



République Tunisienne

Ministère de l'Enseignement Supérieur, de  
la Recherche Scientifique et de la Technologie

Université de Carthage

Faculté des Sciences de Bizerte  
Département Informatique



## MEMOIRE DE MASTERE DE RECHERCHE

Parcours : Sciences de l'Informatique

Option : Informatique

présenté par

**WAJDI BLIDI**

Proposition d'un environnement pour la  
programmation logique en langue arabe

Mastère soutenu le, 10 janvier 2017

devant le jury composé de :

**Habib FATHALLAH**

Maître de conférences FSB / *président*

**Mohamed BARKAOUI**

Docteur FSB / *rapporiteur*

**Dr. Khaled BARBARIA**

Encadrant du mastère



République Tunisienne

Ministère de l'Enseignement Supérieur,  
de la Recherche Scientifique et de la Technologie

Université de Carthage

Faculté des Sciences de Bizerte  
Département Informatique

**MEMOIRE DE MASTERE  
DE RECHERCHE**

**Parcours : Sciences de l’Informatique**

**Option : Informatique**

présenté par

**WAJDI BLIDI**

**Proposition d'un environnement pour la  
programmation logique en langue arabe**

**Mastère soutenu le, 10 janvier 2017**

devant le jury composé de :

**Habib FATHALLAH**

Maître de conférences FSB / *président*

**Mohamed BARKAOUI**

Docteur FSB / *rapporiteur*

**Dr. Khaled BARBRIA**

Encadrant du mastère

**Année Universitaire 2015-2016**

# Sommaire

Introduction générale .....	8
❖ Objectif.....	8
❖ Problématique.....	8
❖ Démarche .....	8
❖ Plan du mémoire .....	9
Chapitre1. Étude de l'état de l'art.....	10
I. La langue arabe.....	10
II. Langages de programmation .....	11
1. Les générations des langages de programmation.....	12
2. Les paradigmes des langages de programmation.....	13
III. Langages de programmation en langue arabe .....	15
Conclusion .....	17
Chapitre2. Le langage Prolog.....	18
I. Syntaxe du langage Prolog .....	18
II. Programmation logique en Prolog .....	20
1. Faits, règles et requêtes.....	20
2. Coupure et Négation.....	22
3. Opérations.....	22
4. Listes.....	24
III. Approches possibles pour développer un environnement de programmation logique en langue arabe .....	26
1. L'interface JPL de SWI-Prolog.....	27
2. L'expert system shell JESS.....	28
3. Implémentation de l'algorithme d'unification .....	29
Conclusion .....	30
Chapitre3. Définition de base de connaissances et unification des expressions logiques	31
I. La logique des prédictats.....	31
II. Description semi-formelle de l'algorithme d'unification .....	33
III. Conception de l'algorithme d'unification .....	37
1. L'unification élémentaire.....	37
2. Modélisation de la base de connaissances .....	40
3. Moteur d'inférences .....	43
Conclusion .....	48

Chapitre4. Évaluation des expressions arithmétiques.....	49
I. Les opérateurs.....	49
II. Démarche pour l'évaluation des expressions .....	51
1. Conversion d'une expression en notation infixe vers sa notation postfixe .....	51
2. Évaluation des expressions en notation postfixe.....	55
III. Exécution des requêtes.....	58
Conclusion .....	59
Chapitre5. Mujeed : environnement de programmation logique en langue arabe.....	60
I. Le Front-end .....	60
1. Le fichier ArabicProlog.g4 .....	61
2. Le compilateur .....	63
II. L'environnement de programmation logique : Mujeed .....	65
1. L'interface graphique .....	65
2. Fonctionnalités de Mujeed.....	66
III. Résultats et tests.....	68
1. Enigme d'Einstein (version arabe).....	69
2. Évaluation des performances .....	71
Conclusion .....	76
Conclusion générale et perspectives .....	77
Références .....	79
Annexe .....	82

## Liste des figures

Figure 1.1	<i>Algorithme de Ada Lovelace _ la note G : calcul de nombres de Bernoulli</i> ....	11
Figure 1.2	<i>Taxinomie de principaux paradigmes de programmation (Peter Van Roy)</i> ....	13
Figure 2.1	<i>Arbres relatifs aux variables X et Y</i> .....	24
Figure 2.2	<i>Diagramme d'activité : Utilisation de l'interface JPL de SWI-Prolog</i> .....	27
Figure 2.3	<i>Diagramme d'activité : Implémentation de l'algorithme d'unification</i> .....	29
Figure 3.1	<i>Graphe And/Or</i> .....	34
Figure 3.2	<i>Arbre de preuve</i> .....	36
Figure 3.3	<i>Diagramme des classes : l'unification élémentaire</i> .....	37
Figure 3.4	<i>La classe SubstitutionSet</i> .....	38
Figure 3.5	<i>Diagramme des classes : modélisation de nœuds opérateurs</i> .....	41
Figure 3.6	<i>Diagramme des classes : modélisation de base de connaissances</i> .....	42
Figure 3.7	<i>La démarche pour construire l'arbre de preuve</i> .....	43
Figure 3.8	<i>Exemple de construction de l'abre de preuve</i> .....	44
Figure 3.9	<i>Diagramme des classes : le moteur d'inférences</i> .....	45
Figure 3.10	<i>Exemple de construction de graphe And/Or</i> .....	48
Figure 4.1	<i>Diagramme des classes _ les nœuds des opérateurs supplémentaires</i> .....	50
Figure 4.2	<i>Exemple de construction de graphe de déductions</i> .....	51
Figure 4.3	<i>Diagramme des classes : l'évaluation des expressions</i> .....	56
Figure 5.1	<i>Le compilateur généré par le framwork ANTLR</i> .....	61
Figure 5.2	<i>Diagramme d'état transition : la compilation</i> .....	63
Figure 5.3	<i>La classe ArabicPrologListener</i> .....	64
Figure 5.4	<i>Capture-écran : interface Mujeed</i> .....	66
Figure 5.5	<i>Le menu fichier</i> .....	66
Figure 5.6	<i>Le menu edition</i> .....	67
Figure 5.7	<i>Le menu paramères</i> .....	67
Figure 5.8	<i>Le menu compiler et exécuter</i> .....	67
Figure 5.9	<i>Le menu aide</i> .....	68
Figure 5.10	<i>Courbe : prédicat sum</i> .....	72
Figure 5.11	<i>Courbe : prédicat quickSort</i> .....	73
Figure 5.12	<i>Courbe : prédicat insertionSort</i> .....	74
Figure 5.13	<i>Courbe : prédicat permutation</i> .....	75

## Liste des tableaux

Tableau 1.1	<i>Implémentations Prolog</i>	15
Tableau 1.2	<i>Les langages de programmation en langue arabe</i>	16
Tableau 2.1	<i>Spécificateur d'opérateurs</i>	19
Tableau 2.2	<i>Priorités et Spécificateurs des opérateurs dans le standard Prolog</i>	19
Tableau 2.3	<i>Base de connaissances : exemple d'une famille</i>	20
Tableau 2.4	<i>Le prédictat fact</i>	23
Tableau 2.5	<i>Les opérateurs de comparaisons</i>	23
Tableau 2.6	<i>Le prédictat member</i>	25
Tableau 2.7	<i>Le prédictat reverse</i>	25
Tableau 2.8	<i>Le prédictat lenght</i>	25
Tableau 2.9	<i>Le prédictat intersect</i>	26
Tableau 2.10	<i>Le prédictat quicksort</i>	26
Tableau 2.11	<i>Table d'équivalences des caractères arabe/latin</i>	28
Tableau 3.1	<i>Opérateurs en logique de prédictats et en Prolog</i>	33
Tableau 3.2	<i>Trace d'exécution : application de la fonction unify</i>	36
Tableau 4.1	<i>Opérateurs supplémentaires</i>	49
Tableau 4.2	<i>Opérateurs arithmétique, relationnels et logiques</i>	52
Tableau 4.3	<i>Trace d'execution : application de la méthode infixeToPostfix</i>	54
Tableau 4.4	<i>Le prédictat sum en ISO Prolog, Prolog en langue arabe, Back-end</i>	58
Tableau 5.1	<i>Syntaxe prolog en langue arabe</i>	62
Tableau 5.2	<i>Exemples de prédictats réalisés</i>	68
Tableau 5.3	<i>Définition du prédictat sum</i>	71
Tableau 5.4	<i>Tableau de mesures : prédictat sum</i>	72
Tableau 5.5	<i>Définition du prédictat quickSort</i>	73
Tableau 5.6	<i>Tableau de mesures : prédictat quickSort</i>	73
Tableau 5.7	<i>Définition du prédictat insertionSort</i>	74
Tableau 5.8	<i>Tableau de mesures : prédictat insertionSort</i>	74
Tableau 5.9	<i>Définition du prédictat permutation</i>	75
Tableau 5.10	<i>Tableau de mesures : prédictat permutation</i>	75

## Introduction générale

---

Prolog est un langage de programmation qui s'inclut dans le paradigme de programmation déclarative et plus précisément, la programmation logique. Prolog est utilisé dans divers domaines, principalement, dans la modélisation et la manipulation de connaissance [1]. Plusieurs implémentations du langage Prolog ont été proposées. Malheureusement, toutes les implémentations que nous avons identifiées sont à base de l'alphabet latin de sorte qu'il n'existe aucune implémentation utilisant l'alphabet chinois (sinogrammes), l'alphabet russe (cyrillique), l'alphabet arabe (abjadie) ou autres alphabets.

### ❖ Objectif

Notre objectif est de développer un environnement de programmation logique (Prolog) en longue arabe. Les domaines d'utilisation potentiels de Prolog en langue arabe tel que le traitement automatique du langage naturel de la langue arabe (analyse syntaxique et sémantique de documents, web sémantique ...) ainsi les sciences islamiques marquent l'importance de ce projet.

### ❖ Problématique

Notre application est en langue arabe qui utilise des caractères arabes et qui s'écrit de droite à gauche. Étant donné que ISO prolog ne convient pas à la programmation logique en longue arabe, il est nécessaire de proposer une nouvelle syntaxe pour Prolog en langue arabe qui s'adapte aux spécificités de l'écriture arabe.

En plus nous sommes amenés dans un premier temps, à implémenter l'algorithme d'unification qui représente le noyau de Prolog. Dans un deuxième temps développer la partie frontale qui joue le rôle d'une interface entre l'utilisateur et l'algorithme d'unification.

### ❖ Démarche

Afin de pouvoir développer une telle application, il faut :

- Avoir une connaissance profonde sur la conception de langages de programmation, le langage Prolog ainsi que ses fondements mathématiques afin de savoir comment le moteur d'inférences fonctionne pour répondre à une requête.
- Proposer une implémentation de l'algorithme d'unification qui représente le noyau de notre application en langage de programmation Java : passage du paradigme orienté-objet vers le paradigme logique.
- Développer un composant pour l'évaluation des expressions arithmétiques et y intégrer dans l'application.

- Développer une interface graphique interactive qui offre toutes les fonctionnalités d'un interpréteur basé sur le paradigme de la programmation logique.

### ❖ Plan du mémoire

La suite de ce mémoire est organisée comme suit :

Le premier chapitre présente une étude de l'état de l'art dans laquelle nous présentons la langue arabe, les langages de programmation d'une manière générale ainsi les langages de programmation en langue arabe.

Le deuxième chapitre est réservé au langage Prolog. Nous commençons par présenter la syntaxe du langage Prolog. Ensuite, la programmation en Prolog ainsi que les approches possibles pour développer un environnement de programmation logique en langue arabe.

Le troisième chapitre présente une proposition d'une implémentation Java de l'algorithme d'unification. Nous introduisons les fondements mathématiques de la programmation logique (la logique de prédicats). Ensuite, nous décrivons l'algorithme d'unification. Finalement, nous présentons la conception de notre système.

Le quatrième chapitre présente une continuation du chapitre précédent dont nous avons étendu l'algorithme d'unification pour qu'il puisse réaliser quelques opérations sur des termes supplémentaires et évaluer les expressions arithmétiques.

Le cinquième chapitre est consacré à la partie frontale de notre interpréteur (front-end) : la couche permettant de traduire une expression écrite en respectant la syntaxe de Prolog en langue arabe vers les objets Java qui lui sont équivalents ainsi une description de l'interface graphique de Mujeed : l'environnement de programmation logique en langue arabe. Finalement nous présentons une étude expérimentale contenant une comparaison de Mujeed avec autres implémentations Prolog.

# Chapitre1. Étude de l'état de l'art

---

Dans la première partie de ce chapitre nous présentons la langue arabe : quelques statistiques, l'écriture et la grammaire arabe ainsi sa contribution scientifique en algorithmique. La deuxième partie est réservée aux langages de programmation : nous nous intéressons tout d'abord à l'histoire des langages de programmation, nous présentons ensuite, les paradigmes. Finalement, nous présentons les langages de programmation développés en langue arabe.

## I. La langue arabe

La langue arabe représente l'une des principales langues au monde. Selon « Ethnologue, Languages of the World » [2] la langue arabe a gagné une place dans le classement des langues les plus parlées au monde en passant de la cinquième place en 2015 à la quatrième place en 2016 avec 267 millions de locuteurs.

Le 18 décembre 1973, la langue arabe a été adopté comme l'une des six langues officielles des Nations Unies, rejoignant le chinois, l'anglais, le français, le russe et l'espagnol ceci est une conséquence directe de son importance mondiale : en fait, elle est la langue maternelle du monde arabe, majoritairement utilisée en République centrafricaine du Tchad, pays non-arabe, et de façon minoritaire dans plusieurs autres pays, dont l'Afghanistan, l'Iran et le Nigeria aussi plus d'un milliard de musulmans dans des pays comme l'Inde, l'Indonésie, le Pakistan et la Tanzanie étudient l'arabe comme langue étrangère principale ou secondaire à des fins religieuses [3].

La langue arabe s'écrit de droite à gauche au moyen de 28 lettres dont trois voyelles : 'alif, wāw et yā' correspondent aux voyelles longues. Cette langue prévoit l'utilisation des diacritiques correspondant aux voyelles courtes. Les diacritiques sont des signes placés sur ou sous la consonne qui modifient la prononciation de cette dernière, elles ne sont généralement pas écrite mais plutôt déduite par le lecteur. Les diacritiques sont la fatha : une petite barre oblique au-dessus de la lettre sa valeur phonétique est /a/, la kasra une petite barre oblique au-dessous de la lettre, sa valeur phonétique est /i/ et la damma un petit wāw miniature au-dessus de la lettre sa valeur phonétique est /u/ en plus de tanwin c'est le redoublement de la vocalisation, le sukun indique qu'une consonne n'est pas suivie par une voyelle, la chadda renforce la prononciation d'une consonne et deux autres diacritiques appropriés à la lettre 'alif: wasla et madda [4].

Il est à noter que les lettres arabes peuvent prendre 4 formes différentes en fonction de leur position dans le mot : début, fin, milieu ou bien isolée. En pratique, dans la plupart des cas les formes début et milieu sont identiques. Plusieurs encodages informatiques supportent l'alphabet arabe parmi lesquels ISO-8859-6 et Unicode.

La langue arabe, comme toutes les langues sémitiques, se caractérise par l'utilisation des schèmes (modèles de formation des mots) permettant d'obtenir des mots à partir de racines abstraites, représentant des notions sémantiques générales ou des significations précises. Ces racines se composent généralement de trois consonnes qui constituent les unités de base pour la formation de nombreux mots dérivés de cette racine. En modifiant les consonnes «de base» des racines, en utilisant diverses combinaisons de voyelles, et en utilisant des préfixes et des suffixes différents, de nombreuses possibilités existent pour dériver des noms, des adjectifs et des adverbes à partir d'une racine donnée ce qui donne une langue très riche en vocabulaire.

La langue arabe a contribué dans divers domaines scientifiques tels que la philosophie, les mathématiques, la médecine, la géographie... .En relation avec l'informatique, Al-Khwarizmi a posé les fondations de l'algorithme. D'ailleurs le mot algorithme est issu du nom latinisé d'Al-Khwarizmi. Ce dernier a étudié la résolution des équations du premier et du second degré dans son ouvrage « L'abrégé du calcul par la restauration et la comparaison » (المختصر في حساب الجبر والمقابلة [5]. Un autre ouvrage, « Livre de l'addition et de la soustraction d'après le calcul indien » (الجمع والنفرق في الحساب الهندي) décrit le système de numération décimale [6].

## II. Langages de programmation

Bien que les premiers langages de programmation n'apparaissent qu'autour de 1950, la notion d'algorithme est beaucoup plus ancienne. Cette notion est, en effet, historiquement liée aux manipulations arithmétiques (comme il est mentionné dans la section précédente). Aujourd'hui, on désigne par algorithme: une description précise d'un processus pour résoudre un problème.

Le premier algorithme a apparu avec le premier ordinateur numérique « analytical engine » du mathématicien anglais Charles Babbage en 1834. Impressionné par le travail rigoureux d'une jeune femme nommée Ada Lovelace qu'elle a passé neuf mois entre 1842 et 1843 à traduire l'article du mathématicien italien Luigi Menabrea [7] décrivant la machine analytique, Babbage lui a demandé d'augmenter la traduction avec ses propres notes. Elle a ajouté à la mémoire sept notes labélisées de A à G [8]. La note G mentionne un véritable algorithme pour calculer les nombres de Bernoulli (Figure 1.1) destiné à être exécuté sur la machine ce qui fait d'Ada le premier programmeur au monde. Le langage Ada a été ainsi nommé en hommage à cette femme.

Number of Operations. Name of Operation.	Variables used.	Variables receiving results.	Indication of change in value on my Variable.	Statement of Results.	Working Variables.								Result Variables.			
					V <sub>1</sub>	V <sub>2</sub>	V <sub>3</sub>	V <sub>4</sub>	V <sub>5</sub>	V <sub>6</sub>	V <sub>7</sub>	V <sub>8</sub>	V <sub>9</sub>	V <sub>10</sub>	V <sub>11</sub>	
1	$\times V_{12} \times V_{13}$	$V_{12}, V_{13}, V_{14}$	$\{V_{12} = V_{13}\}$	$= 2n$	—	2	n	2n	2n	2n	—	—	—	—	—	—
2	$- V_1 - V_2$	$V_1, V_2$	$\{V_1 = V_2\}$	$= 2 - 1$	—	—	—	—	—	—	—	—	—	—	—	—
3	$+ V_3 + V_4$	$V_3, V_4$	$\{V_3 = V_4\}$	$= 2 + 1$	1	—	—	—	—	—	—	—	—	—	—	—
4	$- V_5 - V_6$	$V_5, V_6$	$\{V_5 = V_6\}$	$= 2 - 1$	—	—	0	0	—	—	—	—	—	—	—	—
5	$- V_{12} + V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 2 - \frac{2 - 1}{2}$	—	2	—	—	—	—	—	—	—	—	—	—
6	$- V_{12} - V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 1 - \frac{2 - 1}{2} + A_2$	—	—	—	—	—	—	—	—	—	—	—	—
7	$- V_8 - V_9$	$V_8, V_9$	$\{V_8 = V_9\}$	$= - 1 - (2)$	1	—	n	—	—	—	—	—	—	—	—	—
8	$+ V_5 + V_6$	$V_5, V_6$	$\{V_5 = V_6\}$	$= 2 + 0 = 2$	—	2	—	—	—	—	—	—	—	—	—	—
9	$+ V_{12} + V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 2 - A_1$	—	—	—	—	2n	2	—	—	—	—	—	—
10	$\times V_{12} \times V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 1, \frac{2}{2} = A_1$	—	—	—	—	—	—	—	—	—	—	—	—
11	$+ V_{12} + V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 1 - \frac{2}{2} + 1 + B_1 - \frac{2a}{2}$	—	—	—	—	—	—	—	—	—	—	—	—
12	$- V_{12} - V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= - 2 - (2)$	1	—	—	—	—	—	—	—	—	—	—	—
13	$- V_4 - V_5$	$V_4, V_5$	$\{V_4 = V_5\}$	$= 2n - 1$	1	—	—	—	—	2n - 1	—	—	—	—	—	—
14	$+ V_6 + V_7$	$V_6, V_7$	$\{V_6 = V_7\}$	$= 2 + 1 = 3$	—	—	—	—	—	2	—	—	—	—	—	—
15	$- V_8 - V_9$	$V_8, V_9$	$\{V_8 = V_9\}$	$= 2 - 1$	—	—	—	—	2n - 1	2	—	—	—	—	—	—
16	$\times V_{12} \times V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= \frac{2}{2} = 1$	—	—	—	—	—	—	—	—	—	—	—	—
17	$- V_{12} - V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= 2 - 2$	—	—	—	—	2n - 2	—	—	—	—	—	—	—
18	$+ V_1 + V_2$	$V_1, V_2$	$\{V_1 = V_2\}$	$= 2 + 1 = 4$	—	—	—	—	—	4	—	—	—	—	—	—
19	$- V_3 - V_4$	$V_3, V_4$	$\{V_3 = V_4\}$	$= 2 - 2$	—	—	—	—	2n - 2	4	—	—	—	—	—	—
20	$\times V_{12} \times V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= \frac{2}{2} = 1 - \frac{2 - 2}{2} = A_2$	—	—	—	—	—	—	0	—	—	—	—	—
21	$\times V_{12} \times V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= B_1 - \frac{2}{2} = 1 - \frac{2 - 2}{2} - B_1 A_2$	—	—	—	—	—	—	—	0	—	—	—	—
22	$+ V_{12} + V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= A_2 + B_1 A_1 + B_2 A_2$	—	—	—	—	—	—	—	0	—	—	—	—
23	$- V_{12} - V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= - 2 - (2)$	1	—	—	—	—	—	—	—	—	—	—	—
24	$+ V_{12} + V_{13}$	$V_{12}, V_{13}$	$\{V_{12} = V_{13}\}$	$= B_2$	—	—	—	—	—	—	—	—	—	—	—	—
25	$+ V_1 + V_2$	$V_1, V_2$	$\{V_1 = V_2\}$	$= 1 + 1 = 4 + 1 = 5$	1	—	n + 1	—	—	0	0	—	—	—	—	—

Figure 1.1 Algorithme de Ada Lovelace \_ la note G : calcul de nombres de Bernoulli

En 1936, le mathématicien anglais Alan Mathison Turing a publié l'article fondateur de la science informatique « On Computable Numbers with an Application to the Entscheidungsproblem » [9]. Il y présente sa machine de Turing, le premier calculateur universel programmable. Il invente les concepts et les termes de programmation et de programme. Cet événement représente le coup d'envoi à la création des langages de programmation.

Les langages de programmations ont devenus très nombreux, une statistique publiée par History of Programming Languages [10] (un ensemble de conférences qui s'intéressent à l'histoire des langages de programmation ayant eu lieu en 1978, 1993 et 2007) a révélé qu'il existe plus que 8500 langages de programmation dont plus que le tiers sont développés dans des pays non anglophones (Etats Unis, Royaume Uni, Canada et Australie). D'autres langages utilisent des mots clés anglais ont été développés dans des pays non anglophone tels que Python développé en Pays-Bas, Ruby au Japon, Lua au Brésil, Caml en France.

## 1. Les générations des langages de programmation

On distingue cinq générations de langages de programmation [11][12] :

La première est celle des langages machine ou code machine. Le langage machine est le langage natif d'un processeur. Il s'agit d'une suite de bits interprétée par le processeur lors de l'exécution d'un programme. Ces langages ne sont ni compilés, ni assemblés mais directement entrés par le programmeur. Ce dernier doit donc connaître parfaitement le processeur. Aujourd'hui, le code machine est généré automatiquement par le compilateur ou par l'interpréteur d'un bytecode.

La deuxième génération est le langage assembleur. C'est un langage considérer de bas niveau parce que le code reste trop proche du processeur. Le code est néanmoins plus lisible et compréhensible. Les langages assembleur utilisent des « mnémoniques » qui sont des abréviations alphabétiques faciles à retenir et qui remplacent des portions du code. De plus, les langages assembleur utilisent la notion de pointeur de sorte que le programmeur soit capable de manipuler la mémoire. Un programme spécifique nommé assembleur traduit le code en langage machine. L'avantage majeur des langages d'assemblage est la production de programmes efficaces, utilisant moins de mémoire que les langages de générations supérieures et s'exécutent très rapidement de sorte que ces langages restent utilisés dans le cadre d'optimisations.

Les langages de troisième génération, développés autour de 1960, proposent une syntaxe proche des langues naturelles (très souvent de l'anglais) et offrent un niveau d'abstraction élevé d'où leurs appellation langages de haut niveau. Les langages de haut niveau ont permis un gain énorme en lisibilité et en productivité. Ils ne dépendent plus du processeur, comme c'est le cas de la première et la deuxième génération, mais d'un compilateur spécifique à l'architecture du processeur. L'idée de portabilité des programmes était lancée. On trouve dans cette catégorie les langages de programmation les plus populaires tels que fortran, Algol, Cobol, C, Java...

Les langages de quatrième génération, sont aussi des langages de haut niveau. Ce type de langage de programmation est conçu pour un travail spécifique : gestion de base de données (SQL), production graphique (Postscript, Metafont), calcul numérique (Matlab, Maple)... Le but de cette catégorie était de permettre à des développeurs non experts en informatique de développer leurs propres applications.

Les langages de cinquième génération, sont les langages de programmation déclarative destinés à résoudre des problèmes à l'aide de contraintes, et non d'algorithmes écrits. Ces langages reposent beaucoup sur la logique. Ils sont particulièrement utilisés dans le cadre d'études sur l'intelligence artificielle. Parmi les plus connus, on trouve LISP, Prolog et Haskell. L'approche de la programmation par contraintes pour résoudre des problèmes combinatoires consiste à les modéliser par un ensemble de variables prenant leur valeur dans un ensemble fini et lié par un ensemble de contraintes mathématiques ou symboliques. Eugene Freuder a défini la programmation comme suite: « Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it. » (La programmation par contraintes représente une des avancées que l'informatique ait jamais réalisée qui se rapproche le plus du Saint Graal de la programmation : l'utilisateur définit le problème, l'ordinateur le résout.)

## 2. Les paradigmes des langages de programmation

Un paradigme est une manière de programmer un ordinateur basé sur un ensemble de principes ou sur une théorie. Certains types de problèmes se traitent plus facilement selon un certain paradigme. En conséquence, le choix d'un langage doit se faire selon la nature du problème à appréhender. Il existe de nombreux paradigmes de programmation, la figure ci-dessous présente la taxonomie des principaux paradigmes selon Peter Van Roy [13]:

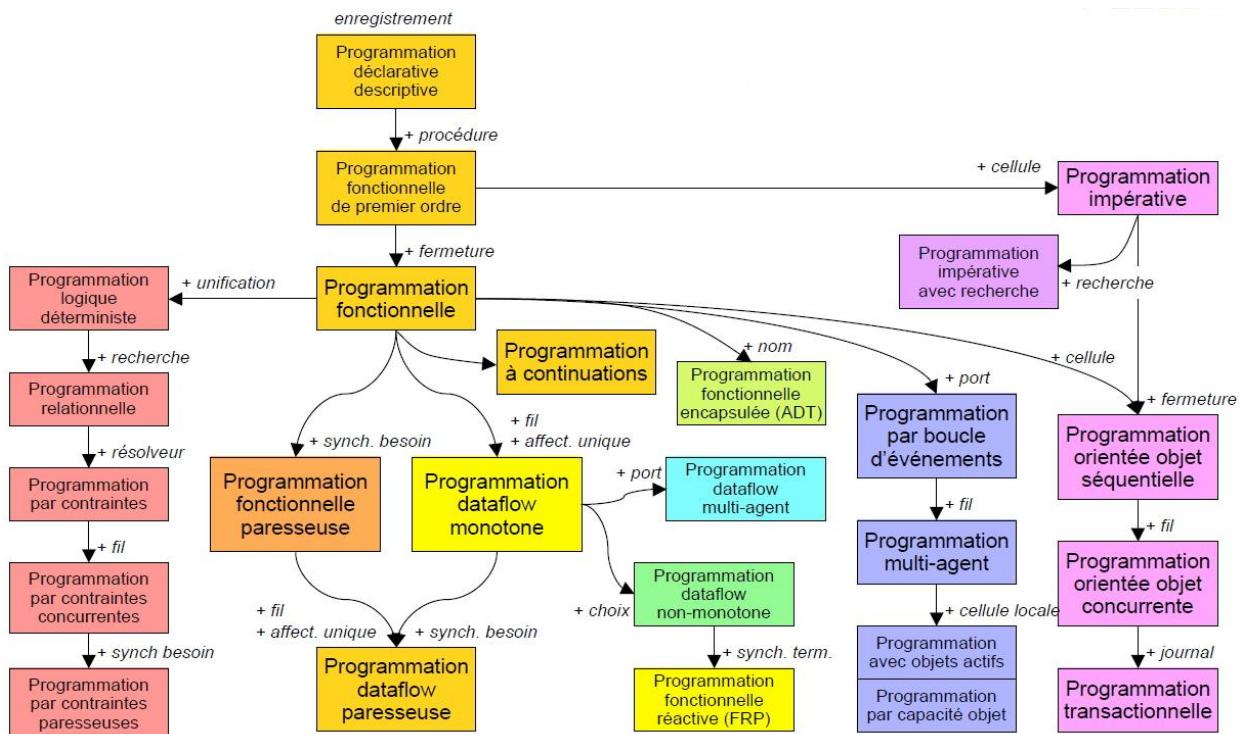


Figure 1.2 Taxinomie de principaux paradigmes de programmation (Peter Van Roy)

Chaque paradigme regroupant un ensemble de langages. Il priviliege un ensemble particulier de stratégies d'analyse et de description. Dans la figure 1.2 les paradigmes à droite sont plus impératifs alors que les paradigmes à gauche sont plus déclaratifs.

Le paradigme impératif, le paradigme orienté-objet, le paradigme fonctionnel et le paradigme logique sont, généralement, considérés comme les principaux paradigmes de programmation.

Le premier paradigme est le paradigme impératif qui décrit les opérations en termes d'état du programme et de séquences d'instructions exécutées par l'ordinateur. Une seule instruction est exécutée par le processeur à la fois. Les données sont examinées puis modifiées. Ce type de programme est conceptuellement proche de l'architecture de la machine de Von Neumann.

La plupart des langages qui sont des descendants de Fortran (Basic, Pascal, C, ...) supportent ce paradigme. La programmation procédurale est impérative, mais elle ajoute des procédures réutilisables.

Le deuxième paradigme est le paradigme orienté-objet. Il est moins fondamental : considéré parfois comme une extension du paradigme impératif. Il est très souvent évoqué et utilisé. Cette approche se concentre sur le concept de l'objet. Ce dernier possède un ensemble de propriétés (ou variables d'instance) et d'un comportement: un ensemble de réactions (ou méthodes) qui peuvent être utilisées par d'autres objets. L'interaction entre les objets via leurs relations permet de concevoir et réaliser les fonctionnalités attendues, de mieux résoudre le ou les problèmes. Plusieurs langages sont inclus dans ce paradigme tels que Java, C#, VB.NET, Python, C++, PHP...

Le troisième paradigme est le paradigme fonctionnel. Dans ce paradigme, le langage de programmation décrit un enchaînement de transformations sur un état initial et produisant un état final. Comme le nom l'indique, cette approche se concentre sur le concept de fonction. On y trouve fréquemment des fonctions qui reçoivent en paramètre des fonctions ou qui retournent des fonctions. Les principaux représentants de ce paradigme sont Lisp, Scheme, Erlang et Haskell.

Le quatrième paradigme est le paradigme logique. La programmation logique est une forme de programmation déclarative reposant sur la logique mathématique. Un programme logique est une description d'un problème suffisamment complète pour que la solution puisse en être déduite. Il est composé d'un ensemble d'axiomes (faits et règles). Un programme est exécuté lorsqu'on pose une question visant à prouver qu'un énoncé peut être déduit à partir de ces axiomes. L'objet de base est la règle qui décrit une relation entre un certain nombre d'individus. Prolog est le représentant le plus connu des langages de programmation logique grâce à ses caractéristiques : la simplicité syntaxique et la puissance du principe d'unification.

Puisque notre projet consiste à développer un environnement pour la programmation logique le tableau ci-dessous présente une étude comparative entre les principaux représentants des implémentations Prolog [14][15] (la dernière ligne représente notre solution : Mujeed (مُجید)) :

Nom	Licence	SE	GUI	Unicode	Code compilé	Interface	Syntaxe
BProlog	Freeware à des fins non commerciales	Unix Windows Mac OS	Non	Oui	Oui	C Java	ISO-Prolog
JIProlog	Shareware Commerciale AGPL	Android	Oui	Oui	Non	Java	ISO-Prolog
Ciao	GPL LGPL	Unix Windows Mac OS	Non	Non	Oui	C Java	ISO-Prolog
DOS-prolog	Shareware	MS-DOS	Oui	Oui	Oui	-	Edinburgh Prolog

ECLiPSe	MPL	Unix Windows Solaris Mac OS	Non	Non	Oui	C Java	Extended Prolog Multi-dialect ISO-Prolog
GNU Prolog	GPL LGPL	Unix Windows Mac OS	Non	Non	Oui	C	ISO-Prolog
SICStus Prolog	Commerciale	Unix Linux Windows Mac OS	Oui	Oui	Oui	C Java	ISO-Prolog
SWI-Prolog	LGPL	Unix Linux Windows Mac OS	Oui	Oui	Oui	C Java	Edinburgh, ISO-Prolog
tuProlog	LGPL	JVM Android	Oui	Oui	Non	C Java	ISO-Prolog
YAP-Prolog	GPL	Unix Windows Solaris Mac OS	Non	Oui	Oui	C Java	Edinburgh, ISO-Prolog, Quintus, SICStus Prolog compatible
Mujeed (مجید)	Freeware	Unix Windows	Oui	Oui	Oui	-	Arabic Prolog

Tableau 1.1 *Implémentations Prolog*

### III. Langages de programmation en langue arabe

Malgré la richesse de la langue arabe et sa contribution scientifique en mathématiques et algorithmique, nous remarquons qu'il y a très peu de langages de programmation en arabe. En plus la majorité de ces langages ont disparu. Le tableau suivant représente plusieurs langages de programmation en langue arabe dans l'ordre chronologique :

Nom du langage	Année	Développeur	université (Pays)	Langage similaire	Ordinateur/ SE	Référence
Ghareeb (غريب)	1978	D. Mohamed Zaki Mohamed khedhr Sahara bd el aziz	Université de Mossoul (Iraq)	Basic	IBM 1130	[16]
Al-Khwarizmi (الخوارزمي)	1978	D. Farouk Rassam	Military Technical College (Iraq)	-	-	
ASM (دنيا)	1978	D. Zakariya Qassem	Iraq	Assembly	-	
Laith (ليث)	1978	D. Zakariya Qassem	Iraq	COBOL	-	
Soltana (سلطانة)	1979	-	Saoudi	Basic	ZX81	
Khwarizmi (خوارزمي)	1979	-	Digital Research (États-Unis)	Basic	CP/M	
Najla (نجلاء)	1979	D. Ridha Siraj Athiqa	King Fahd University of Petroleum and Minerals	Basic	Al-Farabi (أجهزة الفارابي)	

## Chapitre1. Étude de l'état de l'art

Sakhr Basic <i>(صخر بيسيك)</i>	1980	-	Koweït	Basic	-	
Dad <i>(ضاد)</i>	1984	D. Mohamed Ghazali khayat	université du Roi Abdulaziz	Pascal C	Cromemco IBM PC	
Sina <i>(سينا)</i>	1986	D. Muḥammad Al-Afandi	Université de Khartoum (sudan)	Pascal	-	
L.B.A. <i>(لغة برمجة أخرى)</i>	1986	D. Fouad Dahlouli D. Muḥammad Mandoura	université du Roi Abdulaziz, Université du Roi-Saoud	Basic Pascal	-	
Diwan <i>(ديوان)</i>	1987	D. Omar Mekdachi	-	Basic	comodor 64	
Arabic Pascal <i>(باسكال العربي)</i>	1988	D. Ahmed Mahjoub D. Hassan Madhkour	Saoudi	Pascal	VAX-11	
ARBI (Arabic Basic)	1990	Hamoud Aassaadoun Mustafa Yassin Ala Ajalad Mahmoud Ajalad	Koweït	GW-Basic	DOS	
Arabic Natural language processing <i>(سنبلة)</i>	1994	D. Muḥammad Al-Afandi	Saoudi	Pascal	-	
Visual Programming	1995	D. Khaled Sliman	États-Unis	-	-	
arablan	1995	Mansoor Al-A'Ali Mohammed Hamid	Université de Bahreïn	-	-	
Zay <i>(زاي)</i>	1998	D. Jamaleddin zagour	École nationale supérieure d'informatique (Algérie)	Pascal	Dos Windows	
Arabic Logo <i>(لوقو العربية)</i>	1999	PFE : Al-Jouhani et Al-Harbi Encadrement : D. Abdulmalik Salman	King Abdulaziz City for Science and Technology	Logo	Windows	
Pascal al-moutawasi <i>(باسكال المتوازي)</i>	2000	PFE: Khaled Al-Msibih et Abdullah Addakan Encadrement: D. Abdulmalik Salman	King Abdulaziz City for Science and Technology (Saoudi)	Pascal	Windows	
Al-Risalh <i>(الرسالة)</i>	2001	Muhammad Amin	Université de Bahreïn	-	-	[17]
Jeem <i>(ج)</i>	2006	D. Muhammad Ammar Assilka	Syrie	C	Windows	[18]
Ammoria <i>(عموريه)</i>	2006	Abduladhim Ahmed Ammouri	Jordan	C	Windows	[19]
Pheonix <i>(الفنيق)</i>	2007	Youssef Bassil	États-Unis	C	Windows	[20]
MyProlang	2008	Youssef Bassil Aziz Barbar	États-Unis	-	-	[21]
Supernova	2010	Mahmoud Fayed	Égypte	-	Windows Linux	[22]
Ibdaa' <i>(ابداع)</i>	2011	Wael Hassan Mahmoud Ali	Égypte	C	Windows Linux	[23]
Qalb <i>(قلب)</i>	2012	Ramsey Nasser	Eyebeam Art+ Technology Center (États-Unis)	Scheme	Interpréteur en linge	[24]

Tableau 1.2 Les langages de programmation en langue arabe

Nous présentons ci-dessous quelques remarques et informations supplémentaires sur les langages de programmation en langue arabe :

- 1) Entre les années 1978 et 1980, beaucoup de langages ont été proposés. Puis, on remarque un recul (un moyen de sept langages par décade).
- 2) Les langages de programmation en langue arabe ont été développés par des particuliers et non pas par des communautés veillant sur le langage (documentation, versions ...) ce qui explique la manque de réputation et la disparition de la grande majorité de ces langages.
- 3) Tous les langages de programmation identifiés sont inclus dans le paradigme des langages de programmation impératifs (langages similaires : Basic, C ou Pascal) à l'exception d'Al-Risalh : un langage orienté objet qui n'est plus existant et Qalb : un langage fonctionnel.
- 4) La majorité des langages de programmation identifiés ont été proposé pour des fins éducatives.
- 5) Nous remarquons l'absence des langages déclaratifs et en particulier l'inexistence d'un langage de programmation logique (Prolog).

### Conclusion

Dans ce chapitre, nous avons présenté la langue arabe : son importance mondiale, ses caractéristiques et sa contribution scientifique. Nous avons également présenté l'histoire des langages de programmation et leurs classifications. Finalement nous avons présenté les différents langages de programmation en langue arabe dans lesquels on ne trouve aucun langage de programmation déclarative.

Dans le chapitre suivant, nous concentrons sur le langage Prolog.

## Chapitre2. Le langage Prolog

---

Prolog est l'un des principaux langages de programmation déclaratifs. Le nom Prolog est un acronyme de PROgrammation en LOGique. Il a été créé par Alain Colmerauer et Philippe Roussel en 1972 à l'université de Marseille. Depuis, des nombreux interpréteurs Prolog ont été développés.

Ce chapitre a pour objectif d'introduire le langage Prolog. Nous présentons également quelques approches possibles pour développer un environnement de programmation logique en langue arabe.

### I. Syntaxe du langage Prolog

En 1995, l'organisation internationale de standardisation (ISO) a publié le standard du langage de programmation Prolog [25]. La norme décrit le noyau de Prolog y compris la syntaxe et la sémantique opérationnelle. La plupart des implémentations Prolog obéissent à la norme ISO.

Prolog est un langage d'expression des connaissances fondé sur le langage des prédictats. L'utilisateur définit les faits et les règles dans une structure appelée base de connaissances. L'interpréteur Prolog utilise cette base de connaissances pour répondre à des questions. La base de connaissances ainsi que les requêtes sont décrites à l'aide d'une grammaire Backus Naur Form (BNF). La totalité du BNF Prolog est donnée dans le livre « Prolog: The Standard Reference Manual » [26]. Une syntaxe plus réduite est donnée par Ivan Sukin [27]:

```
<program> = <clause list> <query> | <query>
<clause list> = <clause> | <clause list> <clause>
<clause> = <predicate> | <predicate> :- <predicate list>.
<predicate list> = <predicate> | <predicate list> , <predicate>
<predicate> = <atom> | <atom> ( <term list> )
<term list> = <term> | <term list> , <term>
<term> = <numeral> | <atom> | <variable> | <structure>
<structure> = <atom> ( <term list> )
<query> = ?- <predicate list>
<atom> = <small atom> | ' <string> '
<small atom> = <lowercase letter> | <small atom> <character>
<variable> = <uppercase letter> | <variable> <character>
<lowercase letter> = a | b | c | ... | x | y | z
<uppercase letter> = A | B | C | ... | X | Y | Z | _
<numeral> = <digit> | <numeral> <digit>
<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<character> = <lowercase letter> | <uppercase letter> | <digit> | <special>
<special> = + | - | * | / | \ | ^ | ~ | : | . | ? | | # | $ | &
<string> = <character> | <string> <character>
```

Ci-dessous nous présentons une description textuelle de la syntaxe du langage Prolog :

- Les commentaires: Il y a deux types de commentaires: le commentaire ligne désigné par un «% » au début de la ligne et le bloc de commentaires commence par « /\* » et se termine par « \*/ ».
- Les variables : une variable est une chaîne de caractères commençant par une majuscule.
- Les variables anonymes, notées « \_ » : ce sont des variables dont on ne veut pas savoir les valeurs.
- Les constantes : une constante est une chaîne de caractère commençant par minuscule ou un nombre (positif, négatif ou à virgule flottante).
- Les opérateurs : un opérateur est défini par son nom, une priorité et un spécificateur. La priorité détermine l'ordre de l'évaluation des différentes parties d'une expression. Le spécificateur est une mnémonique indiquant la classe d'associativité et si l'opérande est infixe, préfixe ou bien postfixe (Tableau 2.1).

Opérateur	Non associatif	Associatif à droite	Associatif à gauche
Infixe	$xfx$	$xfy$	$yfx$
Préfixe	$fx$	$fy$	-
Postfixe	$xf$	-	$yf$

Tableau 2.1 Spécificateur d'opérateurs

Le tableau ci-dessous représente la liste des opérateurs prédéfinis de la norme ISO Prolog :

Priorité	Spécificateur	Opérateur(s)					
1200	$xfx$	$:$	$-->$				
1200	$fx$	$:$	$?-$				
1100	$xfy$	$;$					
1050	$xfy$	$->$					
1000	$xfy$	$,$					
900	$fy$	$\backslash+$					
700	$xfx$	$=$	$\backslash=$				
700	$xfx$	$==$	$\backslash==$	$@<$	$@=<$	$@>$	$@>=$
700	$xfx$	$=..$					
700	$xfx$	$is$	$=:=$	$=\backslash=$	$<$	$=<$	$>$
500	$yfx$	$+$	$-$	$\wedge$	$\vee$		
400	$yfx$	$*$	$/$	$//$	$rem$	$mod$	$<<$
200	$xfx$	$**$					
200	$xfy$	$^$					
200	$fy$	$-$	$\backslash$				
100	$xfx$	$@$					
50	$xfx$	$:$					

Tableau 2.2 Priorités et Spécificateurs des opérateurs dans le standard Prolog

- Les listes : une liste est définie par une fonction à deux arguments (appeler foncteur « . »). Le premier argument représente l'entête et le deuxième argument représente la

queue (une liste). Une autre manière de définir une liste est d'utiliser des crochets contenant une série de termes séparés par des virgules ou bien une barre verticale séparant la tête de la queue par exemple :

[ ]	:	la liste vide
[a,b,c] = .(a, .(b, .(c, [])))	:	la liste qui contenant les constantes a, b, c
[X,Y,Z] = .(X, .(Y, .(Z, [])))	:	la liste qui contenant les variables X,Y,Z
[H T] = .(H, T)	:	la liste composée de tête H et la queue T

- Les chaînes de caractères : une chaîne de caractères est une séquence de caractères entourés par des guillemets. Elle est équivalente à une liste des code (ASCII ou Unicode) de caractères. Exemple : "Ceci est une chaîne de caractères"

## II. Programmation logique en Prolog

Le principe de la programmation logique est de décrire l'énoncé d'un problème par un ensemble d'expressions et de liens logiques au lieu de définir pas à pas la succession d'instructions que doit exécuter l'ordinateur pour résoudre le problème.

### 1. Faits, règles et requêtes

Les faits, les règles et les requêtes constituent les éléments de base du langage Prolog. Le tableau ci-dessous présente une base de connaissances Prolog :

N°	Base de connaissances Prolog	Explication
1	père(mansour, mahdi).	Fait : Mansour est le père de Mahdi
2	père(mahdi, hadi).	Fait : Mahdi est le père de Hadi
3	père(mahdi, olaya).	Fait : Mahdi est le père de Olaya
4	père(mahdi, rachid).	Fait : Mahdi est le père de Rachid
5	mère(khayzuran, hadi).	Fait : Khayzuran est la mère de Hadi
6	mère(khayzuran, rachid).	Fait : Khayzuran est la mère de Rachid
7	parent(X,Y) :- père(X,Y).	Règle : X est un parent de Y si X est le père de Y <i>Le « :- » signifie « si ».</i>
8	parent(X,Y) :- mère(X,Y).	Règle : X est un parent de Y si X est la mère de Y  On peut fusionner les règles 5 et 6 dans une seul règle : <i>parent(X,Y) :- père(X,Y) ; mère(X,Y).</i> Le « ; » indique une injection (OU logique)
9	frère_ou_sœur(X,Y) :- parent(Z,X), parent(Z,Y), X ≠ Y.	Règle : X est le frère de Y (ou sœur) s'ils ont un parent en commun et X différent de Y <i>Le « , » indique une conjonction (ET logique)</i> <i>Le « ≠ » signifie l'inégalité</i>

Tableau 2.3 Base de connaissances : exemple d'une famille

Les faits sont des données qu'on considère vraies. Une base de connaissances doit obligatoirement contenir des faits car c'est à partir d'eux que Prolog va pouvoir rechercher des preuves pour répondre aux requêtes de l'utilisateur. Un fait est un cas particulier de règle, cette dernière est une relation permettant d'établir de nouveaux faits par déduction. Dans notre exemple les faits décrivent la famille de Mansour. Les règles définissent les liens de parenté.

Il est possible d'interroger la base de connaissances de nombreuses manières grâce aux requêtes. Il existe deux types de requêtes : celles qui confirment ou infirment une proposition (la réponse attendue est Vrai ou faux). Le second type définit la requête à variables. Ceci permet de retourner les substitutions satisfaisant la requête. Voici quelques exemples de requêtes/réponses (selon ISO Prolog les requêtes commencent par ?- )

```
?- parent (mansour, mahdi).
True
```

Cette requête correspond à la question : Mansour est-il un parent de Mehdi?, Le moteur prolog confirme cette proposition en utilisant la règle N°: 7 et le fait N°: 1.

```
?- père (rachid, mahdi).
False
```

Le moteur Prolog infirme cette proposition parce qu'il n'existe pas de fait ni des déductions qui la confirme. L'ordre et le nombre des arguments sont importants. Par exemple le fait « *père(mahdi, rachid)*. » ne permet pas de donner une réponse positive vu que l'ordre des paramètres est différent.

```
?- mère (khayzuran, X).
X = hadi      ;
X = rachid    ;
False
```

Cette requête correspond à la question : Qui sont les fils de Khayzuran? L'ensemble de substitutions de X représente la réponse à cette requête : la première réponse est déduite à partir du fait N°: 5, la deuxième réponse est déduite à partir du fait N°: 6. La dernière réponse est False parce qu'il n'existe pas d'autres réponses à cette requête.

```
?- frère_ou_sœur(X, Y).
X = hadi      Y = olaya      ;
X = hadi      Y = rachid     ;
X = olaya     Y = hadi      ;
X = olaya     Y = rachid     ;
X = rachid    Y = hadi      ;
X = rachid    Y = olaya     ;
False
```

De même, on peut poser des questions à plusieurs variables comme celle de la requête qui correspond à la question: donner tous les frères et sœurs existants dans la base de connaissances.

## 2. Coupure et Négation

La coupure, notée « ! », permet de restreindre l'arbre de recherche en empêchant le développement de branches qui conduisent à des échecs ou à des solutions qui ne sont pas souhaitées. Un simple exemple dans lequel on utilise la coupure est la détermination du maximum de deux entiers :

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

L'idée est la suivante : si  $X$  est supérieur à  $Y$ , il n'y a plus besoin de réaliser la deuxième comparaison parce qu'on sait qu'elle finira par un échec.

La coupure peut changer la sémantique d'un prédicat on parle alors de coupures vertes et rouges. La "coupure verte" désigne que la sémantique d'un prédicat reste inchangée quand on lui ajoute la coupure sinon elle est dite "coupure rouge".

La négation, notée «  $\text{\textbackslash+}$  », ou `not` dans les anciens systèmes prolog, elle a la sémantique suivante :

- Si  $G$  réussis alors  $\text{\textbackslash+} G$  échoue
- Si  $G$  échoue alors  $\text{\textbackslash+} G$  réussis

Voici un exemple d'utilisation de la négation :

```
père(X, Y) :- \+femme(X), fils(Y,X)
```

$X$  est le père de  $Y$  si  $X$  n'est pas une femme et  $Y$  est le fils de  $X$ .

## 3. Opérations

### a. Les opérations arithmétiques

L'évaluation d'une expression arithmétique se fait avec l'opérateur « `is` ». Cet opérateur évalue l'expression à sa droite et unifie le résultat avec la valeur à gauche :

?- *Value is Expression*

Expression doit être calculable, voici quelques exemples d'évaluation des expressions arithmétiques :

- |                                      |   |
|--------------------------------------|---|
| ?- <i>3 is 1+1+1.</i> <i>ou bien</i> | <i>3 is +(1, +(1, 1))</i>   |
| <i>True</i>                          |   |
| ?- <i>X is 1+1+1.</i>                |   |
| <i>X = 3</i>                         |   |
| ?- <i>X is 1+1+Y.</i>                |   |
| <i>Erreur</i>                        | <i>(parce que Y n'est pas instanciée)</i>   |
| ?- <i>Y=2, X is 1+1+Y.</i>           |   |
| <i>Y=2</i>                           | <i>X = 4</i> <i>(correct parce que Y a une valeur numérique lorsque X est évalué)</i> |

Il est possible de définir des prédictats qui utilisent des expressions arithmétiques dans la base de connaissances. Par exemple, le prédictat fact permet de calculer la factorielle d'un entier:

Base de connaissances	Requête
fact(1, 1). fact(A, B) :- C is A-1, fact(C, D), B is A*D.	?- fact(5, R). R = 120

Tableau 2.4 Le prédictat fact

La majorité des implémentations prolog supportent les entiers et les doubles. Prolog offre une dizaine des fonctions prédéfinies telles que : -, +, \*, /, ^, mod, abs, min, max, sign, random, sqrt, sin, cos, tan, log, exp, ...

### a. Comparaison et unification des termes

Le tableau suivant présente l'opérateur d'unification ainsi que la liste des opérateurs de comparaison définis par le standard Prolog :

Opérateur	Notation termes	Notation arithmétique	Signification
Unification	=	-	Réussi si l'unification de ses arguments est possible
Négation de l'unification	\=	-	Réussi si l'unification de ses arguments n'est pas possible
Egalité	==	=:=	Réussi si ses arguments sont identiques/égaux
Inégalité	\==	=\=	Réussi si ses arguments ne sont pas identiques/égales
Inférieur	@<	<	Réussi si le premier argument est inférieur au deuxième argument
Inférieur ou égal	@=<	=<	Réussi si le premier argument est inférieur ou égal au deuxième argument
Supérieur	@>	>	Réussi si le premier argument est supérieur au deuxième argument
Supérieur ou égal	@>=	>=	Réussi si le premier argument est supérieur ou égal au deuxième argument

Tableau 2.5 Les opérateurs de comparaisons

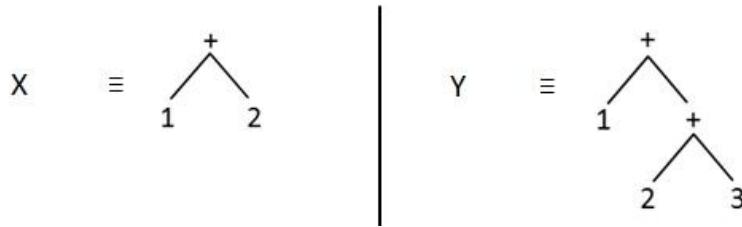
- L'opérateur d'unification (=) cherche les substitutions des variables qui permettent de rendre deux propositions identiques. Il s'agit d'un opérateur d'évaluation d'égalité d'arbres :

```
?- 1+2 = 1+2.
True

?- 1+2 = 2+1.
False

?- X = 1+2, Y = X+3.
X = 1+2      Y = 1+2+3
```

La figure ci-dessous représente les arbres correspondant aux expressions «  $X = 1+2$  » et «  $Y = 1+ (2+3)$  » :



**Figure 2.1** Arbres relatifs aux variables  $X$  et  $Y$

- L'opérateur d'identité (`==`) vérifie si ses deux arguments sont identiques sans faire aucune opération :

?-  $1+2 == 1+2.$   
True

?-  $1+X == 1+2.$   
False

- L'opérateur d'égalité arithmétique (`=:=`) évalue les expressions à gauche et à droite avant de réaliser la comparaison :

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

?-  $1+X =:= 1+2.$   
Erreur (parce que  $X$  n'est pas instanciée : l'expression à gauche n'a pas de valeur arithmétique)

Les opérateurs « `\=` », « `\==` » et « `=\=` » sont les négations correspondantes aux opérateurs « `=` », « `==` », et « `=:=` ». Les opérateurs de comparaison des termes (précédés par `@`) se basent sur l'ordre lexical des caractères. Par exemple :

?-  $famille @< family.$   
True

Les opérateurs de comparaison des expressions arithmétiques évaluent les expressions à gauche et à droite avant d'appliquer l'opérateur. Par exemple :

?-  $1+2 < 3+4.$   
True

#### 4. Listes

Les listes sont essentielles dans plusieurs programmes Prolog. Elles ont une structure récursive : toute liste est composée d'une tête (Head) et d'une queue (Tail). C'est à partir de ces deux éléments qu'on manipule les listes Prolog. Dans cette section nous présentons quelques prédictats basiques appropriés à la manipulation des listes

### b. Le prédictat member

Le prédictat "member" permet de vérifier si un élément appartient à une liste :

Base de connaissances	Requêtes
<pre>member(X,[X _]) :- !. member(X,[_ T]) :- member(X,T).</pre>	<p>?- <i>member(c, [a,b,c,d]).</i>  <i>True</i></p> <p>?- <i>member(e, [a,b,c,d]).</i>  <i>False</i></p> <p>?- <i>\+ member(e, [a,b,c,d]).</i>  <i>True</i></p>

Tableau 2.6 Le prédictat member

### c. Le prédictat reverse

Le prédictat "reverse" permet d'inverser une liste : il construit la liste constituée des mêmes éléments que la liste donnée mais dans l'ordre inverse. Pour inverser une liste il faut :

- i- Inverser la queue de la liste.
- ii- Concaténer la liste résultat et la liste contenant l'entête de la liste à inverser. C'est le prédictat "append" qui a le rôle de concaténer deux listes.

Base de connaissances	Requêtes
<pre>append([],L,L). append([H T],L,[H R]):- append(T,L,R). reverse([],[]). reverse([H T],L):- reverse(T,R),append(R,[H],L).</pre>	<p>?- <i>append([a,b], [a,c], L).</i>  <i>L = [a,b,a,c]</i></p> <p>?- <i>append([a,b], L, [a,b,c,d]).</i>  <i>L = [c,d]</i></p> <p>?- <i>reverse([a,b,c],L).</i>  <i>L = [c,b,a]</i></p>

Tableau 2.7 Le prédictat reverse

### d. Le prédictat length

Le prédictat "length" permet de déterminer le nombre des éléments d'une liste donnée. La longueur de la liste vide est égale à zéro :

Base de connaissances	Requêtes
<pre>length([],0). length([_ T],N):- length(T,N1),N is N1+1.</pre>	<p>?- <i>length([a,b,c], 3).</i>  <i>True</i></p> <p>?- <i>length([], N)</i>  <i>N = 0</i></p>

Tableau 2.8 Le prédictat length

### e. Le prédictat intersect

Le prédictat "intersect" permet de déterminer l'intersection de deux listes :

Base de connaissances	Requêtes
<pre>intersect([],X,[]):- !. intersect([X R],Y,[X T]):- member(X,Y),intersect(R,Y,T),!. intersect([X R],Y,L):- intersect(R,Y,L).</pre>	<pre>?- intersect([a,b,c], [c,e,a], L). L = [a,c]  ?- intersect(L, L, [a,c]). L = [a,c]  ?- intersect([], [a,b,c], []). True</pre>

Tableau 2.9 *Le prédictat intersect*

### f. Le prédictat quicksort

Le prédictat "quicksort" (ou tri rapide) définit un algorithme de tri fondé sur la méthode de conception diviser pour régner. Le principe du tri rapide consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en triant les éléments qui sont inférieurs au pivot à sa gauche et les éléments qui lui sont supérieurs à sa droite. Cette opération s'appelle le partitionnement. Pour chacune des sous-listes, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Base de connaissances	Requêtes
<pre>quicksort(L,T):- qsort(L,[],T).  qsort([],L,L). qsort([P L],Acc,T):- partition(P,L,L1,L2), qsort(L2,Acc,T1), qsort(L1,[P T1],T).  partition(_,[],[],[]). partition(P,[X T],[X U1],U2):- P&gt;X, partition(P,T,U1,U2). partition(P,[X T],U1,[X U2]):- P=&lt;X, partition(P,T,U1,U2).</pre>	<pre>?- quicksort([5,2,4,1,3],L). L = [1,2,3,4,5]</pre>

Tableau 2.10 *Le prédictat quicksort*

## III. Approches possibles pour développer un environnement de programmation logique en langue arabe

Dans cette section, nous présentons trois approches pour pouvoir écrire des bases de connaissances et les interroger en langue arabe: la première en utilisant l'interface prolog /java de swi-prolog. La deuxième en utilisant l'expert system shell JESS. La troisième solution consiste à proposer une implémentation de l'algorithme d'unification (le moteur d'inférences prolog)

Le développement d'un environnement pour la programmation logique est une tâche intéressante. Elle est d'autant plus intéressante qu'il faut gérer les caractéristiques de la langue arabe. Cela nécessite une proposition d'une nouvelle syntaxe. Parmi les points à résoudre : il n'existe pas de différence entre les majuscules dénotant les variables et les minuscules

dénotant les constantes. D'une autre part, la langue arabe s'écrit de droite à gauche ce qui entraîne des changements dans la syntaxe des expressions et des opérateurs.

### 1. L'interface JPL de SWI-Prolog

SWI-Prolog est une implémentation libre du langage Prolog, développé en 1987 à l'université d'Amsterdam. Depuis, il a subi des améliorations continues pour devenir l'une des implémentations Prolog les plus complètes.

SWI-Prolog offre des interfaces/ bibliothèques à des langages de haut niveau tel que JPL (interface avec le langage Java) de telle manière il est possible de définir/interroger une base de connaissances de SWI-Prolog par un programme Java [34]. Le diagramme d'activité suivant modélise le comportement du système utilisant SWI-Prolog pour la définition de base de connaissance et encore la formulation des requêtes en langue arabe :

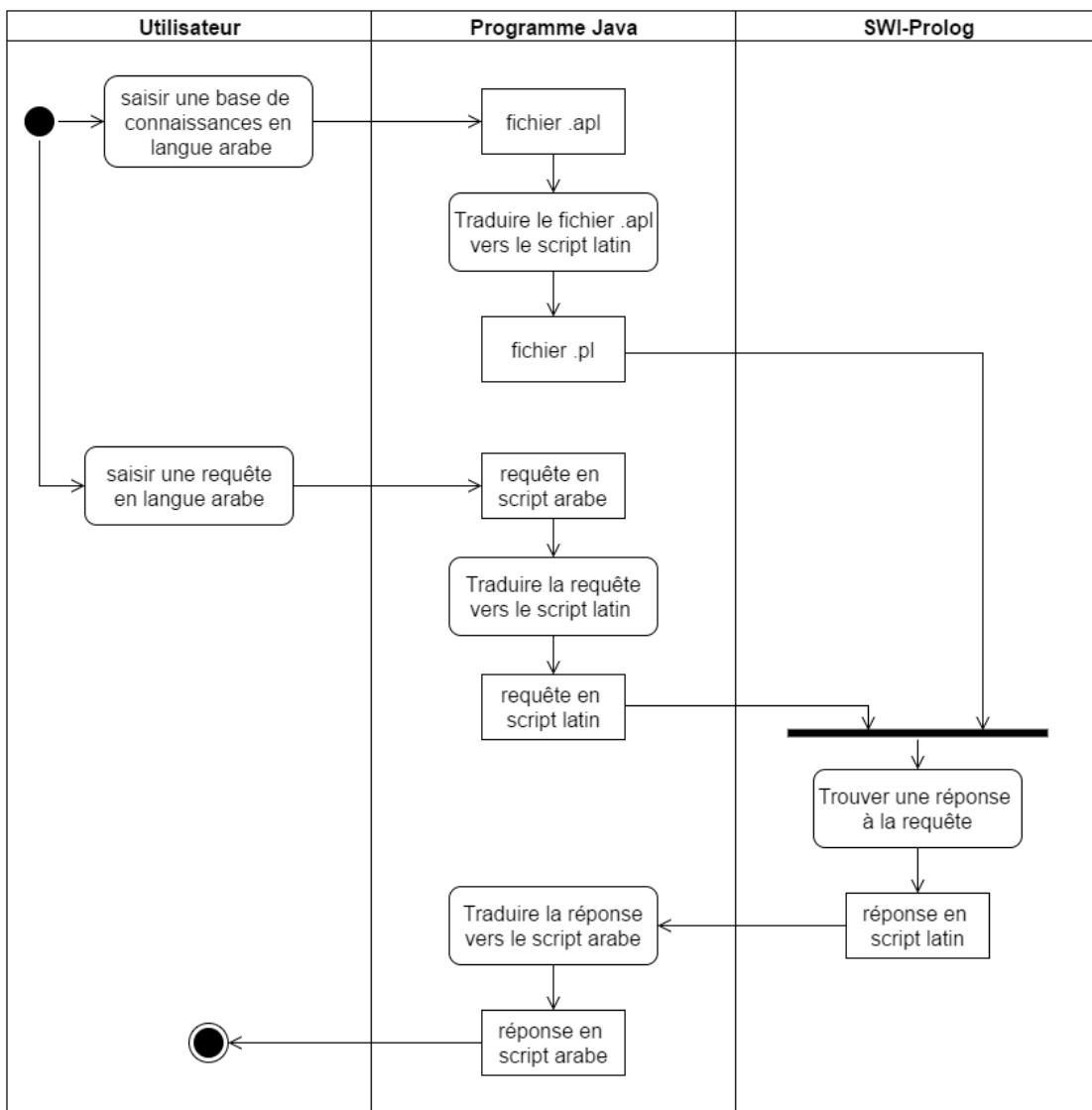


Figure 2.2 Diagramme d'activité : Utilisation de l'interface JPL de SWI-Prolog

Cette solution se base sur le moteur d'inférences de SWI-Prolog. En effet, il s'agit d'utiliser SWI-Prolog via un programme Java. En plus d'une interface graphique interactive et un système de gestion des fichiers (utilisés dans le projet final), le rôle principal du programme java est de traduire une base de connaissances écrite en arabe vers son équivalent

en script latin. De même, une traduction bidirectionnelle des requêtes en script arabe vers le script latin et de script latin vers son équivalent en script arabe.

Concrètement, la traduction se réalise à l'aide d'un analyseur lexical qui a rôle d'identifier les lexèmes de la base de connaissances. Ainsi à chaque lexème on détermine son équivalent en script latin en utilisant une table d'équivalences des caractères (tableau 2.12). Les chaînes de caractères précédées par "?" dénotent les variables. Elles sont remplacées par des majuscules.

Script arabe	Script latin												
ا	a	ح	vh	ز	z	ط	xt	ق	q	ه	h		
ب	B	خ	xh	س	s	ظ	xz	ك	k	و	w		
ت	T	د	d	ش	vs	ع	g	ل	l	ي	y		
ث	vt	ذ	vd	ص	xs	غ	vg	م	m	ة	va		
ج	j	ر	r	ض	xd	ف	f	ن	n				

Tableau 2.11 Table d'équivalences des caractères arabe/latin

Le moteur d'inférences raisonne en se basant sur la requête et la base de connaissances en script latin. Le résultat sera traduit inversement du script latin vers le script arabe.

Nous avons implémenté cette approche. Elle accomplit la majorité des fonctionnalités de SWI-Prolog mais, rapidement, nous atteignons les limites de cette solution :

- 1) On remarque que l'application est composée de deux systèmes ce qui donne un problème de configuration : non seulement il faut disposer d'un Java Development Kit (JDK) et installer SWI-Prolog mais ils doivent être compatibles (la dernière version de JDK compatible avec SWI-Prolog jusqu'à 2016 est le JDK7u71). Nous avons en plus noté l'obligation de l'utilisation de system32 et des autres configurations supplémentaires dans la majorité des cas.
- 2) Le moteur d'inférences qui représente le noyau du langage Prolog est vu comme une boîte noire: on ne sait rien de sa manière de fonctionnement. Donc, on ne peut jamais le modifier ou ajouter de nouvelles fonctionnalités de sorte que nous sommes en dépendance totale de SWI-Prolog.
- 3) La table d'équivalence proposée ne supporte pas les bases de connaissances vocalisées (utilisant les voyelles simples : ـ (fatha), ـ(kasra), ـ(damma)) ainsi "al-hamza" et ses positions (ء ؤ ئ ؛ ؔ ؕ). Ce qui présente une limitation supplémentaires

## 2. L'expert system shell JESS

Une seconde solution possible pour implémenter un environnement de programmation logique est d'utiliser un système expert basé sur le chaînage arrière : Prolog utilise en effet cette technique pour parcourir les arbres représentants les espaces de recherche. Le chaînage arrière représente la différence majeure entre Prolog et les experts systems shells qui, dans la plupart de temps, utilisent le chaînage avant.

Le principe de chaînage avant est simple, il consiste à partir des prémisses pour déduire de nouvelles conclusions. Ces conclusions enrichissent la mémoire et peuvent devenir les prémisses d'autres règles. On désigne par chaînage arrière le fait de commencer par la conclusion, puis, prenant les conditions mènent au but comme nouveaux sous buts et

recommencer la recherche, récursivement, jusqu'à arriver aux faits initiaux. D'où sa seconde appellation raisonnement arrière ou raisonnement guidé par les buts. Le chaînage arrière est naturellement plus adéquat pour Prolog puisque c'est à partir de la requête que le moteur d'inférences commence la recherche.

JESS, l'acronyme de Java Expert System Shell, est l'un des experts systems shells les plus développés [35]. Il est libre à des fins éducatives sinon il est commercial (offre une version d'essai d'un mois). Il supporte le chaînage arrière tout comme le chaînage avant.

L'utilisation de JESS consiste à importer les jars: jess.jar et jsr94.jar dans le projet. JSR, l'acronyme de Java Specification Requests, est l'API java pour communiquer avec l'implémentation du moteur d'inférences jess.jar.

L'environnement de programmation prolog construit en utilisant JESS est très similaire à celui qui utilise l'interface JPL. En effet, nous avons toujours besoin de développer un module permettant de convertir la base de connaissances ainsi que les requêtes de la syntaxe du Prolog arabe vers la syntaxe définie par JSR (Java Specification Requests). Le moteur d'inférences est encore une fois, vu comme une boîte noire. Le seul avantage de l'utilisation de JESS par rapport à l'utilisation de JPL est que l'application en JESS est caractérisée par la simplicité d'installation car le moteur d'inférences est importé dans le projet sous forme d'un fichier d'archive (jess.jar). Il faut en plus ajouter l'API jsr94.jar qui permet la communication avec le moteur d'inférences.

### 3. Implémentation de l'algorithme d'unification

L'algorithme d'unification permet, à partir d'une base de connaissances (le programme), de trouver des réponses aux requêtes. Il repose sur les principes de résolution dans le calcul des prédictats. Le diagramme d'activité suivant présente le comportement du système pour répondre à une requête :

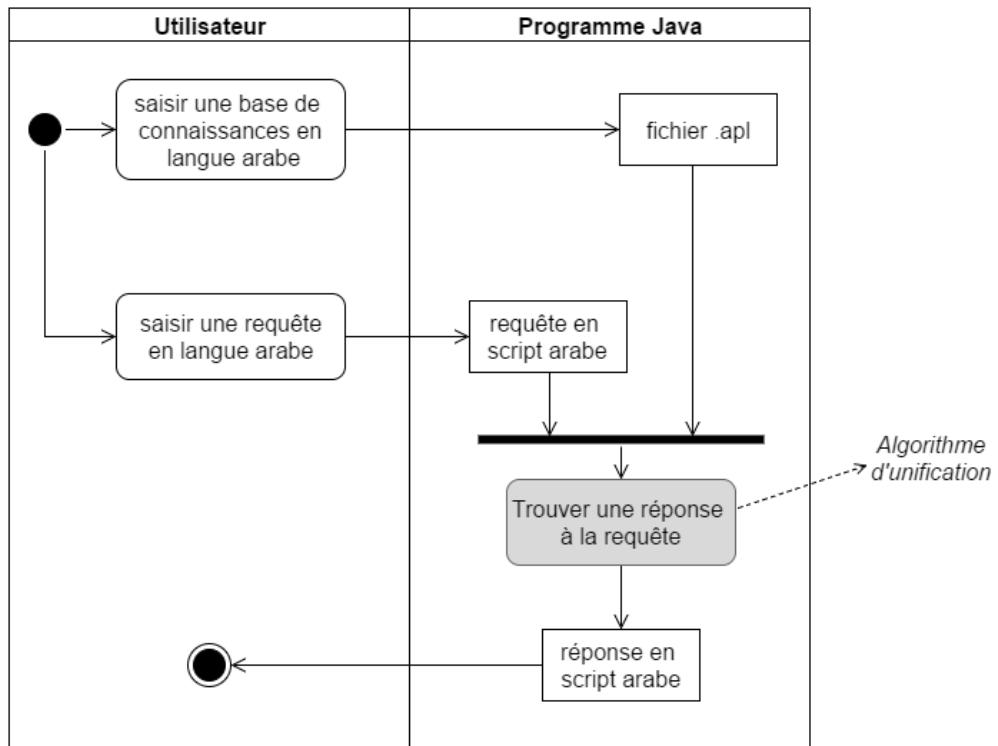


Figure 2.3 Diagramme d'activité : Implémentation de l'algorithme d'unification

Les avantages de cette solution sont nombreux, liés à l'implémentation de notre propre moteur d'inférences (l'algorithme d'unification) :

- 1) Cette solution n'exige pas l'utilisation d'un moteur d'inférences extérieur.
- 2) Les perspectives de cette solution sont plus vastes: on peut, à tout moment, modifier ou ajouter des nouvelles fonctionnalités ou bien travailler sur les performances de l'application.
- 3) Facilité d'installation en plus que l'application finale est multi-plateforme et portable dont nous n'avons besoin que d'un Java Development Kit (JDK).
- 4) On n'a plus besoin de module de traduction. Il faut cependant définir la grammaire (BNF) pour prolog en langue arabe.

## Conclusion

Prolog est un langage de programmation à part entière. Il se distingue par son mode de programmation déclaratif et interactif. Prolog est étonnamment puissant puisqu'il suffit bien souvent de très peu de lignes de code pour mettre en place des programmes intéressants (exemple le tri rapide s'écrit en sept lignes de code).

Dans ce chapitre, nous avons présenté trois solutions possibles pour développer un environnement de programmation logique en langue arabe : la première consiste à emprunter le moteur d'inférences de SWI-Prolog grâce à l'interface JPL. La deuxième consiste à utiliser un expert system shell comme JESS mais ces solutions présentent beaucoup de limites telles que l'impossibilité de modifier le moteur d'inférences. La meilleure solution est d'implémenter l'algorithme d'unification: celle que nous l'avons choisie et celle que nous l'en parlent dans les chapitres suivants.

## Chapitre3. Définition de base de connaissances et unification des expressions logiques

---

Dans ce chapitre, nous commençons par introduire le fondement mathématique de la programmation logique (la logique des prédictats). Puis, nous donnons une description semi-formelle de l'algorithme d'unification. Enfin nous présentons la conception de l'algorithme d'unification qui se décompose en trois parties : la première est de préparer les structures de données pour effectuer une unification élémentaire. La deuxième est de modéliser la base de connaissances et finalement, la troisième est la conception du moteur d'inférences qui, à partir d'une base de connaissances, raisonne pour trouver une réponse à une requête.

Pour notre implémentation, nous sommes inspirés de l'algorithme d'unification proposé par le professeur à Université du Nouveau-Mexique George Luger dans son livre « AI Algorithms, Data Structures and Idioms in Prolog, Lisp, and Java » [30].

### I. La logique des prédictats

La logique des prédictats, ou logique du premier ordre, permet de construire des systèmes formels décrivant des connaissances relatives à des environnements complexes. Elle est construite à partir de la logique propositionnelle et s'inspire du langage naturel pour définir des formules. Par exemple la proposition :

- *George mange une pomme*

S'écrit en logique de premier ordre :

- *mange (george, pomme)*

Où "mange" est le prédictat, "george" et "pomme" sont des termes.

Pour écrire des formules de logique des prédictats, on commence par se donner un vocabulaire composé de symboles de différents types :

- variables ( X , Y , Z , ...)
- constantes ( a , b , c , ...)
- fonctions ( f , g , h , ...)
- prédictats ( p , q , r , ...)

Il est à noter qu'à chaque fonction ou prédictat, on définit l'arité par le nombre d'arguments. Les constantes sont des fonctions d'arité zéro.

La Grammaire de la logique des prédictats est la suivante [31]:

terme	=	constante   variable   fonction (termel, terme2, . . . termen)
atome	=	prédictat (termel, terme2, . . . termen)
formule	=	atome   $\neg$ formule   formule connecteur formule   quantificateur variable formule
connecteur	=	$\wedge$   $\vee$   $\Rightarrow$
quantificateur	=	$\forall$   $\exists$

En logique des prédictats, les constantes, les variables et les symboles de fonctions permettent de définir les termes. Une fonction est un terme composé qui s'évalue uniquement par une seule constante par exemple « père (david) » s'évalue exclusivement à George.

En logique des prédictats, une formule peut prendre plusieurs formes. On distingue en particulier les formules atomiques qui sont composées d'un prédictat suivi par des termes séparés par des virgules.

Il est possible de construire de nouvelles formules à partir de formules existant en utilisant les opérateurs logiques et les quantificateurs, si F et G sont des formules atomiques et x un variable alors ( $\neg F$ ) est une formule, ( $F \wedge G$ ), ( $F \vee G$ ) et ( $F \Rightarrow G$ ) sont des formules. De même  $\forall x.F$  et  $\exists x.F$  sont des formules.

On trouve principalement quatre connecteurs logiques :  $\wedge$  (conjonction),  $\vee$  (disjonction),  $\neg$  (négation) et  $\Rightarrow$  (implication).

Les quantificateurs sont  $\forall$  (quantificateur universel) et  $\exists$  (quantificateur existentiel). Le quantificateur universel exprime le fait que tous les éléments d'un ensemble d'objets sur lequel s'exprime un prédictat vérifient ce prédictat, c'est-à-dire  $\forall x.P(x)$  est vrai revient à considérer que  $P(a_1) \wedge \dots \wedge P(a_n)$  est vrai, si  $\{a_1, \dots, a_n\}$  est le domaine de x. De la même façon, le quantificateur existentiel exprime le fait qu'au moins un des éléments d'un ensemble d'objets sur lequel s'exprime un prédictat vérifie ce prédictat, c'est-à-dire  $\exists x.P(x)$  est vrai revient à considérer que  $P(a_1) \vee \dots \vee P(a_n)$  est vrai, si  $\{a_1, \dots, a_n\}$  est le domaine de x.

Les connecteurs et les quantificateurs permettent de construire des formules logiques plus expressives par exemple :

1) « Certains étudiants assistent à tous les cours »

$$\exists x.(\text{étudiant}(x) \wedge (\forall y.\text{assiste}(x, y)))$$

2) « Aucun étudiant n'assiste à un cours intérressant »

$$\neg \exists x.(\text{étudiant}(x) \Rightarrow (\text{assiste}(x, y) \wedge \neg \text{intéressant}(y)))$$

3) le syllogisme socratique : « Tous les hommes sont mortels, Socrate est un homme, Donc, Socrate est mortel »

$$(\forall x. (\text{homme}(x) \Rightarrow \text{mortel}(x))) \wedge \text{homme}(\text{socrate}) \Rightarrow \text{mortel}(\text{socrate})$$

En développant un moteur d'inférences pour le calcul des prédictats, nous allons suivre les conventions choisies par le langage Prolog : nous nous limitons à une forme d'expression logique appelée clause de Horn. Cette dernière est une disjonction des littéraux (un littéral est une variable propositionnelle ou sa négation) comportant au plus un littéral positif. Par exemple,  $\neg p \vee \neg q \vee r$ ,  $\neg s \vee \neg r$  ou encore simplement le littéral  $t$  sont des clauses d'Horn.

L'idée qui motive la limitation à ce genre de clauses est qu'on peut les écrire comme une conjonction de littéraux positifs impliquant un littéral positif unique. Par exemple  $\neg p \vee \neg q \vee r$  est logiquement équivalent à  $(p \wedge q) \Rightarrow r$ . La partie se situant à gauche de l'implication s'appelle la prémissse (corps), et la partie se situant à sa droite la conclusion (tête).

Selon les conventions prolog, les clauses de Horn sont représentées à l'envers tout en remplaçant la flèche d'implication par « `:-` ». Par exemple la clause de Horn  $(p \wedge q) \Rightarrow r$  s'écrit  $r :- p, q$

opérateur	Logique de prédictats	Prolog
implication	$\Rightarrow$	<code>:-</code>
conjonction	$\wedge$	,
disjonction	$\vee$	;

Tableau 3.1 Opérateurs en logique de prédictats et en Prolog

Cette représentation facilite l'exploration de l'espace de recherche adoptée par le moteur d'inférences Prolog qui est le chaînage arrière : ce moteur commence par la conclusion (but), il cherche toutes les règles menant aux buts. Ceci permet de définir de nouveaux buts et recommencer la recherche récursivement jusqu'à atteindre les faits (succès) ou vérifier que toutes les possibilités ne donnent pas de réponse satisfaisante.

## II. Description semi-formelle de l'algorithme d'unification

L'écriture de la base de connaissances sous forme d'un ensemble de clauses de Horn simplifie le raisonnement logique pour satisfaire une requête. Etant donné le programme suivant :

```

père(mansour, mahdi).
père(mahdi, hadi).
père(mahdi, rachid).

grand_père(X, Y) :- père(X, Z), père(Z, Y).
    
```

On veut savoir si "mansour" est le grand-père de Rachid, donc on pose la question :

```
grand_père(mansour, rachid).
```

En substituant la constante "mansour" par la variable X, la constante "mahdi" par la variable Z et la constante "rachid" par la variable Y, nous obtenons la conclusion désirée. Par conséquent la proposition « mansour est le grand-père de rachid » est correcte. L'algorithme qui détermine les substitutions afin de rendre deux expressions identiques (dans ce cas : `grand-père(X, Y)` et `grand-père(mansour, rachid)`) est appelé l'algorithme d'unification.

Afin de trouver les substitutions des variables, L'algorithme d'unification crée un graphe And/Or relative à la requête. Les nœuds And exigent que toutes les branches fils soient satisfaites alors que dans les nœuds Or n'exigent qu'une seule branche fils soit satisfaite. La figure suivante représente le graphe And/Or relatif à la requête `grand_père(mansour, rachid)`.

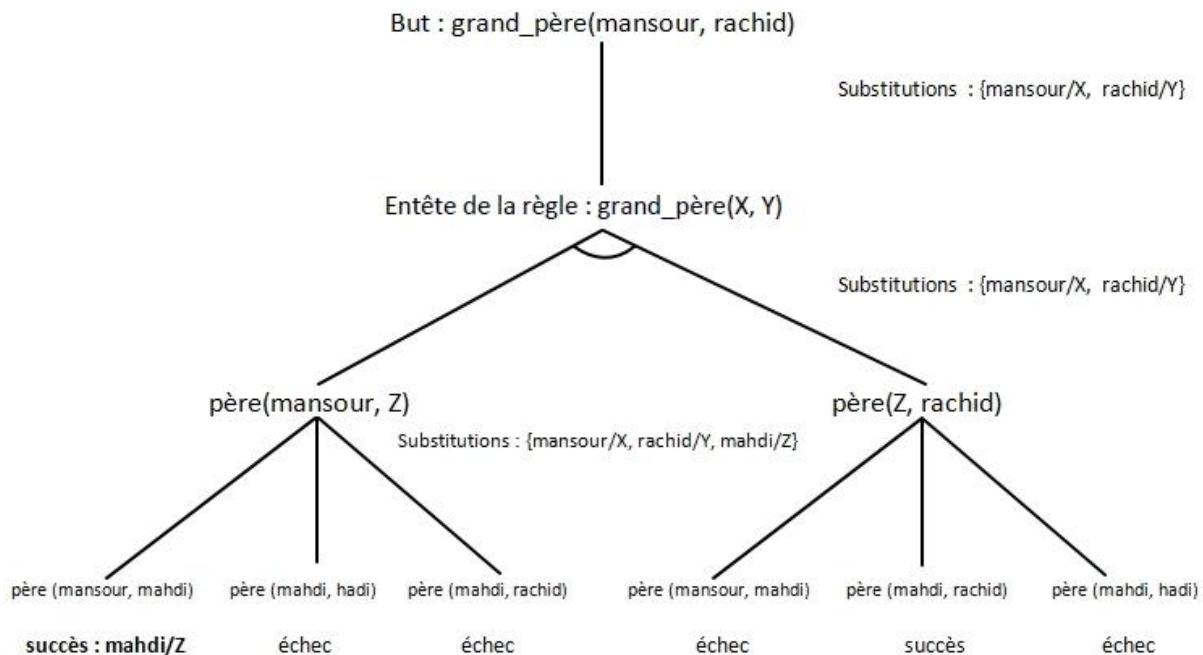


Figure 3.1 Graphe And/Or

La racine du graphe And/Or établie représente le but. Les deux branches menant au but « père(mansour, Z) » et « père(Z, rachid) » doivent toutes les deux être satisfaites parce qu’elles sont connectées avec le connecteur logique "Et". En effet, l’algorithme d’unification cherche dans la base de connaissances les faits ou les règles qui satisfont la première branche tout en effectuant les substitutions des variables. Dans notre cas « père(mansour, mahdi) » s’unifie avec « père(mansour, Z) » en substituant la variable Z avec la constante mahdi. En cas d’échec d’unification comme la tentative d’unifier « père(mansour, Z) » avec « père (mahdi, hadi) » l’algorithme d’unification prévoit un "backtrack" vers le niveau supérieur et passe à la branche suivante. Une fois l’algorithme d’unification trouve les substitutions qui satisfont la première branche il passe à la deuxième et ainsi de suite de manière récursive, jusqu’à satisfaire le but énoncé initialement.

A chaque passage par un nœud dans graphe And/Or établi, l’algorithme d’unification utilise la fonction *unify*. Cette dernière retourne les substitutions qui rendent deux expressions identiques. Dans le cas où l’unification est irréalisable la fonction *unify* retourne : échec. Ci-dessous l’algorithme de la fonction *unify*

```

fonction unify(E1, E2)

cas(1) : E1 et E2 sont des constantes ou la liste vide :
        si E1 = E2 alors
            retourne l'ensemble de substitutions vide;
        sinon
            retourne Echec;

cas(2) : E1 est une variable :
        si E1 est liée alors
            retourne unify(la liaison de E1, E2);

        si E1 apparaît dans E2 alors
            retourne Echec;
    
```

```

        sinon
            retourne l'ensemble de substitutions {E1/E2};
        cas(3): E2 est un variable :
            si E2 est liée alors
                retourne unify(la liaison de E2, E1);
            si E2 apparait dans E1 alors
                retourne Echec;
            sinon
                retourne l'ensemble de substitutions {E2/E1};

        cas(4) : le nombre des éléments de E1 est différent du
                  nombre des éléments de E2 :
                retourne Echec;

        cas(5) : autrement:
            HE1 = premier élément de E1;
            HE2 = premier élément de E2;
            S1 = unify(HE1, HE2);
            si S1 == Echec alors
                retourne Echec;
            TE1 = (récuratif) premier élément de la queue de E1
            TE2 = (récuratif) premier élément de la queue de E2
            S2 = unify (TE1, TE2);
            si S2 == Echec alors
                retourne Echec;
            sinon
                retourner l'ensemble des substitutions S1 ∪ S2
    
```

Le tableau ci-dessous présente quelques exemples d'application de la fonction *unify* :

<b>Unifiable1</b>	<b>Unifiable2</b>	<b>Trace d'exécution</b>	<b>Résultat</b>
mansour	mansour	cas(1)	$\emptyset$
X	mansour	cas(2)	{X/mansour}
père X rachid	père mansour	cas(4)	Echec
père X rachid	père mahdi Y	cas(5) <i>iteration 1</i> HE1 = père HE2 = père cas(1) S1 = $\emptyset$ <i>iteration 2</i> TE1 = X TE2 = mahdi cas(2) S1 = {X/mahdi} <i>iteration 2.1</i> TE1 = rachid TE2 = Y cas(3) S2 = {Y/rachid} S1 ∪ S2 = {X/mahdi, Y/rachid}	{X/mahdi, Y/rachid}

X rachid Y	père Y hadi	$\begin{array}{l} \text{cas(5)} \\ \text{iteration 1} \\ \text{HE1 = X} \\ \text{HE2 = père} \\ \text{cas(2)} \\ S1 = \{X/\text{père}\} \\ \text{iteration 2} \\ \text{TE1 = rachid} \\ \text{TE2 = Y} \\ \text{cas(3)} \\ S1 = \{Y/rachid\} \\ \text{iteration 2.1} \\ \text{TE1 = Y} \\ \text{TE2 = hadi} \\ \text{cas(2)} \\ \text{Echec (Y est liée à rachid!)} \end{array}$	Echec
------------	-------------	--	-------

Tableau 3.2 Trace d'exécution : application de la fonction unify

L'algorithme d'unification ne se limite pas à confirmer ou infirmer une proposition, il permet aussi de déterminer toutes les substitutions possibles d'une variable qui permet une réponse positive. Par exemple pour déterminer les petits-fils de "mansour", on pose la question :

*grand\_père(mansour, X).*

La figure suivante représente l'arbre de preuve relatif à la requête posée :

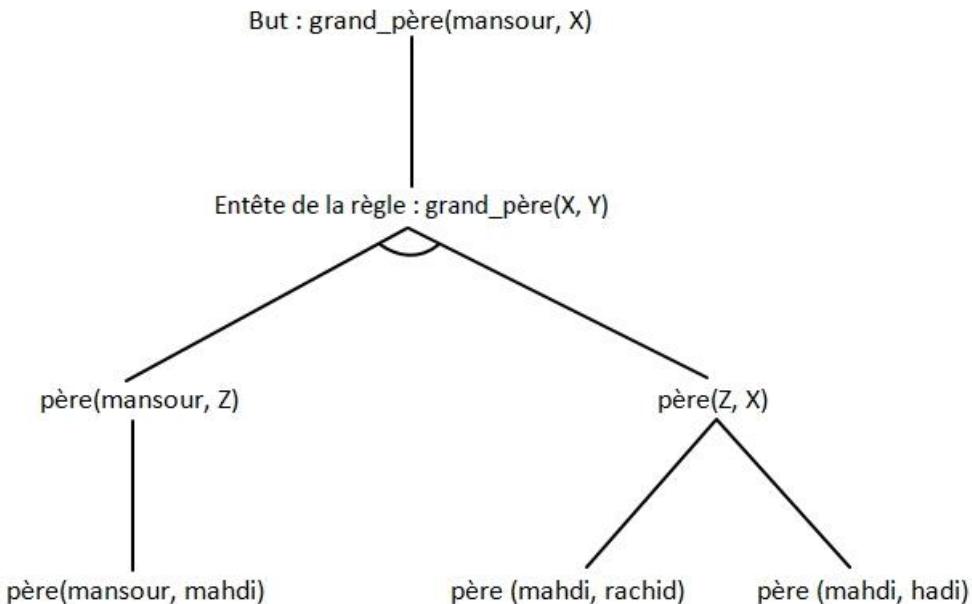


Figure 3.2 Arbre de preuve

L'arbre de preuve (en anglais: proof tree) est une restriction du graphe And/Or où on ne grade que les branches de succès. La réponse est la liste des substitutions de X (X= hadi ; X= rachid).

### III. Conception de l'algorithme d'unification

L'implémentation de l'algorithme d'unification nécessite la définition des éléments de base pour effectuer une unification élémentaire ainsi que la modélisation de la base de connaissances. Finalement, il faut développer un système qui permet de construire le graphe And/Or. Ce système doit en plus être capable de l'explorer afin de construire l'arbre de preuve relatif à une requête.

#### 1. L'unification élémentaire

Nous commençons par la définition des éléments basiques pour la construction d'un système de calcul des prédictats. Nous exposons les classes **Constant**, **Variable** et **SimpleSentence** ainsi que leurs définitions de la méthode *unify* : la méthode permettant d'effectuer une unification. Le diagramme des classes suivant représente la hiérarchie des classes basiques et les relations entre elles :

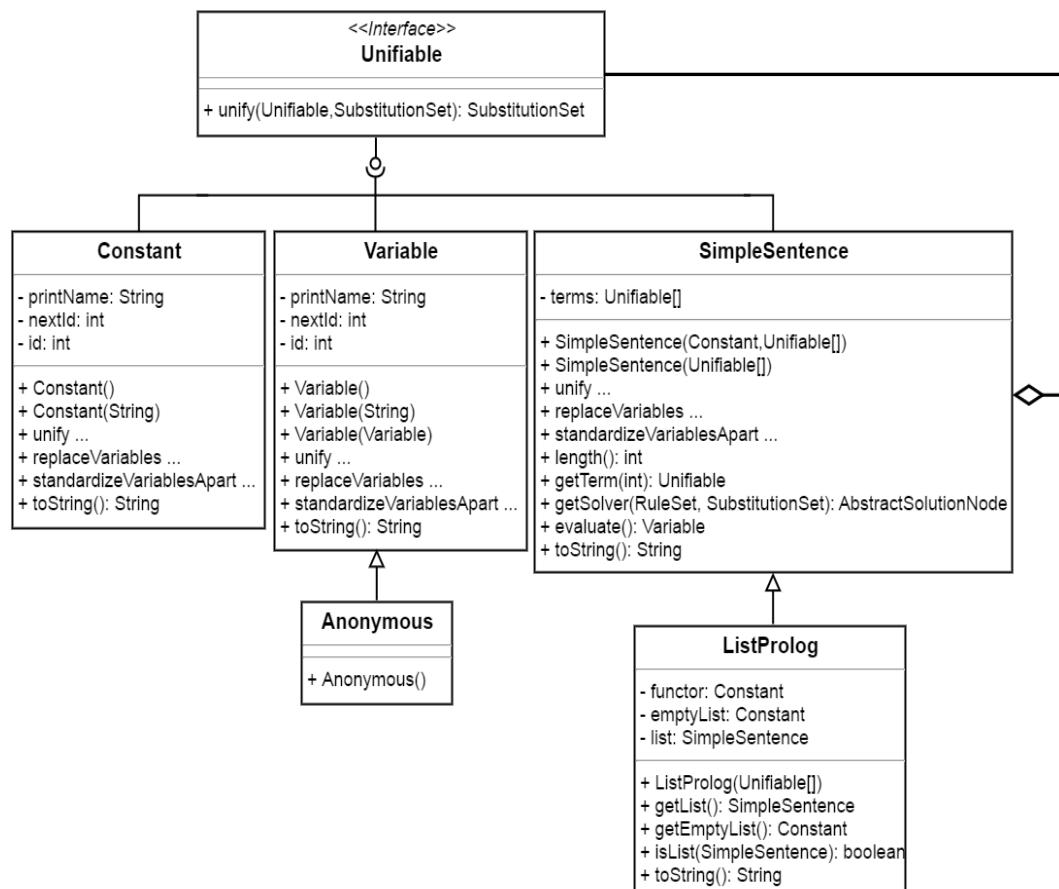
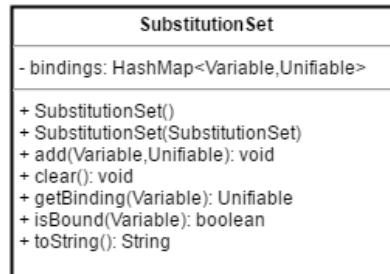


Figure 3.3 Diagramme des classes : l'unification élémentaire

L'interface **Unifiable** définit la signature de la méthode *unify*. Cette méthode est primordiale pour réaliser l'unification. En effet, les définitions de cette méthode dans **Constant**, **Variable** et **SimpleSentence** représentent l'implémentation de la fonction *unify* (dont l'algorithme est donné dans le paragraphe précédent).

#### a. La classe SubstitutionSet

Comme la méthode *unify* a le rôle de trouver les substitutions pour rendre deux unifiables identiques, nous avons besoin d'une structure de données contenant les substitutions permettant d'unifier deux expressions données. C'est la classe **SubstitutionSet**.

**Figure 3.4** La classe *SubstitutionSet*

La classe **SubstitutionSet** est une collection des couples dont le premier élément est une variable et le deuxième élément est un unifiable qui représente sa liaison d'où son nom *bindings*. Cette classe est dotée de la méthode *add* permettant d'ajouter une nouvelle substitution, la méthode *isBound* vérifie si une variable est liée, la méthode *getBinding* retourne la liaison de la variable passée en paramètre et la méthode *clear* permet de vider *bindings*.

### b. La classe Constante

Une instance de la classe **Constante** est caractérisée par un nom *printName* contenant le nom de la constante et un identifiant *id*. L'attribut *nextId* est un compteur qui s'incrémente par un à chaque création d'une nouvelle constante. La méthode *unify* est définie dans la classe **Constante** comme suit:

```

public SubstitutionSet unify(Unifiable exp, SubstitutionSet s) {
    if (this == exp)
        return new SubstitutionSet(s);

    if (exp instanceof Variable)
        return exp.unify(this, s);

    return null;
}
  
```

Si l'unifiable *exp* est égal à l'objet courant alors on garde le *SubstitutionSet* reçu en paramètre. Si *exp* est une variable alors on utilise la méthode *unify* de cette dernière Sinon on retourne *null* pour signaler l'échec d'unification.

### c. La classe Variable

Une instance de la classe **Variable** est caractérisée par un nom *printName* et un identifiant *id*. L'attribut *nextId* est un compteur qui s'incrémente par un à chaque création d'une nouvelle variable. On peut instancier une variable de deux manières : en utilisant son nom ou bien par copie. **Anonymous** est une variable dont le *printName* est égal à `_`. La méthode *unify* est définie dans la classe **Variable** comme suit :

```

public SubstitutionSet unify(Unifiable exp, SubstitutionSet s) {
    if (this == exp)
        return s;
}
  
```

```

        if (s.isBound(this))
            return s.getBinding(this).unify(exp, s);

        SubstitutionSet sNew = new SubstitutionSet(s);
        sNew.add(this, exp);
        return sNew;
    }
}

```

Si l'unifiable *exp* est égale à l'objet courant alors on garde la *SubstitutionSet* reçue en paramètre. Si la variable courante est liée alors on unifie la liaison de la variable courante avec *exp*. Sinon on ajoute à la *SubstitutionSet* une nouvelle substitution : la variable courante avec *exp*.

#### d. La classe SimpleSentence

Les formules atomiques en logique des prédictats sont modélisées par **SimpleSentence** dont, généralement, le premier terme est une constante qui représente le nom du prédictat et les termes qui suivent sont les paramètres. La classe **SimpleSentence** est à la fois un **Unifiable** et une composition de **Unifiable**. Concrètement, il s'agit d'un tableau d'unifiables. Une **SimpleSentence** peut être construite à partir d'une constante (premier terme) suivi d'un tableau d'unifiables ou bien un tableau d'unifiables tout simplement. La méthode *unify* est définie dans *SimpleSentence* comme suit :

```

public SubstitutionSet unify(Unifiable exp, SubstitutionSet s) {
    if (exp instanceof Variable)
        return exp.unify(this, s);

    if (exp instanceof SimpleSentence) {
        SimpleSentence s2 = (SimpleSentence) exp;
        // Compare list lengths
        if (this.length() != s2.length())
            return null;

        // Unify pairs
        SubstitutionSet sNew = new SubstitutionSet(s);
        for (int i = 0; i < this.length(); i++) {
            sNew = this.getTerm(i).unify(s2.getTerm(i), sNew);
            if (sNew == null)
                return null;
        }
        return sNew;
    }

    return null;
}

```

Si l'unifiable *exp* est une variable alors on utilise la méthode *unify* de ce dernier. S'il s'agit d'unification avec une liste d'unifiables (une autre **SimpleSentence**) on vérifie tout d'abord les longueurs : si le nombre des éléments de deux expressions n'est pas égal alors l'unification échoue. Sinon on unifie les unifiables des **SimpleSentence** deux à deux : en cas de succès d'unification dans les deux premiers unifiables on passe aux deuxièmes et ainsi de suite jusqu'à unifier tous les unifiables de deux listes.

Comme le nom l'indique, La méthode *replaceVariables* permet de remplacer une variable par sa liaison. En cas d'un **SimpleSentence**, on parcourt les termes : si une constante est trouvée la méthode *replaceVariables* de la classe **Constante** est appliquée (retourne la constante courante), si c'est une variable la méthode *replaceVariables* de la classe **Variable** est appliquée (retourne la liaison de la variable).

La méthode *getSolver* permet de trouver la solution pour une requête. Ainsi la définition de la méthode *standardizeVariablesApart* dans les Classe **Constante**, **Variable** et **SimpleSentence** est utilisée pour cet objectif. Le processus de la recherche des solutions sera expliqué en détails dans la partie « Le moteur d'inférences » de ce chapitre. Egalement, la méthode *evaluate* permet de réaliser des calculs arithmétiques (cette méthode est expliquée en détails dans le chapitre suivant).

### e. La classe ListProlog

Les listes en Prolog sont aussi des **SimpleSentence** puisqu'elles regroupent un ensemble d'unifiables. Une liste est distinguée par sa manière de construction récursive. Une liste est composée d'un entête et d'une queue. Pour construire une liste, on utilise le foncteur `".."` c'est un prédictat dont le premier argument est l'entête et le deuxième argument est la queue.

La liste [a, b, c] dans notre application est construite comme suit:

```
Constant functor = new Constant("."), emptyList = new Constant("[]");
...
new SimpleSentence(functor, new Constant("a"),
    new SimpleSentence(functor, new Constant("b"),
        new SimpleSentence(functor, new Constant("c"), emptyList)));
```

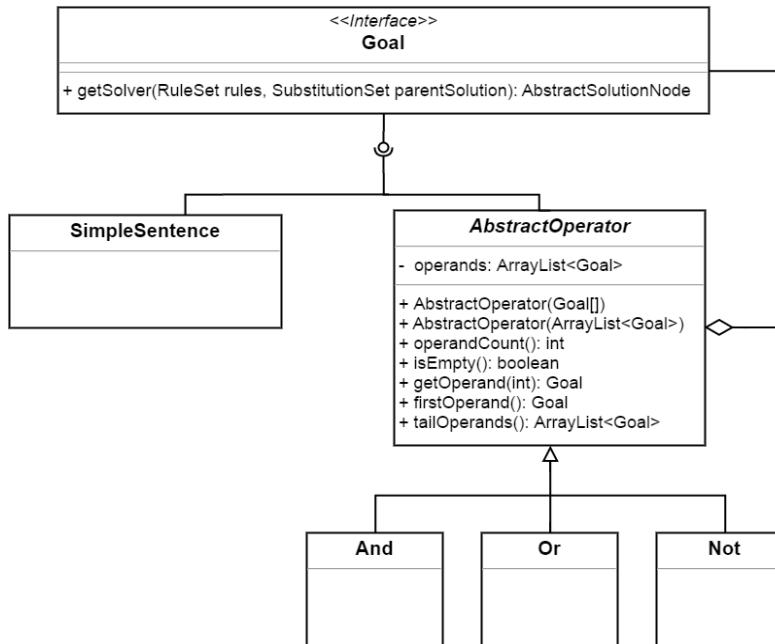
La méthode *getList* est une méthode récursive utilisée par le constructeur de la classe, la méthode *getEmptyList* retourne la liste vide et la méthode *isList* vérifie si une **SimpleSentence** est une liste.

## 2. Modélisation de la base de connaissances

La construction d'une base de connaissances revient à la capacité de construire des graphes And/Or relatifs aux formules compliquées (celles qui contiennent les opérateurs logiques  $\wedge$ ,  $\vee$ ,  $\neg$  et  $\Rightarrow$ ). Ensuite, il faut spécifier la structure d'une règle puisque c'est l'élément de base qui permet de construire la base de connaissances.

### a. Les formules

Une formule atomique est une **SimpleSentence** alors qu'une formule compliquée contient au moins un opérateur ainsi que ses opérandes. Dans le graphe And/Or une formule représente un but. Conséquemment les opérateurs et leurs opérandes expriment les conditions qui mènent à ce but. Autrement dit, une formule est un nœud du graphe And/Or. Le diagramme des classes suivant présente les classes qui entrent en jeu pour construire un nœud du graphe And/Or :



**Figure 3.5** Diagramme des classes : modélisation de nœuds opérateurs

Dans le système de résolution, un nœud du graphe And/Or représente un but à satisfaire. C'est pourquoi l'interface **Goal** définit la signature de la méthode abstraite `getSolver` qui retourne la solution du nœud courant.

**AbstractOperator** est une classe abstraite qui représente le concept formule : indépendamment de la logique de l'opérateur utilisé, une formule est composée d'opérandes dont chacun représente un but à son tour. Puisque **SimpleSentence** est une forme de formule, elle implémente l'interface **Goal**.

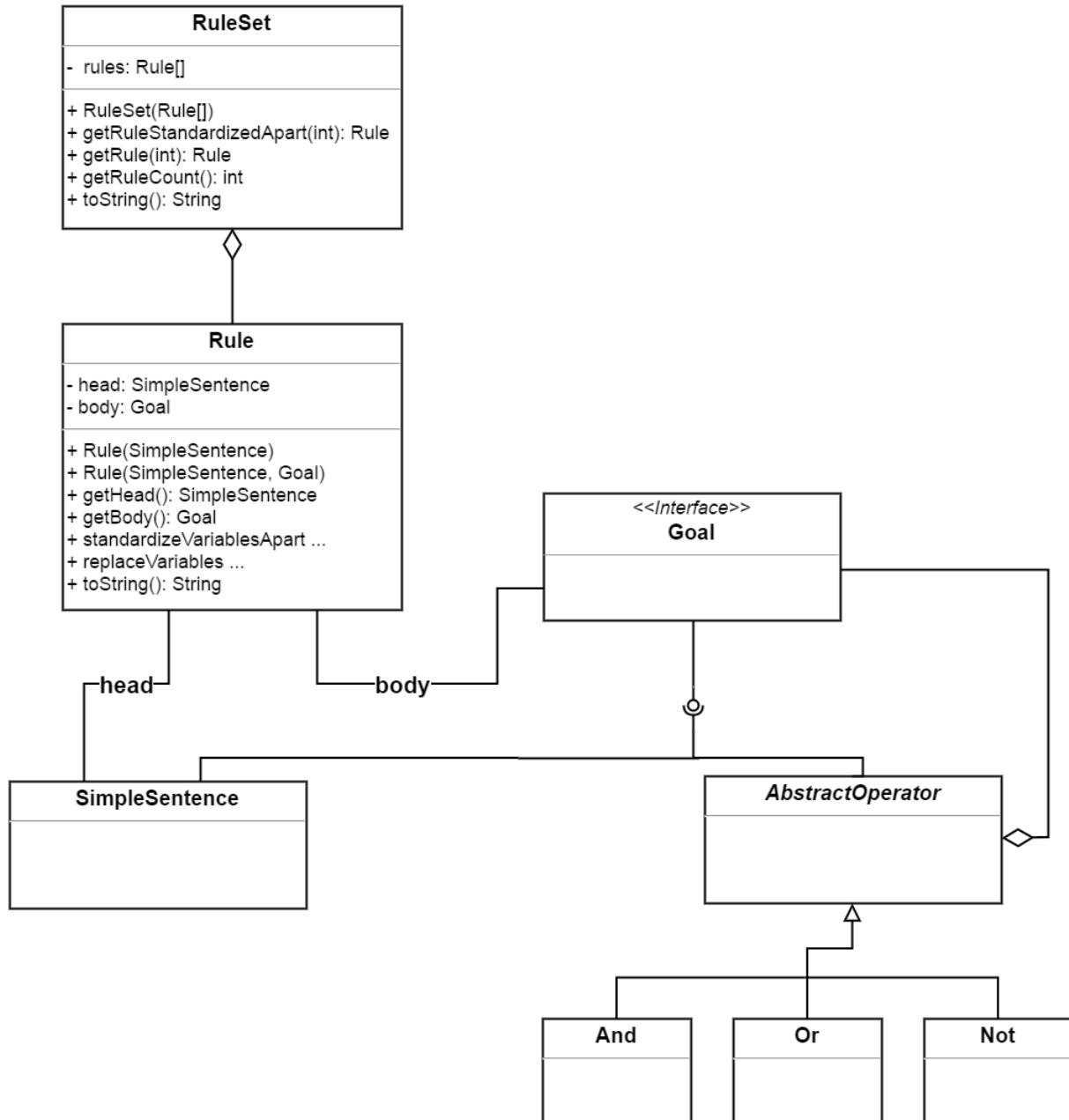
La méthode `operandCount` retourne le nombre d'opérandes dans la formule courante, la méthode `getOperand` retourne l'opérande à la position donnée, la méthode `isEmpty` vérifie si une formule est vide, la méthode `firstOperand` retourne le premier opérande et la méthode `tailOperands` retourne un tableau remplie d'opérandes de la formule sauf le premier.

Les classes **And**, **Or** et **Not** définissent la classe abstraite **AbstractOperator** selon la logique de l'opérateur en question.

La construction récursive des formules comme étant une succession de nœuds (buts, opérandes) permet la construction du graphe And/Or. Le parcours du graphe et la recherche des solutions sont effectué par la méthode `getSolver`. Comme il est mentionné dans la signature, la méthode `getSolver` se base essentiellement sur la base de connaissances (**ruleSet**) et une **substitutionSet**. Dans le paragraphe suivant nous expliquons la manière dont la base de connaissances est représentée et dans le paragraphe d'après nous exposons le moteur d'inférences qui permet de chercher une solution afin de satisfaire une requête.

## b. La structure des règles

Une règle est une structure primordiale en Prolog puisqu'une base de connaissances n'est qu'un ensemble de règles. Une règle est une formule qui admet une syntaxe particulière. Elle est composée de, au moins, une formule atomique (entête) suivie d'une formule qui représente la condition menant au but (queue). Un fait est une règle qui n'admet pas une queue. La structure règle ainsi que la structure d'une base de connaissances est représenté par le diagramme des classes suivant :



**Figure 3.6** Diagramme des classes : modélisation de base de connaissances

Le diagramme de classe ci-dessous reflète parfaitement la structure d'une règle (**Rule**) et la base de connaissances (**RuleSet**).

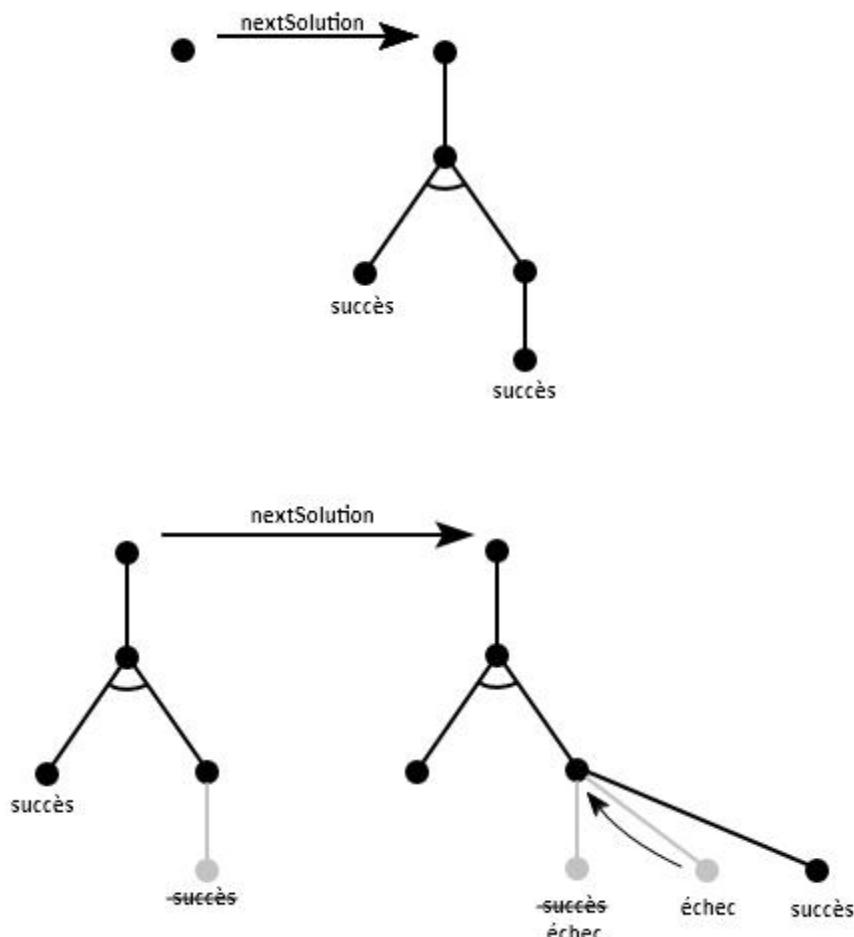
Un **Rule** est composé d'un "head" et d'un "body". Dans le cas où cette classe est instanciée seulement par une **SimpleSentence**, il s'agit d'un fait. Le "body" représente la condition qui confirme l'entête donc le "body" est lui-même un but (**Goal**). On peut extraire le "head" via la méthode *getHead* comme on peut extraire le body via la méthode *getBody*.

La méthode *getRuleStandardizedApart* qui se trouve dans la classe **RuleSet** ainsi que la méthode *standardizeVariablesApart* qui se trouve dans les classes **Rule**, **SimpleSentence**, **Variable** et **Constante** jouent un rôle important dans la recherche des solutions. Elles sont expliquées dans la section suivante.

### 3. Moteur d'inférences

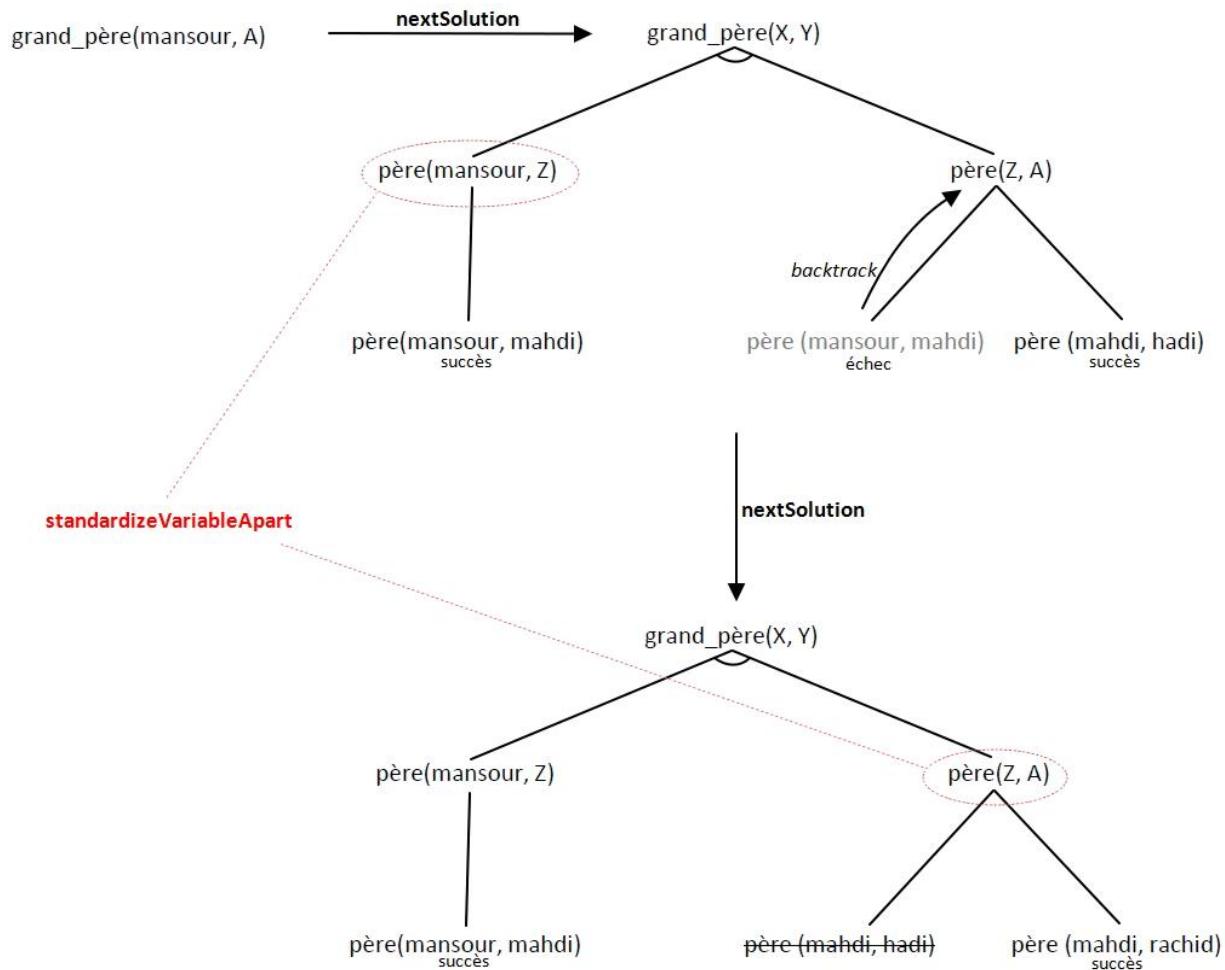
La phase finale de l'implémentation de l'algorithme d'unification est de développer le moteur d'inférences qui a l'objectif de répondre à une requête.

Notre approche consiste à construire le graphe And/Or en gardant seulement les branches qui ne se terminent pas par échec, le graphe obtenu est l'arbre de preuve relatif à une solution. L'arbre de preuve est construit grâce à la méthode *nextSolution* définie dans chaque opérateur. Pour obtenir des solutions supplémentaires, le moteur d'inférences reprend la recherche depuis le sous-but le plus récent. En cas d'échec, le moteur d'inférences retourne en arrière vers le nœud supérieur et continue la recherche. Cette recherche en "backtracking" s'arrête lorsque tout l'espace de recherche est exploré. La figure suivante représente la démarche de construction de l'arbre de preuve :



**Figure 3.7** La démarche pour construire l'arbre de preuve

La figure ci-dessous représente le fonctionnement du moteur d'inférences sur un exemple concret : recherche des solutions afin de satisfaire la requête « grand\_père (mansour, A) » :



**Figure 3.8** Exemple de construction de l'abre de preuve

On remarque que, dans un premier temps, la méthode *nextSolution* a construit l'arbre de preuve relative à la solution ( $A = \text{hadi}$ ). Dans un second temps, cette méthode a repris la recherche depuis la solution précédent pour trouver une autre ( $A = \text{rachid}$ ).

Afin de pouvoir développer un tel système, il faut savoir modéliser un nœud. Un nœud est caractérisé par un ensemble de substitutions de variables (**SubstitutionSet**) et un opérateur. Ce dernier consiste à définir la manière de parcourir le graphe And/Or en se basant sur la base de connaissances. C'est pourquoi la classe abstraite **AbstractSolutionNode** définit la signature de la méthode *nextSolution*. Les classes qui héritent de classe **AbstractSolutionNode** implémentent la méthode *nextSolution* selon la logique de l'opérateur en question. Le diagramme des classes suivant représente les structures de données et les méthodes relatives au moteur d'inférences :

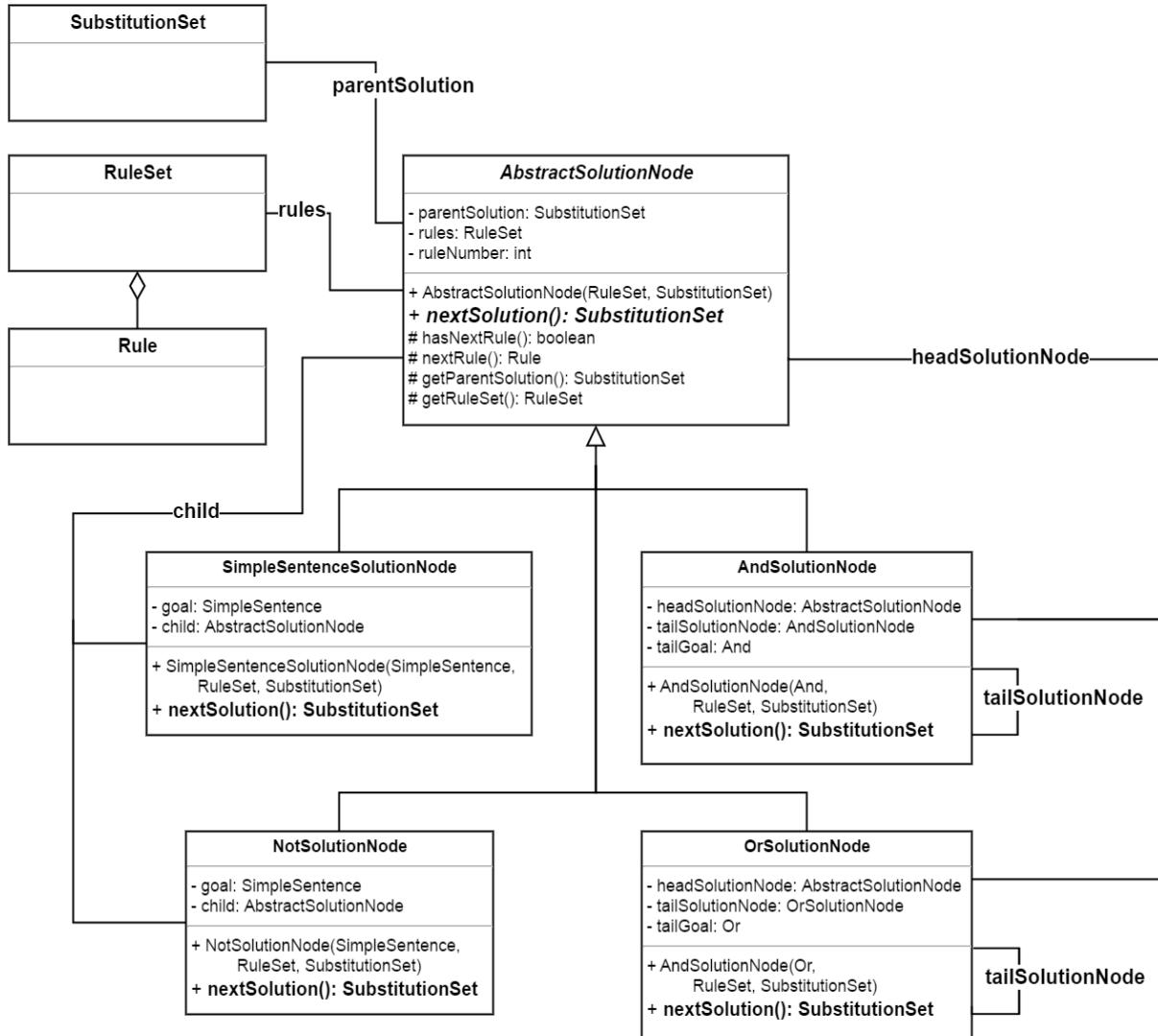


Figure 3.9 Diagramme des classes : le moteur d'inférences

La classe **AbstractSolutionNode** et ses descendants permettent de chercher une solution dans le graphe And/Or, sauvegarder l'état de recherche et reprendre la recherche depuis un sous-but.

Puisqu'à chaque passage par un nœud, le moteur d'inférences essaye de trouver des nouvelles substitutions en consultant la base de connaissances, la classe **AbstractSolutionNode** contient les deux attributs **parentSolution** (l'ensemble des substitutions trouvées) et **rules** (la base de connaissances) ainsi que quelques méthodes pour manipuler la base de connaissances telles que **hasNextRule**, **nextRule** ...

Un concept très important pour que le moteur d'inférences puisse construire des arbres de preuves est la standardisation des règles. Cette-dernière consiste à retourner une règle dans laquelle les variables sont remplacées par leurs substitutions. En cas d'utilisation d'une même règle afin de trouver une solution supplémentaire, la standardisation de règles remplace les variables par des nouvelles substitutions (le moteur d'inférences ne génère plus deux arbres de preuves identiques). Un exemple de standardisation de l'une règle (Figure 3.8)

grand\_père(X,Y) :- père(X,Z) , père(Z,Y) → grand\_père(mansour,A) :- père(mansour,Z) , père(Z,A)

La méthode permettant la standardisation des règles est *getRuleStandardizedApart* définie dans la classe **RuleSet**. Cette méthode fait appel à la méthode *standardizeVariablesApart* définie dans la classe **Rule** qui permet de remplacer les variables par leurs substitutions (existant dans une collection). Puisqu'un **Rule** est composé de "head" (**SimpleSentence**) et "body" (composition des **SimpleSentence**) alors il fait appel, à son tour, à la méthode *standardizeVariablesApart* définie dans la classe **SimpleSentence**. Cette dernière parcourt ses termes: si une constante est rencontrée, la méthode *standardizeVariablesApart* de la classe **Constante** est appliquée (elle retourne la constante courante). Si une variable est rencontrée alors la méthode *standardizeVariablesApart* définie dans la classe **Variable** retourne une collection de substitutions possibles. En cas de recherche d'une solution supplémentaire, la collection de substitutions retournée par *standardizeVariablesApart* définie dans la classe **Variable** est différente de celle qui est utilisée pour trouver la solution précédente pour que le moteur d'inférences ne génère pas deux arbres de preuves identiques (comme dans notre exemple : la collection de substitutions utilisées dans le premier arbre de preuve est {mansour/X, mahdi/Z, Y/A, **hadid/A**} alors que dans le deuxième est {mansour/X, mahdi/Z, Y/A, **rachid/A**}).

### c. La classe SimpleSentenceSolutionNode

La classe **SimpleSentenceSolutionNode** représente un nœud d'une phrase simple comme étant un but. Elle est élémentaire dans la recherche d'une solution puisque les nœuds des opérateurs sont des compositions de cette classe. L'attribut *child* représente le prochain nœud. La méthode *nextSolution* permet de trouver les substitutions des variables relatives au nœud *child*. Elle est implémentée comme suit :

La première étape est de tester si le *child* est nul ou pas. Dans le cas où le *chlid* est non nul, alors il s'agit d'une poursuite d'une recherche. Donc nous appelons la méthode *nextSolution* dans le nœud *chlid* pour chercher des nouvelles solutions dans cette branche. Si le résultat est non nul la méthode retourne cette solution.

Ci-dessous l'ensemble des règles qui définissent la méthode *nextSolution*:

- 1) Si le nœud *child* retourne nul alors la méthode initialise le *child* à nul et essaye de trouver une règle qui s'unifie avec le but. L'extraction des règles pour les tests d'unification se fait dans l'ordre d'apparition de ces règles dans la base de connaissances. Ceci est effectué grâce à une boucle "while" et à l'appel aux méthodes *hasNextRule* et *nextRule*.
- 2) Si "while" consomme toutes les règles de la base de connaissances sans trouver une unification, la méthode retourne *null* indiquant qu'il n'existe pas de solution dans cette branche.
- 3) Si un entête d'une règle s'unifie avec le but alors la méthode vérifie si cette règle a une prémissse. S'il s'agit d'un fait (une règle qui n'admet pas de prémissse) alors cette unification représente une solution partielle.
- 4) Dans le cas où la règle a une prémissse, on applique la méthode *getSolver* pour créer un nouveau nœud dont le but est cette prémissse.
- 5) La méthode *nextSolution* s'applique récursivement dans les nœuds fils jusqu'à avoir un arbre de preuve complet.

#### d. Les nœuds composées

Les classes **NotSolutionNode**, **AndSolutionNode** et **OrSolution** sont les représentants des nœuds "not", "and" et "or". Elles sont des classes composées.

La **AndSolutionNode** est composée d'un entête "head" et une queue "tail". Dans ce nœud la méthode *nextSolution* cherche une solution pour le "head". Si trouvée elle passe à la recherche de solution pour le "tail" qui doit être aussi satisfait. Le code source de la classe **AndSolutionNode** (avec des explications en commentaires) est le suivant :

```

package unification_solver;
import java.util.ArrayList;
public class AndSolutionNode extends AbstractSolutionNode {
    private AbstractSolutionNode headSolutionNode = null;
    private AndSolutionNode tailSolutionNode = null;
    private And tailGoal = null;
    /**
     * Constructeur
     */
    public AndSolutionNode(And goal, RuleSet rules, SubstitutionSet
parentSolution) {
        /*
         * super: le constructeur de la classe abstraite AbstractSolutionNode
         * rules: la base de connaissances
         * parentSolution: l'ensemble de substitutions trouvées
         */
        super(rules, parentSolution);
        /*
         * goal: l'expression à satisfaire
         * headSolutionNode: la solution (SubstitutionSet) du premier opérande
         */
        headSolutionNode = goal.firstOperand().getSolver(rules,
                                                       parentSolution);
        // tail: tableau d'opérandes restants
        ArrayList<Goal> tail = goal.tailOperands();
        // tant qu'il existe d'opérandes: constructions récursives des nœuds
        if (tail.isEmpty() != true)
            tailGoal = new And(tail);
    }
    /**
     * la méthode nextSolution
     */
    public SubstitutionSet nextSolution() {
        SubstitutionSet solution;
        // recherche de solutions pour la queue
        if (tailSolutionNode != null) {
            solution = tailSolutionNode.nextSolution();
            if (solution != null)
                return solution;
        }
        // tous les entêtes doivent être satisfaits
        while ((solution = headSolutionNode.nextSolution()) != null) {
            if (tailGoal == null) // n'existe pas d'autres nœuds
                return solution;
            else { // le reste des nœuds doivent être satisfaits
                tailSolutionNode = new AndSolutionNode(tailGoal,
                                                       getRuleSet(), solution);
            }
        }
    }
}

```

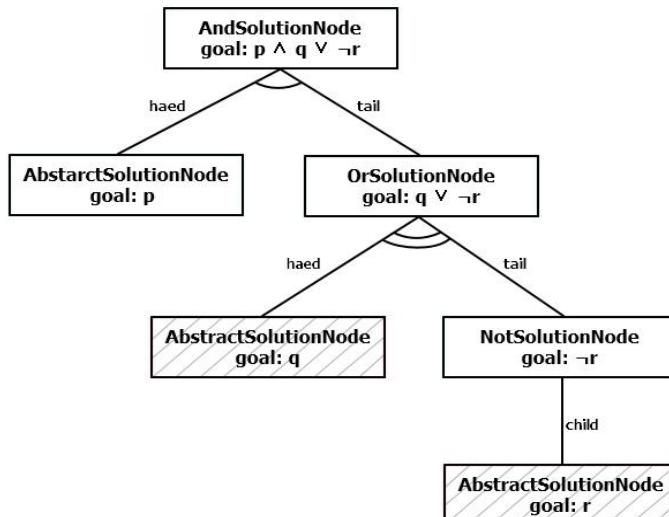
```

        SubstitutionSet tailSolution =
            tailSolutionNode.nextSolution();
        if (tailSolution != null)
            return tailSolution;
    }
}
return null; // pas de solutions
}
    
```

La classe **OrSolutionNode** est aussi composée de *head* est de *tail* mais il suffit que la méthode *nextSolution* trouve une seule solution pour un opérande, la branche sera satisfaite.

La classe **notSolutionNode** représente un nœud unaire qui est satisfait lorsque la méthode *nextSolution* ne trouve pas de solution.

La figure suivante représente une portion du graphe And/Or relative au but " $(p \wedge q) \vee \neg r$ ". (Les nœuds hachurés ne sont pas satisfaits).



**Figure 3.10** Exemple de construction de graphe And/Or

Le moteur d'inférences crée un nœud "and" composé d'un "head"  $p$  et un "tail"  $q \vee \neg r$ . Les opérandes du nœud "and" doivent être satisfaits. Le "tail" représente un nœud or. Puisque l'opérande  $q$  du nœud or n'admet pas de solutions, le moteur d'inférences passe au deuxième opérande  $\neg r$ . Ce dernier est un nœud not qui est satisfait parce que  $r$  n'admet pas de solutions.

## Conclusion

L'algorithme d'unification proposé représente le moteur de notre interpréteur d'expression logique en langue arabe. Nous pouvons ainsi définir une base de connaissances qui n'est qu'un ensemble de clauses de Horn. Puis, l'interroger grâce au moteur d'inférences qui recourt à la notion d'unification et de chaînage arrière afin de trouver les solutions relatives à une requête.

Dans le chapitre suivant nous étendons cet algorithme pour qu'il puisse gérer en plus de la conjonction, la disjonction et la négation, les expressions arithmétiques et les opérations sur les termes.

## Chapitre4. Évaluation des expressions arithmétiques

---

Dans le chapitre précédent nous avons détaillé les éléments relatifs à la conception pour développer l'algorithme d'unification en utilisant le langage de programmation orienté-objet Java.

Dans ce chapitre nous étendons l'algorithme d'unification pour qu'on puisse réaliser des opérations sur les termes. Ceci nécessite, entre autres, l'évaluation des expressions arithmétiques, logiques et relationnelles.

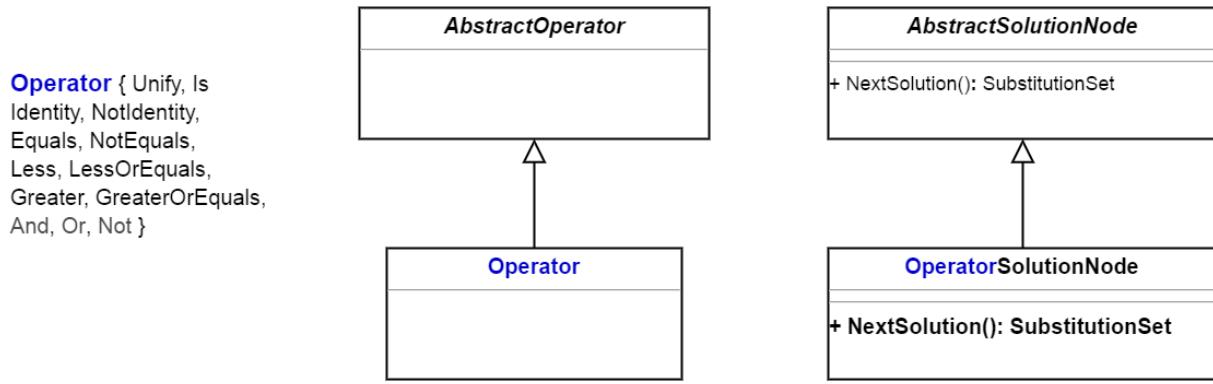
### I. Les opérateurs

Notre objectif est d'étendre l'algorithme d'unification pour qu'il puisse manipuler des nœuds-opérateurs autres que "And", "Or" et "Not" (vus dans le chapitre précédent). Le tableau suivant représente la liste des opérateurs définie par notre application :

Opérateur	Symbol	Signification
Unification	=	Réussi si l'unification de ses arguments est possible
Identité	==	Réussi si ses arguments sont identiques
Négation de l'Identité	\==	Réussi si ses arguments ne sont pas identiques
Egalité	=:=	Réussi si les évaluations de ses arguments sont égales
Inégalité	=\=	Réussi si les évaluations de ses arguments ne sont pas égales
Inférieur	<	Réussi lorsque l'évaluation du premier argument est inférieure à l'évaluation du deuxième
Inférieur ou égale	<=	Réussi lorsque l'évaluation du premier argument est inférieure ou égale à l'évaluation du deuxième
Supérieur	>	Réussi lorsque l'évaluation du premier argument est supérieure à l'évaluation du deuxième
Supérieur ou égale	>=	Réussi lorsque l'évaluation du premier argument est supérieure ou égale à l'évaluation du deuxième
Evaluation des expressions	is	Unifie le premier argument avec l'évaluation de deuxième argument

Tableau 4.1 Opérateurs supplémentaires

Grâce à l'architecture de l'algorithme d'unifications proposée, la définition des nouveaux nœuds est simple. Pour chacun des nœuds relatifs à un opérateur, il est impératif de créer deux classes : la première hérite de la classe abstraite **AbstractOperator** puisque cette dernière représente le squelette d'un opérateur. La deuxième est utilisée par le moteur d'inférences. Elle hérite donc de la classe abstraite **AbstractSolutionNode** et implémente la méthode *NextSolution*. Les classes des opérateurs ainsi que leurs **SolutionNode** sont présentés dans le diagramme de classe suivant :



**Figure 4.1** Diagramme des classes \_ les nœuds des opérateurs supplémentaires

Avec cette architecture l'algorithme d'unification ne se limite plus à un graphe And/Or. Il crée un arbre plus général (le graphe And/Or est un cas particulier). Le moteur d'inférences parcourt l'arbre créé et cherche des solutions en fonction de l'opérateur qui a défini le nœud.

Les implémentations de la méthode `NextSolution` reflètent exactement la définition de l'opérateur en question :

- Dans **UnifySolutionNode** : la méthode `NextSolution` retourne la `substitutionSet parentSolution` augmentée par l'unification du premier argument avec le deuxième argument. En cas d'échec d'unification, elle retourne `null`.
- Dans **IdentitySolutionNode** : si les deux arguments sont identiques, la méthode `NextSolution` retourne `parentSolution (True)` sinon elle retourne `null`.
- Dans **NotIdentity** : si les deux arguments ne sont pas identiques, la méthode `NextSolution` retourne `parentSolution (True)` sinon elle retourne `null`.
- Dans **EqualsSolutionNode**, **NotEqualsSolutionNode**, **LessSolutionNode**, **LessOrEqualsSolutionNode**, **GreaterSolutionNode** et **GreaterOrEqualsSolutionNode** les méthodes `nextSolution` évaluent tout d'abord les arguments de ces opérateurs. Puis elle retourne `parentSolution` si l'expression s'évalue à `True` sinon elle retourne `null` si l'expression s'évalue à `False`.
- Dans **IsSolutionNode**, la méthode `NextSolution` retourne la `substitutionSet parentSolution` augmentée par l'unification du premier argument avec l'évaluation de deuxième argument.

L'évaluation des expressions arithmétique ne fait pas partie de l'algorithme d'unification. Elle est traitée dans un projet appart (expliqué dans la section suivante).

La figure suivante représente les déductions effectuées par le moteur d'inférences pour satisfaire la requête :  $X \text{ is } 3+2$ ,  $Y = X+1$

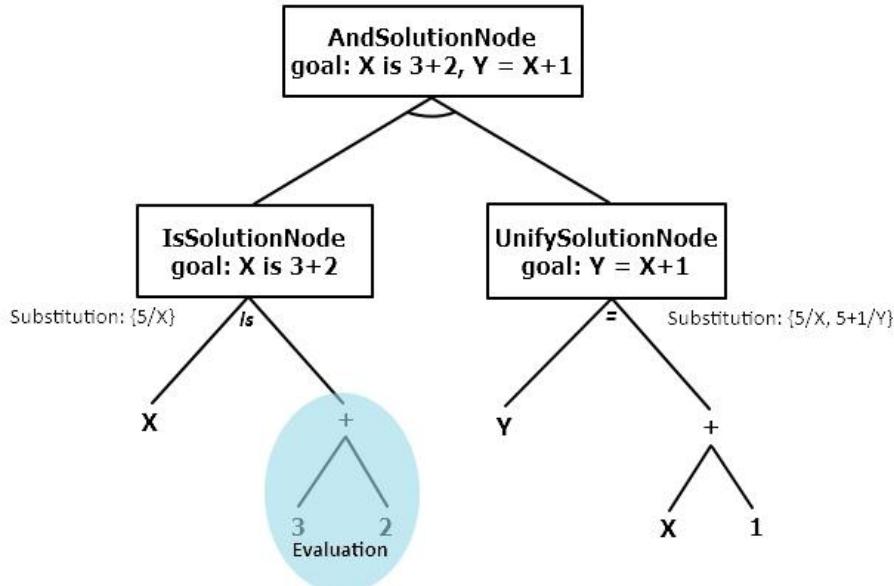


Figure 4.2 Exemple de construction de graphe de déductions

Sans besoin d'une base de connaissances, le moteur d'inférences trouve les substitutions qui satisfont la requête:  $X = 5$        $Y = 5+1$ .

## II. Démarche pour l'évaluation des expressions

Les classes `IsSolutionNode`, `EqualsSolutionNode`, `NotEqualsSolutionNode`, `LessSolutionNode`, `LessOrEqualsSolutionNode`, `GreaterSolutionNode` et `GreaterOrEqualsSolutionNode` nécessitent l'évaluation des expressions arithmétiques, logiques ou bien relationnelles (non pas les expressions arithmétiques seulement comme dans ISO Prolog). Pour ce fait, nous avons besoin d'évaluation de ces expressions.

L'évaluation des expressions supporte les types : entier, double et booléen ainsi que les opérateurs basiques pour chacun de ces types. La méthode `evaluate` permet de solliciter le composant d'évaluation.

### 1. Conversion d'une expression en notation infixé vers sa notation postfixé

Naturellement, les expressions arithmétiques, logiques ou relationnelles s'écrivent en notation infixé alors que dans notre approche d'évaluation, on n'accepte que les expressions en notation postfixé (ou polonaise inversée). Cette notation n'est pas ambiguë et facilite grandement le calcul qui se base alors sur une simple pile.

La première étape consiste à convertir l'expression de sa notation infixé à sa notation postfixé. Par exemple :

L'expression infixée :	$2 + 3 * 4$
S'écrit en notation postfixé :	$2 3 4 * +$ (l'opérateur * est prioritaire)

La fonction `infixToPostfix` permet de convertir une expression en notation infixé vers sa notation en postfixé en prenant en compte la priorité des opérateurs. Elle prend en paramètre une chaîne de caractères contenant l'expression à convertir et retourne une liste qui correspond à la notation polonaise inversée. Les priorités des opérateurs sont définies de

façon que celui qui admet une valeur de priorité minimale soit plus prioritaire. Les valeurs des priorités ont été choisies pour que notre application évalue les expressions tout comme l'évaluation en ISO Prolog. Le tableau suivant représente tous les opérateurs supportés ainsi que leurs priorités et le type des opérandes :

Opérateur	Signification	Type des opérandes	Priorité
<i>Opérateurs arithmétiques</i>			
+	addition	entier, double	3
-	soustraction		3
*	multiplication		2
/	division		2
%	division entière		2
<i>Opérateurs relationnels</i>			
==	égalité	entier, double, booléen	5
=!	inégalité		5
>	inférieur		4
=>	inférieur ou égale		4
<	supérieur		4
=<	Supérieur ou égale		4
<i>Opérateurs logique</i>			
&	et logique	booléen	6
	ou logique		7
!	Non logique		1

Tableau 4.2 Opérateurs arithmétique, relationnels et logiques

Les opérateurs relationnels =!, >, =>, <, =< sont inversés car dans notre projet, les expressions s'écrivent en langue arabe (de droite à gauche). Par exemple inférieur dans une expression arithmétique en arabe s'écrit >.

Lorsqu'une expression contient plusieurs opérateurs, la priorité des opérateurs détermine l'ordre dans lequel chacun des opérateurs est évalué. Les calculs contenus entre parenthèses sont toujours prioritaires.

### a. Algorithme de la fonction infixToPostfix

Dans ce paragraphe nous présentons l'algorithme de la fonction *infixToPostfix* permettant de convertir une expression de sa notation infixe vers sa notation en postfixe [32]:

```

fonction infixToPostfix (String infix): List de token
    List postfix;
    String token;
    Stack operatorStack; //pile
    String tokens[] = getLexemes*(infix);

    pour i de 1 à longueur(tokens)

        cas (1) : tokens[i] == parenthèse ouvrante:
            Empiler tokens[i] dans operatorStack;

        cas (2) : token[i] == parenthèse fermante:
            tantque (operatorStack != vide)
                ou (token != parenthèse ouvrant)
                    token = sommet de operatorStack;

```

```

        si(token = parenthèse ouvrant) alors
            break;
        fin si
        Ajouter token à postfix;
    fin tantque

cas (3) : token[i] est un opérateur:
    tokenPriority = getOperatorPriority**(tokens[i]);
    tantque (operatorStack != vide)
        et
        (getOperatorPriority(sommet(operatorStack))
            <= tokenPriority )
        token = sommet de operatorStack;
        si (token != parenthèse ouvrant) alors
            Ajouter token à postfix;
        fin si
    fin tantque
    Empiler tokens[i] dans operatorStack;

cas (4) : autrement:
    Ajouter tokens[i] à postfix;

fin pour

tantque (operatorStack!= vide)
    token = sommet de operatorStack;
    si (token != parenthèse ouvrant) alors
        Ajouter token à postfix;
    fin si
fin tantque

retourner postfixe

* getLexemes est une fonction permettant de retourner un tableau rempli
de lexèmes de l'expression en notation infixe

** getOperatorPriority est une fonction permettant de retourner l'ordre
de priorité de l'opérateur passé en paramètre

```

### b. Trace d'exécution

Dans cette section, nous allons présenter la trace d'exécution relative à l'application de la fonction *infixToPostfix* sur l'expression :  $2 > 5 \& (3 + 4 != 6)$

infix =  $2 > 5 \& (3 + 4 != 6)$   
 postfix = vide  
 operatorStack = vide  
 tokens = [2, >, 5, &, (, 3, +, 4, !=, 6, )]

<b>Itération 1</b>	tokens[1] = 2 postfix = 2	cas (4)
<b>Itération 2</b>	tokens[2] = > operatorStack = $\boxed{\phantom{0}}$	cas (3)
<b>Itération 3</b>	tokens[3] = 5 postfix = 2->5	cas (4)

<b>Itération 4</b>	tokenPriority = 6 Itération 4.1 : token = > operatorStack = vide postfix = 2->5->> operatorStack = [&]	cas (3)
<b>Itération 5</b>	tokens[5] = ( operatorStack = [& ()]	cas (1)
<b>Itération 6</b>	tokens[6] = 3 postfix = 2->5->>->3	cas (4)
<b>Itération 7</b>	tokens[7] = + tokenPriority = 3 operatorStack = [& (+]	cas (3)
<b>Itération 8</b>	tokens[8] = 4 postfix = 2 -> 5 -> -> 3 -> 4	cas (4)
<b>Itération 9</b>	tokens[9] = ==! tokenPriority = 5 Itération 9.1: token = + operatorStack = [& ()] postfix = 2->5->>->3->4->+ Itération 9.2: token = ( operatorStack = [& ] operatorStack = [& ==!]	cas (3)
<b>Itération 10</b>	tokens[10] = 6 postfix = 2->5->>->3->4->+->6	cas (4)
<b>Itération 11</b>	tokens[11] = ) Itération 11.1: token = ==! operatorStack = [& ] postfix = 2->5->>->3->4->+->6->==! Itération 11.2: token = & operatorStack = vide postfix = 2->5->>->3->4->+->6->==!->&	cas (2)

Tableau 4.3 Trace d'exécution : application de la méthode infixToPostfix

## 2. Évaluation des expressions en notation postfixe

Le principe d'évaluation d'une expression arithmétique est simple. La fonction *evaluate* suivante décrit la démarche pour évaluer une liste de jetons (tokens) représentant une expression en notation polonaise inversée. L'Algorithme de la fonction *evaluate* est le suivant [39]:

```

fonction evaluate (List postfix) : Value

    Stack stack;
    Value token, value1, value2, result;

    pour i de 1 à longeur(postfix)
        token = postix(i)

        cas : token est un type de donnée (entier, double ou booléen) :
            Empiler token dans stack;

        cas : token == opérateur
            si(token == opérateur unaire)alors
                value1 = sommet de stack      //dépiler
                result = application de token sur value1
                Empiler result dans stack;

            sinon (token == opérateur binaire)alors
                value1 = sommet de stack      //dépiler
                value2 = sommet de stack      //dépiler
                result = application de token sur value1 et value2
                Empiler result dans stack;
            fin si
        fin pour

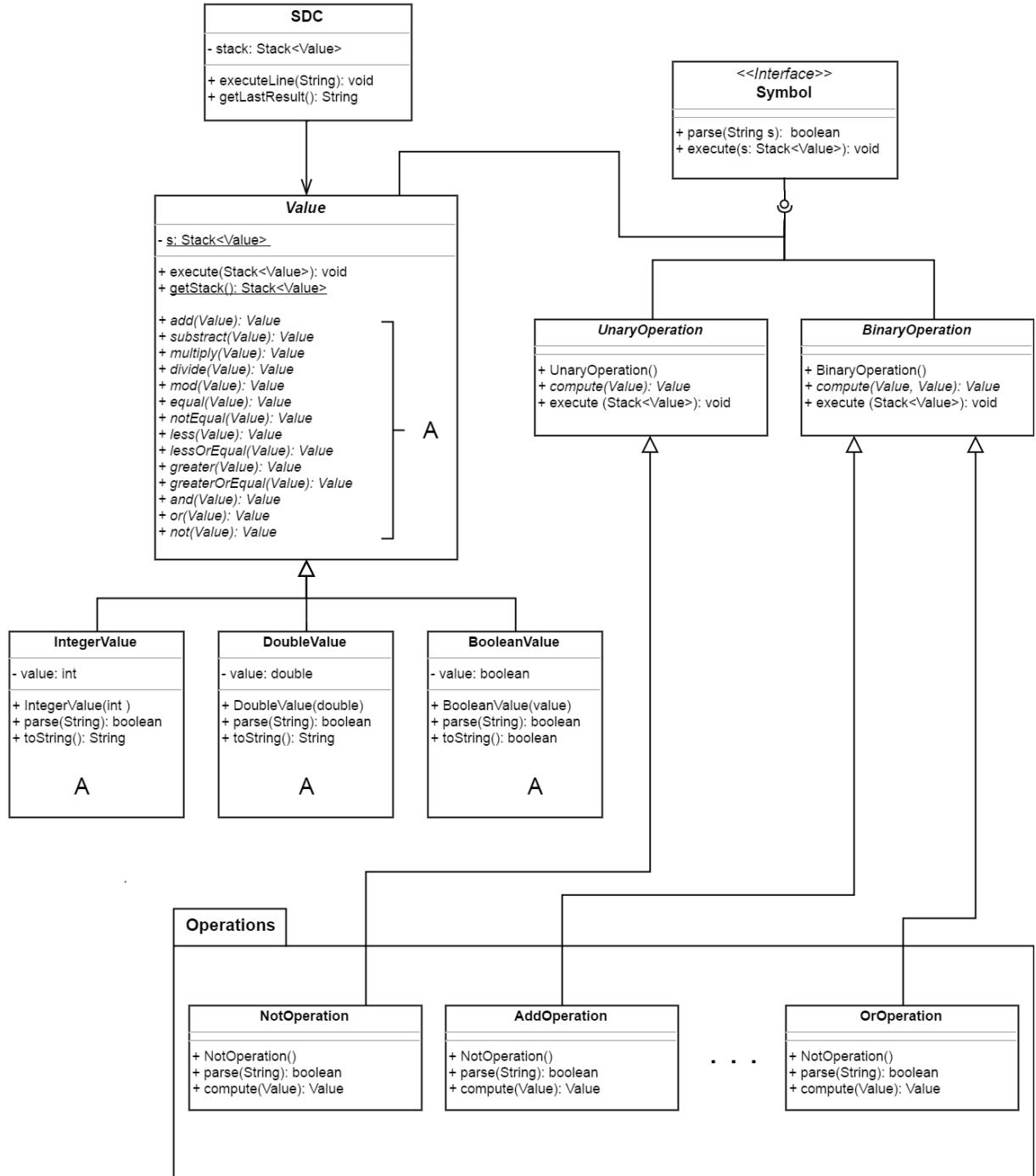
        retourner sommet de stack;
    
```

La fonction *evaluate* évalue correctement l'expression passée en paramètre, si l'expression est bien formulée. Pour implémenter cette fonction en java, il faut :

- 1) Concevoir une pile capable d'empiler les trois types de données : entier, double et boolean)
- 2) Effectuer des calculs élémentaires en fonction d'un opérateur et ses opérandes
- 3) Lancer une exception en cas de :
  - recentrer un symbole inconnu
  - opérandes non compatibles avec l'opérateur
  - expression mal formulée

## Chapitre 4. Évaluation des expressions arithmétiques

L'architecture de notre module d'évaluation des expressions arithmétique est présentée dans le diagramme de classe suivant :



**Figure 4.3** Diagramme des classes : l'évaluation des expressions

Afin de pouvoir évaluer une expression en notation polonaise inversée, la classe **SDC** utilise une pile. Cette dernière sera remplie par des valeurs qui peuvent être entiers, doubles ou bien des booléens d'où la classe abstraite **Value** et ses descendants (**IntegerValue**, **DoubleValue** et **BooleanValue**).

La classe abstraite **Value** contient l'attribut statique **s** du type **Stack** (la pile), la méthode **getStack**, La méthode **execute** qui permet d'empiler une valeur dans la pile **s** et les signatures des méthodes qui sont définies dans ses descendants.

Les classes **IntegerValue**, **DoubleValue** et **BooleanValue** implémentent les méthodes qui correspondent aux opérations dont les signatures sont définies dans la classe mère **Value** (bloc de méthodes A). Chacune de ces méthodes est implémentée selon les types des opérandes. Par exemple la méthode *add* dans **IntegerValue** est implémentée comme suite :

```
public Value add(Value v) throws IncompatibleTypeException {
    if(v instanceof IntegerValue)
        return new IntegerValue(((IntegerValue) v).value + this.value);
    else if (v instanceof DoubleValue)
        return new DoubleValue(((DoubleValue) v).getValue() + this.value);
    else // v instanceof BooleanValue
        throw new IncompatibleTypeException();
}
```

Si la valeur *v* reçue en paramètre est de type **IntegerValue**, on retourne la somme de la valeur courante avec *v* (la somme est de type **IntegerValue**). De même on peut additionner un entier avec un double si *v* est de type **DoubleValue** (la somme est de type **DoubleValue**). Autrement, la méthode lance l'exception d'incompatibilité (**IncompatibleTypeException**) parce qu'on ne peut pas additionner une valeur de type **IntegerValue** avec une autre de type **BooleanValue**.

Les opérations logiques (et, ou est non) sont effectuées dans **BooleanValue** lorsque la valeur reçue en paramètre est aussi de type **BooleanValue**. Les opérateurs relationnels tel qu'inférieur, supérieur... nécessitent des opérandes de type numérique (**IntegerValue** ou **DoubleValue**) et retournent un **BooleanValue**.

Tout symbole lu est consulté par la méthode *parse* qui a comme objectif d'accepter seulement les entiers, les doubles, les booléens et les symboles des opérateurs. Puis en fonction de ce symbole, on manipule la pile *s*. c'est pour cette raison l'interface **Symbol** contient la signature de la méthode *parse*.

Les classes qui héritent de **Value** implémentent la méthode *parse* comme suite :

- 1) Dans **IntegerValue** : on accepte les entiers.
- 2) Dans **DoubleValue** : on accepte les doubles.
- 3) Dans **BooleanValue** : on accepte صحيح (vrai en arabe) ou خطأ (faux en arabe).

Ainsi, chaque opération est représentée par un symbole bien déterminé : **AddOperation** est représentée par +, **AndOperation** est représenté par & ...

Comme dans l'algorithme de la fonction *evaluate*, si le symbole lu est une valeur on l'empile dans la pile *s* (application de la méthode *execute* de la classe **Value**). Si le symbole lu est un opérateur binaire, la méthode *execute* de la classe **BinaryOperation** dépile les deux valeurs et empile le résultat de l'opération élémentaire effectué en utilisant la méthode *compute*. Voici le code source de la méthode *execute* de **BinaryOperation** :

```
public void execute(Stack<Value> s) throws IncompatibleTypeException {
    Value v1 = s.pop();
    Value v2 = s.pop();
    s.push(compute(v1, v2));
}
```

Chacune des classes qui héritent de **BinaryOperation** implémente la méthode *compute* selon le symbole de l'opérateur lu. Par exemple *compute* dans **addOperation** est implémentée comme suite :

```
public Value compute(Value v1, Value v2) throws IncompatibleTypeException {
    return v1.add(v2);
}
```

Si le symbole lu est un opérateur unaire, il s'agit donc d'une opération unaire où on dépile une valeur, lui appliquer l'opération unaire et empiler le résultat.

En plus de la détection des symboles inconnus et les opérandes non compatibles, cet implémentation détecte les expressions malformées : lorsque l'algorithme compte dépiler alors que la pile *s* est vide ou bien lorsque la taille de la pile *s* n'est pas égal à un (après parcours de toute l'expression à évaluer). Le résultat de l'évaluation est le sommet de la pile *s* (qui doit contenir une seule valeur).

### III. Exécution des requêtes

L'algorithme d'unification implémenté représente le back-end de notre langage de programmation. En effet on peut définir des bases de connaissances et les interroger mais ces programmes sont écrits en dur. Le tableau suivant représente la définition du prédictat sum (permettant de calculer la somme des éléments d'une liste) écrit en ISO Prolog, Prolog Arabe et en Back-end :

Base de connaissances	Requête
Arabic Prolog	
جمع([],0). جمع([x xs],n):- جمع(xs,n), جمع([x],n).	<-- جمع([1,2,3],N). 6 = N
ISO Prolog	
sum([],0). sum([H T],N):- sum(T,X), N is H+X.	?- sum([1,2,3], N). N = 6
Back-end	
Constant sum = new Constant("جـمـع");  Variable H = new Variable("هـ"), T = new Variable("تـ"), N = new Variable("نـ"), X = new Variable("خـ");  RuleSet rules = new RuleSet(  new Rule(new SimpleSentence(sum, ListProlog. getEmptyList(), new Constant("0"))),  new Rule(new SimpleSentence(sum, new SimpleSentence(ListProlog.getFunctor(), H, T), N), new And(new SimpleSentence(sum, T, X), new Is(new SimpleSentence(N, Converter.infixToTree("هـ + تـ", variables))))));	solve(new SimpleSentence(sum, new ListProlog( new Constant("1"), new Constant("2"), new Constant("3"), ListProlog.getEmptyList()). getList(), N), rules);  6 = N

Tableau 4.4 Le prédictat sum en ISO Prolog, Prolog en langue arabe, Back-end

On remarque que la définition d'un simple prédicat et une requête écrits en Back-end est relativement difficile : il faut déclarer toutes les variables et les constantes, construction des objets (**RulesSet**, **Rule**, **SimpleSentence** ...), convertir les expressions en notation de l'arbre via la méthode *infixToTree*, appeler la fonction *solve* pour interroger la base de connaissances... d'où la nécessité d'implémenter une couche logicielle supérieure permettant de convertir un programme Prolog écrit en langue arabe (respectant la syntaxe du Prolog en langue arabe) vers les objets qui lui sont équivalents pour qu'il puisse être interprété par notre logiciel.

### Conclusion

Dans ce chapitre, nous avons enrichi notre algorithme d'unification par la définition de nouvelles **SolutionsNode** relatives aux opérations sur termes. Nous avons réalisé un composant pour l'évaluation des expressions arithmétiques, logiques et relationnelles.

Vers la fin de ce chapitre nous avons démontré le besoin d'implémenter une couche logicielle supérieure permettant de convertir un programme Prolog écrit en langue arabe vers son équivalent que notre logiciel puisse l'interpréter. Ce module s'appelle le Front-end qui sera l'objet du chapitre suivant en plus de la présentation de la totalité des fonctionnalités offertes par "Mujeed" : l'environnement de programmation logique en langue arabe.

## Chapitre 5. Mujeed : environnement de programmation logique en langue arabe

---

Nous avons choisi le nom Mujeed (مجيد) pour notre environnement de programmation logique. Le nom « Mujeed » en langue arabe signifie « le compétent » en français : qui connaît parfaitement une question, une matière, un domaine.

Ce chapitre est composé de trois parties : la première est consacrée au compilateur qui représente une couche logicielle permettant de générer les structures de données et d'appeler les méthodes d'évaluation et de résolution. La deuxième partie est réservée à la description de l'interface graphique de Mujeed ainsi que ses fonctionnalités. La troisième partie est consacrée à présenter des exemples de programmes supportés par Mujeed.

### I. Le Front-end

Comme tout environnement de langage de programmation, Mujeed est constitué de deux parties distinctes : le noyau (l'algorithme d'unification) et le Front-end désignant la partie frontale.

Le développement du Front-end consiste à implémenter une fonction qui, à partir d'une base de connaissances et une requête écrites en Prolog en langue arabe, génère les structures de données et fait appel aux méthodes d'évaluation et de résolution. Par exemple la phase simple suivante en Front-end :

ابن (؟, جعفر).

Se traduit en Back-end comme suite :

```
new SimpleSentence(new Constant("ابن"), new Variable("؟"), new Constant("جعفر"))
```

La transformation du code source écrit en Front-end vers son équivalent ne se fait que par la définition d'une grammaire qui reconnaît le langage "Arabic Prolog". Cette dernière est une grammaire hors-contexte. Son implémentation revient à développer un analyseur syntaxique LL (lit l'entrée de gauche à droite et produit une dérivation gauche) ou LR (lit l'entrée de gauche à droite et produit une dérivation droite).

On désigne par grammaire hors-contexte, un 4-uplet  $\langle N, \Sigma, P, S \rangle$  :

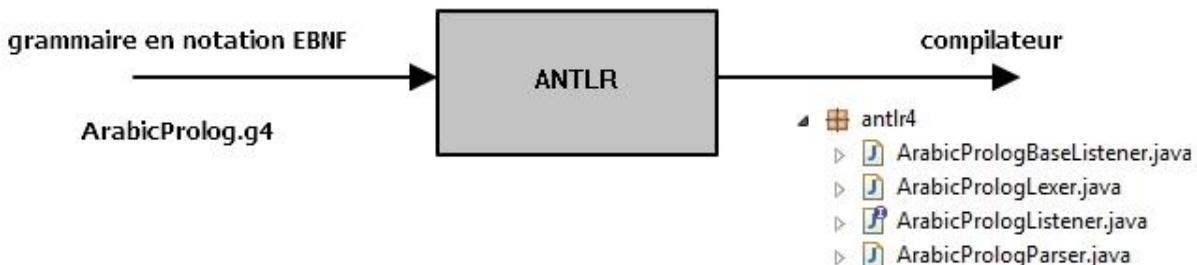
- $N$  est l'ensemble de symboles non terminaux
- $\Sigma$  est l'ensemble de symboles terminaux
- $P$  est un sous ensemble fini de :  $N \times (N \cup \Sigma)^*$   
un élément  $(\alpha, \beta)$  de  $P$ , que l'on note  $\alpha \rightarrow \beta$  est appelé une règle de production
- $S$  est un élément de  $N$  appelé l'axiome de la grammaire

Une notation permettant de décrire les règles syntaxiques pour les grammaires non contextuelles est le BNF (Backus-Naur Form) ou le EBNF (Extended Backus-Naur Form). Dans la section suivante nous donnons le EBNF du langage de programmation Arabic Prolog.

L'analyseur syntaxique pour le langage de programmation Arabic Prolog a été automatiquement générer à partir de sa EBNF. Il existe des frameworks qui génèrent des compilateurs en langage Java dont les principaux représentants :

- ANTLR (ANother Tool for Language Recognition) : une solution libre et gratuite utilisant une analyse LL(\*), c'est le compilateur le plus répondu [33].
- JavaCC (Java Compiler Compiler) : une solution libre et gratuite utilisant une analyse LL(\*), moins connue du grand public [34].
- SableCC: une solution libre et gratuite utilisant une analyse LR(\*) [35].
- JFlex + CUP: JFlex désigne un analyseur lexical dans la plupart de temps couplé avec l'analyseur syntaxique CUP (Construction of Useful Parsers) qui utilise une analyse LR(\*) [36].

Nous avons choisi ANTLR pour ses performances et ses fonctionnalités qui répondent parfaitement à nos besoins comme la génération des arbres syntaxiques abstraits et de l'objet de parcours de l'arbre (**ParseTreeWalker**). ANTLR prend en entrée une grammaire définissant un langage en notation EBNF et produit compilateur dont le code est écrit dans l'un des nombreux langages de programmation cibles (Java, Python, Ruby, C, C++, C#) reconnaissant ce langage. La figure suivante décrit la génération du compilateur pour Prolog en langue arabe :



**Figure 5.1** Le compilateur généré par le framework ANTLR

## 1. Le fichier ArabicProlog.g4

Nous proposons une nouvelle syntaxe de Prolog en langue arabe. La nouvelle syntaxe Prolog est d'une part, proche du standard prolog. D'une autre part, elle est en harmonie avec les spécificités de l'écriture arabe. Le tableau suivant représente l'équivalent de ISO Prolog en langue arabe :

	<b>ISO Prolog</b>	<b>Arabic Prolog</b>
Commentaire	% /* */	% \* *\`
Variable	commence par une majuscule	commence par "?"
Anonymes		-

Constante	commence par une minuscule	chaîne de caractère
Entier	-1, 0, 1, 2, 3 ...	
Double	-1.0, 0.5, 2.5 ...	
Booléen	true, false	صحيح، خطأ
Liste	[a, b, c]	[أ, ب، ج]
	[H  T]	[هـ   تـ]
	[X, Y  Z]	[أـ، بـ   جـ]
Requête	?-	<
Implication	:-	-:
Unification	=	=
Evaluation	is	-->
Conjonction	,	&
Disjonction	;	
Coupe	!	!
Négation	\+	+/-
Egalité	==, =:=	==
Inégalité	\==, =\=	=!
Inférieur	@<, <	>
Inférieur ou égal	@=<, =<	=>
Supérieur	@>, >	<
Supérieur ou égal	@>=, >=	=<
Addition	+	+
Soustraction	-	-
Multiplication	*	*
Division	/	/

Tableau 5.1 Syntaxe prolog en langue arabe

La première étape consiste à définir la grammaire en notation EBNF du Prolog en langue arabe dans un fichier dont l'extension est g4 (la quatrième version de ANTLR).

La grammaire ne doit pas contenir des symboles récursifs à gauche. Un symbole non terminal A est dit récursive à gauche si  $A \rightarrow AB$  (récuriosité gauche directe) ou bien  $A \rightarrow BC$  et  $B \rightarrow AD$  (récuriosité gauche indirecte). Heureusement, tout langage hors-contexte peut être engendré par une grammaire hors-contexte non récursive à gauche en appliquant le système de réécriture suivant :

$$A \rightarrow Ab \mid a \Leftrightarrow \begin{cases} A \rightarrow a \mid aA' \\ A' \rightarrow bA' \mid \epsilon \end{cases}$$

La totalité du EBNF Prolog est donnée dans l'annexe A : contenu du fichier ArabicProlog.g4). Une syntaxe plus réduite est la suivante (les opérateurs ne sont pas considérer) :

```

<program> = <clause list> <query> | <query>
<clause list> = <clause> | <clause list> <clause>
<clause> = <predicate> | <predicate list> :- <predicate>
<predicate list> = <predicate> | <predicate list> , <predicate>
<predicate> = <constant> | <constant> ( <term list> )
<term list> = <term> | <term list> , <term>
<term> = <numeral> | <constant> | <variable> | <structure>
    
```

```

<structure> = <constant> ( <term list> )
<query> = <predicate list> </
<constant> = <arabic char list>
<arabic char list> = <arabic char> <arabic char list> | <arabic char>
<variable> = <arabic char> <arabic char list> | <arabic char> ?
<arabic char> = ا | ب | ت | ... | ه | و | ي
<anonymous> =
<numeral> = <digit> | <numeral> <digit>
<double> = <numeral> . <numeral>
<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  
```

## 2. Le compilateur

Le compilateur généré est composé d'un analyseur lexical (la classe **ArabicPrologLexer**), un analyseur Syntaxique (la classe **ArabicPrologParser**) et un écouteur (l'interface **ArabicPrologListener** et la classe qui l'implémente **ArabicPrologBaseListener**). Le diagramme d'état transition suivant décrit le fonctionnement du compilateur :

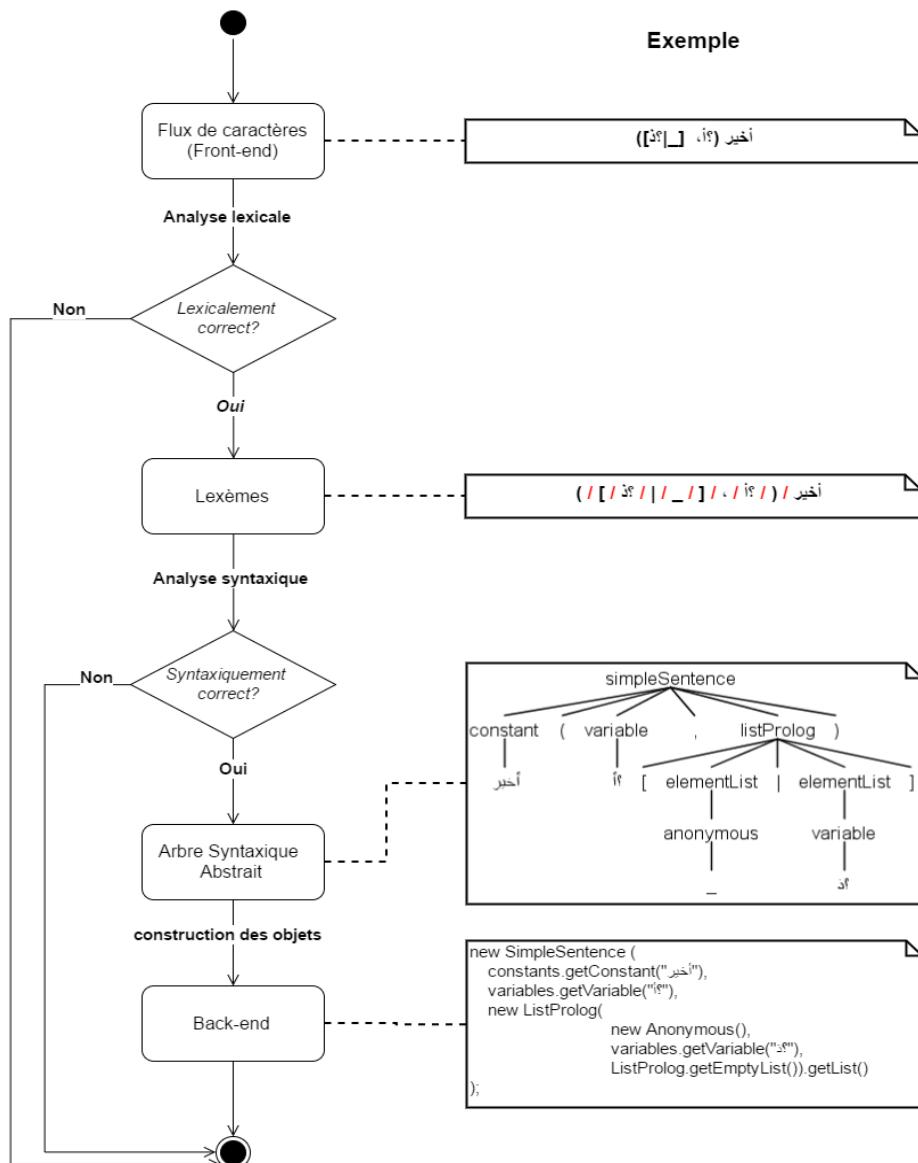


Figure 5.2 Diagramme d'état transition : la compilation

- 1) La classe **ArabicPrologLexer** joue le rôle d'un analyseur lexical : il détermine les lexèmes qui constituent le code source (Front-end). Un lexème est le plus petit mot qui a un sens dans le code. Chaque lexème du Prolog en langue arabe est représenté par une classe générée par ANTLR, c'est en se basant sur ces classes, le "lexer" à déterminer les lexèmes du programme. En cas d'un mot inconnu une exception se lance et un message indiquant qu'il existe un mot est inconnu s'affiche.
- 2) La deuxième étape consiste à faire l'analyse syntaxique : un code est syntaxiquement correct si la classe **ArabicPrologParser** arrive à générer l'arbre syntaxique abstraite (AST) qui le reconnaît, sinon une exception se lance et un message décrivant l'erreur s'affiche.
- 3) L'écouteur joue un rôle primordial pour générer le code source en Back-end. Il est constitué de l'interface **ArabicPrologListener** qui définit les signatures des méthodes de visites des nœuds (comme il est présenté dans la Figure 5.3) et la classe **ArabicPrologBaseListener** qui nous devons prendre le soin d'implémenté ses méthodes (implémentation des méthodes de visites des nœuds)

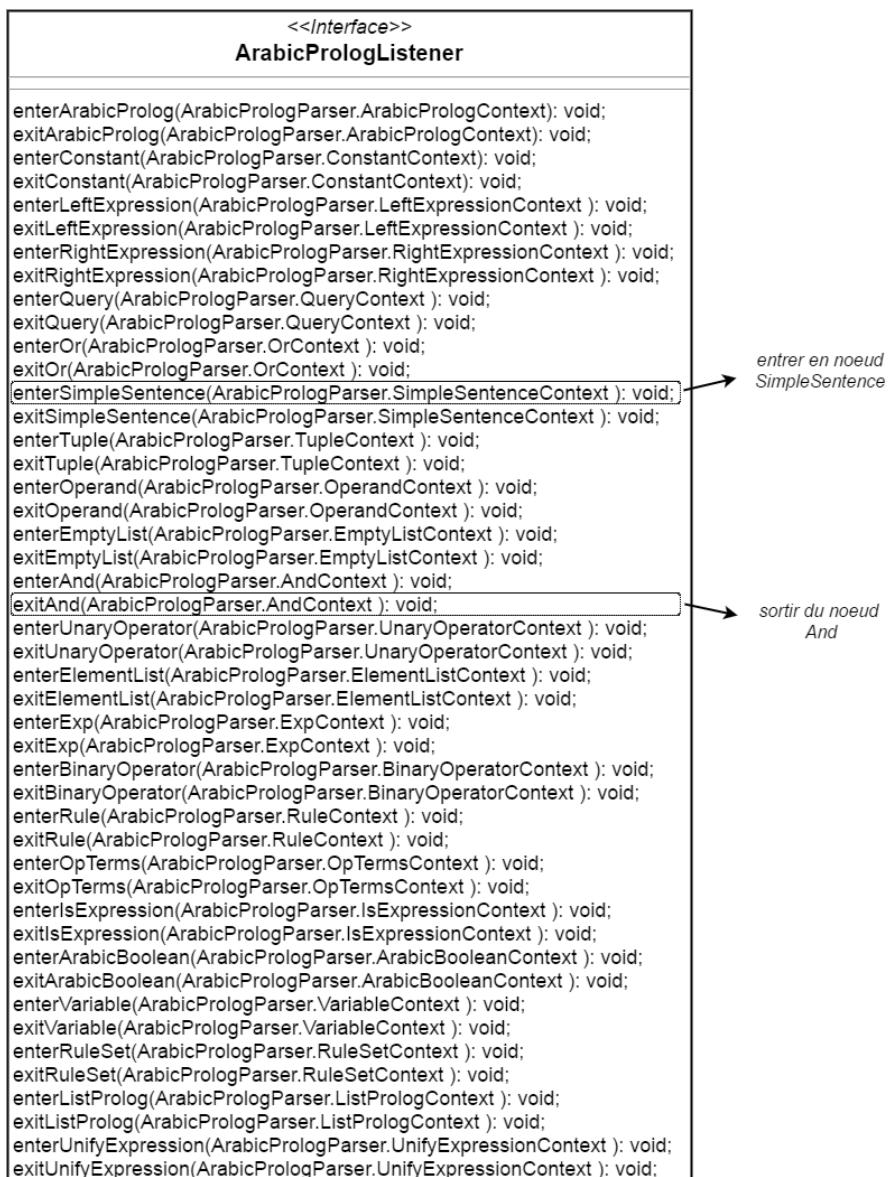


Figure 5.3 La classe ArabicPrologListener

L'idée est de parcourir l'arbre via la méthode *walk* de la classe **ParseTreeWalker**. A chaque passage du nœud (entrée ou sortie) le code de la méthode de visite du nœud courant est exécuté. Par exemple : A l'entrée du nœud **RuleSet**, la méthode *enterRuleSet* s'exécute. Cette méthode permet de créer l'objet représentant une base de connaissance **RuleSet**. Puisque la base de connaissances est constituée d'un ensemble des règles (une règle est représentée par la classe **Rule**) :

- i. La méthode *enterRule* annonce le début de construction d'un nouvel objet **Rule**
- ii. Une fois sortie du nœud **Rule** (la nouvelle règle est bien construite), la méthode *exitRule* ajoute la règle à la base de connaissance

Cette démarche de construction des objets est définie dans tous les méthodes de visites des nœuds.

## II. L'environnement de programmation logique : Mujeed

Mujeed est une application multiplateforme qui hérite sa portabilité de celle de langage Java dont nous n'avons besoin que d'un JDK (Java Developement Kit) pour le faire marcher sur n'importe quel système d'exploitations. Dans cette section nous décrivons l'interface graphique de notre environnement de programmation ainsi que la liste des fonctionnalités.

### 1. L'interface graphique

Nous avons utilisé l'API Swing, qui est une bibliothèque des composants graphiques, pour construire d'interfaces graphique de Mujeed. Swing offre la possibilité de créer des interfaces graphiques identiques quel que soit le système d'exploitation mais aussi grâce à la méthode *setLookAndFeel* de la classe **UIManager**, l'interface s'adapte au thème d'un système d'exploitation passé en paramètre :

```
UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
```

La méthode *getSystemLookAndFeelClassName* permet de déterminer le thème du système d'exploitation courant de sorte que l'interface graphique s'adapte au thème du système d'exploitation dans lequel l'application fonctionne.

L'interface graphique de notre environnement de programmation est interactive. Elle contient un menu permettant de regrouper les diverses fonctionnalités de l'application, une barre de lancement rapide pour les actions fréquemment utilisées, un explorateur de fichiers permettant de visualiser et ouvrir les fichiers des bases de connaissances qui se trouvent dans l'espace du travail, deux interpréteurs : le premier destiner à la rédaction de la base de connaissances. Le deuxième permet de l'interroger. En plus de la possibilité d'ouvrir plusieurs bases de connaissances dans des onglets. La capture-écran suivant représente l'interface graphique de Mujeed (LookAndFeel: Windows 7)

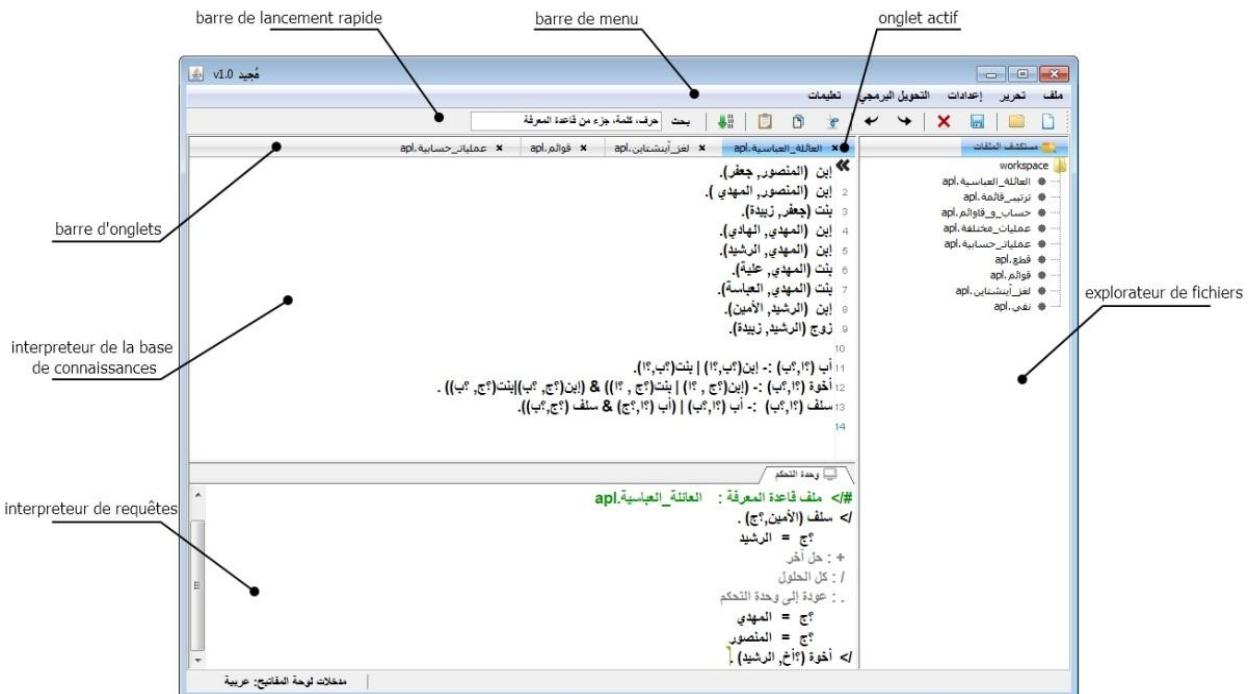


Figure 5.4 Capture-écran : interface Mujeed

## 2. Fonctionnalités de Mujeed

Les fonctionnalités offertes par Mujeed sont regroupées dans un menu: le menu fichier, le menu édition, le menu paramètres, le menu compiler et exécuter et le menu aide.

### a. Le menu fichier (قائمة ملف)

Les fichiers contenant les bases de connaissances ont l'extension .apl (comme Arabic ProLog). Mujeed est mené d'un système de gestion de fichiers de sorte que nous pouvons créer, ouvrir, sauvegarder ou bien supprimer les fichiers *apl*. En plus d'autres fonctionnalités existants dans ce menu comme il est présenté dans la figure suivante :



Figure 5.5 Le menu fichier

(\*) On désigne par courant le fichier *apl* dont le contenu est chargé dans l'interpréteur de la base de connaissances de l'onglet actif

### b. Le menu edition (قائمة تحرير)

Ce menu est destiné à la manipulation du contenu de l'interpréteur de la base de connaissances afin d'accélérer la rédaction de cette dernière. Ce menu offre aussi la possibilité d'annuler une action(Undo) et de restaurer une action(Redo) que les développeurs l'utilisent fréquemment. La figure suivante présente le menu édition :



Figure 5.6 Le menu edition

### c. Le menu paramètres (إعدادات)

Le paramétrage de Mujeed consiste à définir le font de l'écriture souhaité (police, taille, style) ainsi mettre la langue du clavier en arabe et afficher ou cacher : le clavier virtuelle (en langue arabe), l'explorateur des fichiers et l'interpréteur des requêtes. La figure suivante présente le menu paramètres :



Figure 5.7 Le menu paramètres

### d. Le menu compiler et exécuter (قائمة التحويل البرمجي)

C'est ce menu qui permette d'interroger la base de connaissances. En effet en cliquant sur exécuter :

- 1) Le compilateur traduit le code écrit en Front-end vers son équivalent en Back-end (en suivant les étapes décrit dans la section I.2)
  - En cas d'erreur syntaxique, un message décrivant l'erreur s'affiche
- 2) Un message indique que vous pouvez commencer à poser vos requêtes s'affiche

Le menu compiler et exécuter est présenté dans la figure suivante :



Figure 5.8 Le menu compiler et exécuter

Après compilation l'utilisateur écrit des requêtes. Tout comme la base de connaissances, les requêtes sont aussi compilés. Le moteur d'inférences chercher les réponses qui seront

affichées une par une (si l'utilisateur appuie sur "+"), tous les réponses (si l'utilisateur appuie sur "/"), ou bien si l'utilisateur appuie sur ".", l'interpréteur des requêtes rend la main à l'utilisateur.

### e. Le menu aide (قائمة تعليمات)

Le menu aide regroupe tous les informations nécessaires sur cet environnement ainsi que un tutoriel pour programmation logique en langue arabe (écrit en langue arabe) ainsi un lien vers le site officiel de Mujeed contenant l'application et le code source. Voici le menu aide :



Figure 5.9 Le menu aide

## III. Résultats et tests

L'environnement de développement Mujeed réalise la majorité de fonctionnalités Prolog. Il gère les listes, l'arithmétique (y compris l'algèbre de Boole), la coupure et les opérations sur termes. Afin de montrer la validité de l'approche, nous avons développé une pile d'exemple basique. Le tableau ci-dessous représente des prédictats réalisés et testés dans notre application (en plus des exemples présentés en détail dans le reste de cette section):

Prédicat	Eq. anglais	Arité	Rôle
<i>Liste</i>			
عضو	member	2	Vérifie si un élément appartient à une liste
آخر	last	2	Détermine le dernier élément d'une liste
الحق	append	3	Concatène deux listes
حذف	delete	3	Supprime un élément d'une liste
عكس	reverse	2	Inverser une liste
<i>Coupure &amp; liste</i>			
إتحاد	union	3	Union de deux listes
تقاطع	intersect	3	Intersection de deux listes
<i>Arithmétique &amp; opération sur termes</i>			
زيادة وحد	increment	2	Incrémenter un nombre
قيمة مطلقة	abs	2	La valeur absolue d'un nombre
أكبر	max	3	Détermine le maximum entre deux nombres
<i>Arithmétique &amp; liste</i>			
طول	len	2	La longueur d'une liste (nombre des éléments)

*Il est possible d'écrire des requêtes directement dans l'interpréteur de requêtes :*

```

. 1 * 4 == 1 + 3 --> !?
. +/ --> !?
. 6 * 1 < 3 + 10 & --> !?

```

Tableau 5.2 Exemples de prédictats réalisés

Dans cette section nous présentons quelques prédictats en détails. Le premier exemple est l'énigme d'Einstein: cet exemple montre la puissance des langages de programmation déclaratifs. Aussi nous présentons le prédictat sum, le prédictat tri insertion, le prédictat tri rapide et le prédictat permutation. Ces différentes méthodes permettant de mesurer les performances de l'application.

### 1. Enigme d'Einstein (version arabe)

Cette fameuse énigme était inventée par Albert Einstein (parfois attribué à Lewis Carroll). Einstein aurait dit que seulement 2% de la population était capable de résoudre cette énigme. Il existe plusieurs versions de cette énigme d'Einstein, nous aussi nous donnons une version arabe de l'éénigme mais le principe de résolution est inchageable. Cet exemple montre l'avantage des langages déclaratifs sur les langages impératifs pour résoudre ce type de problème.

#### a. Enoncé

Il y a cinq maisons de cinq couleurs différentes, alignées le long d'une route. Dans chacune de ces maisons, vit une personne dont les noms et les nationalités sont différents. Chacune de ces personnes boit une boisson différente et a un animal domestique différent.

- 1) L'Egyptien habite la première maison.
- 2) Le "Shami" habite la maison rouge.
- 3) La maison verte est située juste à gauche de la maison blanche.
- 4) Le Soudanais boit du thé.
- 5) Ali habite à côté de celui qui élève les chats.
- 6) Abbas habite la maison jaune.
- 7) Le Maghrébin s'appelle Mahmoud.
- 8) Celui qui habite la maison du milieu boit de l'eau
- 9) Ali a un voisin qui boit de citronnade.
- 10) Oussama élève le coq.
- 11) Le "Khaliji" élève le cheval.
- 12) L'gyptien habite à côté de la maison bleue.
- 13) Celui qui élève l'oiseau habite à côté de la maison jaune.
- 14) Salah boit du lait
- 15) Dans la maison verte on boit du café.

La question à laquelle il faut répondre est : « Qui élève le poisson ? ».

#### b. Correction (apl.)

Chacune des cinq maisons admet cinq caractéristiques : la couleur, la nationalité, l'animal, la boisson et le nom de l'habitant donc il fallait définir les structures de données qui supportent ces informations.

i- Une maison est une collection de cinq caractéristiques :

(منزل (\_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_))

ii- L'énigme est une collection de cinq maisons :

(اللُّغَزْ (منزل, منزل, منزل, منزل, منزل))

L'auteur donne deux types d'information

- i- Description de la maison en tant que membre dans l'énigme. Par exemple la deuxième proposition : Le "Shami" habite la maison rouge.

- Le prédicat **عضو** (membre en français), une maison peut être dans la première, la deuxième, la troisième, la quatrième ou la cinquième position :

عضو (؟)، عضو (؟).

- ii- Emplacement des maisons : l'auteur a utilisé les expressions qui indiquent les lieux : à gauche, à côté et milieu pour ce fait nous définissons les prédictats correspondant emplacement des maisons :

- Le prédicat **أول** (premier en français)

## أول ((؟، ؟))

- Le prédicat يسار (gauche en français)

- Le prédicat **جَانِبٌ** (voisin en français)

جانب (؟أ, ؟ب, ؟ج) :- يسار (؟أ, ؟ب, ؟ج) | يسار (؟ب, ؟أ, ؟ج).

- Le prédicat **وسط** (milieu en français)

و سط (؟)، (، ؟، ، ؟)

Après avoir défini les prédictats, maintenant, il nous faut qu'écrire le texte de lénigme selon la syntaxe de prolog en langue arabe. La définition du prédictat لغز (énigme en français) est définie comme suite :

لغز (? حل)

---

عضو (منزل (حلب, صالح), حل)  
 & عضو (منزل (أخضر, قهوة), حل)  
 & عضو (منزل (سمكة), حل).

Finalement, on pose la question : « quel est la solution de l'énigme d'Einstein ? »

- لغز (؟ حل). </
- ؟ حل = ( منزل أصفر مصرى قط ليمون عباس )  
 ( منزل أزرق سوداني عصفور شاي على )  
 ( منزل أحمر شامى ديك ماء أسامة )  
 ( منزل أخضر مغاربى سمسك قهوة محمود )  
 ( منزل أبيض خليجي حسان حليب صالح )

## 2. Évaluation des performances

Notre étude expérimentale consiste à mesurer le temps de réponse des plusieurs algorithmes de complexités différentes. Ainsi nous avons choisi de comparer nos résultats avec les résultats de deux implémentations java de Prolog : JIProlog et tuProlog.

JIPROLOG (Java Internet Prolog) est une implémentation java de Prolog. Il est open-source, multiplateforme et supporte les spécifications du standard ISO Prolog. JIPROLOG née en 1998 d'une idée de Ugo Chirico à Naples [37]. Nous avons utilisé JIPROLOG à partir de Java en passant par une API.

tuProlog (2P) est un interpréteur léger de prolog développé par aliCE Research Group à Université à Bologne. Il est apparu en 2001. Il supporte les spécifications du standard ISO Prolog. Tout comme JIPROLOG, tuProlog offre une API java permettant de définir et interpréter une base de connaissances en utilisant un programme Java [38].

Ci-dessous les caractéristiques de l'ordinateur utilisé pour l'étude expérimentale :

- Fabricant et modèle : Acer Aspire 5733
- Système d'exploitation : Microsoft Windows 7 Professionnel
- Type du système : PC à base de x64
- Processeur : Intel(R) Core(TM) i3 CPU M 380 @ 2.53GHz, 2533 MHz, 2 cœurs, 4 processeurs logiques
- Mémoire physique (RAM) : 4,00 Go

### a. Le prédictat sum (جمع)

Le prédictat sum permet de calculer la somme des éléments d'une liste. Il consiste à additionner claculer la somme des éléments d'une liste l'entête de la liste avec la somme et réappliquer le prédictat sur la queue de la liste de manière récursive jusqu'à avoir la liste vide. Il s'agit d'un prédictat de complexité linéaire: O(n). Voici la définition du prédictat somme en ISO Prolog et Arabic Prolog.

ISO prolog	Arabic Prolog
sum([],0). sum([H T],N):- sum(T,N1),N is N1+H.	جمع([],[0]). جمع([_   T], N) :- جمع(T, N1), N is N1 + _.

Tableau 5.3 Définition du prédictat sum

Ci-dessous est le tableau de mesures et la courbe relatifs au prédicat somme pour les trois implémentations Prolog : JIProlog, tuProlog, Mujeed.

Les valeurs des éléments de la liste sont générées aléatoirement. Ce sont les mêmes utilisées dans les trois implémentations de Prolog.

n	25	50	75	100	125	150	175	200
<b>JIProlog</b>	12 ms	11 ms	17 ms	21 ms	25 ms	32 ms	41 ms	48 ms
<b>tuProlog</b>	28 ms	48 ms	57 ms	65 ms	72 ms	80 ms	85 ms	102 ms
<b>Mujeed</b>	27 ms	46 ms	61 ms	100 ms	142 ms	137 ms	227 ms	219 ms

Tableau 5.4 Tableau de mesures : prédicat sum

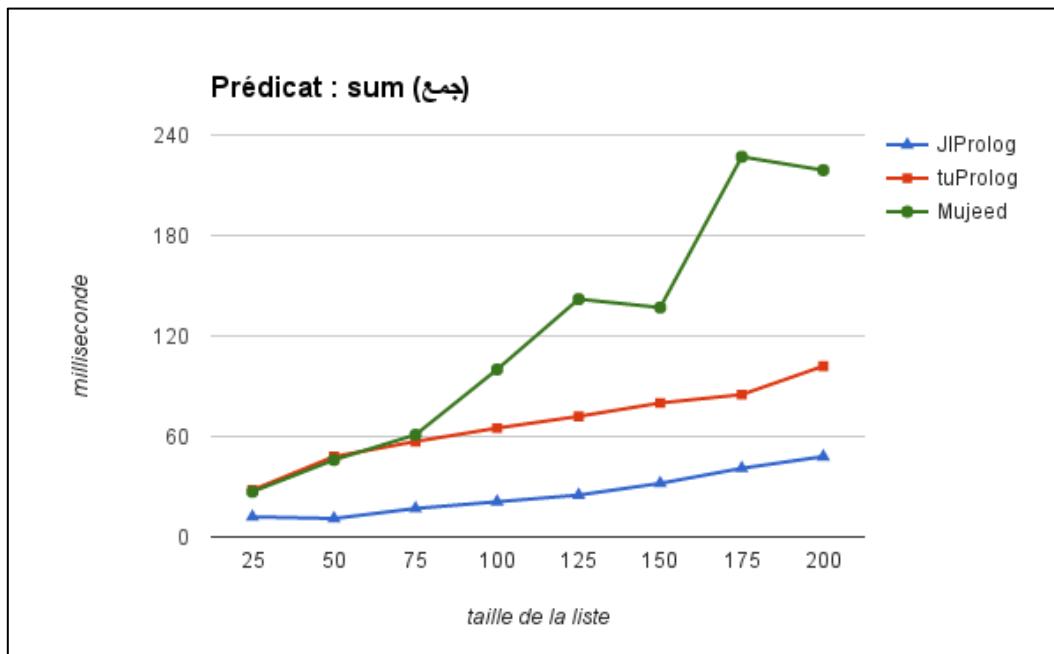


Figure 5.10 Courbe : prédicat sum

JIProlog, tuProlog et Mujeed effectuent le calcul. JIProlog réalise le calcul encore plus rapidement. Ensuite, tuProlog et finalement Mujeed mais la différence reste minimale.

### b. Le prédicat quickSort (الترتيب السريع)

Comme le nom indique, quickSort est l'un des algorithmes de tri les plus (son principe est expliqué dans le deuxième chapitre section : II.4). Il est de complexité semi-linéaire :  $\Theta(n \log(n))$ . Sa définition est la suivante :

ISO prolog	Arabic Prolog
<pre>quickSort(L,T) :- qSort(L,[],T). qSort([],L,L). qSort([P L],Acc,T):- partition(P,L,L1,L2), qSort(L1,Acc,T1), qSort(L2,[P T1],T).</pre>	<p>ترتيب سريع(<math>[L]</math>, <math>T</math>) :- ترتيب(<math>[L]</math>, <math>T</math>).          ترتيب(<math>[]</math>, <math>L</math>).          ترتيب(<math>[P L]</math>, <math>Acc</math>, <math>T</math>) :- تقسيم(<math>P</math>, <math>L</math>, <math>L1</math>, <math>L2</math>),  &amp; ترتيب(<math>L1</math>, <math>Acc</math>, <math>T1</math>), ترتيب(<math>L2</math>, <math>[P T1]</math>, <math>T</math>).</p>

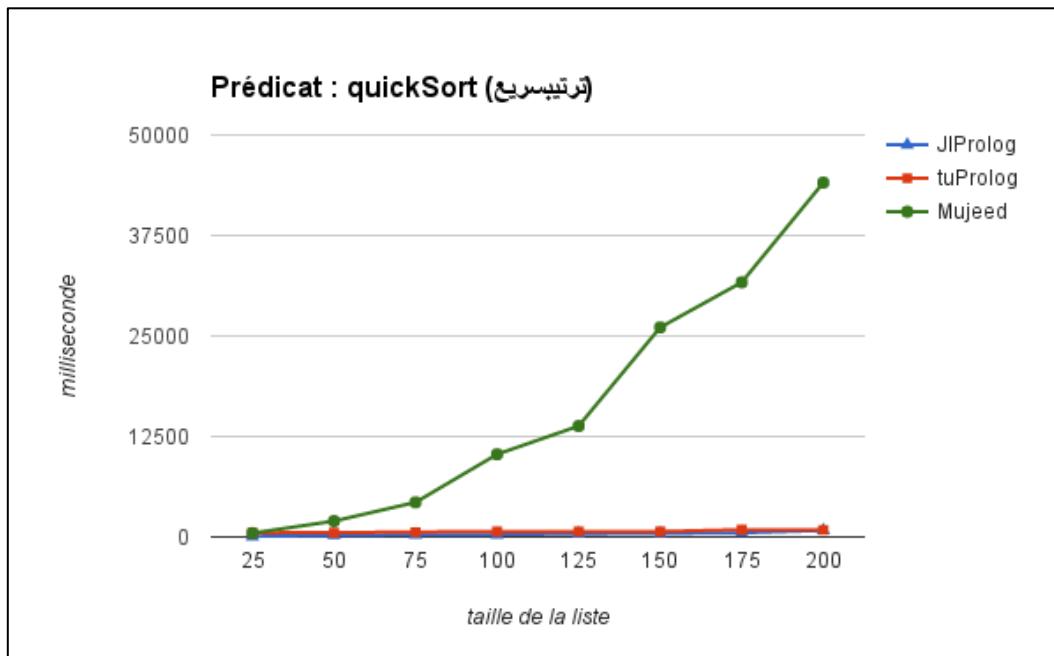
<pre> partition(_,[],[],[]). partition(P,[X T],[X U1],U2) :- P&gt;X, partition(P,T,U1,U2). partition(P,[X T],U1,[X U2]) :- P=&lt;X, partition(P,T,U1,U2). </pre>	<p>تقسيم(_،[],[],[]).</p> <p>تقسيم(أ،[أ أ]،[أ أ]،ج2) :- أ &lt; ج2 &amp;</p> <p>تقسيم(أ،ن،ج1،ج2) :- أ &gt; ج1 &amp;</p> <p>تقسيم(أ،[أ أ]،ج1،ج2) :- أ =&gt; ج1 &amp;</p> <p>تقسيم(أ،ن،ج1،ج2) :- أ &lt; ج1 &amp;</p>
--	---

**Tableau 5.5** Définition du prédictat quickSort

Le tableau de mesures et la courbe suivants présentent les mesures relatives au prédicat quickSort :

<b>n</b>	<b>25</b>	<b>50</b>	<b>75</b>	<b>100</b>	<b>125</b>	<b>150</b>	<b>175</b>	<b>200</b>
<b>JIProlog</b>	66 ms	194 ms	279 ms	294 ms	401 ms	428 ms	522 ms	819 ms
<b>tuProlog</b>	512 ms	538 ms	611 ms	683 ms	672 ms	670 ms	905 ms	874 ms
<b>Mujeed</b>	476 ms	1973 ms	4306 ms	10284 ms	13801 ms	26069 ms	31680 ms	44077 ms

**Tableau 5.6** Tableau de mesures : prédicat quickSort



**Figure 5.11** Courbe : prédicat quickSort

On remarque que les courbes de JIProlog et tuProlog sont presque confondues. Ils réalisent le tri d'une liste de 200 éléments dans un temps qui ne dépasse pas une seconde. Alors que Mujeed met un temps de réponse très élevé pour trier une liste.

### c. Le prédicat insertionSort (نرتیباراچ)

Le tri insertion consiste à insérer un élément d'une liste dans sa bonne position : il faut donc, trouver la bonne position de l'élément à insérer en le comparant aux autres, puis décaler les éléments afin de pouvoir effectuer l'insertion. La complexité du tri par insertion est  $O(n^2)$ .

Ci-dessous la définition du prédicat insertionSort:

ISO prolog	Arabic Prolog
<pre>insertionSort([], []). insertionSort([H T], R) :- insertionSort(T, L),     insertInPlace(H, L, R).</pre>	ترتيبية إدراج ([], []). ترتيبية إدراج ([؟ر   ڈن], [؟]) :- ترتيبية إدراج (ڈن, ڈن) & إدراج (؟ر, ڈن, ڈن).
<pre>insertInPlace(E, [], [E]). insertInPlace(E, [H T], [E L]) :- E =     &lt; H,     insertInPlace(H, T, L). insertInPlace(E, [H T], [H L]) :- E &gt;     H,     insertInPlace(E, T, L).</pre>	إدراج (م، [], [م]). إدراج (م، [؟ر   ڈن], [؟ر   م]) :- ڈن = > م & إدراج (ڈن, ڈن, ڈن). إدراج (م، [؟ر   ڈن], [؟ر   م]) :- م < ڈن & إدراج (م، ڈن، م).

Tableau 5.7 Définition du prédicat insertionSort

Le tableau de mesures et la courbe suivants présentent les mesures relatives au prédicat tri\_insertion :

n	10	20	30	40	50
JIProlog	27 ms	151 ms	259 ms	399 ms	559 ms
tuProlog	41 ms	104 ms	181 ms	213 ms	280 ms
Mujeed	506 ms	1054 ms	3369 ms	7649 ms	11546 ms

Tableau 5.8 Tableau de mesures : prédictat insertionSort

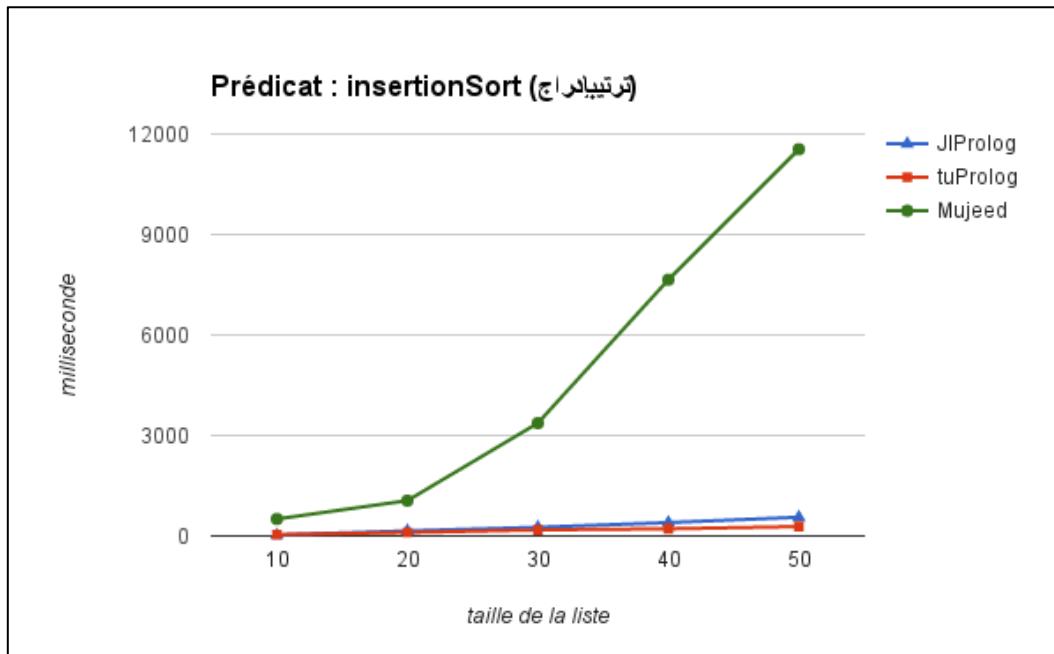


Figure 5.12 Courbe : prédictat insertionSort

Le tri insertion montre, clairement, la lenteur de Mujeed pour les algorithmes de complexité polynomiale. Pour trier une liste de 50 éléments Mujeed prend plus que 10 secondes par contre JIProlog et tuProlog réalisent ce traitement très rapidement.

#### d. Le prédictat permutation (تبديل)

Le prédictat permutation consiste à trouver toutes les permutations possibles des éléments d'une liste. C'est l'un des plus pire algorithme puisque sa complexité est  $O(n !)$ .

Ci-dessous la définition du prédictat permutation:

ISO prolog	Arabic Prolog
<pre>takeOut(X,[X R],R). takeOut(X,[F   R],[F S]) :- takeout(X,R,S).  permutation([X Y],Z) :- permutation(Y,W),     takeOut(X,Z,W). permutation([],[]).</pre>	<pre>إقتبس(؟، [؟، [؟، [؟]).  إقتبس(؟، [؟، [؟، [؟، [؟]) :- إقتبس(؟، [؟، [؟، [؟]).  تبديل(؟، [؟، [؟، [؟]) :- تبديل(؟، [؟)، &amp; إقتبس(؟، [؟، [؟، [؟]).  تبديل(?) .</pre>

Tableau 5.9 Définition du prédictat permutation

Le tableau de mesures et la courbe suivants présentent les mesures relatives au prédictat permutation:

n	3	4	5	6	7	8
JIProlog	7 ms	16 ms	65 ms	291 ms	-	-
tuProlog	39 ms	65 ms	123 ms	2331 ms	-	-
Mujeed	9 ms	26 ms	116 ms	241 ms	819 ms	6455 ms

Tableau 5.10 Tableau de mesures : prédictat permutation

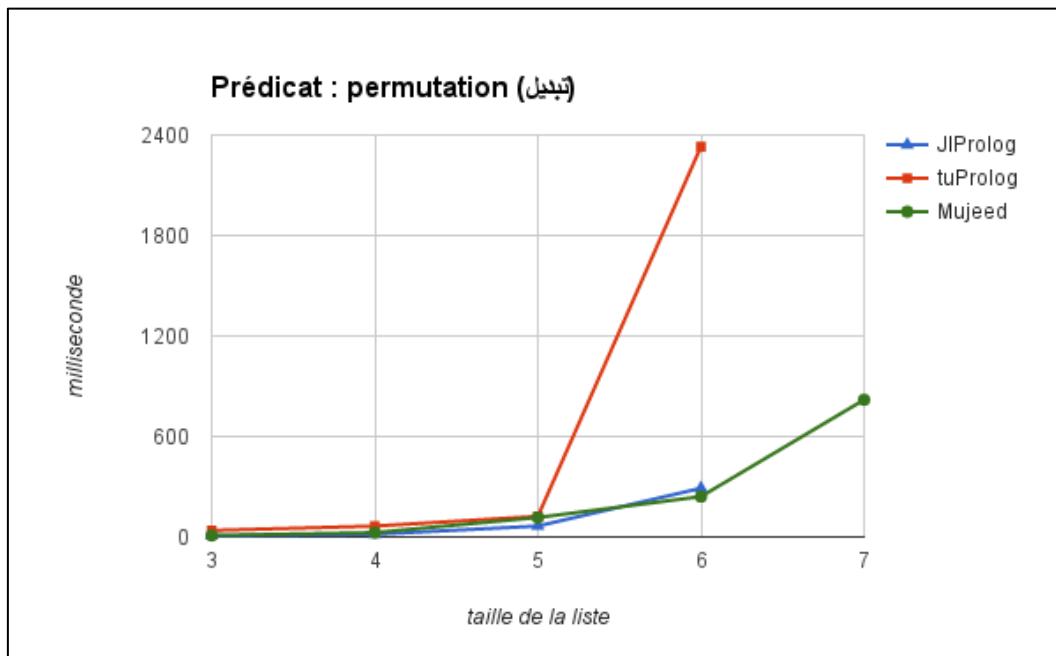


Figure 5.13 Courbe : prédictat permutation

Les trois implémentations trouvent toutes les permutations d'une liste de 3, 4 ou 5 éléments dans un temps presque égal. Pour une liste de 6 éléments tuProlog met un temps

élevé pour trouver toutes les permutations. JIProlog et tuProlog ne peuvent pas appliquer le prédictat permutation pour une liste de taille supérieure à 6.

## Conclusion

Mujeed est un environnement de développement Prolog. Il est caractérisé par sa syntaxe (Arabic Prolog) qui convient aux caractéristiques de l'écriture arabe, son interface graphique interactive et ses fonctionnalités multiples.

Il est maintenant possible d'écrire des programmes de nature déclarative telle que l'énigme d'Einstein et les différents algorithmes de tri en utilisant l'application Mujeed.

En Comparant les performances de Mujeed avec les deux implémentations Java de Prolog JIProlog et tuProlog, nous avons constaté que ces derniers sont beaucoup plus puissants. Dans la prochaine version, Nous nous intéressons à l'optimisation des performances de Mujeed

## Conclusion générale et perspectives

Notre objectif était de concevoir et de développer un environnement pour la programmation logique en langue arabe. L'existence de ce genre d'outil est, en effet, important pour la programmation en langue arabe en général et pour la définition et l'analyse de bases de connaissances en langue arabe en particulier.

Développer un langage de programmation logique en langue arabe présente un défi puisque ISO Prolog et les solutions existantes utilisent les caractéristiques de l'alphabet latin : les variables sont notées avec une lettre en majuscules en début, les constantes sont notées avec une lettre en minuscule au début et le sens d'écriture de gauche à droite. Par contre l'alphabet arabe s'écrit de droite à gauche et ne définit pas le majuscule ni le minuscule.

Nous avons pu concevoir et mettre en place une architecture orienté-objet permettant de supporter le mécanisme d'unification qui représente le noyau du Prolog et d'y intégrer des composants pour la gestion des expressions arithmétiques et la gestion des listes.

Nos objectifs ont été globalement atteints avec notamment les contributions suivantes :

- Etude de l'état de l'art pour classifier les langages de programmation, les langages de programmation en langue arabe, ainsi la programmation logique en Prolog.
- Implémentation de l'algorithme d'unification en tant qu'une architecture orientée objet permettant de définir des bases de connaissances et les interroger.
- Extension de l'algorithme d'unification pour la gestion des listes et réalisation d'un composant pour la gestion des expressions arithmétiques intégré dans l'algorithme d'unification.
- Définition de la partie frontale La partie frontale sous forme de BNF (Backus Naur Form), que le framework ANTLR utilise pour générer un compilateur. Ainsi les bases de connaissances et les requêtes sont définies avec des expressions écrites en langue arabe.
- Conception et développement d'une interface homme machine conviviale qui offre la possibilité de définir, interroger, sauvegarder les bases de connaissances.
- Développement d'une large pile d'exemples pratiques permettant de montrer la validité de l'approche.

Notre approche de développement d'un environnement de programmation logique en langue arabe consiste, tout d'abord, à créer le langage de programmation : nous nous sommes intéressés aux fonctionnalités offertes par l'environnement de programmation sans penser aux performances. Dans la seconde version de Mujeed, nous nous travaillerons sur les performances. L'optimisation de la solution existante peut se faire en changeant le langage Java par un autre langage de bas niveau ou bien en utilisant des algorithmes dont la complexité est inférieure.

## Conclusion générale et perspectives

---

Ce travail ouvre des perspectives de recherche intéressantes comme par exemple la science du hadith (علم الحديث). Les musulmans donnent aux communications orales du prophète Mohammed (صَلَّى اللَّهُ عَلَيْهِ وَسَلَّمَ) une importance majeure de sorte qu'ils ont classé les hadiths selon des degrés d'authenticité. Les spécialistes des hadiths utilisent des règles bien précises pour évaluer chaque hadith en se basant sur le contenu (المتن) et la chaîne de transmission (السند) du hadith. La définition d'une base de connaissances alimentées les hadiths authentiques (الأحاديث الصحيحة) et la modélisation des techniques utilisées par les spécialistes des hadiths en des règles permettent de savoir le degré d'authenticité du hadith en requête.

## Références

- [1] Colmerauer, Alain, and Philippe Roussel. “The Birth of Prolog.” *History of Programming Languages--II*, ACM, New York, USA, 1996, pp. 331–367.
- [2] “Summary by Language Size.” *Ethnologue*, 2016, [www.ethnologue.com/statistics/size](http://www.ethnologue.com/statistics/size).
- [3] Djebbar, Amine. “Histoire Et Évolution De La Langue Arabe.” *Les Cahiers De l'Islam*, 30 Sept. 2012, [www.lescahiersdelislam.fr/Histoire-et-evolution-de-la-langue-arabe\\_a137.html](http://www.lescahiersdelislam.fr/Histoire-et-evolution-de-la-langue-arabe_a137.html).
- [4] “Alphabet Arabe.” *Wikipedia*, Wikimedia Foundation, [fr.wikipedia.org/wiki/Alphabet\\_arabe](http://fr.wikipedia.org/wiki/Alphabet_arabe).
- [5] Al-Khuwārizm, Muhammad Ibn Mūsā. *كتاب الجبر و المقابلة*. Caire, Egypte, Université Du Caire.
- [6] Ayyubi, N. Akmal. “Contribution of Al-Khwarizmi to Mathematics and Geography.” *The Proceedings of the International Symposium on Ibn Turk, Khwârazmî, Fârâbî, Beyrûnî and Ibn Sînâ*, Ankara, 1990 .
- [7] Boley, Bruno A. “Memorie . Luigi Federico Menabrea , Letterio Briguglio , Luigi Bulferetti.” *Isis*, vol. 66, no. 1, 1975, pp. 146–147.
- [8] Kim, Eugene Eric, and Betty Alexandra Toole. “Ada and the First Computer.” *Scientific American*, vol. 280, no. 5, 1999, pp. 76–81.
- [9] Turing, A. M. “On Computable Numbers, with an Application to the Entscheidungsproblem.” *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, Jan. 1937, pp. 230–265.
- [10] Bergin, Thomas J. (Tim). “A History of the History of Programming Languages.” *Communications of the ACM*, vol. 50, no. 5, May 2007, pp. 69–74.
- [11] Nielsen, J. “International Conference on Fifth Generation Computer Systems 1988.” *SIGCHI Bull.* ACM SIGCHI Bulletin, vol. 21, no. 1, 1989, pp. 68–71.
- [12] Borderie, Xavier. “Expliquez-Moi... L’Évolution Des Langages Informatiques.” *JDN*, Apr. 2006, [www.journaldunet.com/developpeur/tutoriel/theo/060406-generations-langages-informatiques.shtml](http://www.journaldunet.com/developpeur/tutoriel/theo/060406-generations-langages-informatiques.shtml).
- [13] Van Roy, Peter. “Les Principaux Paradigmes De Programmation.” Jan. 2008, Université Catholique De Louvain Louvain-La-Neuve, Belgium.
- [14] Wielemaker, Jan, and Vítor Santos Costa. “Portability of Prolog Programs: Theory and Case-Studies.” Sept. 2010. Online proceedings of the Joint Workshop on Implementation of Constraint Logic Programming Systems and Logic-based Methods in Programming Environments (CICLOPS-WLPE 2010), Edinburgh, Scotland, U.K., July 15, 2010
- [15] “Comparison of Prolog Implementations.” *Wikipedia*, Wikimedia Foundation, [en.wikipedia.org/wiki/Comparison\\_of\\_Prolog\\_implementations](http://en.wikipedia.org/wiki/Comparison_of_Prolog_implementations).
- [16] Zarrouki, Taha. *وادي التقنية*, ”دراسة مقارنة لثلاث لغات برمجة عربية حديثة (جيم، زاي، لوغو) ” Mar. 2008, [itwadi.com/node/362](http://itwadi.com/node/362).

## Références

---

- [17] Amin, M.r. "The Arabic Object-Oriented Programming Language Al-Risalh." *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*, June 2001, pp. 424-424.
- [18] Assalka, Muhammad Ammar. **كتاب تعريف لغة ج** Damas, Syrie, 2007, [www.jeemlang.com/documentation/webframe.html](http://www.jeemlang.com/documentation/webframe.html).
- [19] Ammouria, Abdualdhim Ahmed. "عمورية أول لغة برمجة عربية مفتوحة المصدر" "ammoria." [sourceforge.net/ar/programmers\\_page.html](http://sourceforge.net/ar/programmers_page.html).
- [20] Bassil, Youssef. "The 1st General-Purpose Arabic Programming Language." *Phoenix* **الفنق**, [phoenixlanguage.8k.com/](http://phoenixlanguage.8k.com/).
- [21] Bassil, Youssef, and Aziz Barbar. "MyProLang - My Programming Language: A Template-Driven Automatic Natural Programming Language."
- [22] Fayed, Mahmoud. "Supernova Programming Language." [supernova.sourceforge.net/](http://supernova.sourceforge.net/).
- [23] Muhammed Ali, Wael Hassan. **رسالة البرمجة ببادع**
- [24] "Qalb (Programming Language)." *Wikipedia*, Wikimedia Foundation, [en.wikipedia.org/wiki/Qalb\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Qalb_(programming_language)).
- [25] "Information Technology -- Programming Languages -- Prolog -- Part 1: General Core." *The International Organization for Standardization (ISO), the International Electrotechnical Commission (IEC)*, 1995.
- [26] Deransart, Pierre et al. *Prolog: the Standard; Reference Manual*. Berlin, Allemagne, Springer, 1996.
- [27] Krenger, Simon. "Prolog BNF Grammar." *GitHub*, Dec. 2013, [github.com/simonkrenger/ch.bfh.bti7064.w2013.PrologParser/blob/master/doc/prolog-bnf-grammar.txt](https://github.com/simonkrenger/ch.bfh.bti7064.w2013.PrologParser/blob/master/doc/prolog-bnf-grammar.txt).
- [28] Singleton, Paul et al. "JPL: A Bidirectional Prolog/Java Interface." *SWI-Prolog*, [www.swi-prolog.org/packages/jpl/](http://www.swi-prolog.org/packages/jpl/).
- [29] Ernest, J. Friedman-Hill. "Online Jess Documentation." *Jess, the Rule Engine for the Java Platform*, [www.jessrules.com/jess/docs/index.shtml](http://www.jessrules.com/jess/docs/index.shtml).
- [30] Luger, George F., and William A. Stubblefield. *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp, and Java*. Boston, MA, USA, Pearson Addison-Wesley, 2009.
- [31] G. Falquet. "Logique Des Prédicats." Université De Genève.
- [32] "Infix to Postfix Java Converter Implementation." *CODE MILES*, Jan. 2013, [www.codemiles.com/java-examples/infix-to-postfix-converter-t10792.html](http://www.codemiles.com/java-examples/infix-to-postfix-converter-t10792.html).
- [33] Parr, Terence. "ANTLR 4 Documentation." *GitHub*, [github.com/antlr/antlr4/blob/master/doc/index.md](https://github.com/antlr/antlr4/blob/master/doc/index.md).
- [34] "JavaCC [Tm]: Documentation Index." *Java.net*, [javacc.java.net/doc/docindex.html](http://javacc.java.net/doc/docindex.html).
- [35] Gagnon, Etienne. "SableCC, an Object-Oriented Compiler Framework." School of Computer Science McGill University, Montreal, 1998.

## Références

---

- [36] Klein, Gerwin et al. “JFlex User's Manual.” *JFlex*, Apr. 2015, [jflex.de/manual.html](http://jflex.de/manual.html).
- [37]“JIPProlog : A Prolog Interpreter, Pure Java 100%, Cross-Platform and Open Source.” JIPProlog, [www.jiprolog.com/](http://www.jiprolog.com/).
- [38] Denti, Enrico. TuProlog Manual. Université De Bologne, Italie.

---

## Annexe

Le fichier "ArabicProlog.g4" contient la grammaire de notre langage de programmation en notation EBNF (en plus d'une explication en commentaire) :

```

grammar ArabicProlog;

//un programme prolog est représenté par une base de connaissances et des requêtes
arabicProlog      :   ruleSet| query*;

/*
Une requête est syntaxiquement correcte, si elle est sous forme de :
 * une simpleSentence
 * une évaluation d'une expression
 * une unification de deux expressions
 * une opération sur termes
*/
query           :   (simpleSentence|isExpression|unifyExpression|opTerms) Point;

// Une base de connaissances est constituée d'un ensemble des règles
ruleSet         :   rule*;

/*
Une règle est syntaxiquement correcte, si elle est sous forme d'une clause de Horn:
 * une simpleSentence (fait)
 * une simpleSentence (entête) :- corps (plusieurs formes possibles)
*/
rule            :   simpleSentence Point
                  | simpleSentence If (simpleSentence|and|or|Cut|isExpression|
                           unifyExpression|opTerms)+ Point;

//la conjonction et la disjonction sont définies par au moins l'opérateur
//ainsi que ses opérandes
and             :   (((simpleSentence|Cut|isExpression|unifyExpression|opTerms)|
                     LeftParen and RightParen|LeftParen or RightParen) And)+
                  (((simpleSentence|Cut|isExpression|unifyExpression|opTerms)|
                     LeftParen and RightParen|LeftParen or RightParen);

or              :   (((simpleSentence|Cut|isExpression|unifyExpression|opTerms)|
                     LeftParen and RightParen|LeftParen or RightParen) Or)+
                  (((simpleSentence|Cut|isExpression|unifyExpression|opTerms)|
                     LeftParen and RightParen|LeftParen or RightParen);

/*
Une simpleSentence est syntaxiquement correcte, si elle est sous forme de
 * un prédicat (ou sa négation) suivi par des arguments séparés par des
   virgules et entre parenthèses
 * un tuple
 * une simpleSentence entre parenthèses
*/
simpleSentence :   NotOp? constant LeftParen
                  (((variable|constant|tuple|simpleSentence|listProlog|
                     emptyList) Comma)*
                   (variable|constant|tuple|simpleSentence|listProlog|
                     emptyList)) RightParen
                  | tuple
                  | LeftParen simpleSentence RightParen;

```

## Annexe

---

```
// Un tuple est une suite des arguments séparés par des virgules entre parenthèses
tuple      : LeftParen (((variable|constant|tuple) Comma)*
                      (variable|constant|tuple)) RightParen;

/*
Une listProlog est syntaxiquement correcte, si elle est sous forme de
 * une liste vide
 * la structure élément1, élément2|tail entre crochets
 ** Un élément est syntaxiquement correcte, si elle est sous forme d'une
     Variable, une constante, un tuple, une simpleSentence, une liste
*/
emptyList   : LeftBracket RightBracket;
listProlog   : LeftBracket elementList ((Comma elementList)* | 
                                         (Or elementList))? RightBracket;
elementList  : (variable|constant|tuple|simpleSentence|listProlog) ;

/*
Une constante est syntaxiquement correcte, si elle est sous forme de:
 * un caractère arabe suivi d'une chaîne peut comporter des chiffres
 * un nombre entier, un nombre double (précéder par "-" s'il est négatif)
 * un booléen en langue arabe ( صحيح/خطأ )
*/
constant     : Constant | Number | arabicBoolean ;
Constant      : ArabicChar+ (('0'..'9')|ArabicChar)*;
ArabicChar    : '\u0600'..\u06FF';
Number        : '-'? ('0'..'9')+ (Point ('0'..'9'))+)?;
arabicBoolean : ArabicTrue|ArabicFalse;
ArabicTrue    : ' صحيح';
ArabicFalse   : ' خطأ';

/*
Une variable est syntaxiquement correcte, si elle est sous forme de:
 * un anonyme "_"
 * la syntaxe d'une constante précédée par ?
*/
variable     : Variable | Anonymous;
Variable      : '?' Constant;
Anonymous     : '_';

// Une unification et une évaluation sont représentées par une expression à gauche
// (leftExpression), l'opérateur et une expression à droite (rightExpression)
unifyExpression : leftExpression UnifyOperator rightExpression;
isExpression   : leftExpression IsOperator rightExpression;

// Une opération sur termes a la même syntaxe que rightExpression
opTerms       : rightExpression;

/*
La partie gauche d'une expression est syntaxiquement correcte, si elle est sous
forme de:
 * une variable
 * une constante
*/
leftExpression : variable|constant;
```

## Annexe

---

```
/*
La partie droite d'une expression est syntaxiquement correcte, si elle est sous
forme de:
    * une constante
    * une variable
    * une ou plusieurs expressions
        ** une expression contenant un opérateur binaire
        ** une expression contenir un opérateur unaire
*/
rightExpression : constant
                  | variable
                  | (operand binaryOperator)* operand binaryOperator operand
                    (binaryOperator operand)*
                  | (operand binaryOperator)* unaryOperator operand
                    (binaryOperator operand)*;

/*
Un opérande est syntaxiquement correcte, s'il est sous forme de :
    * une constante
    * une variable
    * la partie gauche d'une expression entre parenthèses
*/
operand      : constant|variable|exp;
exp          : LeftParen rightExpression RightParen;

// Opérateur unaire
unaryOperator : NotOp;

// Opérateur binaire
binaryOperator : AddOp|SubOp|MulOp|DivOp|ModOp|And|Or|EqualOp|NotEqualOp|
                 LessOp|LessOrEqualOp|GreaterOp|GreaterOrEqualOp;

// Opérateurs & Symboles
Point       : '.';
If          : ':-' ;
Cut         : '/';
Or          : '|';
And         : '&';
UnifyOperator : '=';
IsOperator   : '<--';
NotOp       : '!';
AddOp       : '+';
SubOp       : '-';
MulOp       : '*';
DivOp       : '\';
ModOp       : '%';
EqualOp     : '==';
NotEqualOp  : '!=';
LessOp       : '>';
LessOrEqualOp : '<=';
GreaterOp   : '<';
GreaterOrEqualOp: '>=';
Comma        : ',';
LeftParen    : '(';
RightParen   : ')';
LeftBracket  : '[';
RightBracket : ']';
```

## Annexe

---

```
Whitespace      : [ \t]+           -> skip; // Ignorer les espaces
Newline         : ('\\r' '\\n'?|'\\n')-> skip; // Ignorer les retours chariots
BlockComment    : '/*' .*? '*'\\' -> skip; // Ignorer les blocs de commentaires
LineComment     : '%' ~[\\r\\n]*   -> skip; // Ignorer les commentaires ligne
```

## Résumé

Le développement d'un environnement de programmation logique en langue arabe est important pour la programmation en langue arabe en général et pour la définition et l'analyse de bases de connaissances en langue arabe en particulier. D'abord, nous avons conçu et implémenté l'algorithme d'unification en nous basant sur une architecture orientée objet permettant de définir des bases de connaissances et les interroger. Ensuite, nous avons étendu le noyau responsable d'unification par un composant de gestion des listes et un composant d'évaluation des expressions arithmétiques. Puis, nous avons défini la grammaire du Prolog en langue arabe sous forme de BNF que le framework ANTLR utilise pour générer un compilateur. Finalement, nous avons configuré le compilateur généré par le framework ANTLR pour qu'il soit possible de définir et interroger les bases de connaissances en langue arabe.

Mujeed, l'environnement de programmation logique en langue arabe est caractérisé par une interface graphique interactive qui offre pratiquement toutes les fonctionnalités d'un interpréteur basé sur le paradigme de la programmation logique.

Ce travail ouvre des perspectives de recherche intéressantes comme par exemple l'optimisation des performances et l'exploitation de certaines bases de connaissances de taille importante.

**Mots-clés :** Arabic Prolog, Mujeed langage de programmation, algorithme d'unification, évaluation des expressions arithmétiques, BNF, ANTLR.

## Abstract

The development of a logical programming environment in Arabic is important for programming in Arabic language in general and for the definition and analysis of knowledge bases in Arabic language in particular. First, we designed and implemented the unification algorithm based on an object-oriented architecture to define knowledge bases and interrogate them. Then we extended the kernel responsible for unification by a component of processing lists and a component of the evaluation of the arithmetic expressions. Then, we defined the Prolog grammar in Arabic as BNF that the ANTLR framework uses to generate a compiler. Finally, we have configured the compiler generated by the ANTLR framework so that it is possible to define and query the knowledge bases in Arabic language.

Mujeed, the logical programming environment in Arabic language is characterized by an interactive graphical interface that offers practically all the functionality of an interpreter based on the paradigm of logical programming.

This work opens interesting research perspectives such as optimizing performance and exploiting some large knowledge bases.

**Keywords:** Arabic Prolog, Mujeed programming language, unification algorithm, Evaluation of arithmetic expressions , BNF, ANTLR