \longrightarrow \exists y B$ par $\neg \forall x A \vee \exists y B$

$\neg \forall x A \vee \exists y B = \exists x \neg A \vee \exists y B$

– On remplace $\exists x \neg A \vee \exists y B$ par $\exists x(\neg A \vee \exists y B)$

– On remplace $\exists x(\neg A \vee \exists y B)$ par $\exists x \exists y(\neg A \vee B)$. Cette formule est bien entendu en forme normale de prenexe conjonctive.

Exemple 1.2.3. Mettre la formule suivante sous forme normale de prenexe conjonctive.

$$\forall x \exists y A(x, y) \longrightarrow \exists y B(y, y)$$

– On renomme les variables liées. On obtient la formule suivante :

$\forall x \exists y A(x, y) \longrightarrow \exists y_1 B(y_1, y_1)$

– On remplace $\longrightarrow$ par $\neg$ et $\vee$. On obtient $\neg \forall x \exists y A(x, y) \vee \exists y_1 B(y_1, y_1)$. Cette formule est équivalente à $\exists x \forall y \neg A(x, y) \vee \exists y_1 B(y_1, y_1)$.

– On applique les règles précédentes. On obtient finalement la formule suivante en forme de prenexe conjonctive. $\exists x \forall y \exists y_1(\neg A(x, y) \vee B(y_1, y_1))$.

1.2.2 Forme normale de Skolem

Définition 1.2.2. Une formule A en forme normale de Skolem est une formule A_s en forme normale prenexe où les connecteurs existentiels ont été supprimés de telle

sorte que si A est satisfiable alors A_s est satisfiable.

Exemple 1.2.4. La formule $\forall x \forall y P(x, y, z)$ est en forme normale de Skolem. Par contre $\forall x \exists y P(x, y, z)$ n'est pas en forme normale de Skolem à cause de la présence du quantificateur $\exists$.

Algorithme de mise sous forme normale de Skolem

Il existe un algorithme permettant de transformer n'importe quelle formule du calcul des prédicats vers une formule en forme normale de Skolem conservant la satisfiabilité.

Algorithme 1 *Algorithme de mise sous forme de Skolem*

Input : une formule A
Output : une formule en forme normale de Skolem.

1: Mettre A en forme normale conjonctive de prénexe
2: Remplacer chaque occurrence d'une variable y quantifiée existentiellement par un terme de la forme $f(x_1, \ldots, x_n)$ où $x_1, \ldots, x_n$ sont les variables quantifiées universellement et dont le quantificateur apparaît avant celui de y dans A.
3: Si aucun quantificateur universel n'apparaît avant $\exists y$ dans A, alors remplacer y par une constante de Skolem.

Remarque 1.2.1. Pour chaque variable y, quantifiée existentiellement, choisir un nom différent pour la fonction de Skolem (resp constante de Skolem).

Exemple 1.2.5. La formule $\forall z \forall x \exists y P(x, y, z)$ de l'exemple 1.2.4 n'est pas en forme de Skolem. Sa skolémisation donne une formule de la forme $\forall z \forall x P(x, f(z, x), z)$ où $f(z, x)$ est une fonction de Skolem dont les arguments sont les variables quantifiées universellement et dont le quantificateur apparaît avant le $\exists$ considéré.

Exemple 1.2.6. Soit la formule $\exists h \forall z \forall x \exists y P(h, x, y, z)$. La skolemisation de cette

formule donne $\forall z \forall x P(b, x, g(z, x), z)$ où b est une constante de Skolem et g(z,x) est une fonction de Skolem.

Exemple 1.2.7. Soit la formule $\exists h \exists z \exists x \exists y P(h, x, y, z)$. La skolemisation de cette formule donne $P(b, c, d, e)$ où b, c,d et e sont des constantes de Skolem.

1.2.3 Clauses

Définition 1.2.3. Une clause est une formule (universellement quantifiée) dont le corps est une disjonction de formules atomiques ou de la négation de formules atomiques.

Exemple 1.2.8. $A(x) \wedge B(y)$ n'est pas une clause.

$\forall x \forall y(A(x) \vee B(y))$ est une clause.

$\forall x \forall y(A(x) \vee B(y) \vee C(x, y, z))$ est une clause.

$\forall x \forall y(A(x) \vee \neg B(y))$ est une clause.

$\forall x \forall y(\neg A(x) \vee B(y))$ est une clause.

$\forall x \forall y(\neg A(x) \vee \neg B(y))$ est une clause.

$\forall x \forall y((\neg A(x) \wedge B(x)) \vee \neg B(y))$ n'est pas une clause.

$\forall x \forall y((\neg A(x) \longrightarrow B(x)) \longleftrightarrow \neg B(y))$ n'est pas une clause.

Remarque 1.2.2. Une formule en forme normale conjonctive est une conjonction de clauses.

Algorithme de mise sous forme clausale

L'algorithme 2 permet de transformer n'importe quelle formule du calcul des prédicats vers une ou plusieurs clauses.

Algorithme 2 *Algorithme de mise sous forme clausale*

Input : une formule A
Output : une formule en forme clausale.

1: Mettre A en forme normale conjonctive de prénexe
2: Mettre la formule obtenue en forme de Skolem
3: Remplacer partout le connecteur $\longrightarrow$ par $\{\neg, \vee\}$ ($A \longrightarrow B$ par $\neg A \vee B$).
4: Remplacer les $\wedge$ de la formule obtenue par des virgules "," donnant lieu à une ou plusieurs clauses.

Exemple 1.2.9. Soit la formule $\forall x A(x) \longrightarrow \exists y B(y)$. La clause correspondante est obtenue comme suit :

1. *Forme normale conjonctive de prenexe*

   $\exists x \exists y(\neg A(x) \vee B(y))$.

2. *Forme normale de Skolem*

   $\neg A(u) \vee B(v)$. u et v sont des constantes de skolem.

3. *Clause*

   Cette formule est déjà une clause.

Exemple 1.2.10. Soit la formule $\forall x(A(x) \vee B(x)) \longrightarrow C(x))$. La clause correspondante est obtenue comme suit :

1. *Forme normale conjonctive de prenexe*

   $\exists y(\neg(A(y) \vee B(y)) \vee C(x))$ qui est équivalente à $\exists y((\neg A(y) \wedge \neg B(y)) \vee C(x))$ elle-même équivalente à $\exists y((\neg A(y) \vee C(x)) \wedge (\neg B(y) \vee C(x)))$.

2. *Forme normale de Skolem*

   $(\neg A(u) \vee C(x)) \wedge (\neg B(u) \vee C(x))$. u est une constante de skolem.

3. *Clauses*

   En remplaçant le $\wedge$ de la formule par une virgule ",", on obtient deux clauses,

à savoir.

$\neg A(u) \vee C(x)$

$\neg B(u) \vee C(x)$.

1.3 Exercices

Exercice N°1

Mettre les énoncés suivants sous forme de formules du calcul des prédicats.

- Tous les êtres humains sont mortels
- Ali est fils de Mohamed
- A l'exception de Said, tous les étudiants sont brillants.
- Certains étudiants ne participent pas aux TDs
- Quelques étudiants préparent tous les exercices de la série de TD.
- Son chat est gris ou noir.
- Ali et Omar sont des étudiants brillants.
- Si un étudiant travaille, il réussira.
- A la fin du semestre, un étudiant est soit admis soit ajourné.
- Tous les jeunes hommes âgés de plus de 20 ans sont mariés.

Exercice N°2

Indiquer les occurrences libres et liées de la variable x dans les formules suivantes :

$Q(x) \vee P(x,x) \rightarrow \neg Q(x) \wedge \exists x P(x,y)$

$\exists x Q(x) \vee P(x,x) \rightarrow \neg Q(x) \wedge P(x,y)$

$\forall x P(x,y) \rightarrow \exists x Q(x) \vee P(x,z) \wedge Q(y)$

$$Q(x) \vee \exists x \forall y P(x,y) \rightarrow \neg Q(y) \wedge P(x,y)$$

Exercice N°3

Mettre les parenthèses pour les formules de l'exercice précédent en respectant la priorité des connecteurs.

Exercice N°4

Transformer chaque formule obtenue au niveau de l'exercice 1 en forme normale conjonctive de prenexe puis en forme normale de skolem.

Exercice N°5

Transformer les formules suivantes en clauses

- $A \longrightarrow B$
- $\forall x A(x) \longrightarrow B$
- $\forall x A(x) \longrightarrow \exists x B(x)$
- $A(x) \longrightarrow \forall x B$
- $A \longrightarrow B \longrightarrow C \wedge D \vee H$
- $\exists x \exists y P(x,y,n)$
- $\forall x \exists y A(x,y) \longrightarrow \exists z Q(z,y) \wedge \forall x B(r,x)$
- $\exists x A(x,y) \longrightarrow \exists y (Q(x,y) \longrightarrow \forall m \forall n R(m,n))$.
- $x = y \longrightarrow \forall z (z > 0) \longrightarrow (x + z < y + z)$

Chapitre 2

Programmation logique

Dans ce chapitre, nous introduisons les concepts fondamentaux de la programmation logique, à savoir la notion de langage logique, la substitution, l'unification et la résolution.

2.1 Langage logique

Définition 2.1.1. Un langage logique est un sous-ensemble de la logique mathématique et l'exécution en est la preuve (démonstration).

Les principales caractéristiques d'un langage logique sont :

- symbolique : dans le sens où tous les objets manipulés sont des symboles (constantes, variables, etc.).
- haut niveau : aucune gestion des dispositifs de la machine.
- déclaratif : on s'intéresse plus à "quoi faire" qu'à "comment faire".
- relationnel : dans ce style de programmation, un programme décrit un état donné du monde réel par un ensemble d'entités et de prédicats (relations) entre elles. On ne s'occupe pas de la manière d'obtenir le résultat. Le programmeur

doit faire la description du problème à résoudre en listant les objets concernés, les propriétés et les relations qu'ils vérifient.

- indéterministe : le mécanisme de résolution (pris entièrement en charge par le langage) est général et universel. Il parcourt de façon non déterministe toutes les possibilités du problème et peut donc retourner plusieurs solutions.

Définition 2.1.2. La programmation logique en clauses de Horn est définie par la donnée :

- d'un langage des données qui est un ensemble de termes. Un terme peut être une constante ou une variable. Si $t_1, \cdots, t_n$ sont des termes et f est un symbole de fonction alors $f(t_1, \cdots, t_n)$ est aussi un terme. Si p est un symbole de prédicat et $t_1, \cdots, t_n$ sont des termes alors $p(t_1, \cdots, t_n)$ est un atome (formule).
- d'un langage de programme définissant un ensemble de clauses définies. On distingue 3 types de clauses définies :

  1. ***fait*** : un fait est une relation sur des objets du monde considérée comme vraie. Par exemple *intelligent(ali)* est un fait du monde réel. *intelligent* est un prédicat et *ali* est une constante. Cette formule factuelle exprime le fait que ali est intelligent. Une base de faits est une suite de faits exprimant des connaissances du monde réel.

     Exemple 2.1.1. Voici un ensemble de faits.
     pere(ali, mohamed).
     freres(youcef, mohamed).
     homme(mohamed)

  2. ***règle*** : une règle est une clause constituée d'un littéral positif (appelé tête de la clause) et plusieurs littéraux négatifs (queue de la clause). Sa syntaxe

est la suivante : $A \leftarrow B_1, \cdots, B_n$. Elle se lit : A est vrai si $B_1, \cdots, B_n$ sont vraies.

Exemple 2.1.2. Voici un ensemble de clauses.

freres(X, Y) ← pere(X,Z), pere(Y,Z), male(X).

freres(X, Y) ← pere(X,Z), pere(Y,Z), male(Y).

sœurs(X, Y)← pere(X,Z), pere(Y,Z), femelle(X), femelle(Y).

3. ***but*** : un but est une formule dont on cherche à savoir si elle est vraie ou fausse. Sa syntaxe est comme suit : $? - B_1, \cdots, B_m$.

   Exemple 2.1.3. Voici quelques buts.

   ?-freres(X, Y).

   ?-sœurs(X, Y).

   ?-male(ali).

   ?-=(X, Y).

   ?->(X, Y).

Variables logiques

Une variable logique représente une entité du monde réel mais une fois instanciée, cette variable ne peut pas changer de valeur comme c'est le cas dans les langages classiques. En d'autres termes, une variable logique n'est pas une zone mémoire.

Remarque 2.1.1. Les variables logiques sont locales et elles sont renommées à chaque utilisation.

Exemple 2.1.4. La formule *étudiant(X)* signifie que X est un étudiant.

2.2 Substitution

Une des notions les plus importantes en programmation logique est la substitution.

Définition 2.2.1. Une substitution σ est un ensemble de couples <variable, terme>, représenté comme suit : $\sigma = \{X_1 \leftarrow t_1, \cdots, X_n \leftarrow t_n\}$.

L'application d'une substitution σ ($\sigma = \{X_1 \leftarrow t_1, \cdots, X_n \leftarrow t_n\}$) à un terme ou une formule t consiste seulement à remplacer les occurrences de $X_1, \cdots, X_n$ dans t par les termes $t_1, \cdots, t_n$.

Exemple 2.2.1. Soit la substitution $\sigma = \{X_1 \leftarrow ali, X_2 \leftarrow omar\}$ et soit la formule t= freres(X,Y). Alors $\sigma(t)$=freres(ali,omar).

Remarque 2.2.1.
- si $X \leftarrow t \in \sigma$, alors $\sigma(X) = t$.
- si f est un symbole de fonction ou de prédicat, alors
  $\sigma(f(t_1, \cdots, t_n)) = f(\sigma(t_1)), \cdots, f(\sigma(t_n))$

2.3 Unification

Définition 2.3.1. Deux termes t_1 et t_2 sont dits unifiables s'il existe une substitution σ telle que $\sigma(t_1) = \sigma(t_2)$.

Définition 2.3.2. Soit E un ensemble de termes E=$\{t_1, \cdots, t_n\}$. Une substitution σ est dite unificateur le plus général de E si les deux conditions suivantes sont satisfaites.
- $\sigma(t_1) = \cdots = \sigma(t_n)$ et
- si σ' est un autre unificateur de E alors il existe une substitution μ telle que $\sigma' = \sigma\mu$.

Algorithme d'unification de Martelli-Montanari

L'algorithme 3 retourne l'unificateur le plus général pour les deux expressions exp_1 et exp_2.

Algorithme 3 *Algorithme d'unification de Martelli-Montanari*

Initialisation :
$\theta_0 = \epsilon$, $E_0 = \{exp_1 = exp_2\}$
Output : Unificateur le plus général de exp_1 et exp_2.

1: **while** $E_i \neq \emptyset$ **do**
2: **if** $E_i = \{f(s_1, \cdots, s_p) = f(s'_1, \cdots, s'_p)\} \cup E'_i$ **then**
3: $E_{i+1} = \{s_j = s'_j\}_{1 \leq j \leq p} \cup E'_i$ et $\theta_{i+1} = \theta_i$
4: **if** $f(s'_1, \cdots, s'_p) = g(s'_1, \cdots, s'_m) \in E_i$ avec $f \neq g$ ou $m \neq p$ **then**
5: échec de l'unification
6: **if** $E_i = \{X = X\} \cup E'_i$ **then**
7: $E_{i+1} = E_i$ et $\theta_{i+1} = \theta_i$
8: **if** $E_i = \{t = X\} \cup E'_i$ ou $E_i = \{X = t\} \cup E'_i$, $t \neq X$ et $X \notin V(t)$ **then**
9: $E_{i+1} = [X \leftarrow t]E'_i$ et $\theta_{i+1} = \theta_i[X \leftarrow t]$
10: **if** $E_i = \{t = X\} \cup E'_i$ ou $E_i = \{X = t\} \cup E'_i$, $t \neq X$ et $X \in V(t)$ **then**
11: échec de l'unification

Interprétation d'un programme logique

Etant donnée une clause de Horn quelconque $B \longleftarrow B_1, \cdots, B_n$. Une double interprétation peut être faite pour cette clause.

- *interprétation déclarative* : cette interprétation stipule que B est vraie si B_1,$\cdots$ et B_n sont vraies en même temps.
- *interprétation procédurale* : dans ce cas, la même règle peut être interprétée comme une règle de réécriture. Ainsi pour prouver B, il faut prouver B_1,$\cdots$ et B_n. Chaque B_i est un atome qu'on cherche à éliminer jusqu'à obtenir un *succès*

(clause vide) ; ceci par application des différentes clauses du programme. En cas de blocage (impossibilité d'éliminer des buts), c'est un *échec*.

Il est à signaler que les deux interprétations (déclarative et procédurale) sont équivalentes. Cette équivalence constitue le principe de la programmation logique.

2.4 Résolution SLD

Dans ce paragraphe, nous présentons le principe de résolution SLD (sélection linéaire définie) qui fonde la sémantique opérationnelle des programmes logiques en général.

Principe de résolution SLD

Le principe de résolution SLD est une règle de simplification des buts qui procède par unification avec les têtes de clauses du programme. C'est une variante du principe de résolution proposé par Robinson pour toutes les clauses du calcul des prédicats (non nécessairement de Horn).

Définition 2.4.1. Soit le but courant $B_1, \cdots, B_{i-1}, B_i, B_{i+1}, \cdots, B_n$ et soit la clause $A \leftarrow A_1, \cdots, A_p$. Supposons que l'unificateur le plus général entre A et B_i est σ via une substitution de renommage des variables θ. Le principe de la résolution SLD est défini par l'équation suivante :

$$\frac{B_1,\cdots,B_{i-1},B_i,B_{i+1},\cdots,B_n}{\sigma(B_1,\cdots,B_{i-1},A_1,\cdots,A_p,B_{i+1},\cdots,B_n)}$$

C'est une réécriture du but $B_1, \cdots, B_{i-1}, B_i, B_{i+1}, \cdots, B_n$ où l'atome B_i est remplacé par $A_1, \cdots, A_p$. Ceci bien sûr en supposant que A et B sont unifiables via une certaine substitution σ.

Remarque 2.4.1. On voit dans le principe de résolution SLD deux sources d'indéterminisme :

- quel but à résoudre
- quelle clause à appliquer

Arbres SLD

L'arbre SLD pour un but $\alpha_1, \cdots, \alpha_n$ est construit comme suit :

- La racine est étiquetée par $\alpha_1, \cdots, \alpha_n$.
- On choisit parmi $\alpha_1, \cdots, \alpha_n$ un sous-but α_k à résoudre (selon la règle de selection).
- Pour chaque clause $C = \beta \leftarrow \beta_1, \cdots, \beta_m$ dont la tête peut s'unifier avec α_k (même prédicat)
  - on renomme les variables de la clause C en $\beta' \leftarrow \beta'_1, \cdots, \beta'_m$ de sorte qu'elle n'utilise que des variables qui n'apparaissent nulle part ailleurs ;
  - si l'unification de α_k avec β' échoue, alors on passe à la clause suivante dans le programme ; sinon, on crée un nouveau fils pour le nœud étiqueté par $\alpha_1, \cdots, \alpha_n$. On labelle la branche associée à ce nouveau nœud avec la clause renommée $\beta' \leftarrow \beta'_1, \cdots, \beta'_m$ et avec l'ufg (unificateur le plus général) θ de β' et α_k. Le nouveau nœud est étiqueté par :
    $(\alpha_1, \cdots, \alpha_{i-1}, \beta'_1, \cdots, \beta'_m, \alpha_{i+1}, \cdots, \alpha_n)\theta$.
- Ce processus est répété jusqu'à ce qu'il ne soit plus possible de rajouter des nœuds (c'est-à-dire jusqu'à atteindre des labels de nœuds qui sont soit vides soit irréductibles).
- Chaque nœud vide représente une solution pour la question initiale (ie. une

instance du but initial qui est une conséquence du programme). Pour trouver cette instance, il faut composer toutes les substitutions (l'unificateur le plus général) menant de la racine au nœud qui a un label vide et réduire le domaine aux variables de $\alpha_1, \cdots, \alpha_n$.

Remarque 2.4.2. Il peut y avoir des branches infinies dans l'arbre SLD.
Les réponses qu'on obtient en un temps fini dépendent particulièrement de deux paramètres : la manière de choisir l'atome à réduire à chaque étape ainsi que l'ordre dans lequel les clauses sont exploitées : recherche en profondeur ou en largeur d'abord.

Exemple 2.4.1. Soit le programme suivant :

$h(0) \longleftarrow .$

$h(s(x)) \longleftarrow h(x).$

La figure 2.1 montre un arbre infini pour le but h(y).

Exemple 2.4.2. Soit le programme suivant :

$h(a) \longleftarrow .$

$h(b) \longleftarrow .$

$r(b) \longleftarrow .$

$r(c) \longleftarrow .$

La figure 2.2 montre deux arbres pour les buts *?-h(x),r(y).* et *?-h(x),r(x).* respectivement.

Dérivation et réfutation SLD

Définition 2.4.2. Une dérivation SLD est une suite de buts obtenus par résolution SLD à chaque étape :

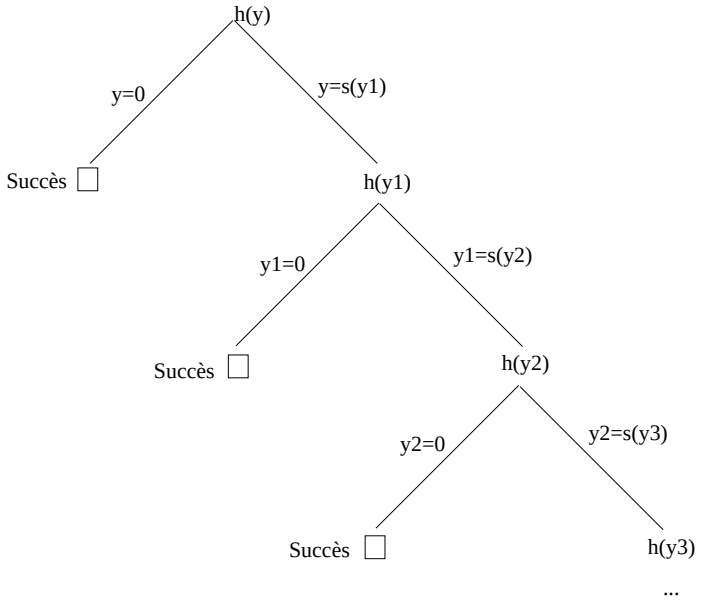

FIG. 2.1 – Arbre de résolution SLD pour l'exemple 2.4.1

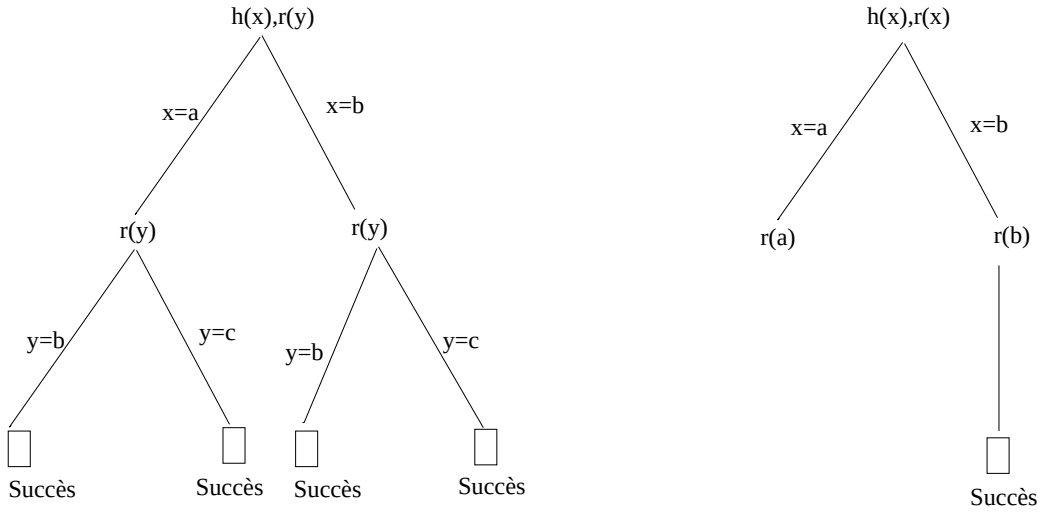

FIG. 2.2 – Arbres de résolution SLD pour l'exemple 2.4.2

$$G_0 \longrightarrow^{C_{i0}} G_1 \cdots G_{n-1} \longrightarrow^{C_{in}} G_n.$$

C_{ij} est la clause appliquée au but G_j pour obtenir le but G_{j+1}.

Une dérivation SLD correspond à une branche complète de l'arbre SLD. Nous distinguons trois cas :

- le dernier but est vide : dans ce cas, on a un succès. Ce cas correspond à la réfutation de la négation du but.
- aucune règle ne peut être appliquée au but courant : c'est un échec
- branche infinie.

Remarque 2.4.3. Pour éviter les conflits de variables, il est important de renommer les variables à chaque étape de la résolution.

2.5 Domaines d'application de la programmation logique

Les domaines d'utilisation les plus connus de la programmation logique sont :

- Traitement du langage naturel,
- Bases de données
- Systèmes experts
- Automates
- Vérification de la correction de programmes
- Outils d'aide au diagnostique des pannes
- ...

2.6 Exemples de programmes logiques

Exemple 2.6.1. Soit le programme suivant :

homme(ali).

homme(omar).

homme(said).

femme(céline).

femme(lila).

pere(said, ali).

pere(omar, ali).

mere(omar, céline).

mere(said, céline).

parents(X,M,P)← mere(X, M), pere(X, P).

frere(X,Y)← homme(Y), parents(X,M,P), parents(Y,M,P).

Soit le but suivant : ?-frere(omar, said).

Le prédicat *frere* s'unifie uniquement avec la tête de la dernière clause du programme. avec la substitution X/omar, Y/said. Cela donne le but suivant (constitué de 3 sous-buts) : *homme(said), parent(said,M,P), parent(omar,M,P).*

Le premier sous-but correspond à un fait donc il est retiré. Il reste les deux autres. En renommant M par M1 et P par P1,on obtient :

mere(said, M1), pere(said, P1), parent(omar, M1, P1).

La substitution M1/céline permet d'unifier le premier sous-but avec un fait du programme. Cela donne : *pere(said,P1), parent(omar, céline, P1).*

Pour P1/ali, le premier sous-but correspond à un fait du programme donc il est effacé. On obtient *parent(omar,céline,ali).* Il s'unifie avec la tête de l'avant dernière clause.

Cela donne *mere(omar,céline), pere(omar, ali).*

Évidemment le premier puis le deuxième sous-but correspondent à des faits du programme donc ils sont tous effacés. Cela correspond à un succès. Donc omar et said sont des frères et Prolog répond par Yes.

L'arbre SLD correspondant est donné par la figure 2.3.

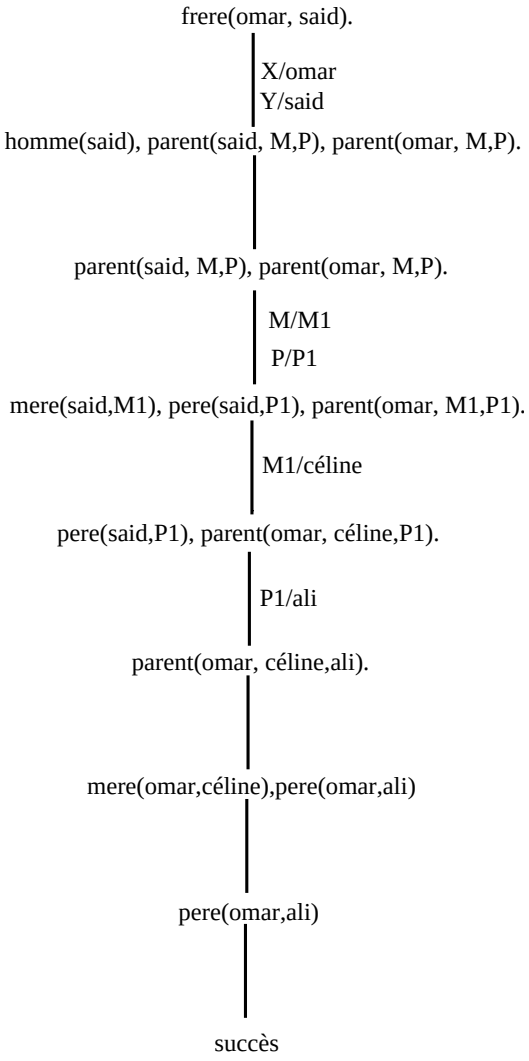

FIG. 2.3 – Arbre de résolution SLD

Exemple 2.6.2. Soit le programme

p(a).

p(b).

p(c).

q(c).

q(d).

r(a).

p(X)⟵ q(X).

q(X)⟵ r(X).

Soit le but ?-p(X).

L'arbre SLD correspondant à ce but est donné par la figure 2.4.

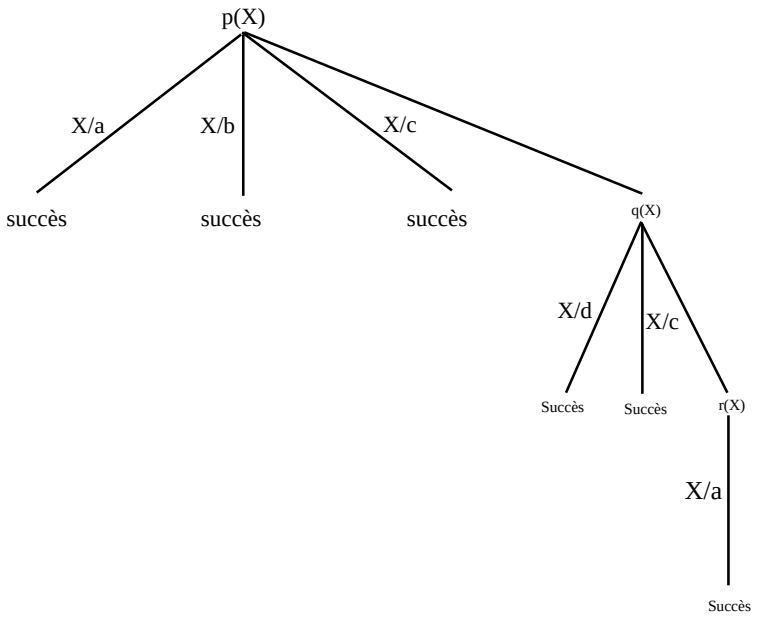

FIG. 2.4 – Arbre de résolution SLD

2.7 Exercices

Exercice N°1

- Expliquer pourquoi uniquement les clauses de Horn peuvent être exploitées pour définir un langage de programmation logique.
- Quelle est la principale différence entre un langage logique et un langage classique comme C ou Pascal ?
- Pourquoi les langages logiques sont considérés comme non déterministes.

Exercice N°2

Ali s'est marié avec Djamila et ont donné naissance à un garçon Said et une fille Malika. Said s'est marié avec Yasmine et ont deux filles Leila et Saida. Malika quant à elle s'est mariée avec Youcef.

On souhaite décrire cette famille en Prolog. On utilisera les prédicats suivants :

- *male*(X) pour signifier que X est de sexe masculin
- *femelle*(X) pour signifier que Y est de sexe féminin
- *enfant*(X, Y) pour signifier que X est enfant de Y
- *mari*(X, Y) pour signifier que X est le mari de Y.

a- Décrire la famille de Ali en définissant un ensemble de faits utilisant les prédicats précédents.

b. Définir le prédicat femme(X, Y) qui signifie que X est la femme de Y.

c- Ecrire les questions suivantes en prolog et donner leurs réponses respectives :

d. Qui est la femme de Youcef ?

e. Qui est le mari de Djamila ?

e. Définir le prédicat garçon(X, Y) qui spécifie que X est un enfant de Y de sexe mâle.

Exercice N°3

Soit le programme logique suivant :

a(X)← b(X,Y),c(Y).

a(X) ← c(X).

b(a,b).

b(a,c).

c(b).

c(a).

a/ Construire l'arbre SLD associé au but ***?-a(X).***

b/ Construire l'arbre SLD associé au but ***?-b(X, Y).***

Exercice N°4

Soit le programme suivant :

p(0).

p(s(x))← p(x).

Donner un arbre SLD pour le but ?-p(y).

Exercice N°5

Soit le programme Prolog suivant :

p(a).

p(b).

q(b).

q(c).

Donner un arbre SLD pour le but ?-p(x),q(y), puis un autre pour le but ?-p(x),q(x).

Chapitre 3

Le langage PROLOG

Ce chapitre est consacré au langage Prolog. nous présentons essentiellement la syntaxe d'un programme Prolog, l'algorithme d'unification, l'algorithme d'exécution d'un programme Prolog ainsi que quelques prédicats prédéfinis.

3.1 Présentation de Prolog

Prolog, acronyme de "PROgmmation en LOGique", est un langage logique qui repose essentiellement sur les clauses de Horn.

3.2 Syntaxe

3.2.1 Constantes

En Prolog, toute chaîne de caractères qui commence par une minuscule est une constante.

Exemple 3.2.1. ali, 5 sont des constantes.

Remarque 3.2.1. Une chaîne de caractères entre cotes ' ' est une constante même si elle commence par une majuscule.

Exemple 3.2.2. 'ali', 'CHAINE' et 'Chaîne' sont des constantes.

3.2.2 Variables

En Prolog, toute chaîne de caractères qui commence par une majuscule est une variable.

Exemple 3.2.3. Ali, Variables et VAR sont des variables.

Variables anonymes

Une variable est dite anonyme si elle commence par _. Les valeurs affectées à ces variables après unification importent peu. Elles sont utilisées pour des variables dont on ne désire pas connaître les valeurs.
Par exemple quand on cherche à savoir si ali a un frère, alors il suffit de poser la question (but) suivante :

?-frere(ali,_).

Yes

On s'en faut de la valeur prise par cette variable anonyme_.

3.2.3 Prédicat

Un prédicat est une chaîne de caractère commençant par une minuscule. A ne pas confondre avec une constante. Un prédicat a des arguments qui peuvent être des constantes, des variables ou des termes composés.

Exemple 3.2.4. freres(X, ali), pere(ali, omar), jumeaux(YY, ZZ) sont des formules atomiques. freres, pere et jumeaux sont des prédicats.

3.2.4 Faits

Un fait est une clause de Horn positive (sans littéraux négatifs). Il exprime une connaissance factuelle. Sa syntaxe est illustrée par l'exemple suivant. Un fait se termine par un point.

Exemple 3.2.5. Soit le fait suivant :

homme(ali).

Cette formule exprime le fait que ali est un homme.

Exemple 3.2.6. freres(ali, youcef).

Cette formule nous renseigne que ali et youcef sont des frères.

Exemple 3.2.7. petit(X, succ(X)).

Cette formule exprime le fait que X est inférieur à son successeur (on suppose que X est un entier positif) quelque soit la valeur de X.

3.2.5 Règles

Les règles sont des clauses ayant un littéral positif et un ou plusieurs littéraux négatifs. Elles sont de la forme $A : -B_1, \cdots, B_m$. Cette formule veut dire pour que A soit satisfaite, il faut que $B_1, \cdots, B_m$ soient simultanément satisfaites. On peut aussi la lire comme suit : Si $B_1, \cdots, B_m$ sont simultanément vraies alors A est aussi vraie. Chaque règle se termine par un point.

Exemple 3.2.8. Soient les deux règles suivantes :

grand-pere(X, Y) :-pere(X,Z),pere(Z,Y).

grand-pere(X, Y) :-mere(X,Z'),pere(Z',Y).

Ces deux règles expriment la connaissance suivante :

Pour que Y soit grand père de X, il faut qu'il existe un Z tel que Z est père de X et Y est père de Z ; ou Z' est mère de X et Y est père de Z'.

Exemple 3.2.9. Soient les deux règles suivantes :

freres(X, Y) :-pere(X,Z),pere(Y,Z).

freres(X, Y) :-mere(X,H),mere(Y,H).

Ces deux règles expriment la connaissance suivante :

Pour que X et Y soient des frères, il suffit qu'ils aient même père ou même mère.

Remarque 3.2.2. On peut noter les règles des exemples 3.2.8 et 3.2.9 respectivement comme suit :

grand-pere(X, Y) :-pere(X,Z),pere(Z,Y) ;mere(X,L),pere(L,Y).

freres(X, Y) :-pere(X,Z),pere(Y,Z) ;mere(X,H),mere(Y,H).

Les virgules "," expriment la conjonction et les point-virgules " ;" exprime la disjonction.

3.2.6 Buts

Appelés parfois **questions**, les buts permettent d'exprimer une interrogation sur l'existence de données vérifiant les prédicats. La syntaxe d'un but est la suivante : ?-B. Ou plus généralement : ?-$B_1, \cdots, B_m$.

Exemple 3.2.10. ?-freres(X, Y).

Ce but vérifie l'existence de données X et Y avec X est frère de Y.

Exemple de programme PROLOG

Soit le programme suivant :

homme(ali).

homme(said).

pere(said, omar).

pere(ali, omar).

frere(X,Y) :-pere(X,Z), pere(Y, Z).

Soient les buts suivants :

?-frere(said, ali).

Prolog répond par Yes.

?-frere(M, N).

Prolog répond par M=said, N=ali, Yes.

En Prolog, si on veut avoir une autre réponse possible, on tape ;

Dans ce cas, on aura M=ali , N=said, Yes. Et si on tape encore le caractère " ;" on aura la réponse No.

3.3 Algorithme d'unification de PROLOG

Pour comprendre l'exécution d'un programme écrit en Prolog, on présente d'abord l'algorithme d'unification utilisé par l'interpréteur de PROLOG.

En prolog, l'unification est désignée par l'opérateur "=". Ainsi

?-pere(X,ali)=pere(djamila, Y).

X=djamila,

Y=ali.

Remarque 3.3.1. Quand Prolog tente d'unifier un but avec un fait ou une tête de clause, il peut s'avérer nécessaire de renommer les variables.

Exemple 3.3.1. Soit le but *?-aime(X,ali)* et soit le fait *aime(omar, X).*
Pour que ces deux expressions soient unifiables, on doit renommer les variables comme suit :

?-aime(X1,ali)=aime(omar, X2).

Soient deux expressions A et B quelconques (formules ou termes). L'algorithme 4 retourne un unificateur θ de A et B s'il existe, et retourne ECHEC sinon.

Algorithme 4 *Algorithme d'unification de Prolog*

Input : deux expressions A et B à unifier
Output : Unificateur θ de A et B.

1: $\theta = \emptyset$
2: empiler $A = B$
3: **while** pile n'est pas vide **do**
4: Dépiler l'équation $X = Y$
5: **if** X est une variable qui n'apparaît pas dans Y **then**
6: Substituer toutes les occurrences de X par Y dans la pile et dans θ
7: Ajouter $X = Y$ dans θ
8: **if** Y est une variable qui n'apparaît pas dans X **then**
9: Substituer toutes les occurrences de Y par X dans la pile et dans θ
10: Ajouter $Y = X$ dans θ
11: **if** X et Y sont des constantes ou des variables identiques **then**
12: Continuer
13: **if** X est $f(X_1, \cdots, X_n$ et Y est $f(Y_1, \cdots, Y_n$ **then**
14: Empiler les n équations $X_i = Y_i$;
15: **if** Aucun des cas précédents **then**
16: retourner échec ;
17: Retourner θ

Exemples d'unification

?-1+2=2+1.

No

?-1+2= :=2+1.

Yes

?-X=ali.

X=ali

?-ali=omar.

No

?-f(X,5)=f(10,Y).

X=10

Y=5

?-p(X,X,X)=g(X,X,X).

No

?-p(X,X,X)=p(a,a,a).

X=a

?-p(X,X,X)=p(11,11,12).

No

?-p(X,X,a)=g(b,Y,Y).

No

?-homme(ali) $\neq$ homme(farid).

Yes

Le symbole $\neq$ est utilisé pour non unification. La réponse est évidemment ***Yes*** car les deux constantes *ali* et *farid* ne sont pas identiques.

3.3.1 Extraction de l'information

Voyons maintenant comment extraire de l'information en exploitant le principe de l'unification. On va illustrer cela par l'exemple de programme Prolog suivant.

ami(omar,ali).

ami(omar, farid).

garçon(omar).

garçon(farid).

fille(djamila).

fille(zahira).

Soit maintenant le but suivant :

?-garçon(X).

X=omar ;

Si on tape le caractère " ;", Prolog affiche

X=farid ;

Si on tape encore le caractère " ;", Prolog affiche

No

Pour le traitement du but : *?-garçon(X).*, Prolog essaie d'unifier garçon(X) avec chacun des faits du programme dans l'ordre de leur apparition. Ainsi, la tentative d'unification de garçon(X) avec ami(omar,ali) échoue car on est en présence de deux prédicats différents. Puis il tente d'unifier garçon(X) avec ami(omar,farid). Le résultat est aussi un échec car les deux prédicats sont différents aussi. Puis, il tente d'unifier garçon(X) avec garçon(omar). Cette fois, la tentative réussit car les deux expressions ont le même prédicat et le même nombre d'arguments via la substitution X/omar. Dans ce cas, Prolog retourne X=omar. Si on tape " ;" (on demande une autre réponse),

Prolog tente le même principe avec le reste des faits du programme.

3.4 Exécution d'un programme Prolog

Pour évaluer un but, Prolog exécute l'algorithme 5 ci-après qui exploite une pile (appelée *résolvante*) pour sauvegarder tous les choix possibles à un moment donné.

3.4.1 Algorithme d'exécution

Algorithme 5 *Algorithme d'exécution d'un programme Prolog*

Input : La clause but à évaluer
Output : Instanciations des variables du but.

```
1:  Pile ← clause but
2:  while Pile n'est pas vide do
3:    Obj← Dépiler premier sous-but
4:    label : Chercher la première clause C du programme (fait ou règle) dont la tête
      s'unifie avec Obj.
5:    if succès then
6:      Empiler tous les sous-buts du corps de C
7:      Mémoriser la position de C dans le programme pour d'éventuels retour-
        arrières
8:    else
9:      if il existe des points de choix then
10:       Retourner au dernier point de choix
11:       Aller à label
12:     else
13:       Retourner échec
14: Retourner l'instanciation des variables du but
```

3.4.2 Points de choix

Quand il y a un retour-arrière, il faut retourner au point de choix correspondant au dernier but considéré pour lequel il restait encore d'autres possibilités d'unifications.

Exemple 3.4.1. Soit le programme Prolog suivant :

grandpere(X,Y) :-pere(X,Z),pere(Z,Y),homme(Y).

grandpere(X,Y) :-mere(X,Z),pere(Z,Y),homme(Y).

homme(ali).

pere(omar,said).

pere(said,ali).

mere(yacine,anissa).

pere(anissa,ali).

Soit le but ?-grandpere(X,Y).

L'arbre de résolution est donné par la figure 3.1. Le point de choix est au niveau du nœud 1.

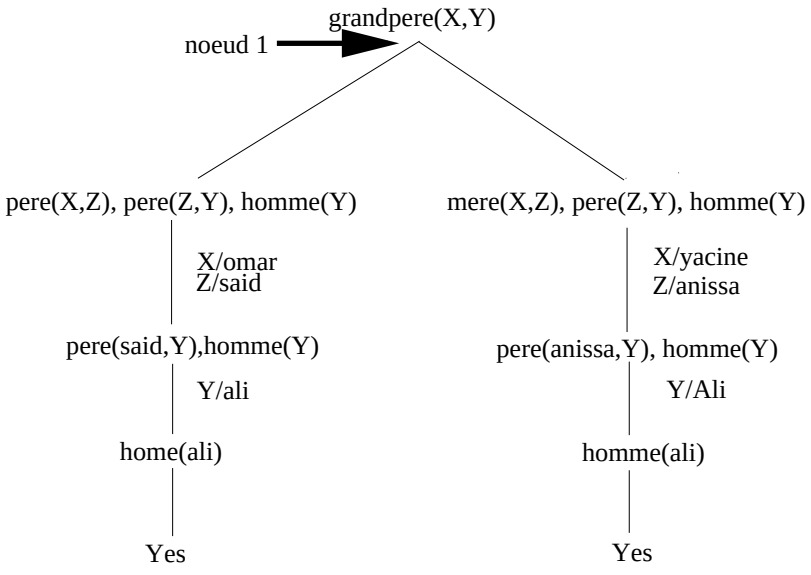

FIG. 3.1 – Arbre de résolution

3.5 Récursivité

En Prolog, la récursivité est définie par la donnée :

- du pas de la récursivité,
- de la condition d'arrêt.

En Prolog, il n'y a pas de ***loop*** et de ***for***, donc la récursivité est le seul moyen d'exprimer cette notion d'itération.

Exemple 3.5.1. Soit le programme qui calcule la factorielle d'un entier. Il est bien connu que la factorielle de n'importe quel entier n est n *factorielle de $(n-1)$. Il est évident que le pas de la récursivité dans ce cas est 1 et la condition d'arrêt est le fait que factoriel de 0 est 1. Le programme Prolog correspondant est :

$factorielle(0,1).$

$factorielle(N,L)$:-M is $N-1, factorielle(M,L1)$, L is $N*L1$.

Alors, soit le but suivant :

?-factorielle(5,N).

N=120

Il est bien évident que le but suivant échoue :

?-factorielle(X,120).

No

En effet, l'instruction : -M *is* $N-1$ tente de décrémenter une variable non instanciée, ce qui est exigé par l'opérateur *is*.

La récursivité est importante en Prolog, nous allons l'utiliser beaucoup dans la suite du cours.

3.6 Calcul en Prolog

Prolog offre les opérateurs arithmétiques suivants : $+, -, *, /, mod$.

Le prédicat is

Le prédicat ***is*** permet l'affectation d'une valeur ou d'un résultat d'une opération à une variable.

Exemple 3.6.1. ?-X *is* 5.

Yes

?-Y *is* 5+2.

Y=7

?-M *is* X+Z.

M=12

?-12 is R+10.

Error

Remarque 3.6.1. Signalons que l'expression à gauche de l'opérateur *is* ne peut pas être une expression quelconque. Elle ne peut être q'une valeur numérique.

Exemple 3.6.2. ?- X = 8 + 3, X is 7 + 4.

No

L'expression à gauche de *is* n'est pas une valeur numérique, c'est pour cela que la requête a échoué.

Le prédicat = :=

Le prédicat $=:=$ permet la comparaison de deux expressions après leur évaluation.

Les opérateurs relationnels

Les opérateurs $<, >, =<, >=$ *et* $=\backslash=$ effectuent le calcul avant de comparer les valeurs. Pour utiliser ces opérateurs, les variables doivent être *instanciées*.

Exemple 3.6.3. ?-X>2.

No

?-4>3.

Yes

?-X=3.

Yes

?-Y=5.

Yes

?-X>Y.

No

3.6.1 Calcul de toutes les solutions

Prolog permet de calculer toutes les solutions possibles à un ou plusieurs buts. Cependant, à chaque fois qu'il donne une réponse, il perd la précédente. Les prédicats *findall*, *bagof* et *setof* permettent de sauvegarder toutes les solutions dans une liste.

Le prédicat *findall*

Sa syntaxe est comme suit :

findall(Objets,But, Liste).

Ce prédicat permet de retourner la liste **Liste** de tous les objets **Objets** vérifiant But. Si le But n'a pas de solution, le prédicat *findall* retournera la liste vide : [ ].

Exemple 3.6.4. Soient les prédicats suivants :

travaille(ali, alger).

travaille(youcef, bejaia).

travaille (said, alger).

travaille(mohamed, alger).

Soit le but suivant :

?- findall(X, travaille(X,alger), Liste).

Prolog retourne Liste=[ali, said, mohamed].

Notons que *findall* retourne une liste **vide** si pas de solution.

Le prédicat *bagof*

Il se comporte comme *findall* à la différence que *bagof* échoue en cas de non solution. Aussi *bagof* diffère de *findall* quand il y a des variables libres dans But. Dans ce cas, *bagof* sépare les résultats en fonctio des valeurs de ces variables libres : Par exemple dans le but suivant : ?-*bagof*(X,repas(X,Y,melon),L).

Y est libre. En fait, il retourne tous les résultats possibles pour chaque valeur possible de la variable libre.

Le prédicat *setof*

Ce prédicat se comporte comme *bagof* mais il élimine les doublons et trie la liste des résultats.

3.7 Stratégie de recherche

La stratégie de recherche correspond au choix de la clause à appliquer pour réduire le but sélectionné par la stratégie de sélection. Pour le cas de Prolog, le choix des clauses se fait du haut vers le bas dans le programme. Cette stratégie n'est évidemment pas complète car un choix de clause donné peut mener vers une branche infinie alors qu'un autre choix mènerait vers un succès.

3.8 Stratégie de sélection

La stratégie de sélection de Prolog correspond au choix du sous-but le plus à gauche. Cette stratégie peut mener aussi à des branches infinies.

3.9 Manipulation dynamique des programmes

En Prolog, il est possible de modifier dynamiquement le programme en **ajoutant** ou en **retirant** des clauses. Dans ce qui suit, nous allons présenter deux prédicats prédéfinis *assert* et *retract* permettant de réaliser cette modification dynamique.

Le prédicat assert

Ce prédicat permet d'ajouter dynamiquement une clause au programme.

Exemple 3.9.1. Soit le programme suivant :

garçon(ali).

garçon(omar).

fille(saida).

garçon(malik).

fille(malika).

fille(leila).

Si nous voulons ajouter une clause pour dire que youcef est un garçon, on exécute la requête suivante :

assert(garçon(youcef)).

Le prédicat retract

Ce prédicat permet le retrait de la première clause du programme qui s'unifie avec son argument.

Exemple 3.9.2. retract(garçon(ali)).

Remarque 3.9.1. En SWI-Prolog, pour pouvoir exploiter assert et retract avec les prédicats définis dans le programme, il faut que les prédicats concernés soient déclarés dynamiques grâce à la commande suivante au début du programme :
:-dynamic suivie de tous les prédicats dynamiques avec leur arité (c'est-à-dire leur nombre d'arguments). Par exemple, si le prédicat freres et amis sont dynamiques et leurs arités respectives sont 2 et 3, on écrit :- dynamic freres/2, amis/3.

3.10 Trace d'exécution

Le prédicat trace(but). permet de voir tous les appels et leur résultats logiques lors de la résolution d'un but.

La syntaxe est la suivante : *?-trace.*

3.11 SWI-Prolog

Dans cette section, nous présentons quelques instructions permettant de travailler avec SWI-Prolog.

Taper ***pl*** suivie de entrée.

Pour charger le programme, taper

:-***consult***(nom_fichier).

ou

:-nom_fichier.

Si on veut entrer directement des clauses au terminal, on tape

:-***consult***(user).

ou :-user.

Puis [CTRL] D.

Pour lister l'ensemble des clauses dont la tête est le prédicat h, taper ***listing***(h).

Pour lister l'ensemble des clauses, taper ***listing***.

Pour exécuter une commande shell, taper ***sh***.

Pour retourner à Prolog, taper ***[CTRL] D***.

Il existe deux façons de faire des commentaires :

Soit en précédent le commentaire par le caractère % soit comme en C entre /*commentaire */

3.12 Exercices

Exercice N°1

a- Ecrire un programme PROLOG pour les énoncés suivants :

Ali et Omar sont des frères. Le père de Ali est Said. La mère de Omar est Nassima.

b- Ecrire une requête Prolog pour le but suivant :

Quelle est la mère de Ali.

c- Ecrire la clause Prolog pour modéliser que X et Y sont des frères s'ils ont le même père et la même mère.

d-Donner l'arbre de recherche pour le but *?-freres(X,Y).*

Exercice N°2

Ecrire les clauses Prolog pour définir les relations suivantes :

grand_parent(X,Y) (Y est la grand-mère ou le grand-père de X)

cousin(X,Y) (X et Y sont cousins germains)

Exercice N°3

Etant donnés deux entiers n_1 et n_2. Ecrire un programme Prolog qui calcule le quotient entier Q et le reste de la division de n_1 par n_2.

Exercice N°4

Quelles sont les réponses de Prolog aux buts suivants.

?-2=1+1.

?-X=2, X=1+1.

?- X is 3+4.

?-X=5, X is 3+3.

?-P(X,X)=P(a,b).

?-P(X,X)=Q(a,b).

Exercice N°5

Soit le programme PROLOG suivant :

mange(omar, pommes).

mange(omar, carottes).

mange(said, bananes).

fruit(pommes).

fruit(bananes).

legume(carottes).

sante(X) :- mange(X, Y), fruit(Y).

Donnez les réponses de PROLOG aux buts suivants :

?- sante(X).

?- mange(X, pommes).

?- fruit(X).

?- legume(carrotte).

?- legume(oignon).

Chapitre 4

Les listes et arbres en PROLOG

Dans ce chapitre, on s'intéressera aux listes et aux arbres en Prolog.

4.1 Listes

Une liste est une séquence d'objets qui peuvent être de types différents. C'est une structure de données très utilisée en PROLOG. La notation usuelle est $[a, b, \cdots]$. Par exemple $[a, 11, 3, homme, gateau]$ est une liste contenant des objets de types différents. Aussi une liste peut contenir d'autres listes, par exemple $[a, [1, 2, 3], X, Y]$. Il existe aussi une notation permettant de distinguer la tête de la liste du reste de ses éléments. Cette notation est comme suit : $[X/L]$ où X est l'élément de tête et L est le reste de la liste. Par exemple, considérons la liste $[a, b, c]$ alors X correspond à a et $L = [b, c]$. Cette notation est très utilisée en Prolog. Plus généralement, on peut isoler plusieurs éléments en tête de n'importe quelle liste. Par exemple $[1, a, 2, b/L]$ indique n'importe quelle liste commençant par les éléments $1, a, 2, b$.

4.1.1 Unification des listes

Nous allons illustrer l'unification de deux listes par une série d'exemples.

?-[a,b,c]=[X/L].

X=a.

L=[b,c]

?-[1/L]=[1,2,3].

L=[2,3]

?-[1,2,3/L]=[1,2,3].

L=[ ]

?-[a,b/L]=[1,2,3].

No

?-[a,b/L]=[a,b,3,4].

L=[3,4]

?-[Y]=[ ].

No

?-[X,Y]=[a,b,c].

No

?-[X|L]=[X,Y/L2].

L=[Y/L2]

Il est aussi possible d'imbriquer des listes.

?-$[X, Y] = [1/[2]]$

X=1

Y=2

?-$[X/L] = [1, [\,]]$

X=1

L= [[ ]]

4.1.2 Quelques opérations sur les listes

Concaténation de deux listes

La concaténation de deux listes L1 et L2 est réalisée par le prédicat **append** prédéfini en Prolog. Cette opération exploite la récursivité. Son principe est comme suit :

1. isoler la tête de la première liste L1.
2. Résoudre récursivement le prédicat pour concaténer le reste de la liste L1 avec la seconde liste L2.
3. Ajouter au début de la liste résultante l'élément qui avait été isolé à la première étape.

Le programme correspondant est comme suit :

```
concatenation([ ], L2, L2).
concatenation([X/L], L2, X/L3) :-concatenation(L,L2,L3).
```

Comme on l'a déjà signalé, il y a un prédicat *append* prédéfini en Prolog qui réalise cette opération. Ce prédicat permet différentes utilisations du fait que Prolog ne distingue pas les données des résultats. L'exemple 4.1.1 donne un aperçu sur les possibilités d'utilisation de append.

Exemple 4.1.1. Soient L1, L2 et L3 trois listes telles que L3 est le résultat de la concaténation de L1 avec L2.

?-append([1,2],[3,4,5], L3).

L3=[1,2,3,4,5]

?-append(L1,[3,4,5], [1,2,3,4,5]).

L1=[1,2]

?-append([1],L2, [1,2,3,4,5]).

L2=[2,3,4,5]

?-append(L1,L2, [1,2,3,4,5]).

Plusieurs réponses possibles

L1=[ ]

L2=[1,2,3,4,5] ;

L1=[1]

L2=[2,3,4,5] ;

L1=[1]

L2=[2,3,4,5] ;

L1=[1,2]

L2=[3,4,5] ;

L1=[1,2,3]

L2=[4,5] ;

L1=[1,2,3,4]
L2=[5] ;

L1=[1,2,3,4,5]
L2=[ ] ;

No

Remarque 4.1.1. On peut utiliser le prédicat même lorsque les deux derniers arguments ne sont pas instanciés.

?- append([a,b,c],L2,L3).

L2 = L
L3 = [a,b,c|L]

Inversion d'une liste

Le prédicat ***inverser*** suivant réussit si la deuxième liste est l'inverse de la première. Le programme Prolog correspondant est le suivant :

?- inverser([ ],[ ]).
?- inverser([X|L1],L2) :- inverser(L1,L3),append(L3,[X],L2).

Exemple 4.1.2. Voici un exemple d'utilisation du prédicat inverser.

?-inverser ([1, h, gateau], L).

L=[gateau,h,1]

?-inverser (L, [1, h, gateau]).

L=[gateau,h,1]

Recherche d'un élément dans une liste

Etant donnés un objet et une liste, l'objectif est d'écrire un programme Prolog qui vérifie si cet objet fait partie des éléments de la liste ou non.

L'idée est simple :

- vérifier si l'objet et la tête de la liste sont identiques et dans ce cas, le programme s'arrête avec un succès.
- sinon, vérifier dans le reste de la liste.

Le programme correspondant est le suivant :

?-appartient(X,[X/L]).

?-appartient(X,[_/L]) :-appartient(X,L).

Signalons qu'il existe un prédicat *member* prédéfini en Prolog pour cette opération.

Exemple 4.1.3. Soit la liste [b,c,a,d]. Le programme Prolog suivant vérifie si a appartient à cette liste.

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

Yes

Remarque 4.1.2. Le prédicat prédéfini *member* permet de lister les éléments d'une liste. L'exemple suivant illustre cette possibilité.

Exemple 4.1.4. Le programme Prolog liste les objets de la liste [b,c,a,d].

?-member(X,[b,c,a,d]).

X=b

Si on tape " ;", on aura une autre réponse

X=c

Si on tape " ;", on aura une autre réponse

X=a

Si on tape " ;", on aura une autre réponse

X=d

Si on tape " ;", on aura la réponse suivante car il n'y plus d'autres réponses possibles

X=No

Longueur d'une liste

Il s'agit de compter le nombre d'objets que contient la liste. Le programme Prolog associé est :

longueur([ ],0).

longueur([X/L],N) :-longueur(L,M), N is M+1.

Notons qu'il existe un prédicat *length* prédéfini en Prolog qui calcule la longueur de n'importe quelle liste.

Exemple 4.1.5. La requête(but) suivante calcule la longueur de la liste [b,c,a,d].

?-longueur([b,c,a,d],N).

N=4

Somme des éléments d'une liste

Le programme suivant calcule la somme des éléments d'une liste.

```
somme([ ],0).
somme ([X/L], N) :-somme(L,M), N is M+X.
```

Exemple 4.1.6. La requête(but) suivante calcule la somme des éléments de la liste [5,10,1,2].

```
?-somme([5,10,1,2],S).
S=18
```

Tri des éléments d'une liste

Plusieurs techniques sont proposées pour trier les éléments d'une liste. Les plus utilisées sont *le tri par sélection*, *le tri par fusion* et le *tri par insertion* que nous allons présenter dans ce cours.

Tri par sélection

Le tri par sélection consiste à :

- Chercher le minimum de la liste puis le déplacer vers la tête de la liste.
- Refaire le même processus avec la liste privée de son premier élément
- Le processus s'arrête quand la liste contient un seul élément.

Le programme Prolog correspondant peut être :

```
tri_Selection([ ],[ ]).
tri_Slection([X],[X]).
tri_Selection([X,Y|L],[Min|L1]) :- minimum([X,Y|L],Min),
                                   enlever(Min,[X,Y|L], L2),
```

```
                                        tri_Selection(L2,L1).
minimum([X],X).
minimum([X,Y|L], X) :- minimum([Y|L],M), X =< M.
minimum([X,Y|L], M) :- minimum([Y|L],M), X>M.
enlever(X,[ ],[ ]).
enlever(X,[X|L],L).
enlever(X,[U|L],[U|M]) :- X=U, enlever(X,L,M).
```

Un exemple d'appel de ce programme est :

```
?-tri_selection([2,3,1,4],[1,2,3,4]).
True

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

Tri par insertion

Le principe de ce tri est d'enlever le premier élément de la liste puis de trier le reste de la liste, et on insère l'élément à sa place dans la liste triée obtenue. Ceci bien sûr est réalisé récursivement.

Le programme correspondant est :

Premièrement on définit le prédicat insertion(X,L,L1) permettant d'insérer l'élément X dans L pour donner L1.

```
insertion(X,[ ],[X]).
insertion(X,[Y|L],[X,Y|L]) :- X=<Y.
```

```
insertion(X,[Y|L],[Y|L1]) :- X>Y, insertion(X,L,L1).
```

Puis on définit le prédicat tri_insertion(L,L1) où L1 est la liste L triée.

```
tri_insertion([ ],[ ]).
tri_insertion([X|L],L1) :- tri_insertion(L,L2), insertion(X,L2,L1).
```

Tri par fusion

Le principe de ce tri consiste à **diviser** les n éléments de la liste à trier en deux sous-listes de $n/2$ éléments puis de **trier** par fusion les deux sous-listes récursivement et de **fusionner** les deux sous-listes triées obtenues.

Le programme correspondant en Prolog est :

```
/*Le prédicat tri_fusion réalise le tri par fusion d'une liste */
        tri_fusion([ ],[ ]).
        tri_fusion([X],[X]).
        tri_fusion([A,B|R], S) :- diviser([A,B|R],L1,L2), tri_fusion(L1,S1),
                                  tri_fusion(L2,S2), fusionner(S1,S2,S).

    /*Diviser une liste en deux sous-listes*/
        diviser([ ],[ ],[ ]).
        diviser([A],[A],[ ]).
        diviser([A,B|R],[A|Ra],[B|Rb]) :- diviser(R,Ra,Rb).

    /*Fusionner deux listes triées*/
        fusionner(A,[ ],A).
```

```
fusionner([ ],B,B).
fusionner(A|Ra], [B|Rb], [A|M]) :- A=<B, fusionner(Ra,[B|Rb],M).
fusionner(A|Ra], [B|Rb], [B|M]) :- A>B, fusionner([A|Ra],Rb,M).
```

4.2 Arbres

Les arbres sont très importants en Prolog. Ils permettent de représenter beaucoup de données du monde réel. Leur principal avantage est la recherche efficace des données. Au niveau de chaque nœud, il y a une valeur qui est la donnée qu'on veut stocker. Pour exploiter les données stockées dans un arbre, il faut le parcourir soit en profondeur d'abord soit en largeur d'abord.

4.2.1 Arbres binaires

Un arbre binaire est un arbre où chaque nœud est une feuille ou possède exactement deux fils. Dans ce qui suit, nous allons présenter la représentation par des listes.

4.2.2 Représentation par des listes

Dans ce cas, un nœud de l'arbre est représenté par 3 éléments : le premier représente la valeur stockée ; le deuxième représente le sous-arbre gauche et le troisième représente le sous-arbre droit.

Exemple 4.2.1. L'arbre binaire de la figure 4.1 est représenté par une liste comme suit : [A,[B,[D,[ ],[ ]],[E,[ ],[ ]]],[C,[D,[ ],[ ]],[E,[ ],[ ]]]].

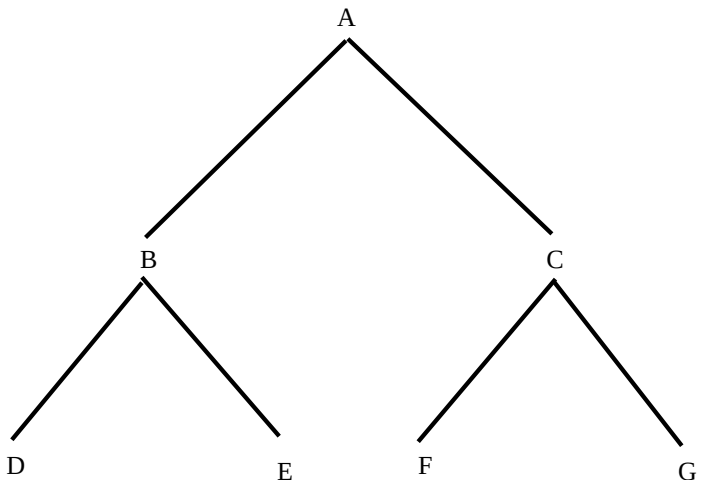

FIG. 4.1 – Arbre binaire

L'arbre binaire peut être ordonné ou non.

Arbres binaires non ordonnés

Un exemple d'opérations qu'on peut réaliser sur ces arbres sont :

- **Appartenance d'un élément à un arbre**

  appartient(X,t(_,X,_)) :- !.

  appartient(X,t(G,_,_)) :- appartient(X,G).

  appartient(X,t(_,_,D)) :- appartient(X,D).

  Signalons que : appartient(X,nil) échoue.

Arbres binaires ordonnés

Les valeurs des nœuds peuvent être ordonnées. Au niveau de chaque nœud, la valeur du nœud est supérieure à toutes celles du sous-arbre enraciné au niveau de son fils gauche et inférieure à toutes celles du sous-arbre enraciné au niveau de son fils

droit. Un exemple d'opérations qu'on peut réaliser sur ces arbres sont :

- **Appartenance d'un élément à un arbre**

  dans(X,t(_,X,_)).

  dans(X,t(G, Racine,_)) :- X< Racine, dans(X,G).

  dans(X,t(_,_,D)) :- X> Racine, dans(X,D).

- **Ajout d'éléments.**

  ajout(A, X, A1) : insérer X dans A donne A1

  ajout(nil, X, t(nil,X,nil)).

  ajout ((t(G, X, D), X, t(G, X, D)).

  ajout(t(G, R, D), X, t(Ag, R, D)) :- X<R, ajout(G, X, Ag).

  ajout(t(G, R, D), X, t(G,R,Ad)) :- X>R, ajout(D, X, Ad).

- **Suppression d'éléments.**

  suppr(t(nil, X, D), X, D).

  suppr(t(G, X, nil), X, G).

  suppr(t(G, X, D), X, t(G, Y, D1)) :- effMin(D, Y, D1).

  suppr(t(G, Racine, D), X, t(G1, Racine, D)) :- Racine> X, suppr(G, X, G1).

  suppr(t(G, Racine, D), X, t(G, Racine, D1)) :- Racine< X, suppr(D, X, D1).

  effMin(t(nil, Y, D), Y, D).

  effMin(t(G, Racine, D), Y, t(G1, Racine, D)) :- effMin(G, Y, G1).

4.3 Exercices

Exercice N°1

Quelles sont les réponses de Prolog aux buts suivants :

a/

?- [X,a|Y] = [c,a,a|K].

?- [X,a|Y] = [c,c,a|K].

?- [1|X] = [1,2,3,5].

?- [1,2|X] = [1,2].

?- [1,2|X] = [1,2,7,3,4].

?- [X] = [ ].

?- [X] = [1,2]

?- [X,Y/Z] = [1,2]

?- [[X|Y]|Z] = [[a,b,c],[d,e],[f]].

b/

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

p([je,vais,[à ,l'université]]).

p([je,bois,[une,limonade]]).

?- p([X|Y]).

?- p([_,_,[_|Y]]).

?- p([_,_,[X|Y]]).

Exercice N°2

Ecrire une programme PROLOG pour :

- calculer la longueur d'une liste
- afficher le dernier élément d'une liste
- inverser une liste d'entiers
- afficher le nième élément d'une liste
- supprimer le dernier élément d'une liste
- supprimer le n^{ieme} élément d'une liste
- concaténer deux listes d'entiers

Exercice N°3

Ecrire un programme Prolog permettant de trier une liste d'entiers de façon ascendante puis descendante.

Exercice N°4

Ecrire un programme Prolog qui vérifie si une liste donnée représente un arbre binaire.

Exercice N°5

Ecrire un programme Prolog qui calcule la somme des entiers représentés par un arbre binaire.

Chapitre 5

La coupure

Dans ce chapitre, on s'intéressera au prédicat prédéfini "cut" qui est un mécanisme de contrôle de la résolution. Il permet de contrôler l'espace de recherche, de définir la notion de négation et de réduire le coût temporel d'une requête quand l'objectif est le calcul d'une seule solution.

5.1 Coupure

La notion de coupure est introduite pour ne pas sauvegarder les points de choix en attente sur les prédicats précédant la coupure. Ceci bien sûr permet de réduire l'arbre de recherche en élaguant certaines branches. Ce qui peut conduire à une plus grande rapidité d'exécution, à une réduction de l'espace mémoire et à une modification de la sémantique du programme. Le symbole de la coupure est " !". Le prédicat de la coupure réussit toujours.

Soit la clause suivante :

$$H : -B_1, \cdots, B_i, !, B_{i+1}, \cdots, B_m.$$

Une fois le prédicat ! est exécuté, tous les choix sur $H, B_1, \cdots, B_i$ sont supprimés. Par contre, il y a possibilité d'exploiter les autres choix sur les prédicats après ! (ie.

$B_{I+1}, \cdots, B_m$).

Exemple 5.1.1. Dans cet exemple (repris de la littérature), nous allons illustrer l'utilisation de la coupure pour réduire l'arbre de recherche. Soit le programme PROLOG suivant :

idiot(X) :-idiot2(X,Y),idiot3(Y).

idiot(X) :-idiot3(X).

idiot2(a,b).

idiot2(a,c).

idiot3(b).

idiot3(a).

Soit le but *idiot(X)*.

Construire l'arbre SLD associé à ce but.

En supposant que l'utilisateur veuille toutes les solutions (mêmes multiples) au but *idiot(X)*, indiquer les branches non parcourues de l'arbre SLD pour les modifications de programme suivantes :

La clause 1 est remplacée par :

idiot(X) :-idiot2(X,Y), !,idiot3(Y).

La clause 3 est *remplacée par :*

idiot2(a,b) :- !.

La clause 5 est remplacée par :

idiot3(b) :- !.

Solution

L'arbre de résolution SLD est donné par la figure 5.1.

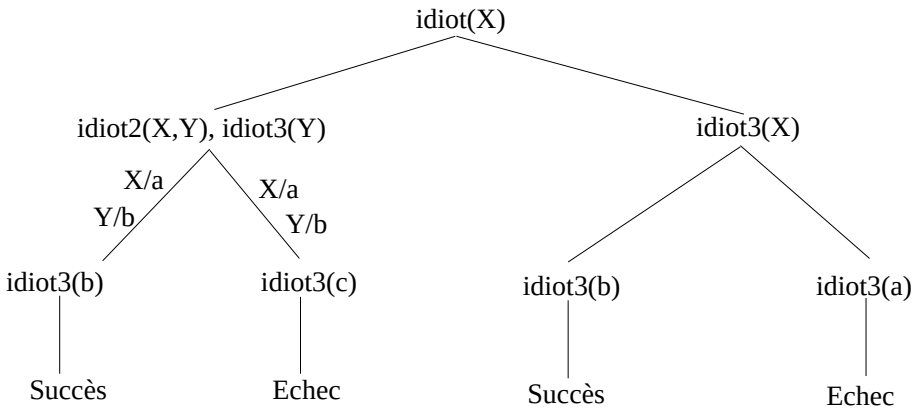

FIG. 5.1 – Arbre de résolution SLD

b- Les branches non parcourues sont les branches de la figure 5.1 qui sont absentes dans la figure 5.2.

c- Les branches non parcourues sont les branches de la figure 5.1 qui sont absentes dans la figure 5.3.

d- Les branches non parcourues sont les branches de la figure 5.1 qui sont absentes dans la figure 5.4.

Autres utilisations de la coupure

Plusieurs autres utilisations sont possibles pour la coupure. Dan ce qui suit, nous allons présenter quelques unes.

Limitation du nombre de solutions

La coupure permet de limiter le nombre de solutions.

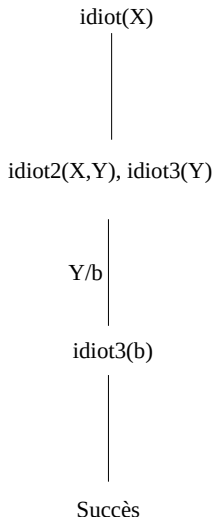

FIG. 5.2 – Arbre de résolution SLD après la modification de la clause 1.

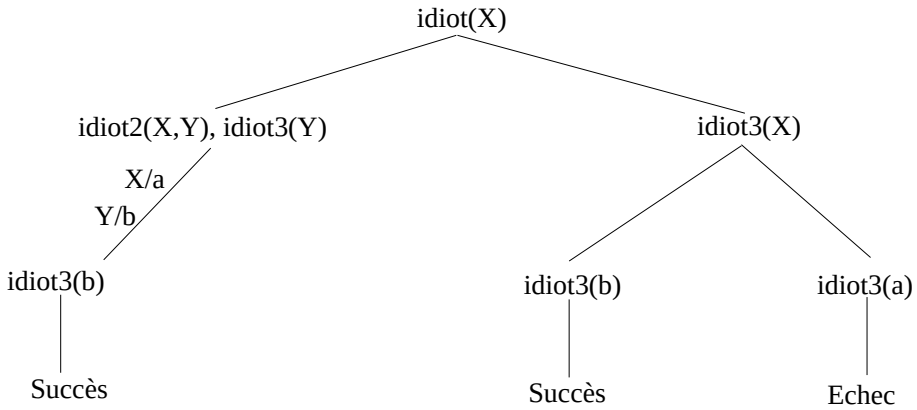

FIG. 5.3 – Arbre de résolution SLD après la modification de la clause 3.

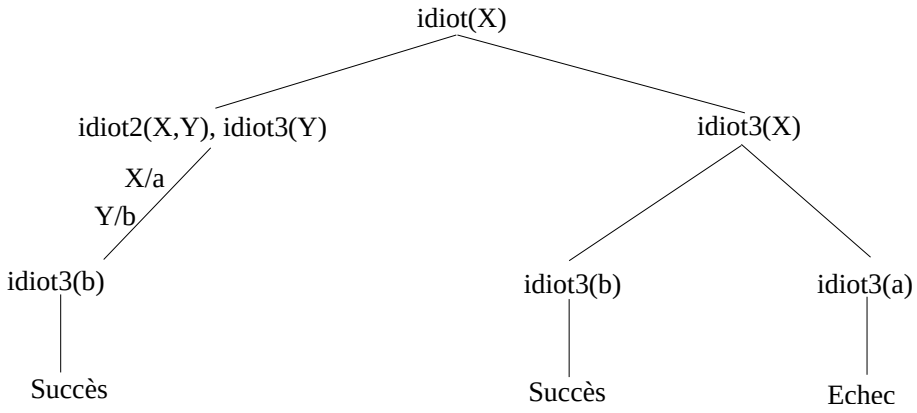

FIG. 5.4 – Arbre de résolution SLD après la modification de la clause 5.

Simulation de if then else

L'instruction if <condition> then <inst1> else <inst2> peut être réalisée comme suit en Prolog :

if(Condition, Instruction1, Instruction2) :-Condition, !,Instruction1.

if(Condition, Instruction1, Instruction2) :-Instruction2.

Exemple 5.1.2. Déclaration des valeurs d'une fonction en fonction des intervales

f(X,0) :- X $<$ 3, !.

f(X,2) :- X $<$ 6, !.

f(_,4).

Éviter les tests d'exclusion mutuelle

Exemple 5.1.3. Un être humain est soit un homme ou une femme.

humain(X) :- homme(X), !.

humain(X) :- femme(X), !.

Calcul d'une seule solution

Etant donné que PROLOG calcule toutes les solutions à un but donné, on pourra ajouter un cut à la clause but si on désire s'arrêter à la première réponse.
?-q($X_1, \cdots, X_n$), !.

5.1.1 Dangers avec la coupure

Quand on décide d'utiliser la coupure, il faut penser à tous les cas possibles d'utilisation. Notons que l'introduction ou la suppression de la coupure peut modifier la sémantique déclarative du programme.

Green cut

Dans ce cas, la sémantique déclarative du programme n'est pas modifiée. Par exemple si on considère le programme suivant :

Exemple 5.1.4. homme(X) :-homme(X), !.
homme(X) :-femme(X), !.

On voit que la coupure ne change pas la sémantique déclarative du programme. Elle sert juste à son optimisation.

Red cut

Dans ce cas, la sémantique déclarative est modifiée. Le retrait ou l'ajout du coupe-choix conduit souvent à un comportement erroné.

Exemple 5.1.5. min(X,Y,Z) :- X < Y, !, Z =X.
min(_,Y,Y).

Dans cet exemple, si on enlève le coupe-choix le comportement du programme sera différent.

5.2 Négation

L'opérateur \+ (en GNU Prolog) ou le prédicat *not* représente la négation par échec en Prolog. Le sous-but \+ A réussit si A échoue.
Notons que le but A ne peut pas être une variable libre. Par exemple, pour le but : ?- \+ X., Prolog retourne *Error*.

Exemple 5.2.1. ?- not member(3,[2,3,4])).

No

?- not member(3,[4,5,6])).

Yes

Le principe de la négation est bien résumé par le programme Prolog suivant :

not(p) :-p, !,fail.

not(p).

fail est un prédicat prédéfini qui est toujours évalué à faux.
Cela permet de définir la négation de beaucoup d'opérateurs.

Exemple 5.2.2. On peut définir un prédicat different pour spécifier que X et Y ne sont pas unifiables.

different(X,X) :- !,fail.

different(X,Y).

Remarque 5.2.1. not p(X) ne veut pas dire que p(X) est faux, mais on échoue à déduire p(X).

5.3 Exercices

Exercice N°1

Soit le programme suivant :

```
h :- m.
h :- s.
m :- u.
m :- q, v.
m :- w.
```

a- Donner l'arbre de recherche au but ?-h.

b- Donner l'arbre de recherche au but ?-h. si on modifie la clause m :-q, v. en m :-q, !, v. puis en en m :-q, v, !.

Exercice N°2

Soit le programme suivant :

```
p(1).
p(2).
d(2).
f(1).
c(2).
b(1).
b(2).
r(X) :- p(X).
r(X) :- f(X).
q(X) :- r(X), !,c(X).
```

q(X) :- b(X).

pred(X) :- q(X),f(X).

pred(X) :- d(X).

Donner l'arbre de recherche pour le but ?- pred(1).

Exercice N°3

Soit le programme PROLOG suivant :

a(X) :-b(X,Y), c(Y).

a(X) :-c(X).

b(a,b).

b(a,c).

c(b).

c(a).

a- Construire l'arbre SLD associé au but a(X)

b- En supposant que l'utilisateur veuille toutes les solutions (mêmes multiples) au but a(X), indiquer les branches non parcourues pour les modifications suivantes :

La clause 1 est remplacée par : a(X) :-b(X,Y), !,c(Y).

La clause 3 est remplacée par : b(a,b) :- !.

La clause 5 est remplacée par : c(b) :- !.

Exercice N°4

Soit le programme PROLOG suivant :

r(a).

q(b).

p(X) :- not r(X).

Donner la réponse de PROLOG au but ?-q(X),p(X).

Donner la réponse de PROLOG au but ?-p(X),q(X).

Exercice N°5

Soit le programme Prolog suivant :

member(X,[X|_]).

member(Y,[_|L]) :-member(Y,L).

a- Donner l'arbre de résolution au but ?-member (2, [1, 2, 3]).

b- Modifier le programme de telle sorte à ne garder que les branches succès dans l'arbre de la question a.

Exercice N°6

Ecrire un programme Prolog qui déclare les valeurs de la fonction g suivante :

$g(x) = 0$ si $x < 0$

$g(x) = 1$ si $0 < x < 10$

$g(x) = 2$ si $10 < x < 20$

$g(x) = 3$ si $x > 20$

Bibliographie

1. ***A. Colmerauer, An Introduction to Prolog III***

   Communications of the ACM, 33(7) :69-90.

2. ***A. Colmerauer, H. Kanoui, P.Roussel, R.pasero, Un Système de Communication Homme-Machine en Français***

   Technical report, Technical Report, Group d'Intelligence Artificielle, Marseille, (1973).

3. ***J. W. Lloyd, Foundations of Logic Programming***

   Springer-Verlag New York, Inc. New York, NY, USA ©1984 ISBN :0-387-13299-6

4. ***F. Giannesini, H. Kanoui, R. Pasero et M. Van Caneghem, Prolog***

   Inter-éditions, 1985

5. ***W.F. Clocksin, C.S. Mellish, Programming in Prolog***

   Springer-Verlag Telos ; 4th edition (September 1994)

6. ***Ulf Nilsson and Jan Maluszynski, Logic Programming and Prolog (2nd ed)***

   free e-book, 2000

7. ***Max Bramer, Logic programming with Prolog***

   springer online.com, 2005, free e-book, ISBN-13 : 978-1852-33938-8