Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Prolog avec Truffle et GraalVM
Rapport final

Projet de semestre 5
I-3
Automne 2019/2020

Martin Spoto
martin.spoto@edu.hefr.ch

Responsable :
Frédéric Bapst
Frederic.Bapst@hefr.ch

30 janvier 2020

Table des matières

1 Introduction **5**
- 1.1 Contexte . . . 5
- 1.2 Objectifs . . . 5
- 1.3 Déroulement . . . 5
- 1.4 Structure du rapport . . . 6

2 Technologies du projet **6**
- 2.1 GraalVM . . . 6
- 2.2 Truffle . . . 6
- 2.3 Prolog . . . 6
- 2.4 ANTLR . . . 7
- 2.5 Java . . . 7
- 2.6 Maven . . . 7
- 2.7 Relations entre les technologies . . . 7

3 Approche et méthodologie **7**
- 3.1 Gestion du projet . . . 7
- 3.2 Phases du projet . . . 8
- 3.3 Démarrage du projet . . . 8
- 3.4 Littérature et références . . . 8
  - 3.4.1 Utilisation de Truffle . . . 8
  - 3.4.2 Implémentation d'un interpréteur Prolog . . . 8
  - 3.4.3 Autres . . . 8

4 Phase 1 : Fonctionnalités de base **9**
- 4.1 Besoins . . . 9
- 4.2 Approche . . . 9
  - 4.2.1 Choix d'une philosophie . . . 9
  - 4.2.2 Technologies . . . 9
- 4.3 Plan . . . 9
- 4.4 Prise en main . . . 10
- 4.5 Implémentation . . . 10
- 4.6 Implémentation : launcher . . . 10
- 4.7 Implémentation : language . . . 10
  - 4.7.1 Point d'entrée : Analyse . . . 11
  - 4.7.2 Point d'entrée : Modifications . . . 12
  - 4.7.3 Parser : Analyse . . . 12
- 4.8 Représentation des éléments . . . 12
- 4.9 Implémentation : parser . . . 13
  - 4.9.1 Lexer . . . 14
  - 4.9.2 Parser . . . 14
  - 4.9.3 Génération des fichiers Java . . . 15
  - 4.9.4 Implémentation du listener . . . 16
- 4.10 Implémentation : language (suite) . . . 18
  - 4.10.1 Liaison au parser . . . 18
  - 4.10.2 Interpréteur : Analyse . . . 19

4.10.3 Adaptation du `EvalRootNode` : analyse . . . 19
4.10.4 Adaptation du contexte . . . 20
4.10.5 Adaptation du `EvalRootNode` : implémentation . . . 20
4.10.6 Interpréteur : Implémentation . . . 20
4.10.7 Noeud de résolution . . . 21
4.11 Synthèse . . . 21

5 Phase 2 : Variables **22**
5.1 Besoins . . . 22
5.2 Plan . . . 22
5.3 Représentation des variables . . . 22
5.4 Adaptation du parser . . . 22
5.4.1 Lexer . . . 23
5.4.2 Parser . . . 23
5.4.3 Listener . . . 23
5.4.4 Retour du parser . . . 24
5.5 Adaptation du noeud de résolution . . . 25
5.6 L'unification . . . 25
5.6.1 Principe général . . . 25
5.6.2 Atomes et nombres . . . 25
5.6.3 Structures . . . 26
5.6.4 Variables . . . 26
5.7 Affichage des résultats . . . 27
5.8 Synthèse . . . 28

6 Phase 3 **28**
6.1 Besoins . . . 28
6.2 Plan . . . 28
6.3 Représentation des éléments . . . 29
6.3.1 Listes . . . 29
6.3.2 Clauses . . . 30
6.4 Adaptation du parser . . . 30
6.4.1 Lexer . . . 31
6.4.2 Parser . . . 31
6.4.3 Listener . . . 32
6.4.4 Retour du parser . . . 35
6.5 Arbre de preuve . . . 35
6.5.1 Noeud de résolution . . . 36
6.5.2 Noeuds de l'arbre de preuve . . . 36
6.5.3 Synthèse . . . 37
6.6 Correction des problèmes d'unification . . . 37
6.6.1 Valeur racine . . . 38
6.6.2 Test d'occurrence . . . 38
6.7 Affichage des résultats . . . 39
6.8 Synthèse . . . 40

7 Phase bonus **40**
7.1 Résultats multiples . . . 40
7.1.1 Besoins . . . 40
7.1.2 Demande du prochain succès . . . 40
7.1.3 Reprise de la recherche . . . 41
7.1.4 Implémentation : interpréteur . . . 41
7.1.5 Implémentation : noeud de résolution . . . 43
7.1.6 Implémentation : noeuds de l'arbre de preuve . . . 43
7.2 Prédicats intégrés . . . 44
7.2.1 Implémentation : partie commune . . . 44
7.2.2 Prédicat `var` . . . 45
7.2.3 Prédicat `write` . . . 46
7.2.4 Les raisons de la méthode `execute` . . . 47
7.2.5 Intégration des prédicats . . . 47
7.3 Questions multiples . . . 47
7.4 Synthèse . . . 48

8 Tests unitaires **48**
8.1 Fonctionnement . . . 48
8.2 Améliorations . . . 48
8.3 Tests . . . 48

9 Problèmes rencontrés **49**
9.1 JLine3 . . . 49

10 Améliorations possibles **49**
10.1 Optimisations et intégration plus forte à Truffle . . . 49
10.1.1 Améliorations du noeud de résolution . . . 49
10.1.2 Amélioration du processus de résolution . . . 49
10.1.3 Intégration plus forte avec Truffle . . . 50
10.2 Support de l'interopérabilité avec Truffle . . . 50
10.3 Support du prédicat cut . . . 50
10.4 Support d'autres prédicats/de la librairie standard . . . 50

11 Conclusion **51**
11.1 Atteinte des objectifs . . . 51
11.2 Conclusion personnelle . . . 51

Références **52**

12 Glossaire **53**

13 Annexes **56**
13.1 Lien du projet . . . 56
13.2 Librairies utilisées . . . 56
A Planning . . . 57
B Manuel d'installation et d'utilisation . . . 58

1 Introduction

Prolog avec Truffle et GraalVM est un projet de semestre réalisé durant le semestre d'automne 2019-2020. Ce rapport contient une trace du déroulement, des choix et des problèmes rencontrés et résolus au fil du projet.

1.1 Contexte

GraalVM est une machine virtuelle universelle basée sur la *Java Virtual Machine* conçue pour permettre l'interprétation à haute performance de n'importe quel langage de programmation, pour autant qu'un interpréteur pour ce langage soit disponible. Truffle est le nom du framework permettant l'implémentation en Java d'un tel interpréteur.
Des interpréteurs sont disponibles pour de nombreux langages, tels que JavaScript, Python, Ruby et même C et C++, mais il n'existe actuellement aucun interpréteur pour le langage de programmation logique Prolog.
Le but de ce projet est donc de réaliser cet interpréteur, afin de prouver la faisabilité technique de l'implémentation d'un langage de programmation logique sur cette plateforme.

1.2 Objectifs

L'objectif principal du projet est de fournir un interpréteur pour le langage Prolog basé sur GraalVM et Truffle. Cependant, en raison des contraintes temporelles du projet, implémenter la totalité du langage Prolog n'est pas réalisable. Un sous-ensemble de fonctionnalités jugées "de base" a donc été sélectionné. Les fonctionnalités retenues sont les suivantes :

- Gestion de termes simples et composés
- Gestion de clauses comprenant des appels récursifs
- Gestion des variables
- Gestion de la notation simplifiée pour les listes
- Unification de termes, variables
- Gestion de la résolution par *backtracking*
- Gestion des requêtes depuis la ligne de commande

Un certain nombre de fonctionnalités sont également retenues si l'implémentation s'avère plus rapide ou plus simple que prévue. Ces fonctionnalités sont listées sans ordre de priorité ci-dessous :

- Gestion de calculs avec is/2
- Opérateur "!" (*cut*)
- *assert* et *retract*
- Gestion des exceptions
- Traceur de résolution
- Opérateurs mathématiques et logiques usuels (+ - * / = < >)
- Librairie standard, I/O
- Profiter de GraalVM/Truffle pour fournir une interface avec d'autres langages
- Profiter de GraalVM/Truffle pour fournir un débogueur

1.3 Déroulement

Le projet a été découpé en 4 phases clés :

0. Analyse et prise en main des technologies
1. Version minimale de l'interpréteur, fonctionnant uniquement pour des faits
2. Ajout du support des variables, toujours uniquement sur des faits
3. Ajout du support des clauses composées et récursives

Ces phases sont détaillées dans la section 3.2.

1.4 Structure du rapport

Le rapport est découpé selon les différentes phases du projet. Chaque phase contient une partie d'analyse des besoins puis décrit la réalisation de ladite phase.
Ce rapport fait l'hypothèse que le lecteur est familier avec le langage de programmation logique Prolog ainsi qu'avec le language Java, pour se concentrer sur les explications de la réalisation de l'interpréteur. Toutefois un glossaire des termes techniques utilisés est disponible en fin de document (section 12).

2 Technologies du projet

Cette section présente les différentes technologies liées au projet, en fournissant pour chacune une description ainsi que leur rôle dans le projet.

2.1 GraalVM

GraalVM est une machine virtuelle universelle, conçue pour permettre l'exécution d'un grand nombre de langages de programmation différents et de faciliter la création de nouveaux langages. GraalVM essaie de résoudre les problèmes liés à la création d'un nouveau langage : fournir un débogueur, un compilateur et d'autres outils...
GraalVM permet également l'interopérabilité entre différents langages : on peut par exemple écrire une méthode en Javascript et l'invoquer dans un code source Python.
Du côté technique, GraalVM est basé sur la *Java Virtual Machine*, mais avec de nombreuses modifications sur le compilateur à la volée permettant d'augmenter les performances. GraalVM est également capable de créer des images natives compilées avant l'exécution, afin de s'affranchir du besoin d'avoir une machine virtuelle Java disponible.
Le rôle de GraalVM dans ce projet est de fournir la plateforme permettant de lancer l'application finale. À noter qu'il est également possible de lancer l'application sur une JVM classique, mais les performances sont réduites sans les capacités d'optimisations et de compilation à la volée de GraalVM.

2.2 Truffle

Truffle est une librairie Java open source permettant de réaliser des interpréteurs pour des langages de programmation. C'est la librairie qui permet de faire le lien entre le langage de programmation et les capacités de GraalVM. Ce lien donne toute sa puissance à la suite de technologies, car il permet en écrivant uniquement un interpréteur de bénéficier de performances proches d'un compilateur, et écrire un interpréteur est en général bien plus facile qu'écrire un compilateur.
Son rôle dans le projet est central : la majeure partie de l'implémentation se fait en utilisant l'API de Truffle.

2.3 Prolog

Prolog est un langage de programmation logique. Il permet de réaliser des programmes en utilisant une syntaxe déclarative plutôt que impérative. Prolog a été choisi en raison de l'absence d'implémentations de langages de programmation logique avec Truffle et GraalVM.
Prolog sert donc de base au projet pour lui donner une direction et un but. Son fonctionnement interne sera un élément central, vu qu'il s'agit de ce qui sera au final implémenté. En revanche, le langage en lui-même ne sera pas utilisé pour programmer des parties du projet, à part pour écrire des tests permettant de vérifier le fonctionnement de l'interpréteur.

2.4 ANTLR

ANTLR est un outil permettant de générer un lexer et un parser dans différents langages à partir d'un fichier décrivant la syntaxe du langage.
ANTLR sert à générer les fichiers Java permettant de parser un fichier Prolog et d'en ressortir les différents éléments.

2.5 Java

Java est le langage de programmation dans lequel sera réalisé le projet.

2.6 Maven

Maven est l'outil permettant de gérer le cycle de vie du projet Java, en gérant automatiquement les tests et la compilation, ainsi que les dépendances ou encore la génération du fichier JAR contenant l'application.

2.7 Relations entre les technologies

La figure 1 schématise les différentes relations entre les technologies du projet.

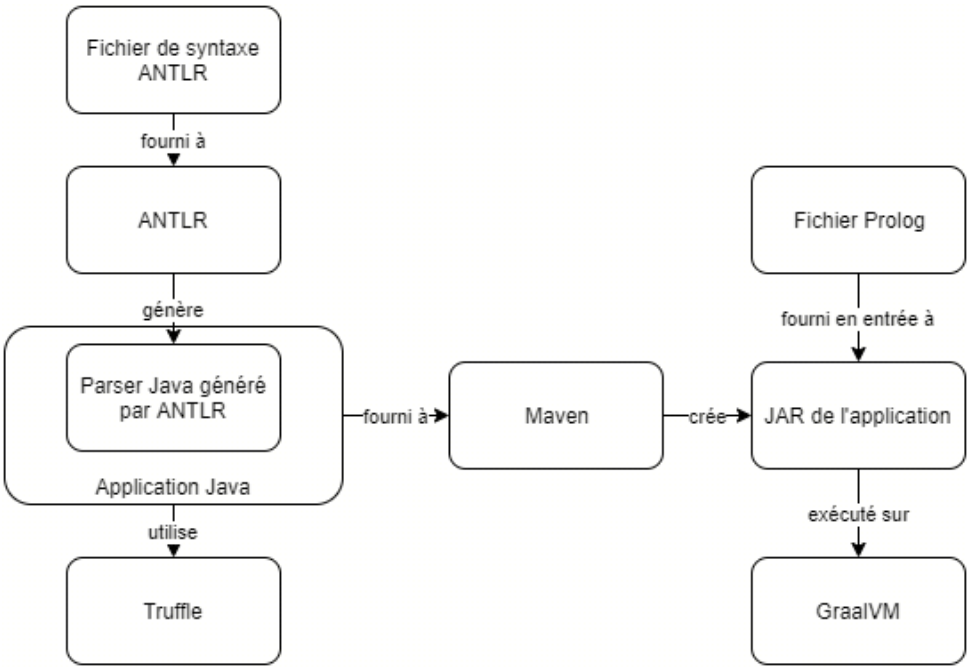


FIGURE 1 – Relations entre les différentes technologies

3 Approche et méthodologie

Cette section décrit la méthodologie suivie pour le projet ainsi que l'approche initiale utilisée pour démarrer le projet.

3.1 Gestion du projet

Le projet suit un planning établi préalablement, basé sur le cahier des charges, et des séances hebdomadaires avec le responsable de projet sont tenues afin de contrôler le bon déroulement de celui-ci. Un procès-verbal de chaque séance est rédigé, afin de garder une traces des décisions et constatations faites lors des séances. Le planning est disponible en annexe (annexe A).

Pour la partie technique, un répertoire du logiciel de gestion de versions *Git* est utilisé pour gérer les fichiers du projet. Le répertoire utilise principalement deux branches : la branche "master" contenant le dernier prototype fonctionnel du projet, et la branche "wip" contenant les dernières fonctionnalités, potentiellement encore en cours d'implémentation ou non testées. Sur la branche "master", des "tags" sont utilisés pour marquer les étapes clés du projet. Un lien vers le répertoire du projet est disponible en annexe (13.1).

3.2 Phases du projet

Sur la base du cahier des charges, le projet a été découpé en trois phases principales. Ces phases correspondent à des paliers de fonctionnalités à atteindre, dans le but d'obtenir un prototype fonctionnel à la fin de chacune de ces phases. Ces phases sont décrites en détail dans le cahier des charges disponible en annexe, mais un rappel des différents objectifs de chaque phase sera disponible avant leur section dédiée plus loin dans ce rapport.
En résumé, les phases du projet sont les suivantes :

1. Fonctionnalités de base : parsing de faits simples, sans variables
2. Ajout du support des variables pour ces faits
3. Ajout du support des clauses complexes et des listes

3.3 Démarrage du projet

Le projet a été démarré en énumérant les connaissances requises pour celui-ci. On en trouve deux principales : l'utilisation de Truffle et l'implémentation d'un interpréteur Prolog.
Pour l'utilisation de Truffle, le point de départ est principalement le langage de démonstration fourni en tant que documentation officielle, SimpleLanguage, qui est une implémentation complète d'un langage de programmation fictif en utilisant Truffle. Différents articles de blogs donnent également une bonne base pour se lancer dans l'implémentation. Ces articles sont listés dans la section suivante.
Pour l'implémentation d'un interpréteur Prolog, le choix a été fait d'étudier les différentes manières existantes, mais de quand même repartir de zéro, en gardant uniquement la "philosophie" de certaines implémentations, pas de base de code. Les différents articles, publications scientifiques et implémentations existantes servant de référence sont listés dans la section suivante.

3.4 Littérature et références

Cette section liste les différentes références servant de base et de guide au projet.

3.4.1 Utilisation de Truffle

— [Gra15] : Projet de démonstration officiel utilisant Truffle
— [Wim16] : Conférence Oracle sur l'utilisation de Truffle et GraalVM
— [Esq15] : Tutoriel sur l'implémentation d'un langage avec Truffle

3.4.2 Implémentation d'un interpréteur Prolog

— [Aït91] : Livre décrivant une implémentation abstraite possible d'un moteur Prolog
— [JIP98] : Projet open source implémentant un moteur Prolog en Java

3.4.3 Autres

— [ISO95] : Référence officielle pour la syntaxe Prolog
— [Tom17] : Tutoriel sur l'utilisation de ANTLR

4 Phase 1 : Fonctionnalités de base

Cette section documente le travail réalisé pour atteindre les fonctionnalités de base. On définit en tant que fonctionnalités de base le fait d'avoir un interpréteur capable de lire un fichier Prolog, et de répondre à des requêtes au clavier portant sur des faits simples.
Cette phase relativement simple nécessite déjà d'avoir toutes les technologies du projet fonctionnelles et travaillant ensemble.

4.1 Besoins

Les besoins de cette phase sont les suivants :

- Préparation de l'environnement GraalVM et Truffle
- Lecture de fichiers
- Lexing et parsing de faits écrits en Prolog
- Récupération de l'entrée clavier et requêtes dans les faits lus

En plus de ces besoins technologiques, il faut également décider la "philosophie" et l'architecture générale de l'interpréteur.

4.2 Approche

Cette section décrit l'approche pour subvenir aux différents besoins.

4.2.1 Choix d'une philosophie

Afin de pouvoir choisir une architecture et une philosophie de développement pour l'interpréteur, une recherche préalable sur les implémentations existantes et sur les manières d'implémenter un interpréteur Prolog a été effectué. On apprend par exemple dans [Aït91](*Warren's Abstract Machine : A Tutorial Reconstruction*) qu'il existe des techniques pour réaliser un interpréteur optimisé, mais ces techniques se reposent sur des mécanismes de bas niveau qui seraient difficiles à mettre en place avec Truffle, en tout cas dans un premier temps.
Pour des exemples plus proches de ce que l'on cherche à réaliser, on trouve des implémentations en Java d'interpéteurs Prolog, par exemple [JIP98](*JIProlog*). Certaines de ces versions se basent sur une approche complètement orientée objet ; c'est cette philosophie qui sera retenue pour notre implémentation.

4.2.2 Technologies

Pour prendre en main la technologie Truffle, quelques ressources sont à disposition en ligne. On trouve par exemple la conférence [Wim16]« One VM to Rule Them All » qui présente quelques aspects liés à Truffle. On trouve également une série d'articles de blog décrivant le parcours réalisé par un développeur californien pour implémenter son propre nouveau langage de programmation, d'abord en pur Java puis en intégrant Truffle ([Esq15]).
Mais la meilleur ressource disponible pour prendre en main la suite de technologies reste le projet de démonstration *SimpleLanguage : A simple example language built using the Truffle API*, qui fournit un exemple complet et relativement bien documenté sur l'utilisation de toutes les technologies et des différents liens entre celles-ci.

4.3 Plan

Le plan de réalisation de cette phase est donc le suivant : prendre comme base l'implémentation de démonstration *SimpleLanguage*, et commencer par retirer tout ce qui est spécifique à cette dernière, afin de se retrouver avec une base "minimale". Puis, les fonctionnalités propres à notre interpréteur seront rajoutées une à une à cette base, de manière incrémentale.

4.4 Prise en main

Le projet *SimpleLanguage* se présente sous la forme d'un projet *Maven*. Il est composé de quatre sous-parties :

- component : contient les fichiers nécessaires pour créer une version ajoutable à GraalVM (un *plugin*).
- language : contient les sources de l'interpréteur du langage *SimpleLanguage*
- launcher : contient les sources permettant de lancer l'interpréteur *SimpleLanguage*
- native : contient les fichiers nécessaires pour créer une version "native" d'un programme *SimpleLanguage*

Après l'installation de GraalVM et de Maven, puis la mise en place des variables d'enviromments nécessaires (opérations décrites en annexe de ce document), on peut taper `mvn package` pour créer les différents fichiers JAR correspondant aux parties du projet *SimpleLanguage*. On peut ensuite utiliser le script fourni, sobrement appelé "sl", pour lancer l'interpréteur fraîchement compilé en lui donnant un des fichiers ".sl" fourni et vérifier que tout fonctionne.

4.5 Implémentation

Maintenant que l'on sait que tout compile et fonctionne, on peut commencer à retirer les fonctionnalités propres à *SimpleLanguage* pour obtenir notre base pour la suite. On commence donc par supprimer les sous-parties "component" et "native" qui ne seront pas utilisées pour l'instant. On continue en modifiant les différents fichiers de description de projet Maven "pom.xml" afin de modifier le nom du projet en "ProloGraal", nom original pour le sous-ensemble de fonctionnalités Prolog développé lors de ce projet. On peut ensuite commencer à étudier le code fourni, en commençant par le launcher, vu que celui-ci est composé uniquement d'une seule classe.

4.6 Implémentation : launcher

Le launcher contient le "main" de l'application, et permet donc de lancer l'interpréteur de manière autonome. Il se charge de créer une représentation abstraite du code source à exécuter à l'aide d'un objet de type `Source` et est également responsable de la création d'un contexte d'exécution pour ce code source, en lui donnant notamment les flux d'entrée et de sortie. Ce mécanisme est pratique car il permet de donner des flux fictifs pour les tests unitaires par exemple (cf. section 8).
Au final on se rend compte que tout le code déjà existant est assez générique, et qu'il suffit de remplacer les références au langage SL par des références vers le nôtre.

```
private static final String SL = "sl";
// devient
private static final String PROLOGRAAL = "pl";
```

4.7 Implémentation : language

Le launcher étant terminé pour le moment, on peut passer au gros morceau : le composant language. Ce composant contient tout le code propre à *SimpleLanguage*, du lexer au parser en passant par le code Truffle. Il va donc falloir trier les fonctionnalités afin de distinguer celles qui nous seront utiles telles quelles, celles qui devront être adapatées et celles qui peuvent être supprimées.
On commence en étudiant la structure du code. La figure 2 donne un aperçu de celle-ci. On distingue quatre packages et quelques fichiers racines. Ces packages sont les suivants :

- builtins : contient des classes décrivant des méthodes prédéfinies propres au langage SL. A priori ne semble pas intéressant, vu que le contenu est très spécifique.

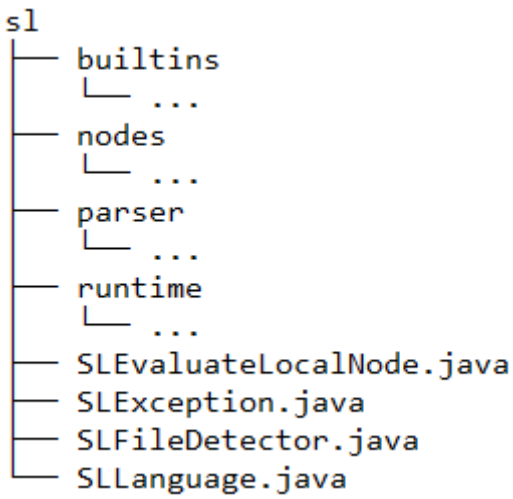


FIGURE 2 – Structure du code du module "language" de *SimpleLanguage*

— nodes : contient des classes décrivant des bouts de code (*statements*) SL. Bien que le code soit à nouveau spécifique, la structure est néanmoins intéressante pour la suite.
— parser : contient les classes gérant le lexing et le parsing d'un fichier source SL. Contient également un fichier ".g4" qui est un fichier de grammaire ANTLR servant à décrire le langage SL. Ce package est donc intéressant pour ce que l'on souhaite réaliser à ce stade.
— runtime : contient des classes décrivant les types propres à SL ainsi qu'un conteneur de contexte `SLContext.java`. Comme pour le package "nodes", la structure est intéressante, particulièrement le conteneur de contexte qui se retrouvera sûrement dans notre interpréteur.

Parmi les fichiers racines, les fichiers intéressants semblent être `SLFileDetector.java`, qui permet de vérifier qu'un fichier contient du code SL, et `SLLanguage.java` qui semble être le fichier principal du module.

4.7.1 Point d'entrée : Analyse

On va maintenant repartir du fichier principal trouvé précédemment, `SLLanguage.java`, et analyser son contenu. Il contient une classe `SLLanguage` héritant de `TruffleLanguage<SLContext>`. Cette classe est précédée d'une annotation décrivant le nom de langage et lui donnant un ID. Elle redéfinit plusieurs méthodes, et certaines semblent particulièrement intéressantes : `createContext` et `parse`.
On commence par la fonction `parse`. On comprend qu'elle utilise le parser pour récupérer la liste des fonctions présentes dans le fichier sous la forme d'une `Map<String, RootCallTarget>`. Les `RootCallTarget` sont des objets de Truffle représentant des racines de l'arbre de syntaxe abstrait, autrement dit, ils représentent des fonctions appelables. Une fois la liste des fonctions récupérée, la méthode récupère la fonction "main" si elle existe et crée un noeud de type `SLEvalRootNode` en lui donnant cette fonction. Finalement la méthode retourne un `CallTarget` contenant cet `SLEvalRootNode` via l'environnement Truffle :
`Truffle.getRuntime().createCallTarget(evalMain)`
Un `CallTarget` représente du code appelable. Ici, le `CallTarget` sera invoqué automatiquement plus tard par Truffle.

4.7.2 Point d'entrée : Modifications

Pour guider nos modifications, on peut assimiler la liste de fonctions à la liste des faits du programme Prolog. On doit donc modifier le parser, premièrement pour qu'il fonctionne pour du Prolog et plus du SL, mais également pour qu'il retourne correctement la liste de faits. Il faut également adapter le `SLEvalRootNode`.
Concernant les autres méthodes de la classe, leur utilité restant pour le moment floue, elles sont réduites au minimum et laissées de côté.

4.7.3 Parser : Analyse

Le parser se compose de quatre parties principales :

- `SimpleLanguage.g4` qui est le fichier de grammaire ANTLR décrivant comment lexer et parser un fichier SL. ANTLR s'en sert pour générer le lexer et le parser en Java. Il contient également des bouts de code permettant d'invoquer des appels à la `SLNodeFactory` décrite plus bas.
- `SimpleLanguageLexer.java` qui est un fichier généré automatiquement par ANTLR et qui contient le code Java nécessaire au lexing d'un fichier SL.
- `SimpleLanguageParser.java` qui est un fichier généré automatiquement par ANTLR et qui contient le code Java nécessaire au parsing d'un fichier SL.
- `SLNodeFactory.java` qui contient le code appelé depuis celui généré par ANTLR. Il s'agit d'une "Factory", ou usine, qui est un patron de développement faciliant la création d'objets. En l'occurrence, cette Factory permet de créer les différents noeuds Truffle représentant les parties du code source SL.

La grammaire et la Factory vont donc devoir être modifiées pour fonctionner avec du Prolog. Mais avant, il faut décider comment on va représenter du code Prolog de manière abstraite à l'aide de classes.

4.8 Représentation des éléments

Pour représenter un code Prolog en Java, un rappel sur Prolog s'impose. On sait des standards ([ISO95]) que Prolog contient une seule unité de données, le terme. Un terme peut être de plusieurs types :

- un atome : un atome est une séquence de caractères, qui ne transporte pas d'informations supplémentaires
- un nombre : les nombres en Prolog peuvent être de type entier ou à virgule flottante
- une variable : les variables en Prolog ont la fonction des variables en logique ; elles peuvent être remplacées par un autre type de données pour satisfaire une requête
- un terme composé : un terme composé est constitué d'un atome appelé foncteur et d'un certain nombre d'arguments. Ces arguments sont eux-mêmes des termes. Le nombre d'arguments d'un terme composé est appelé arité.

Il est maintenant trivial de transposer cette hiérarchie en Java. La figure 3 montre une représentation possible de ces différents types, en omettant volontairement les variables car nous ne nous en occuperons pas encore dans cette phase.
En plus des données, un concept vital à Prolog est la réussite ou l'échec d'une requête. On pourrait représenter cela à l'aide d'un simple booléen en Java, mais le choix d'abstraire cette représentation à l'aide de nouvelles classes a été fait, en prévision pour les phases suivantes. La figure 4 montre la représentation choisie pour ce concept.

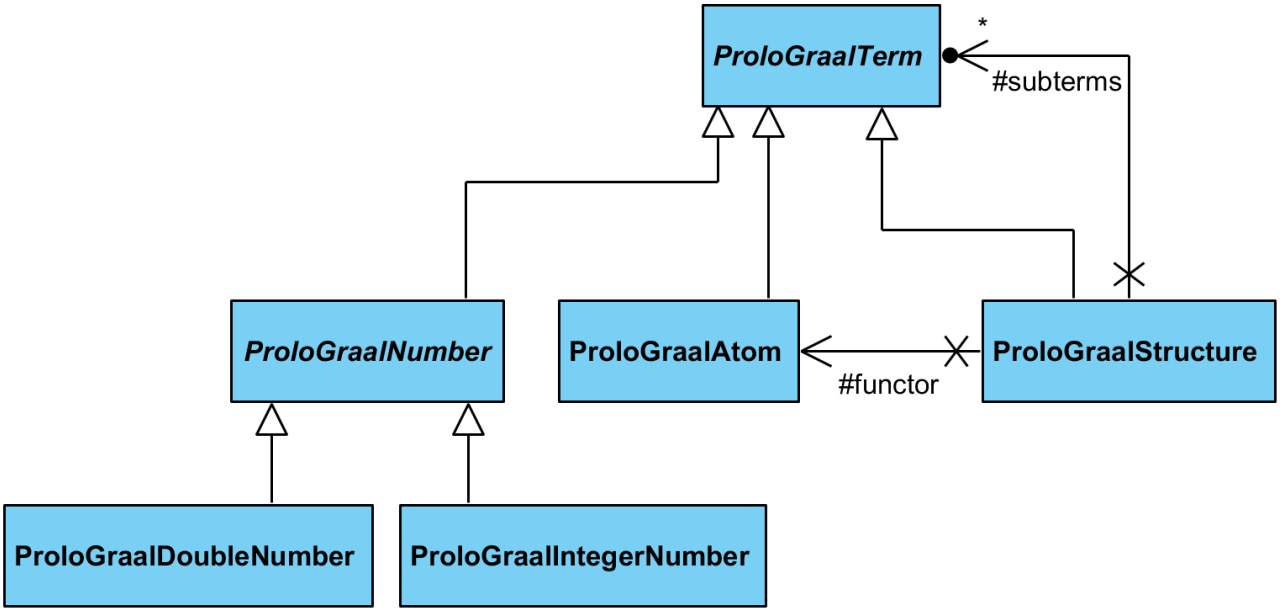


FIGURE 3 – Diagramme de classes des éléments Prolog en Java

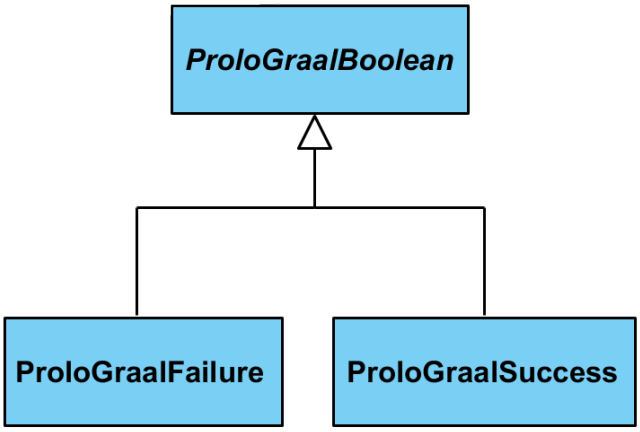


FIGURE 4 – Diagramme de classes représentant la réussite et l'échec Prolog en Java

4.9 Implémentation : parser

Maintenant que nous avons la structure des éléments représentée en Java, il faut écrire une grammaire ANTLR permettant de générer le code. Pour démarrer l'implémentation, nous avons les pistes suivantes : la grammaire de *SimpleLanguage*, et l'excellent tutoriel [Tom17]. Après lecture de ce tutoriel, il est possible d'analyser la grammaire écrite pour *SimpleLanguage*. On se rend compte qu'elle utilise une méthode hybride, avec du code Java imbriqué dans la grammaire, pour faire des appels à une autre classe se chargeant de créer les représentations Java des différents éléments au fur et à mesure. Cette méthode n'est pas particulièrement lisible, et elle n'est également pas recommandée par ANTLR. Les méthodes recommandées par ANTLR suivent deux patterns de développement :

— Le pattern Visitor
— Le pattern Listener

Le pattern Visitor est un peu plus flexible mais plus compliqué à mettre en place que le pattern Listener. Le langage Prolog n'étant pas particulièrement difficile à parser et à transformer en

Java, le pattern Listener a donc été retenu pour sa simplicité d'implémentation.
Ces choix techniques effectués, il reste le principal : écrire la grammaire. ANTLR sépare les règles de lexing et les règles de parsing.

4.9.1 Lexer

Comme décrit précédemment, Prolog ne contient pas beaucoup de symboles différents. Pour cette première phase, on en distingue uniquement deux : les atomes et les nombres.
Un atome commence par une lettre minuscule, et peut être suivi d'un certain nombre de caractères, pouvant être des lettres minuscules ou majuscules, des chiffres, ou un underscore. Un atome peut également commencer par une apostrophe et contenir n'importe quel caractère jusqu'à rencontrer une nouvelle apostrophe. Voici comment se traduisent ces règles dans une grammaire ANTLR :

```
ATOM : (LOWERCASE+ (LOWERCASE | UPPERCASE | DIGITS | '_')*) | ('\'' .*? '\'');
```

Pour rendre la règle plus lisible, des fragments sont utilisés. Les fragments sont des sortes de macro remplaçant une certaine classe de caractères :

```
fragment LOWERCASE : [a-z];
fragment UPPERCASE : [A-Z];
fragment DIGITS : [0-9];
```

Pour les nombres, ceux-ci doivent commencer par un chiffre, et être suivis d'un certain nombre d'autres chiffres. Ils peuvent également contenir un point suivi d'autres chiffres encore pour repésenter un nombre à virgule. D'autres notations sont disponibles en Prolog classique, comme la notation exponentielle par exemple, mais pour des raisons de simplicité, celles-ci sont omises. Voici l'implémentation de ces règles :

```
NUMBER : DIGITS+ ('.' DIGITS+)?;
```

Nous devons encore définir des règles pour les espaces blancs et les commentaires, tous deux devant être ignorés :

```
WHITESPACE : [ \t\r\n]+ -> skip;
COMMENT : '%' ~[\r\n]* -> skip;
```

On définit encore un dernier élément de lexing, le caractère terminal point servant à délimiter les différentes clauses du programmes :

```
TERMINATOR : '.';
```

La partie lexer est maintenant terminée pour cette phase, on peut maintenant utiliser ces règles pour écrire les règles de parsing.

4.9.2 Parser

Pour la partie parser, on se base sur les règles définies à la section 4.8. On sait également qu'un programme Prolog est composé d'un certain nombre de clauses. Pour cette phase, nous savons que ces clauses seront uniquement des faits. Ces règles se traduisent à nouveau de manière très élégante dans la grammaire ANTLR :

```
prolograal
:
clause clause* EOF
;

clause
:
fact
;

fact
:
term TERMINATOR
;
```

Comme décrit à la section 4.8, un terme peut être de plusieurs types. On retrouve ces différentes possibilités dans la grammaire :

```
term
:
functor
(
    '(' term (',' term)* ')'
) |
atom |
number
;
```

Le reste des éléments ne sera pas détaillé, mais il s'agit à chaque fois d'une chaîne de règles se terminant par un élément dit "terminal", qui contient uniquement une référence vers un élément du lexer. Par exemple pour un atome :

```
atom
:
ATOM
;
```

La grammaire est ensuite testée en utilisant un outil fourni par ANTLR permettant de visualiser l'arbre de syntaxe généré après le parcours d'un fichier. La figure 5 montre et valide le résultat obtenu avec la grammaire actuelle, pour le fichier d'exemple Prolog suivant :

```
hello(world).
test(antlr).
multiples(arg1, arg2).
```

4.9.3 Génération des fichiers Java

Pour générer les fichiers Java correspondant à la grammaire, on utilise l'exécutable JAR de ANTLR. Des explications détaillées sur l'utilisation de ce JAR sont disponibles dans le guide d'installation fourni en annexe (Annexe B). Cet exécutable génère plusieurs fichiers :

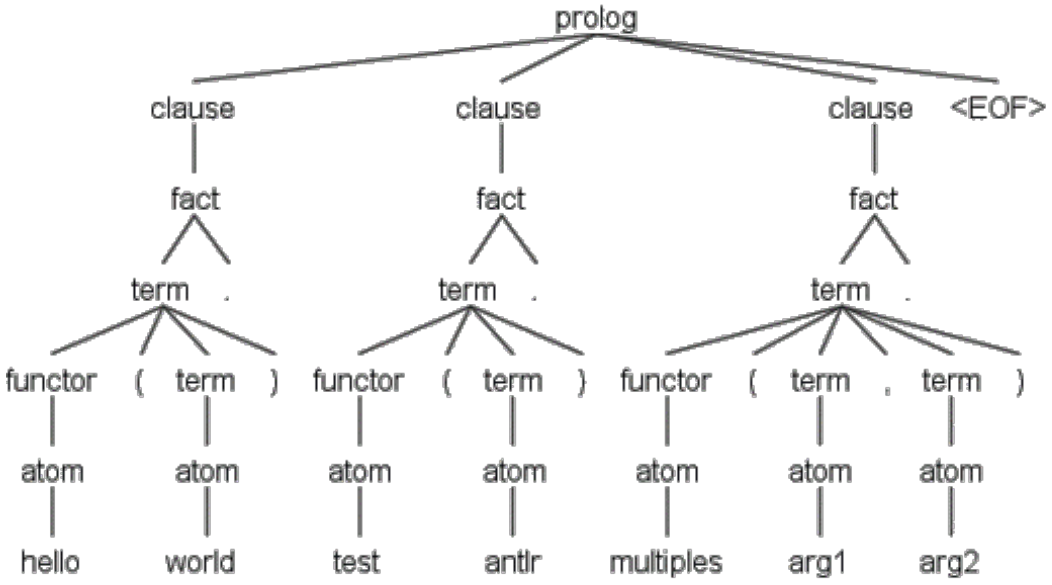


FIGURE 5 – Arbre de syntaxe obtenu après le parcours d'un fichier Prolog en utilisant la grammaire ANTLR de la phase 1

- ProloGraalLexer.java : fichier contenant le code du lexer
- ProloGraalParser.java : fichier contenant le code du parser
- ProloGraalListener.java : interface Java décrivant les fonctions à implémenter pour le pattern Listener. L'interface définit deux méthodes pour chaque élément du parser : une méthode "enter" et une méthode "exit". Ces méthodes seront appelées automatiquement lorque l'on parcourra le fichier.
- ProloGraalBaseListener.java : implémentation vide de l'interface Java du Listener

L'étape suivante est donc de créer une classe héritant de l'implémentation vide du Listener, pour ajouter la logique permettant de créer nos éléments Java correctement pendant le parcours du fichier source Prolog.

4.9.4 Implémentation du listener

L'implémentation du listener de parcours se base sur le principe suivant : on doit obtenir au final la liste des différents termes qui sont à la racine du fichier (qui sont donc les différents faits). La principale difficulté à surmonter est le fait que les termes composés peuvent être imbriqués. Par exemple, quand on entre dans un atome dans le listener, il est impossible de savoir si l'on est dans une structure imbriquée, et si oui dans laquelle on se trouve. En raison de cela, pour suivre le contexte actuel, les structures de données suivantes sont utilisées :

- une liste de termes pour stocker les termes racines
- une pile d'atomes représentant les foncteurs. Le sommet de la pile représente donc le foncteur de la structure la plus récente.
- une pile de listes de termes, représentant les arguments de chaque structure. Le sommet de la pile représente donc les arguments de la structure la plus récente.

En plus de ces structures, un compteur de profondeur sera également tenu à jour, incrémenté lors de l'entrée dans un terme possédant plus d'un descendant (terme composé), et décrémenté à la sortie de ceux-ci. Ce compteur est nécessaire pour choisir s'il faut ajouter le terme courant à la liste des arguments de la structure courante, ou directement à la liste des faits (si la

profondeur est égale à 0). [1]
On commence donc par initialiser nos différentes structures au moment de l'entrée dans le fichier :

```
public void enterProlograal(ProloGraalParser.ProlograalContext ctx) {
    clauses = new ArrayList<>();
    structArgs = new Stack<>();
    functors = new Stack<>();
    depth = 0;
}
```

Lorsque l'on entre dans un terme, si celui-ci a plus d'un enfant, alors il s'agit d'un terme composé, et il faut donc incrémenter le compteur de profondeur en conséquence :

```
public void enterTerm(ProloGraalParser.TermContext ctx) {
    if (ctx.getChildCount() > 1)
        depth++;
}
```

Lorsque l'on sort d'un terme, si celui-ci a plus d'un enfant, alors il faut décrémenter le compteur de profondeur, créer un nouveau terme composé avec le foncteur et les arguments les plus récents (ceux au sommet des piles), et ajouter cette nouvelle structure :

```
public void exitTerm(ProloGraalParser.TermContext ctx) {
    if (ctx.getChildCount() > 1) {
        depth--;
        ProloGraalStructure struct = new ProloGraalStructure(functors.pop(),
        ↪  structArgs.pop());
        add(struct);
    }
}
```

Lorsque l'on ajoute un élément, il faut l'ajouter au bon endroit. Si la profondeur est à zéro, cela signifie que nous sommes à la racine, et il faut donc ajouter l'élément directement à la liste des faits. Sinon, il faut l'ajouter à la liste des arguments du terme composé le plus récent :

```
private void add(ProloGraalTerm term) {
    if (depth == 0)
        clauses.add(term);
    else
        structArgs.peek().add(term);
}
```

Lorsque l'on entre dans un atome, il faut vérifier s'il s'agit d'un foncteur ou non. S'il s'agit d'un foncteur, la logique est gérée au moment de l'entrée dans un foncteur et il ne faut donc rien faire de plus. Sinon, il faut simplement créer un nouvel atome et l'ajouter :

```
public void enterAtom(ProloGraalParser.AtomContext ctx) {
    // if the parent is a functor
```

1. en rétrospective, ce compteur n'est pas utile et aurait pu être remplacé par une simple vérification de la taille de la pile de foncteurs (si elle est vide, nous sommes à la racine...)

```
    if (ctx.parent instanceof ProloGraalParser.FunctorContext) {
        return; // already handled
    }
    add(new ProloGraalAtom(ctx.getText()));
}
```

Lorsque l'on entre dans un foncteur, il faut créer un nouvel atome pour ce foncteur et l'ajouter à la pile des foncteurs. Il faut également créer une nouvelle liste d'arguments et l'ajouter à sa pile :

```
public void enterFunctor(ProloGraalParser.FunctorContext ctx) {
    functors.push(new ProloGraalAtom(ctx.getText()));
    structArgs.push(new ArrayList<>());
}
```

Restent uniquement les nombres, qui doivent juste être créés avec le bon type, entier ou à virgule, puis ajoutés :

```
public void enterNumber(ProloGraalParser.NumberContext ctx) {
    String n = ctx.getText();
    try {
        add(new ProloGraalIntegerNumber(Integer.parseInt(n)));
    } catch (NumberFormatException ex) {
        add(new ProloGraalDoubleNumber(Double.parseDouble(n)));
    }
}
```

4.10 Implémentation : language (suite)

La partie parser étant terminée, il est temps de la lier avec la partie principale. Faisons d'abord un point sur ce que nous avons actuellement :

- un parser fonctionnel, nous retournant la liste des faits du programmes sous la forme de termes (section 4.9.4)
- une représentation en Java de ces termes (section 4.8 et figure 3)
- une représentation en Java des concepts de succès et d'échec (section 4.8 et figure 4)
- une compréhension basique du fonctionnement de la partie principale (section 4.7.1)

On va donc maintenant repartir de cette partie principale, et la lier à notre parser en récupérant la liste des différents faits.

4.10.1 Liaison au parser

Dans *SimpleLanguage*, l'appel au parser se fait via une méthode statique rajoutée dans le parser généré par ANTLR, appelée `parseSL`. On va donc simplement copier cette méthode et l'adapter à notre fonctionnement dans une nouvelle classe.
Cette méthode est relativement triviale. Elle crée une instance du lexer et du parser, et donne au lexer le fichier source qu'elle reçoit en paramètre. Elle attache également un écouteur d'erreur à ces deux instances. Enfin, elle récupère un `ParseTree` depuis le parser, représentant l'arbre de syntaxe du fichier. On peut donc facilement l'adapter pour qu'elle utilise notre lexer et parser :

```
public static List<ProloGraalTerm> parseProloGraal(ProloGraalLanguage
↪  language, Source source) {
    ProloGraalLexer lexer = new
    ↪  ProloGraalLexer(CharStreams.fromString(source.getCharacters().toString()));
```

```
    ProloGraalParser parser = new ProloGraalParser(new
    ↪  CommonTokenStream(lexer));
    lexer.removeErrorListeners();
    parser.removeErrorListeners();
    BailoutErrorListener errListener = new BailoutErrorListener(source);
    lexer.addErrorListener(errListener);
    parser.addErrorListener(errListener);
    ProloGraalParserImpl.source = source;
    ParseTree tree = parser.prolograal();
    // ...
```

Il reste à ajouter le code nécessaire pour utiliser notre listener. Il faut en créer une instance, et l'appeler à l'aide d'un `ParseTreeWalker`. Le `ParseTreeWalker` se charge d'appeler les méthodes "enter" et "exit" au bon moment. Enfin, il faut également retourner la liste des faits à l'appelant :

```
    // ...
    ProloGraalListenerImpl listener = new ProloGraalListenerImpl();
    ParseTreeWalker.DEFAULT.walk(listener, tree);

    listener.debug();

    return listener.getClauses();
}
```

On peut maintenant récupérer les clauses dans notre fichier principal :

```
clauses = ProloGraalParserImpl.parseProloGraal(this, source);
```

4.10.2 Interpréteur : Analyse

Nous pouvons maintenant lire un fichier Prolog, et manipuler les faits qu'il contient en Java. Il nous faut donc à présent en faire quelque chose. Pour cela, nous avons besoin d'un interpréteur pour lire les requêtes de l'utilisateur.
Deux endroits sont possibles pour l'implémentation d'un tel interpréteur :

- Dans la partie "launcher". L'interpréteur serait externe au langage même, et il faudrait trouver un mécanisme pour envoyer les requêtes et récupérer leurs résultats.
- Directement dans la partie "language". L'interpréteur serait donc interne et fortement lié au langage, ce qui facilite le traitement des requêtes.

Le second choix semble donc meilleur. De plus, *SimpleLanguage* possède une implémentation des fonctions "eval" (permettant d'évaluer du code à la volée) et "read" (permettant de lire l'entrée standard), qui sont précisément ce que l'on va chercher à faire. Conceptuellement, nous allons donc faire comme si notre fichier Prolog se terminait toujours par une instruction "read" qui enverrait son contenu à une fonction "eval".
Mais d'abord, il faut encore se pencher sur le `EvalRootNode` qui est, comme indiqué à la section 4.7.1, le point d'entrée du programme après le parsing du code source.

4.10.3 Adaptation du `EvalRootNode` : analyse

Dans la classe `SLEvalRootNode`, on trouve une méthode particulièrement intéressante : la méthode `execute`. Cette méthode sauvegarde les fonctions obtenues après l'évaluation du code

source dans le contexte, et crée un nouveau noeud pour la fonction "main", qui est ensuite appelé. Dans notre cas, les fonctions correspondent aux prédicats Prolog, et il n'existe pas de prédicat "main". On en déduit donc deux modifications à effectuer : adapter le contexte pour qu'il puisse sauvegarder nos faits, et créer puis exécuter un noeud pour notre interpréteur, qui sera l'équivalent de notre fonction "main".

4.10.4 Adaptation du contexte

Le contexte de *SimpleLanguage*, `SLContext`, contient de nombreuses méthodes. Celles qui nous intéressent actuellement sont : les méthodes pour récupérer les flux d'entrée et de sortie, qui nous seront utiles pour l'interpréteur, et la méthode pour récupérer le `SLFunctionRegistry`. On va donc simplifier la classe : on enlève toutes les méthodes qui n'ont pas l'air utile pour le moment. On garde les méthodes d'entrée/sortie, sans même avoir besoin de les adapter. On enlève également le `SLFunctionRegistry`, que l'on remplace par une simple liste de termes correspondant à nos faits. Nous obtenons un contexte capable de stocker nos faits et de fournir les flux d'entrée et de sortie, tout ce dont nous avons besoin pour réaliser l'interpréteur.

4.10.5 Adaptation du `EvalRootNode` : implémentation

Le contexte étant maintenant prêt, on peut adapter le `EvalRootNode`. On enregistre les clauses dans le contexte :

```
reference.get().registerClauses(clauses);
```

On va également créer et enregister dans le contexte les noeuds d'interprétation et de résolution (discutés dans les sections suivantes), puis finalement invoquer le noeud d'interprétation.

```
ProloGraalInterpreterNode interpreterNode = new
↪  ProloGraalInterpreterNode(language);
ProloGraalResolverNode resolverNode = new ProloGraalResolverNode(language);

reference.get().registerInterpreter(interpreterNode);
reference.get().registerResolver(resolverNode);
// ...

return Truffle.getRuntime()
    .createCallTarget(reference.get().getInterpreterNode()).call();
```

4.10.6 Interpréteur : Implémentation

En s'inspirant du code des fonctions d'évaluation et de lecture d'entrée de *SimpleLanguage*, `SLEvalBuiltin` et `SLReadInBuiltin`, on crée une classe `ProloGraalInterpreterNode` qui hérite de `RootNode`, ce qui lui permet d'être exécutée via sa méthode `execute`. Le déroulement de cette méthode est très simple : on lit l'entrée standard, on parse la ligne ainsi obtenue, on exécute cette ligne et on affiche le succès ou l'échec de l'opération, puis on recommence toutes ces opérations jusqu'à l'arrêt du programme par l'utilisateur.
Pour interpréter la ligne lue, on peut réutiliser le parser principal, de la même façon que pour lire le fichier Prolog initial :

```
Source source = Source.newBuilder("pl", line, null).build();
List<ProloGraalTerm> terms = ProloGraalParserImpl.parseProloGraal(language,
↪  source);
```

Il faut ensuite vérifier que l'on a bien lu un fait, et surtout lancer le processus pour vérifier si ce fait est présent dans ceux lus depuis le fichier Prolog. Cette tâche étant vouée à devenir de plus en plus complexe au fil du projet, il a été décidé de séparer le traitement dans une nouvelle classe, le `ProloGraalResolverNode`. Comme pour l'interpréteur, cette classe sera de type `RootNode`. On lui fournit la liste des clauses lues depuis l'entrée standard (pour l'instant, il doit y en avoir exactement une), et elle doit nous retourner si oui ou non la clause est compatible avec un des faits de la liste construite depuis le fichier Prolog.
Dans l'interpréteur, ces opérations prennent la forme suivante :

```
System.out.println(Truffle.getRuntime()
    .createCallTarget(context.getResolverNode()).call(terms));
```

4.10.7 Noeud de résolution

Pour cette phase, il n'y pas de variables ou de clauses, uniquement des faits, et on ne gère que le cas où la requête est composée d'un unique fait. Le noeud de résolution doit donc uniquement trouver un fait dans la liste construite depuis le fichier Prolog qui est exactement égal au fait que l'on reçoit depuis l'interpréteur. Comme pour l'interpréteur, ce noeud hérite de `RootNode` et a donc une méthode `execute` qui sera appelée par Truffle.
Récupération du but, qui est le fait dont on veut vérifier l'existence :

```
ProloGraalTerm goal = ((List<ProloGraalTerm>) frame.getArguments()[0]).get(0);
```

Vérification de l'existence du fait dans la liste :

```
return ProloGraalBoolean
    .fromBoolean(terms.stream().anyMatch(x -> x.equals(goal)));
```

Cette implémentation se base sur la méthode "equals", ce qui implique que tous nos termes implémentent correctement cette méthode. On s'en asssure en déclarant cette méthode comme abstraite dans notre classe de base, `ProloGraalTerm` (cf. section 4.8, figure 3) :

```
@Override
public abstract boolean equals(Object obj);
```

À ce stade, vu qu'il n'y pas de variables, l'implémentation est triviale pour les atomes et les nombres, où il faut simplement vérifier qu'ils ont le même nom/la même valeur. Pour les structures, on doit vérifier récursivement chaque sous-terme :

```
if(obj == null) return false;
if(!(obj instanceof ProloGraalStructure)) return false;
ProloGraalStructure other = (ProloGraalStructure)obj;
return subterms.equals(other.subterms);
```

Ceci conclut l'implémentation du noeud de résolution, et de la phase 1 du projet.

4.11 Synthèse

La phase 1 du projet est maintenant terminée. Voici ce que nous avons actuellement :

— Un parser capable de transformer un fichier Prolog en sa représentation Java. Ce parser est pour l'instant capable de comprendre des faits, composés d'atomes, de nombres et de structures composées, mais pas de variables. Il est également réutilisable pour transformer l'entrée standard en sa représentation Java dans l'interpréteur.

— Une représentation "intelligente" des différents types de termes Prolog, capable de vérifier si un terme est exactement égal à un autre.
— Un contexte stockant la liste des faits obtenus depuis le parser, ainsi que les références vers l'interpréteur et le noeud de résolution.
— Un interpréteur capable de lire l'entrée standard, d'interpréter une ligne reçue depuis cette entrée à l'aide du parser et de demander au noeud de résolution de vérifier si cette ligne produit un succès ou un échec.
— Un noeud de résolution très simple capable de vérifier via l'intelligence des types si le fait reçu est exactement égal à un des faits présents dans la liste du contexte.

5 Phase 2 : Variables

Cette section documente le travail réalisé pour atteindre les fonctionnalités de la phase 2 du projet. Ces fonctionnalités sont les mêmes que celles de la phase 1, mais avec en plus le support des variables dans les requêtes et dans les fichiers Prolog.

5.1 Besoins

Les besoins de cette phase sont les suivants :
— Ajout du support des variables au parser
— Ajout d'une représentation en Java d'une variable Prolog
— Implémentation de l'unification
— Support de l'affichage des variables ayant conduit à un succès

5.2 Plan

En se basant sur l'état actuel du projet, on peut dresser le plan de réalisation suivant : modifier les composants existants pour intégrer les variables, en commençant par trouver une représentation en Java pour les variables et adapter le parser pour celles-ci, puis continuer en adaptant l'interpréteur et le noeud de résolution.

5.3 Représentation des variables

En se basant sur la structure décrite à la section 4.8, on sait que les variables sont également des termes. Les variables logiques ont cependant une particularité : elles représentent des inconnues, et peuvent donc prendre une valeur si cela s'avère nécessaire pour s'unifier. En conséquence, pour modéliser ce comportement, elles peuvent se retrouver liées à un autre terme (qui peut être une autre variable). La figure 6 montre une représentation possible des éléments Prolog en Java, avec les variables.

5.4 Adaptation du parser

Pour adapter le parser afin que celui-ci intègre les variables, trois éléments sont à modifier, dans l'ordre :

1. Le lexer dans la grammaire ANTLR, pour qu'il comprenne que les chaînes commençant par une majuscule(ou un tiret bas) sont des variables
2. Le parser dans la grammaire ANTLR, pour qu'il comprenne qu'une variable est composée de l'unité "variable" du lexer
3. Le listener, pour créer les représentations Java des variables

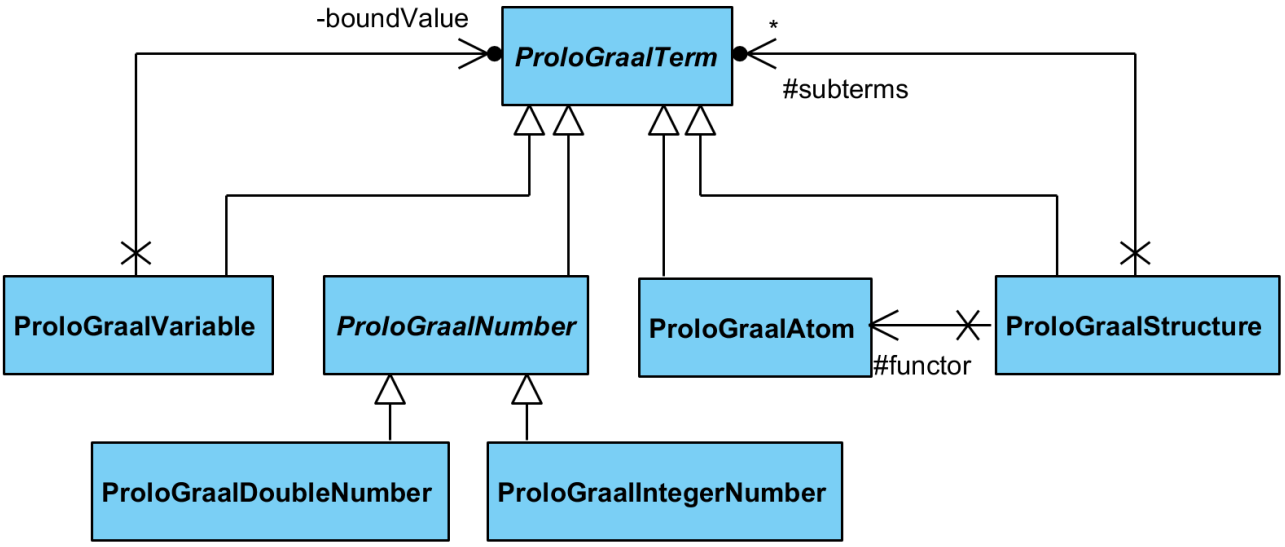


FIGURE 6 – Diagramme de classes des éléments Prolog en Java, variables incluses

5.4.1 Lexer

Dans la grammaire ANTLR, on rajoute un nouvel élément représentant une variable au lexer. Une variable commence soit par une majuscule, soit par un tiret bas, suivi d'un certain nombre de caractères.

```
VARIABLE : ((UPPERCASE | UNDERSCORE) (LOWERCASE | UPPERCASE | DIGITS |
↪  UNDERSCORE)*);
```

5.4.2 Parser

Pour la partie parser, on doit rajouter dans la grammaire ANTLR deux choses : un terminal composé de l'élément du lexer "variable", et dire qu'un terme peut être aussi une variable.

```
variable :
VARIABLE
;
term :
functor ('(' term (',' term)* ')') |
atom |
number |
variable
;
```

5.4.3 Listener

Pour le listener, on pourrait penser qu'il suffit de créer une nouvelle variable avec le bon nom et de l'ajouter à la structure courante. Mais cette approche sera rapidement victime d'un problème : si la même variable est référencée plusieurs fois dans la même structure (exemple : `fait(X, X).`), alors deux instances indépendantes seront créées, ce qui pose problème au moment où une de ces variables prendra une valeur, car l'autre doit obligatoirement prendre la même valeur pour que le comportement soit correct, ce qui nécessiterait de gérer la synchronisation manuellement.

Pour pallier ce problème, on va profiter du mécanisme de référence de variables en Java. L'algorithme imaginé est le suivant [2] : on tient à jour une liste des variables de la structure courante. Quand on rencontre une variable, on vérifie si elle existe déjà dans cette liste. Si oui, alors on la récupère et on place sa référence dans la structure courante. Si non, on la crée et on l'ajoute à la structure courante et à sa liste de variables.
Dans l'implémentation, la liste de variables est en fait une liste de listes : chaque entrée correspond à une entrée dans la liste des clauses, et pour chaque clause on peut avoir un certain nombre de variables :

```
private List<ProloGraalTerm<?>> clauses;
private List<List<ProloGraalVariable>> variablesForEachClause;
```

Dans la méthode `enterVariable`, on va réaliser l'algorithme présenté précédemment. On commence par créer une variable représentant celle que l'on vient de rencontrer dans le code source, pour pouvoir la comparer avec les existantes. On récupère également la liste des variables de la clause courante :

```
ProloGraalVariable var = new ProloGraalVariable(ctx.getText());
List<ProloGraalVariable> variablesForThisClause =
 ↪  variablesForEachClause.get(variablesForEachClause.size() - 1);
```

On va ensuite chercher dans la liste des variables pour voir si la variable actuellement traitée est nouvelle ou non, et l'ajouter si nécessaire. À l'aide de la *Stream API* de Java et de quelques astuces, on peut écrire cette opération de manière très concise et élégante :

```
final ProloGraalVariable var2 = var;
// if the variable was previsouly encountered in this scope, do not add a
 ↪  new one
var = variablesForThisClause.stream().filter(x ->
 ↪  x.equals(var2)).findFirst().orElseGet(() -> {
    variablesForThisClause.add(var2);
    return var2;
});
add(var);
```

5.4.4 Retour du parser

Pour simplifier les opérations sur les variables, comme par exemple la remise à zéro après la résolution d'un but, il serait intéressant de conserver directement la liste des variables et de la rendre disponible pendant la résolution. On va donc créer une nouvelle classe `ProloGraalParseResult`, dont l'unique but est de stocker à la fois une référence vers la liste des clauses, et une référence vers la liste de listes des variables. Le retour final du parser est donc le suivant :

```
return new ProloGraalParseResult(listener.getClauses(),
 ↪  listener.getVariables());
```

Dans la classe `ProloGraalLanguage`, on va donc récupérer cet objet et le transmettre au `ProloGraalEvalRootNode`, qui se charge d'enregister la référence vers les clauses et celle vers les variables dans le contexte. Ces deux éléments sont donc accessibles pour notre `ResolverNode`.

2. une version améliorée de cet algorithme basée sur des "Map" est présentée plus tard à la section 6.4.3

5.5 Adaptation du noeud de résolution

Vu que nous avons maintenant des variables, il ne suffit plus de chercher les termes étant exactement égaux au but courant ; il faut trouver tous les termes *unifiables* avec le but courant. L'algorithme est le suivant : pour chacun des termes du contexte (nos faits), on regarde s'il est unifiable avec le but. Si oui, on l'ajoute à la liste des succès. Il faut ensuite défaire tous les changements que le processus d'unification aurait pu engendrer, en réinitialisant les variables du but et de chaque fait. Cet algorithme se code de cette manière :

```
for(int i = 0; i < terms.size(); i++) {
    ProloGraalTerm<?> clause = terms.get(i);
    if(clause.isUnifiable(goal)) {
        System.out.println("Success with clause : " + clause.toString());
        successes.add(new ProloGraalSuccess(clause, variables.get(i)));
    }
    variables.get(i).forEach(ProloGraalVariable::unbind);
    goalVariables.forEach(ProloGraalVariable::unbind);
}
```

Reste à définir le processus d'unification et sa logique propre à chaque type de terme.

5.6 L'unification

Le processus d'unification dépend du type du terme en question. Pour représenter ce comportement, la surcharge de fonctions avec notre structure présentée à la figure 6 se prête particulièrement bien. On déclare donc notre méthode d'unification de la manière suivante, dans notre classe mère [3] :

```
public abstract boolean isUnifiable(ProloGraalTerm<?> other);
```

5.6.1 Principe général

Le principe général de l'unification mis en place est que chaque type de terme est responsable de l'unification avec un élément de son propre type. Mais si l'autre élément est une variable, alors la logique est déléguée à la variable. L'unification étant bidirectionnelle, le résultat ne change pas, mais l'implémentation est simplifiée avec toute la logique inter-types présente uniquement dans le code des variables.

5.6.2 Atomes et nombres

Pour les atomes et les nombres, le processus est très simple : pour que l'unification soit possible, il faut que l'autre élément soit du même type, avec la même valeur, ou qu'il soit une variable :

```
@Override
public boolean isUnifiable(ProloGraalTerm<?> other) {
    if(this.equals(other)) {
        return true;
    } else if(other instanceof ProloGraalVariable) {
        return ((ProloGraalVariable)other).isUnifiable(this);
```

3. Son nom n'est pas particulièrement bien choisi, vu qu'elle peut modifier les objets pour les rendre unifiables. Elle sera plus tard renommée en simplement `unify`.

```
    }
    return false;
}
```

On remarque également le principe expliqué précédemment de déléguation de la logique à la variable.

5.6.3 Structures

Pour les structures, il faut que les deux aient le même foncteur et la même arité, et il faut que chaque sous-terme s'unifie avec son correspondant direct (le premier sous-terme de la première structure doit s'unifier avec le premier sous-terme de la seconde structure). L'unification est donc un succès uniquement si tous les sous-termes sont unifiables. Même si l'on peut potentiellement modifier les sous-termes, les opérations seront quand même correctement défaites en cas d'échec, car toutes les variables sont dans la même liste, même celles qui sont imbriquées (cf. 5.4.3).

```
@Override
public boolean isUnifiable(ProloGraalTerm<?> other) {
    if(this.equals(other)) {
        for(int i = 0; i < this.arity; i++) {
            if(!this.subterms.get(i)
                .isUnifiable(((ProloGraalStructure)other).subterms.get(i))) {
                return false;
            }
        }
        return true;
    } else if(other instanceof ProloGraalVariable) {
        return ((ProloGraalVariable)other).isUnifiable(this);
    }
    return false;
}
```

5.6.4 Variables

Pour les variables, le processus se décline en deux variantes :

- Si la variable est déjà liée à une autre valeur, alors on essaie d'unifier cette valeur avec l'autre élément.
- Si la variable n'est pas déjà liée, alors on la lie avec l'autre élément et l'unification est un succès.

Il y a également tous les cas spéciaux dans le cas où l'autre élément est également une variable, et les problèmes engendrés par les références cycliques. Ces problèmes sont analysés et partiellement résolus plus tard dans la section 6.6. Mais pour l'instant, on en reste à une implémentation basique :

```
@Override
public boolean isUnifiable(ProloGraalTerm<?> other) {
    // can always unify to itself
    if(other instanceof ProloGraalVariable && (ProloGraalVariable)other ==
    ↪  this)
        return true;
```

```
    if(this.isBound) {
        if(other instanceof ProloGraalVariable) {
            ProloGraalVariable otherVar = (ProloGraalVariable)other;
            if(otherVar.isBound) {
                return this.boundValue.isUnifiable(otherVar.boundValue);
            } else {
                otherVar.bind(this);
                return true;
            }
        } else {
            return this.boundValue.isUnifiable(other);
        }
    } else {
        this.bind(other);
        return true;
    }
}
```

Pour aider à ce processus, une méthode permettant de lier la variable à un élément a été implémentée. Cette méthode pratique également un test très simple [4] pour tenter de détecter les cas cycliques :

```
public void bind(ProloGraalTerm<?> other) {
    if(other instanceof ProloGraalVariable) {
        ProloGraalVariable otherVar = (ProloGraalVariable)other;
        if(otherVar.isBound && otherVar.boundValue == this) {
            return;
        }
    }
    this.isBound = true;
    this.boundValue = other;
    System.out.println(name + " = " + other);
}
```

Ainsi que sa correspondante permettant de défaire la liaison :

```
public void unbind() {
    this.isBound = false;
    this.boundValue = null;
}
```

5.7 Affichage des résultats

Pour le retour des résultats depuis le noeud de résolution et ensuite leur affichage dans l'interpréteur, l'état des variables du but est sauvegardé à chaque succès. Pour cela, une méthode de copie de l'état actuel d'une variable est implémentée :

```
@Override
public ProloGraalVariable copy() {
```

4. et très incomplet...

```
    ProloGraalVariable var = new ProloGraalVariable(this.name);
    var.isBound = this.isBound;
    if(this.isBound) {
        var.boundValue = this.boundValue.copy();
    }
    return var;
}
```

Ces succès se retrouvent encapsulés dans une liste de `ProloGraalSuccess` qui est remontée au noeud d'interprétation. Celui-ci se charge de lire l'état des variables pour chaque succès et de l'afficher :

```
ProloGraalBoolean callResult = (ProloGraalBoolean)
↪  Truffle.getRuntime().createCallTarget(context.getResolverNode()).call(runtime);
if(callResult.asBoolean()) {
    ProloGraalSuccess success = (ProloGraalSuccess)callResult;
    if(success.getSuccesses().size() > 0) {
        for(ProloGraalSuccess succ : success.getSuccesses()) {
            succ.getVariables().forEach(x -> {
                if(x.isBound())
                    writer.println(x);
            });
        }
    }
}
writer.println(callResult);
```

5.8 Synthèse

La phase 2 est maintenant terminée. Il reste de nombreux bugs (dont beaucoup qui seront découverts par la suite), mais l'objectif de cette phase est atteint : nous avons une représentation des variables Prolog en Java, ainsi qu'un mécanisme d'unification sommaire.

6 Phase 3

Cette section documente le travail réalisé pour atteindre les fonctionnalités de la phase 3 du projet. En plus de fonctionnalités de la phase 2 (faits avec support des variables), on rajoute le support des clauses "complexes", c'est à dire avec un corps. On rajoute également le support des listes.

6.1 Besoins

Les besoins de cette phase sont les suivants :

- Ajout du support des clauses complexes au parser, avec leur représentation en Java
- Ajout du support des listes au parser, avec leur représentation en Java
- Implémentation de l'arbre de preuve pour résoudre les buts

6.2 Plan

Comme pour les phases précédentes, le plan va être de commencer par trouver une représentation Java pour les nouveaux éléments, puis adapter le parser pour ceux-ci. Enfin, on va adapter le noeud de résolution pour inclure l'arbre de preuve.

6.3 Représentation des éléments

Quelques ajouts et modifications sont nécessaires pour représenter correctement les nouveaux éléments. Ces changements sont décrits en détail dans les sections suivantes, et le diagramme de classe résultant est montré avec la figure 8.

6.3.1 Listes

Une liste en Prolog n'est rien d'autre qu'une structure récursive avec le foncteur `'.'`, comme le montre la figure 7. On va donc profiter de cette représentation interne pour ne pas devoir dupliquer les opérations d'unification et autres. Ce qui se traduit en code par un héritage depuis de `ProloGraalStructure`. En plus des sous-termes d'une structure classique, on va également garder directement une liste interne des éléments de la liste, ainsi qu'une référence directe vers la queue de la liste. Ces éléments ne sont pas directement utilisés à part au moment de la construction ; cependant, ils pourraient servir à d'éventuelles optimisations [5]. La méthode clé

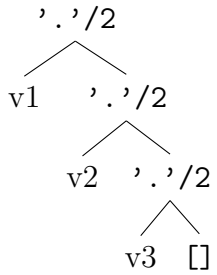


FIGURE 7 – Représentation interne de la liste `[v1,v2,v3]`

de la représentation est celle permettant de créer la représentation interne de la liste. Elle va remplir les sous-termes de la structure avec les bon élément `'.'`. Comme les listes Prolog sont récursives par nature, l'implémentation l'est également. Le procédé est le suivant : on garde un lien vers la structure courante. On met son foncteur au foncteur `'.'`. Pour chaque élément de la liste jusqu'à l'avant-dernier, on l'ajoute comme premier sous-terme de la structure courante. On crée une nouvelle structure avec comme foncteur toujours `'.'`. Enfin, on ajoute cette nouvelle structure comme deuxième sous-terme de la structure courante, et la nouvelle structure devient la courante. Pour le dernier élément, on l'ajoute en temps que sous-terme de la structure courante, comme pour les précédents, mais le second sous-terme n'est plus une structure `'.'/2` mais la queue de la liste.

```
// list elements
protected List<ProloGraalTerm<?>> items;
// tail of the list. Empty by default, but a setter is provided.
protected ProloGraalTerm<?> tail = ProloGraalBuiltinAtoms.EMPTY_LIST;

/**
 * Create the parent structure based on the internal representation of a
↪  Prolog list.
 */
public void buildInteralRepresentation() {
    this.setFunctor(ProloGraalBuiltinAtoms.DOT_OPERATOR);
```

5. En l'état actuel, il n'est donc pas très utile de faire la différence entre les listes et les structures...

```
        ProloGraalStructure struct = this;
        // for each item until the tail
        for(int i = 0; i < items.size()-1; i++) {
            ProloGraalTerm<?> item = items.get(i);
            // add "left side" to the representation
            struct.addSubterm(item);
            // create "right side" of the representation
            ProloGraalStructure next = new ProloGraalStructure(variables);
            next.setFunctor(ProloGraalBuiltinAtoms.DOT_OPERATOR);
            // add "right side"
            struct.addSubterm(next);
            // switch to the new structure on the right side
            struct = next;
        }
        ProloGraalTerm<?> item = items.get(items.size()-1);
        struct.addSubterm(item);
        struct.addSubterm(this.tail);
    }
```

6.3.2 Clauses

Une clause est composée d'une tête, qui est un terme, et une liste de buts, qui sont également des termes. La représentation d'une clause n'a pas vraiment d'intelligence, elle sert uniquement à stocker la tête et les buts.

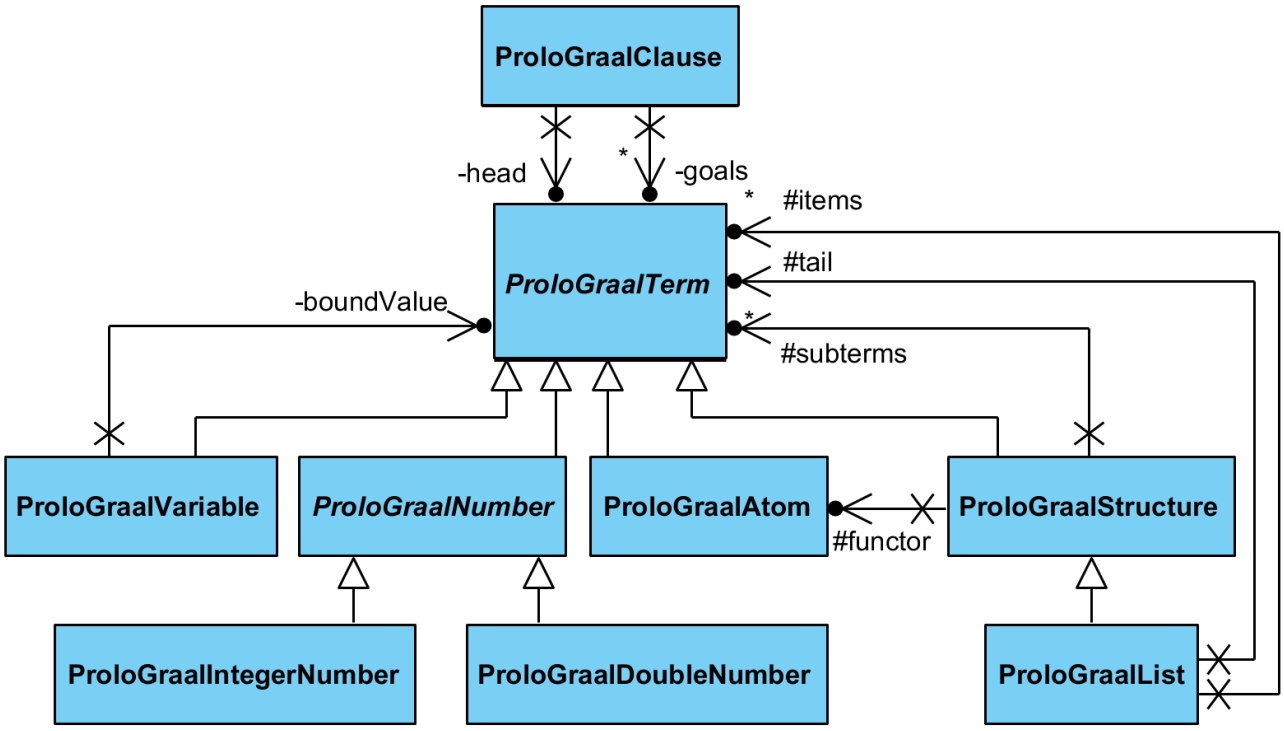


FIGURE 8 – Diagramme de classes des éléments Prolog en Java, avec listes et clauses

6.4 Adaptation du parser

Les modifications suivantes sont à faire dans tout le trio lexer-parser-listener :

— Ajouter les listes
— Ajouter les clauses complexes

6.4.1 Lexer

Dans le lexer, on doit rajouter les nouveaux éléments suivantes : les délimiteurs de liste `[` et `]`, le marqueur de queue de liste `|`, la virgule pour séparer les buts et les éléments des listes, et finalement le marqueur de corps de clause `:-` :

```
LIST_START : '[';
LIST_END : ']';
LIST_ENDING : '|';
SEPARATOR : ',';
CLAUSE_MARKER : ':-';
```

6.4.2 Parser

Dans le parser, on doit maintenant utiliser nos nouveaux éléments de lexer. On commence par la liste vide, qui est un atome :

```
atom :
ATOM |
LIST_START LIST_END // empty list
;
```

Puis on définit la liste, qui commence par un `[` et se termine par un `]`, qui contient un ou plus termes séparés par une virgule, et qui se termine optionnellement par un `|` suivi d'une queue. La queue est un terme quelconque.

```
tail :
term
;

list :
LIST_START
term (SEPARATOR term)* (LIST_ENDING tail)?
LIST_END
;
```

On continue en modifiant le terme pour inclure la possibilité qu'il soit une liste. On extrait également la définition du terme composé, qui était auparavant incluse dans la définition du terme, pour pouvoir la réutiliser indépendamment plus tard :

```
composedTerm :
functor ('(' term (',' term)* ')')
;

term :
composedTerm |
atom |
number |
```

```
variable |
list
;
```

On définit ensuite la clause, qui peut être soit un fait, soit une clause composée (ou clause complexe).

```
clause :
fact |
composedClause
;
```

Une clause composée est constituée d'une tête, qui peut être un atome ou un terme composé, suivie du marqueur de corps `:-`, suivi d'un certain nombre de buts séparés par des virgules qui peuvent chacun être un atome ou un terme composé. Enfin, la clause se termine par un point.

```
head :
atom |
composedTerm
;

goal :
atom |
composedTerm
;

composedClause :
head CLAUSE_MARKER goal (SEPARATOR goal)* TERMINATOR
;
```

Même si la tête et les buts peuvent être composés par les mêmes éléments, la distinction est quand même faite pour faciliter le travail du listener.

6.4.3 Listener

Le listener a été complètement repensé pour cette étape, avec une manière de faire bien plus simple qu'avant. Au lieu des différentes listes (cf. 4.9.4), on utilise deux piles : une pour les éléments divers que l'on rencontre au fur et à mesure, et une autre pour les clauses.

```
private Deque<ProloGraalTerm<?>> elements = new ArrayDeque<>();
private Deque<ProloGraalClause> clauses = new ArrayDeque<>();
```

On utilise la classe `ArrayDeque` plutôt que la classe `Stack` pour les raisons suivantes, tirées de la documentation officielle :

- Stack : *A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class.* [6]
- ArrayDeque : *This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.* [7]

6. https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html
7. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html

On va ensuite profiter des événements d'entrée mais surtout de ceux de sortie pour ajouter les éléments aux bons endroits. Le principe est le suivant : quand on arrive à un élément "primitif", on en crée la représentation Java et on l'ajoute à la pile des éléments, sans se soucier de l'ajouter à une structure ou autre :

```
@Override
public void enterAtom(ProloGraalParser.AtomContext ctx) {
    ProloGraalAtom atom = new ProloGraalAtom(clauses.peek().getVariables(),
    ↪  ctx.getText());
    elements.push(atom);
}

@Override
public void enterList(ProloGraalParser.ListContext ctx) {
    elements.push(new ProloGraalList(clauses.peek().getVariables()));
}

@Override
public void enterComposedTerm(ProloGraalParser.ComposedTermContext ctx) {
    elements.push(new ProloGraalStructure(clauses.peek().getVariables()));
}
```

De manière analogue, quand on entre dans une clause, on en crée une nouvelle et on l'ajoute à la pile des clauses :

```
@Override
public void enterClause(ProloGraalParser.ClauseContext ctx) {
    clauses.push(new ProloGraalClause());
}
```

La logique s'effectue ensuite au moment de la sortie des éléments. Par exemple, quand on quitte un terme composé, on va enlever des éléments de la pile et les stocker dans une liste jusqu'à ce qu'on arrive à un élément de type "terme composé" encore vide. Cela signifie que l'on a atteint l'élément qui devra contenir les sous-termes accumulés dans la liste (en sens inverse, vu que les éléments arrivent d'une pile).

```
@Override
public void exitComposedTerm(ProloGraalParser.ComposedTermContext ctx) {
    List<ProloGraalTerm<?>> subterms = new ArrayList<>();
    while(true){
        while(!(elements.peek() instanceof ProloGraalStructure)) {
            subterms.add(elements.pop());
        }
        // check if element is a composed term already handled
        if(((ProloGraalStructure)elements.peek()).getArity() > 0) {
            subterms.add(elements.pop());
        } else {
            break;
        }
    }
```

```
    ProloGraalStructure struct = (ProloGraalStructure) elements.peek();
    for(int i = subterms.size()-1; i >= 0; i--) {
        struct.addSubterm(subterms.get(i));
    }
}
```

Cette logique est très simple à utiliser pour les autres éléments également. Par exemple, quand on sort d'un foncteur, on sait que l'élément précédent était forcément la structure qui doit avoir ce foncteur :

```
@Override
public void exitFunctor(ProloGraalParser.FunctorContext ctx) {
    try {
        ProloGraalAtom functor = (ProloGraalAtom) elements.pop();
        ProloGraalStructure struct = (ProloGraalStructure) elements.peek();
        struct.setFunctor(functor);
    } catch(ClassCastException ex) {
        throwParseError(ctx, "Invalid state : " + ex.getLocalizedMessage());
    }
}
```

Pour les variables, on reprend toujours le même concept de vérifier si la variable existe déjà avant de l'ajouter, mais pour assurer le fonctionnement il faut ajouter de toute façon la variable à la pile des éléments.
On remarque qu'on l'ajoute également à la liste des variables de la clause. Cette liste correspond à celle qui était présente de manière globale dans les versions précédentes. Ce n'est cependant plus une liste, mais un dictionnaire, pour rendre la vérification plus rapide (plus besoin de chercher dans toute la liste). En fait, logiquement il s'agit d'un ensemble, mais l'implémentation du `Set` de Java ne permet pas de récupérer une valeur, uniquement de tester si oui ou non elle est présente à l'intérieur. Pour pallier ce problème, une `Map<Variable, Variable>` est donc utilisée.

```
@Override
public void enterVariable(ProloGraalParser.VariableContext ctx) {
    ProloGraalVariable variable = new
    ↪  ProloGraalVariable(clauses.peek().getVariables(), ctx.getText());
    if(clauses.peek().getVariables().containsKey(variable)) {
        // add a reference to the previously added one
        elements.push(clauses.peek().getVariables().get(variable));
    } else {
        elements.push(variable);
        clauses.peek().getVariables().put(variable, variable);
    }
}
```

Il faut encore gérer les listes. Une astuce est de retenir la queue de la liste au moment où l'on en sort. Cette astuce marche même dans le cas récursif où la queue serait également une liste, vu que l'on sortira d'abord de la liste interne, puis on passera dans `exitTail` ce qui mettra la queue pour la liste externe.

```
@Override
public void exitTail(ProloGraalParser.TailContext ctx) {
    tail = elements.pop();
}
```

Le code pour créer la liste est en suite très similaire à celui pour les structures, avec en plus la gestion de la queue et la création de la représentation interne :

```
public void exitList(ProloGraalParser.ListContext ctx) {
    // ...
    if(tail != null) {
        list.setTail(tail);
        tail = null;
    }
    list.buildInteralRepresentation();
}
```

6.4.4 Retour du parser

Pour le retour du parser de la phase 2 (cf. 5.4.4), nous avions introduit la classe `ProloGraalParseResult` pour stocker la liste des faits ainsi que les variables. Cette classe est renommée en `ProloGraalRuntime`, et ne contient plus que les clauses du programme, les variables étant déjà inclues dans ces dernières. Mais ces clauses ne sont pas stockées dans une simple liste. On utilise un dictionnaire reliant une tête à une liste de clauses. Ceci permet de retrouver efficacement les clauses compatibles avec un certain but dans le noeud de résolution. Cette implémentation nécessite cependant l'effort supplémentaire de fournir une méthode `hashCode` fonctionnelle pour chacun des types de terme.

```
private final Map<ProloGraalTerm<?>, List<ProloGraalClause>> clauses;

public ProloGraalRuntime(List<ProloGraalClause> clauseList) {
    clauses = new HashMap<>();

    for (ProloGraalClause clause : clauseList) {
        clauses.putIfAbsent(clause.getHead(), new ArrayList<>());
        List<ProloGraalClause> clauses1 = clauses.get(clause.getHead());
        clauses1.add(clause);
    }
}
```

Comme pour le `ProloGraalParseResult`, ce résultat de parser est stocké dans le contexte et accessible depuis le noeud de résolution.

6.5 Arbre de preuve

Pour l'implémentation de l'arbre de preuve, plusieurs méthodes sont possibles. Les deux ayant été considérées sont les suivantes :

- Méthode récursive utilisant la pile d'appels comme mécanisme pour le parcours de l'arbre
- Méthode utilisant une boucle pour simuler le parcours de l'arbre

La méthode récursive est plus facile à implémenter, mais est rapidement soumise aux limitations de taille de la pile d'appels. Malgré ce désavantage, c'est quand même cette version qui est retenue.

Le fonctionnement de l'arbre de preuve est le suivant : on maintient une liste de buts. On va ensuite remplacer le premier but par un corps possible (dont la tête s'unifie), ce qui crée un noeud par remplacement possible. On va ensuite recommencer cette opération jusqu'à ce qu'on atteigne des faits, ce qui aura pour effet de diminuer la taille de la liste des buts. On continue de recommencer le processus jusqu'à ce que la liste de buts soit vide, ce qui correspond à un succès, ou jusqu'à ce qu'on ne puisse plus remplacer, ce qui correspond à un échec.

6.5.1 Noeud de résolution

Le noeud de résolution sert maintenant à initialiser l'arbre de preuve. Il crée la liste de buts et donne le premier but, qui est la requête qui arrive depuis l'interpréteur. Il va ensuite démarrer le processus de résolution en créant et en exécutant un `ProloGraalProofTreeNode`.

```
ProloGraalTerm<?> head = goalRuntime.getFirstClause().getHead();

Deque<ProloGraalTerm<?>> goals = new ArrayDeque<>();
goals.push(head);
ProloGraalProofTreeNode proofTreeNode = new ProloGraalProofTreeNode(clauses,
↪  goals);

ProloGraalBoolean r = treeNode.execute();
```

Un `Deque` est utilisé plutôt qu'une simple liste pour les buts, car les buts sont insérés au début (on pourrait aussi utiliser un `Stack` ou une `Queue`, mais les raisons citées à la section 6.4.3 s'appliquent également ici).

6.5.2 Noeuds de l'arbre de preuve

Les noeuds de l'arbre de preuve sont responsables de remplacer le premier but par ses possibles corps, et de relancer le traitement récursivement. On commence donc par vérifier si la liste de buts est vide : cela signifie que nous avons atteint un succès, il faut donc le signaler en retournant un `ProloGraalSuccess`.

```
if(goals.isEmpty()) { // leaf node
    return new ProloGraalSuccess();
}
```

On va ensuite récupérer le but actuel (le premier de la liste) ainsi que les candidats potentiellement unifiables. S'il n'y en a pas, alors on lève une erreur.

```
ProloGraalTerm<?> currentGoal = goals.getFirst();
List<ProloGraalClause> possibleClauses = clauses.get(currentGoal);
// if no match, throw an error
if(possibleClauses == null || possibleClauses.isEmpty())
    throw new ProloGraalExistenceError();
```

On va ensuite essayer chacun de ces candidats, en essayant de les unifier avec le but courant. Mais on doit d'abord créer une copie du candidat, pour s'assurer de ne pas corrompre la base de clauses au moment de l'unification. On sauvegarde également l'état du but courant pour les mêmes raisons.

```
for(ProloGraalClause possibleClauseOriginal : possibleClauses) {
    ProloGraalClause possibleClause = possibleClauseOriginal.copy();

    currentGoal.save();

    // if the head of the clause is unifiable with the current goal
    if(possibleClause.getHead().unify(currentGoal)) {
```

Si l'unification ne fonctionne pas, alors il suffit de défaire tous les changements sur le but en appelant la méthode `undo`. Sinon, si l'unification a fonctionné, alors on va créer une nouvelle liste de buts et remplacer le premier par le corps du candidat :

```
// create a copy of the current goals
Deque<ProloGraalTerm<?>> newGoals = new ArrayDeque<>(goals);

List<ProloGraalTerm<?>> body = possibleClause.getGoals();

// always remove the first goal since it will be replaced
newGoals.pollFirst();

// no need for distinction between facts and regular clauses
Collections.reverse(body);
body.forEach(newGoals::addFirst);
```

On remarque l'inversion des buts du corps du candidat. Elle est nécessaire vu que l'on ajoute les buts toujours au début de la liste.
Il ne reste plus qu'à créer et appeler un nouveau noeud en lui donnant la nouvelle liste de buts. À ce stade, on ne gère pas les résultats multiples, donc on retourne immédiatement si cet appel récursif donne un succès.

```
if(new ProloGraalProofTreeNode(clauses, newGoals).execute().asBoolean()) {
    return new ProloGraalSuccess();
}
```

Finalement, il ne faut pas oublier que si l'on a essayé tous les candidats mais que tous ont échoué, alors il faut retourner un échec.

6.5.3 Synthèse

L'implémentation de l'arbre de preuve est maintenant terminée, en assumant un fonctionnement correct du mécanisme d'unification. Ce n'est cependant pas le cas, la phase précédente ayant laissé une unification très approximative. On va donc s'y repencher.

6.6 Correction des problèmes d'unification

Il y a principalement deux gros problèmes dans le mécanisme d'unification actuel des variables qui ont été omis :

- Le concept de valeur racine
- Le test d'occurrence

6.6.1 Valeur racine

Quand on essaie d'unifier des variables, particulièrement entre-elles, il est important de toujours travailler sur la "valeur racine" de celles-ci. La valeur racine est le début de la possible chaîne de références d'une variable. Par exemple, si la variable `A` est liée à `B`, qui est elle même liée à la variable `C`, qui est liée à l'atome `x`, alors la valeur racine de `A` doit être `x`. L'implémentation précédente ne prenait pas en compte ce genre de cas, en ne regardant qu'à une distance maximum de 1.
On implémente donc une méthode permettant de récupérer la valeur racine d'une variable, que l'on utilisera chaque fois que l'on a besoin de la valeur de cette variable. L'algorithme est simple : si la variable n'est pas liée, alors il s'agit de la racine. Sinon, on regarde à quoi est liée la variable. Si c'est une autre variable, alors on appelle cette méthode récursivement. Sinon, on retourne simplement la valeur liée.

```
@Override
public ProloGraalTerm<?> getRootValue() {
    if (this.isBound) {
        if (this.boundValue instanceof ProloGraalVariable) {
            return this.boundValue.getRootValue();
        } else {
            return this.boundValue;
        }
    } else {
        return this;
    }
}
```

On utilise ensuite cette méthode lors de l'unification par exemple :

```
ProloGraalTerm<?> rootValue = other.getRootValue();
if (this.isBound) {
    return this.boundValue.getRootValue().unify(rootValue);
} else {
    this.bind(rootValue);
    return true;
}
```

6.6.2 Test d'occurrence

Le test d'occurrence empêche l'unification d'une variable avec une structure la contenant. Par exemple, `X` ne doit pas pouvoir s'unifier à `f(X)`. L'implémentation de ce test est assez lourde, car on a besoin de maintenir la liste des variables présentes dans chaque structure, et on doit vérifier toutes les variables pour voir si leur valeur racine revient vers la variable que l'on cherche. On doit également vérifier récursivement si ces variables sont liés à d'autres structures :

```
private boolean occursCheck(ProloGraalStructure root) {
    Map<ProloGraalVariable, ProloGraalVariable> subVariables =
     ↪  root.getSubVariables();
    if (subVariables.get(this) == this) { // contains directly this
        return true;
    }
```

```
        for (ProloGraalVariable var : subVariables.values()) {
            ProloGraalTerm<?> rootValue = var.getRootValue();
            if (rootValue == this) { // or contains indirectly
                return true;
            }
            if (rootValue instanceof ProloGraalStructure) {
                if (occursCheck((ProloGraalStructure) rootValue)) {
                    return true;
                }
            }
        }

        return false;
    }
```

On glisse ce test durant le processus d'unification :

```
ProloGraalTerm<?> rootValue = other.getRootValue();
if (rootValue instanceof ProloGraalStructure) {
    if (occursCheck((ProloGraalStructure) rootValue)) {
        return false;
    }
}
```

6.7 Affichage des résultats

L'affichage des résultats est assez similaire à celui-ci de la phase 2. Le noeud de résolution encapsule les variables du but originel dans un succès :

```
if(r.asBoolean()) {
    r = new ProloGraalSuccess(head.getVariables());
}
return r;
```

Et comme à la phase 2 ce succès est récupéré par l'interpréteur, et ses variables sont affichées.

```
if(callResult.asBoolean()) {
    ProloGraalSuccess success = (ProloGraalSuccess)callResult;
    for(ProloGraalVariable variable : success.getVariables().values()) {
        if(variable.isBound()) {
            ProloGraalTerm<?> root = variable.getRootValue();
            String rootStr;
            if(root instanceof ProloGraalStructure) {
                rootStr = ((ProloGraalStructure) root).toRootString();
            } else {
                rootStr = root.toString();
            }
            writer.println(variable.getName() + " = " + rootStr);
        }
    }
}
```

Seule particularité, l'appel à une méthode `toRootString` dans le cas où une variable est liée à une structure. Cette méthode sert particulièrement dans le cas où la structure est une liste ; elle va en effet formatter correctement la liste en la retransformant sous sa forme `[ ... ]`. Elle va également s'appliquer récursivement pour chaque variable contenue à l'intérieur de la structure, afin de produire un résultat correct pour l'affichage.

6.8 Synthèse

La phase 3 est maintenant terminée. Nous avons à présent un interpréteur fonctionnel, capable de résoudre des clauses complexes et de gérer des cas spéciaux de l'unification. Les objectifs initiaux du projet sont atteints. Les problèmes principaux restants sont les suivants :

— Pas de gestion des requêtes multiples dans l'interpréteur.
— Impossible d'obtenir plus que le premier résultat.

7 Phase bonus

Maintenant que la base du projet est là, on peut s'attaquer à des fonctionnalités supplémentaires. Les fonctionnalités retenues sont les suivantes :

— Résultats multiples : pouvoir obtenir plusieurs résultats dans l'interpréteur
— Prédicats intégrés : mettre en place l'architecture nécessaire à l'intégration de prédicats intégrés
— Questions multiples : possiblité de demander plusieurs buts séparés par des virgules dans l'interpréteur

Bien que non vitales, ces fonctionnalités semblent être nécessaires à tout interpréteur Prolog.

7.1 Résultats multiples

Actuellement, l'interpréteur permet uniquement de récupérer le premier succès. Il faut trouver un moyen de permettre d'en trouver plus.

7.1.1 Besoins

Voici ce qu'il faut pouvoir faire afin d'être capable de récupérer plusieurs résultats :

— Avoir une commande ou une touche permettant de demander le prochain succès. Par exemple, dans l'interpréteur *GNU Prolog*, on peut obtenir le prochain succès avec la touche `';'`.
— Être capable de recommencer la recherche depuis l'endroit où l'on s'est arrêté dans l'arbre de preuve.

7.1.2 Demande du prochain succès

Pour la demande du prochain succès, on peut imaginer demander à l'utilisateur d'appuyer sur une touche, ou alors d'entrer une commande spéciale. C'est cette deuxième option qui a été choisie, car elle présente les avantages suivants :

— On ne supporte pas actuellement le fait de pouvoir réagir à une touche particulière, vu qu'on lit l'entrée utilisateur ligne par ligne. C'est donc plus facile à implémenter.
— On ne casse pas la compatibilité avec les tests unitaires existants (voir section 8), ce qui permet de rajouter des tests pour valider le fonctionnement des résultats multiples sans devoir réécrire les tests précédents.

La commande choisie est la suivante : `redo`. Son nom est assez explicite sur sa fonctionnalité. Voici le comportement attendu :

1. L'utilisateur fait une requête dans l'interpréteur. Il obtient le premier résultat.
2. L'utilisateur entre la commande `redo`. Il obtient le prochain résultat.
3. L'utilisateur peut recommencer l'étape 2 jusqu'à ce qu'il obtienne un échec, ce qui signifie qu'il n'y a plus d'autres résultats.
4. Si l'utilisateur entre de nouveau la commande, alors il continue d'obtenir des échecs.

7.1.3 Reprise de la recherche

Pour la reprise de la recherche, une stratégie possible est de se rappeler du parcours effectué jusqu'à l'obtention du succès. Cela revient à se rappeler des branches choisies dans l'arbre de preuve. Pour simplifier et profiter du fait qu'on utilise la pile d'appels, on peut effectuer cette sauvegarde au moment où l'on obtient le succès, pendant que l'on remonte la pile d'appels.
Quand on souhaite obtenir le prochain résultat, on utilise ce parcours sauvegardé quand on descend dans l'arbre de preuve, pour sauter directement à l'embranchement qui nous intéresse. Il faut cepedant faire attention à ne pas oublier de quand même unifier le but courant avec la tête de l'embranchement choisi, pour que les variables se retrouvent dans le bon état quand on arrive en bas de l'arbre.

7.1.4 Implémentation : interpréteur

Dans l'interpréteur, il faut plusieurs choses : reconnaître la commande `redo`, garder en mémoire le parcours effectué, et garder en mémoire le contexte de la dernière requête, pour éviter de devoir parser à nouveau l'entrée à chaque fois. L'utilisation du parcours effectué est décrite dans l'implémentation des noeuds de l'arbre de preuve (7.1.6). Il s'agit d'un `Deque`, mais il sera utilisé comme un `Stack`, toujours pour les raisons citées à la section 6.4.3.

```
Deque<Integer> currentBranches = new ArrayDeque<>();
ProloGraalRuntime lastRuntime = null;
boolean skipParsing = false;
```

Quand l'utilisateur effectue la requête initiale, on parse normalement, et on sauvegarde le contexte. On doit également réinitialiser la sauvegarde du parcours, vu qu'il s'agit d'une nouvelle requête.

```
Source source = null;
if (!skipParsing) {
    // new goal, so clear the current branches
    currentBranches.clear();

    source = Source.newBuilder("pl", line, null).build();
}
ProloGraalRuntime runtime;
try {
    if (!skipParsing) {
        // parse the source to get a runtime
        runtime = new ProloGraalRuntime(
                ProloGraalParserImpl.parseProloGraal(source),
                language.getContextReference().get()
        );
```

```
            // save runtime
            lastRuntime = runtime;
        } else {
            // ... handle redo
        }
    } catch (ProloGraalParseError parseError) {
        // clear latest runtime on error so we cannot redo
        lastRuntime = null;
        writer.println(parseError.getMessage());
        continue;
    }
```

Sinon, si l'utilisateur entre la commande `redo`, il faut vérifier qu'il existe déjà un contexte, et indiquer que l'on est dans cette situation à l'aide d'un booléen.

```
if (line.equals("redo.")) {
    // triggers a redo instead of a normal parsing
    if (lastRuntime == null) {
        writer.println("no");
        continue;
    }
    skipParsing = true;
}
```

On se sert ensuite de ce booléen pour passer outre le parsing, en récupérant à la place simplement le précédent contexte.

```
ProloGraalRuntime runtime;
try {
    if (!skipParsing) {
        // ... handle first request (see above)
    } else {
        runtime = lastRuntime;
    }
} catch (ProloGraalParseError parseError) {
    // ... handle parse error (only possible when not redoing)
}
```

On va ensuite appeler le noeud de résolution normalement, mais en lui donnant la sauvegarde du parcours. On détecte également si ce parcours est vide après l'appel au noeud de résolution, car cela signifie qu'il n'y a plus de solutions et qu'on peut donc effacer le contexte courant.

```
skipParsing = false;
// get the result from the resolver node
callResult =
        (ProloGraalBoolean) Truffle.getRuntime()
        .createCallTarget(context.getResolverNode())
        .call(runtime, currentBranches);
if (currentBranches.isEmpty()) {
    // there are no more solutions
    lastRuntime = null;
}
```

7.1.5 Implémentation : noeud de résolution

Dans le noeud de résolution, rien de spécial à faire si ce n'est récupérer et transmettre le parcours au premier noeud de l'arbre de preuve.

```
Deque<Integer> branches = (Deque<Integer>) frame.getArguments()[1];
// ...
ProloGraalBoolean r = proofTreeNode.execute(branches);
```

7.1.6 Implémentation : noeuds de l'arbre de preuve

Dans les noeuds de l'arbre de preuve, on reçoit le parcours en paramètre. Il y a deux possibilités : si le parcours est vide, alors on est en train de descendre pour la première fois dans l'arbre. Sinon, on est dans un `redo`.
Dans le cas d'un `redo`, il faut passer directement à la bonne branche, mais en unifiant quand même le but courant avec la tête de la clause. Plusieurs changements sont nécessaires. On commence par changer un peu la logique générale : avant, on essayait toutes les clauses dont la tête était la même que celle du but courant. On va maintenant filtrer directement celles qui ont la même tête mais aussi qui sont unifiables avec le but courant.

```
// filter clauses that are unifiable with the current goal
// creating copies and saving variables state as needed
List<ProloGraalClause> unifiableClauses =
        IntStream.range(0, possibleClauses.size())
            .filter(x -> { // filter clauses that are unifiable
                ProloGraalClause clause = possibleClauses.get(x).copy();
                currentGoal.save();
                boolean r = clause.getHead().unify(currentGoal);
                currentGoal.undo();
                return r;
            })
            .mapToObj(x -> possibleClauses.get(x).copy()) // copy clauses
            .collect(Collectors.toList());
```

Ceci permet, quand on obtient un succès, de savoir directement s'il reste ou non des branches possibles, et donc de faire une optimisation sur la sauvegarde de parcours nous envoyant directement sur le prochain succès possible lors du `redo`.
Pour la sauvegarde du parcours, l'algorithme est donc le suivant : au moment où on atteint un succès, dans la feuille de l'arbre de preuve, le parcours est forcément vide. Par contre, les noeuds précédents ne savent pas si le succès a été trouvé juste en dessous d'eux par la feuille, ou bien plus bas dans la pile d'appels. Ce qui signifie que plus haut dans l'arbre, le parcours peut déjà être en partie rempli. On ajoute donc la trace du parcours si une ou plusieurs des conditions suivantes est vraie :

- il reste une clause possible après celle ayant donné le succès dans le noeud actuel. Il y a donc un potentiel succès supplémentaire.
- le parcours n'est pas vide. Cela signifie que la première condition s'est déjà activée plus bas dans l'arbre, et il faut donc retenir le parcours exact qui mène à cet embranchement, même si le noeud actuel n'a lui même plus d'autres succès possibles.

Ces deux conditions permettent de remonter l'arbre sans sauvegarder le parcours jusqu'à ce qu'on arrive à un noeud qui a encore des possiblités. On pourra donc continuer la recherche

directement depuis le prochain succès possible, sans devoir redescendre au précédent succès d'abord.

```
if (!branches.isEmpty() || i + 1 < unifiableClauses.size()) {
    if (branches.isEmpty()) {
        i = i + 1; // if we're at the bottom go directly to next node
    }
    branches.push(i); // add the path that gave the success
}
```

Pour la descente et l'utilisation du parcours, on s'appuie sur le fait que l'on utilise une boucle sur les différentes clauses possibles afin de passer directement à celle qui nous intéresse.

```
int start = 0;
if (!branches.isEmpty()) {
    // if we're redoing we need to skip to the right branch directly
    start = branches.pop();
}

for (int i = start; i < unifiableClauses.size(); i++) {
    // ...
```

Et ceci conclut l'implémentation de notre `redo`.

7.2 Prédicats intégrés

Le langage Prolog contient de nombreux prédicats intégrés, facilitant certaines opérations et permettant de réaliser des opérations qui ne sont pas directement issues de la logique, comme manipuler des fichiers par exemple. Le but ici n'est évidemment pas d'ajouter tous les prédicats de la librairie de Prolog, mais d'en intégrer un petit nombre en guise de preuve de fonctionnement. Les prédicats choisis sont les suivants :

- var(X) : permet de déterminer si X est une variable (encore inconnue). Ce prédicat a été choisi car il a un effet au moment de l'unification : var(X) doit retourner faux, alors que var(a) doit retourner vrai.
- write(X) : écrit dans la sortie standard la représentation textuelle de X. Ce prédicat a été choisi car il a un effet de bord. Il a donc un comportement non trivial lors de l'exécution et surtout quand on `redo`.

7.2.1 Implémentation : partie commune

Pour la partie commune à tous les prédicats, une sous-classe de `ProloGraalClause` a été définie : `ProloGraalBuiltinClause`. Cette classe introduit deux méthodes. La première, `execute`, permet aux prédicats intégrés d'effectuer des actions ayant un potentiel effet de bord. La deuxième est une surcharge de la méthode de copie des clauses, et ne modélise pas directement un comportement de Prolog ; elle permet simplement d'éviter que les clauses intégrées perdent leur statuts quand on les copie au moment de la résolution.

```
public void execute() {
    // default behaviour is to do nothing...
}
```

```
// this method is abstract to force subclasses to implement it
@Override
public abstract ProloGraalClause copy();
```

7.2.2 Prédicat var

Le prédicat `var` a un effet au moment de l'unification. Il faut donc pouvoir intervenir au moment de l'unification de la tête de la clause. Pour cela, on va remplacer la tête par une extension de la classe `ProloGraalStructure`, afin de pouvoir réimplémenter la méthode `unify`.

```
public final class ProloGraalVarBuiltin extends ProloGraalBuiltinClause {
    private static class VarPredicateHead extends ProloGraalStructure {
        public VarPredicateHead(Map<ProloGraalVariable, ProloGraalVariable>
        ↪  variables) {
            super(variables);
            // add the correct functor for this predicate, and an anonymous
            ↪  variable since we do not need it
            setFunctor(new ProloGraalAtom(variables, "var"));
            addSubterm(new ProloGraalVariable(variables, "_"));
        }
        // ...
```

On réimplémente donc la méthode `unify` en lui donnant le comportement que l'on désire, c'est-à-dire vérifier si l'argument est une variable ou non.

```
@Override
public boolean unify(ProloGraalTerm<?> other) {
    if (other instanceof ProloGraalStructure) {
        ProloGraalStructure struct = (ProloGraalStructure) other;
        if (struct.getFunctor().equals(getFunctor()) && struct.getArity() ==
        ↪  1) {
            // checks if the root value of the first argument resolves to a
            ↪  variable
            return struct.getArguments().get(0).getRootValue() instanceof
            ↪  ProloGraalVariable;
        }
    }
    return false;
}
```

Il ne faut pas oublier d'également implémenter la méthode de copie pour éviter de perdre le comportement par défaut, comme expliqué précédemment :

```
// override the default copy so we do not lose the custom unification
↪  behaviour
@Override
public ProloGraalStructure copy(Map<ProloGraalVariable, ProloGraalVariable>
↪  variables) {
    return new VarPredicateHead(variables);
}
```

Il faut ensuite remplacer la tête de notre prédicat intégré par notre implémentation :

```
public ProloGraalVarBuiltin() {
    super();
    // creates our custom head and set it
    VarPredicateHead head = new VarPredicateHead(getVariables());
    setHead(head);
}
```

Et voilà, l'implémentation du prédicat intégré est terminée.

7.2.3 Prédicat `write`

Le prédicat `write` n'a pas d'effets sur le mécanisme d'unification, par contre, il un effet de bord. Il a également besoin du contexte pour avoir accès à la sortie standard.
Il faut donc récupérer la sortie standard, ainsi que préparer le préparer le prédicat en choissant sa tête. Contrairement à `var`, une structure classique avec le foncteur `write` et une variable en argument suffit. On garde juste une référence vers la variable pour pouvoir récupérer son contenu au moment de l'affichage.

```
private final PrintWriter writer; // used for outputting
private final ProloGraalContext context; // we keep the context for the copy method
private ProloGraalVariable arg; // the variable X in write(X).

public ProloGraalWriteBuiltin(ProloGraalContext context) {
    super();
    // get printer from context
    this.writer = new PrintWriter(context.getOutput(), true);
    this.context = context;

    // create the head of this clause
    // since we do not need custom unification, a simple structure is enough
    ProloGraalStructure head = new ProloGraalStructure(getVariables());
    head.setFunctor(new ProloGraalAtom(getVariables(), "write"));
    // we create and store the variable to access it more easily later in the execute
    arg = new ProloGraalVariable(getVariables(), "_");
    head.addSubterm(arg);
    setHead(head);
}
```

Il ne reste plus qu'à implémenter l'affichage. Pour cela, on surcharge la méthode `execute` définie dans `ProloGraalBuiltinClause`. Il suffit d'afficher la représentation en chaîne de caractères de la valeur racine de l'argument, en enlevant éventuellement les apostrophes des atomes (car c'est ce que fait le moteur *GNU Prolog*).

```
@Override
public void execute() {
    String str = arg.getRootValue().toString();
    if(str.startsWith("'") && str.endsWith("'")) {
        // strip single quotes
        str = str.substring(1, str.length()-1);
```

```
    }
    writer.print(str);
    writer.flush();
}
```

Ne reste que la méthode de copie qui est triviale, et le prédicat est implémenté.

7.2.4 Les raisons de la méthode `execute`

On peut se demander l'utilité de la méthode `execute`. On pourrait en effet effectuer les opérations au moment de l'unification. Mais l'implémentation actuelle de la résolution se base sur le fait que l'on peut réaliser l'unification sans provoquer d'effets de bord autres que l'unification des variables. Il est donc nécessaire de séparer les comportements ayant de tels effets, pour éviter des problèmes lors de la descente de l'arbre de preuve, spécialement quand on fait un `redo`. L'exécution de la méthode `execute` se fait donc uniquement lors de la première traversée d'un certain noeud de l'arbre de preuve, après l'unification.

```
// only execute built-in the first time we traverse their nodes
if (branches.isEmpty()) {
    if (unifiableClause instanceof ProloGraalBuiltinClause) {
        // if the clause is a built-in, execute its internal behaviour
        ((ProloGraalBuiltinClause) unifiableClause).execute();
    }
}
```

7.2.5 Intégration des prédicats

Les prédicats sont implémentés, mais ils ne sont pas encore utilisables. Il faut encore les greffer à la liste des clauses du programme. Cette opération se fait au moment où on construit le `ProloGraalRuntime`, juste après le parsing. Une méthode `installBuiltins` est appelée à la fin de la construction. Elle se charge d'instancier les différents prédicats intégrés et de les ajouter à la liste des clauses :

```
private void installBuiltins() {
    ProloGraalClause varBuiltin = new ProloGraalVarBuiltin();
    clauses.put(varBuiltin.getHead(), Collections.singletonList(varBuiltin));

    ProloGraalClause writeBuiltin = new ProloGraalWriteBuiltin(context);
    clauses.put(writeBuiltin.getHead(), Collections.singletonList(writeBuiltin));
}
```

7.3 Questions multiples

Dans la plupart des moteurs Prolog, on peut demander plusieurs buts directement dans l'interpréteur, en les séparant avec des virgules. Un problème qui peut apparaître est le fait que l'on réutilise le parser pour traiter la requête utilisateur. En effet, si `test :- test(A), write(A).` est une ligne tout à fait valide, juste `test(A), write(A).` ne l'est pas. Heureusement, on peut résoudre ce problème avec une astuce très simple : avec de parser la ligne, on rajoute une tête valide.

```
line = "goals :- " + line;
```

Il suffit ensuite dans le noeud de résolution de donner au premier noeud de l'arbre de preuve non plus la tête de la clause qu'on reçoit depuis le noeud d'interprétation, mais son corps :

```
Deque<ProloGraalTerm<?>> goals = new ArrayDeque<>(clause.getGoals());
// create the root node of the proof tree
ProloGraalProofTreeNode proofTreeNode = new ProloGraalProofTreeNode(clauses, goals);
```

Et nous pouvons maintenant effectuer plusieurs requêtes dans l'interpréteur.

7.4 Synthèse

Cette phase bonus touche à sa fin. Elle corrige les derniers problèmes de la phase 3, et prouve qu'il est possible d'ajouter des prédicats intégrés à notre moteur.

8 Tests unitaires

Afin de contrôler le fonctionnement et d'éviter les problèmes de régression lors de l'implémentation, des tests unitaires ont été réalisés. Ces tests se basent sur le modèle de ceux implémentés dans le projet d'exemple *SimpleLanguage*.

8.1 Fonctionnement

Ces tests fonctionnent de la manière suivante : dans le dossier tests, on trouve des ensembles de 3 fichiers, avec le même nom mais des extensions différentes :

- Le fichier source, avec son extension normale (.sl pour *SimpleLanguage*, .pl pour Prolog)
- Un fichier .input, qui contient l'entrée clavier qui sera fournie au programme lors de son exécution.
- Un fichier .output, qui contient la sortie attendue après l'exécution du programme.

Ce système est assez bien fait, car il n'y a donc pas besoin d'écrire de code Java pour rajouter des tests supplémentaires. Cependant, l'implémentation de démonstration est assez basique, et ne donne pas beaucoup d'informations sur l'endroit du test qui a échoué. C'est compréhensible pour *SimpleLanguage*, vu que chaque test contient uniquement un programme. Mais pour Prolog, vu qu'il faut tester différentes entrées par prédicat, ce n'est pas suffisant.

8.2 Améliorations

Des améliorations ont donc été apportées au système de test, pour permettre de comparer la sortie après chaque entrée, et ainsi trouver exactement et directement la requête qui n'a pas obtenu le résultat attendu. Ces améliorations ajoutent également le support de commentaires aux fichiers .input et .output.

8.3 Tests

Les tests ont été découpés par fonctionnalités qu'ils testent. Les tests implémentés sont les suivants :

- 01_facts : teste les faits simples, sans corps. Teste également le parser avec des structures complexes et sur plusieurs lignes.
- 02_variables : teste les variables, et plus généralement l'unification avec des faits simples. Teste également le test d'occurence.
- 03_lists : teste les listes, et l'unification entre celles-ci.
- 04_clauses : teste différentes clauses connues, comme la concaténation de liste, de D-listes, le prédicat owns, etc... Teste des prédicats récursifs et d'autres non récursifs.
- 05_builtins : teste les prédicats intégrés `write` et `var`.
- 06_redo : teste la fonctionnalité permettant d'obtenir plus de résultats, en prenant soit de vérifier le fonctionnement avec le prédicat avec effet de bords `write`.

9 Problèmes rencontrés

Aucun gros problème n'a été rencontré durant la réalisation de ce projet. Cependant, de nombreux bugs ont été découverts au fur et à mesure. L'implémentation des tests unitaires a grandement aidé à la détection et la résolution de ces bugs, et ceux-ci auraient dû être réalisés plus tôt dans le projet.
Voici quand même un problème ayant coûté un certain temps et réduit la qualité du projet.

9.1 JLine3

JLine est une librairie open source permettant de faciliter et d'améliorer grandement la gestion de l'entrée console, en proposant par exemple des fonctionnalités comme un historique similaire à celui de tous les terminaux. La librairie fonctionnait parfaitement, jusqu'à l'arrivée des tests unitaires... En effet, comme tous les terminaux, elle utilise des caractères de contrôle invisibles pour contrôler l'affichage. Ces caractères faisaient planter les tests, et j'ai perdu un temps fou à essayer de les enlever, à l'aide d'expressions régulières, de remplacement de chaînes, et même de simple `substring`, pour obtenir uniquement la sortie texte, sans succès. J'ai donc décidé à mon grand regret d'abandonner la librairie et de revenir à une simple ligne de commande sans historique ou autres fonctionnalités, mais également sans problèmes avec les tests.

10 Améliorations possibles

Cette section donne quelques améliorations possibles du projet, en discutant de la faisabilité technique et de l'implémentation de chacune.

10.1 Optimisations et intégration plus forte à Truffle

Le moteur est actuellement très peu optimisé (pour ne pas dire pas optimisé du tout...). Il y a beaucoup d'améliorations possibles à beaucoup d'endroits du projet, mais le principal serait d'améliorer le processus de résolution.

10.1.1 Améliorations du noeud de résolution

Dans sa version actuelle, chaque noeud de l'arbre de preuve crée deux copies de chaque clause potentiellement unifiable avec le but courant. En parlant de celui-ci, son état est également sauvegardé (ce qui produit des ajouts de l'état des variables à des piles) et restoré une fois pour chaque clause potentielle, puis une fois pour chaque clause unifiable... Les clauses sont également unifiées au but courant deux fois, une fois pour les filtrer, puis une fois pour mettre l'état correct des variables.
Il est évidemment possible de faire mieux. Une piste serait par exemple de garder l'état du but courant en mémoire pour chaque clause, plutôt que d'oublier son état et de le restaurer à l'aide d'une nouvelle unification plus tard.

10.1.2 Amélioration du processus de résolution

De manière plus générale, le processus de résolution s'appuie actuellement sur la pile d'appels, ce qui limite les possibilités de récursion à la taille de ladite pile. On pourrait simuler le comportement à l'aide d'une boucle et d'une pile externe pour enlever cette limite.
Mais il faudrait d'abord améliorer l'intégration avec Truffle, et vérifier les performances, car Truffle est capable de faire de la *Tail Call Optimization*[8] et est donc capable de potentiellement

8. https://cesquivias.github.io/blog/2015/01/15/writing-a-language-in-truffle-part-4-adding-features-the-truffle-way/#tail-call-optimization-in-truffle

enlever la limitation de la pile d'appels sans que l'on doive nous même faire le changement.

10.1.3 Intégration plus forte avec Truffle

Le projet s'appuie finalement très peu sur les fonctionnalités de Truffle. Très peu de changements seraient requis pour s'en passer complètement. Cela veut également dire que l'on ne profite pas du tout des apports en vitesse promis par Truffle.
Une piste d'utilisation de Truffle est dans le mécanisme de spécialisations : on fournit plusieurs versions d'une même fonction avec des arguments différents, et Truffle se charge d'appeler la bonne (un peu comme la surcharge de méthodes, mais avec moins de coût à l'exécution grâce à la compilation partielle). Cette approche est fondamentalement différente de celle actuellement utilisée. L'implémentation actuelle est sous cette forme : une classe représentant un certain composant de Prolog contient les différentes actions possibles pour ce composant, par exemple l'unification. L'approche Truffle serait la suivante : une classe représentant une action, par exemple l'unification, et plusieurs méthodes représentant la réalisation de cette opération pour les différents types.

10.2 Support de l'interopérabilité avec Truffle

Truffle propose une API pour faciliter l'interopérabilité entre différents langages de programmation. Cette API se base sur le principe de messages, indiquant les fonctionnalités d'un certain objet. Par exemple, un objet peut avoir le message `executable` pour indiquer qu'il peut être exécuté (comme une méthode, ou un prédicat par exemple dans notre cas).
La première étape pour rendre notre langage interopérable serait d'étudier la partie *component* du projet de démonstration *SimpleLanguage* (cf. section 4.4). Cette partie avait été volontairement mise de côté, mais elle sert à rendre notre langage ajoutable en tant que plugin à GraalVM, afin de pouvoir l'utiliser dans d'autres langages et pas uniquement en version *standalone*.
On pourrait ensuite hypothétiquement utiliser le terminal multi-langages de GraalVM pour définir des prédicats en Prolog et les exécuter dans un environnement Java. Mais cela nécessite également que notre implémentation utilise correctement les fonctionnalités de Truffle, ce qui n'est pas le cas actuellement. Il faudrait donc d'abord améliorer l'intégration avec Truffle pour ensuite pouvoir proposer l'interopérabilité.

10.3 Support du prédicat cut

Le support du parsing du prédicat peut s'ajouter très facilement. Son comportement, un peu moins... Pour rappel, le cut empêche la remise en question de ce qui le précède, y compris le prédicat qui l'a introduit. Pour l'implémenter, il faudrait donc regarder au moment où on remplace le but courant par le corps de la clause, si elle contient un cut. Ensuite, il faudrait se rappeler de si ce cut a été exécuté ou non (il peut y avoir un échec avant qu'on y arrive). Si ce cut a été exécuté, et qu'il y a un échec après, alors on doit retourner jusqu'au prédicat l'ayant introduit en retournant des échecs même s'il reste d'autres branches possibles. Le prédicat l'ayant introduit doit également retourner un échec. La principale difficulté serait donc de gérer correctement la sauvegarde de qui a introduit le cut, et d'être capable de remonter l'information dans la pile d'appels qu'il faut retourner jusqu'à ce point.

10.4 Support d'autres prédicats/de la librairie standard

Avec l'infrastructure mise en place, l'ajout de la plupart des prédicats simples comme par exemple `select` est facile. Certains prédicats pourraient cependant poser problème, comme par exemple ceux permettant d'ajouter/enlever dynamiquement des clauses, `assert` et `retract`. L'ajout ou la suppression en eux-mêmes ne poseraient pas trop de problèmes, vu qu'il est

trivial d'ajouter des éléments à la liste des clauses du programme si l'on possède une référence vers le contexte. Par contre, cela pourrait poser des problèmes avec la fonctionnalité `redo`, vu que l'arbre de preuve pourrait contenir de nouvelles branches qui n'étaient pas là lors de la précédente descente. Il faudrait donc porter une attention particulière à la correction de la sauvegarde de la descente, ou alors interdire la possibilité de modifier un prédicat en cours d'utilisation...

11 Conclusion

Ce rapport touche à sa fin. Il est temps de faire le point sur le travail réalisé, au travers d'une comparaison avec les objectifs du projet, et de donner un ressenti personnel sur le projet et son déroulement.

11.1 Atteinte des objectifs

Les fonctionnalités retenues à la section 1.2 ont toutes été implémentées avec succès. Les fondations pour certains objectifs optionnels également, aux travers des prédicats intégrés `var` et `write`, et une gestion basique des exceptions avec une erreur en cas de prédicat inconnu. Les objectifs ayant été fixés au début du projet sont donc atteints.
Il reste un point noir : le projet s'appelle "Prolog avec Truffle et GraalVM", mais au final, c'est plutôt "Prolog en Java"... Ce n'est cependant pas si grave, vu que le projet laisse de bonnes fondations pour un éventuel approfondissement, et que cette étape était nécessaire pour ensuite intégrer plus profondément Truffle. De plus, l'objectif principal du projet était bien de voir s'il est possible d'implémenter un langage de programmation logique avec Truffle, et nous pouvons maintenant répondre de manière affirmative à cette question.

11.2 Conclusion personnelle

J'ai eu du plaisir à réaliser ce projet. J'ai appris beaucoup de choses sur le fonctionnement interne de Prolog et de manière plus générale sur le fonctionnement d'un interpréteur de langage. Le projet s'est bien déroulé, étant resté dans les temps prévus initialement. Je regrette juste un peu de ne pas avoir eu plus de temps pour creuser l'intégration avec Truffle au cours du projet, car il aurait été très intéressant d'essayer d'optimiser l'interpréteur après l'avoir écrit.
Au niveau de la gestion de l'implémentation, il aurait peut-être fallu planifier un peu mieux les différentes phases en pensant un peu plus au futur de chaque fonctionnalité dans le projet final, car je me retrouvais souvant à devoir presque totalement défaire ce qui avait été fait précédemment pour intégrer les nouvelles fonctionnalités (comme on peut le constater particulièrement entre la phase 2 et 3).
Mais de manière globale, je suis content d'avoir réalisé ce projet et satisfait de son résultat. Je remercie mon superviseur monsieur Frédéric Bapst pour son accompagnement le long de ce projet, ses conseils avisés et son oeil d'aigle pour les fautes de programmation(et d'orthographe !).

Déclaration d'honneur

Je, soussigné, Martin Spoto, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toute autre forme de fraude. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Références

[Aït91] Hassan Aït-Kaci, *Warren's Abstract Machine : A Tutorial Reconstruction*, MIT Press, 1991, ISBN : 0-262-51058-8.

[Esq15] Cristian Esquivias, *Writing a Language in Truffle*, 2015, URL : `https://cesquivias.github.io/tags/truffle.html`.

[Gra15] GraalVM, *SimpleLanguage : A simple example language built using the Truffle API*, 2015, URL : `https://github.com/graalvm/simplelanguage`.

[ISO95] ISO, *Information technology – Programming languages – Prolog*, Standard, Geneva, CH : International Organization for Standardization, juin 1995.

[JIP98] JIProlog, *JIProlog*, 1998, URL : `https://github.com/jiprolog/jiprolog/`.

[Tom17] Federico Tomassetti, *The ANTLR Mega Tutorial*, 2017, URL : `https://tomassetti.me/antlr-mega-tutorial/`.

[Wim16] Christian Wimmer, « One VM to Rule Them All », Oracle Labs, Santa Barbara CA : PLDI 2016, 2016, URL : `https://lafo.ssw.uni-linz.ac.at/pub/papers/2016_PLDI_Truffle.pdf`.

12 Glossaire

ANTLR

Ensemble de librairies utilisées pour la génération du parser et du lexer.

Deque

Structure de données supportant l'insertion et la suppression efficace au début et à la fin.

GraalVM

Plateforme supportant Truffle. Contient une JVM.

Java

Langage de programmation utilisé pour le projet et ses librairies.

Java Virtual Machine (JVM)

Machine virtuelle permettant l'exécution de programmes Java.

Maven

Gestionnaire de dépendances et de cycle de développement.

SimpleLanguage

Projet de démonstration utilisant Truffle et GraalVM.

Stack (pile)

Structure de données représentant une pile : le dernier élément inséré est le premier élément qui ressort.

Truffle

Ensemble de librairies utilisant GraalVM pour fournir des fonctionnalités d'implémentation de langages.

arbre de preuve

Arbre créé en suivant les règles logiques d'un programme.

arité

Nombre d'arguments d'une structure Prolog.

backtracking

Technique de programmation basée sur l'essai systématique de toutes les possiblités.

clauses

En Prolog, décrit une relation logique. Il existe deux types de clauses : les clauses simples, appelées faits, et les clauses composées.

- Un fait est de la forme : `fact.`
- Une clause composée est de la forme : `head :- rule1, rule2.`

compilateur à la volée

Technique permettant de recompiler des bouts de code de manière dynamique lors de l'exécution du programme, pour les rendre plus performants.

contexte

Contexte de l'application, contenant des informations sur les clauses et sur l'environnement du programme. Dans le projet, le contexte peut désigner deux classes :

- La classe `ProloGraalRuntime`, qui représente le contexte d'exécution et contient les clauses du programme
- La classe `ProloGraalContext`, qui contient le contexte globale avec des références vers la sortie et l'entrée standards du programme

faits

Clause Prolog de la forme : `fact.`.

fichier de grammaire

Fichier contenant du code compréhensible par ANTLR.

foncteur

Désignateur d'une structure Prolog. Dans la structure `a(b, c)`, `'a'` est le foncteur.

framework

Ensemble de librairies et autres composants.

interpréteur

Peut désigner deux choses :

- Un interpréteur de langage, qui lit un code source et se charge de l'exécuter (sans le compiler).
- L'interpréteur Prolog, qui est une interface en ligne de commande permettant d'effectuer des requêtes sur un programme Prolog.

lexer

Composant chargé de lire un fichier source, et de créer à partir des caractères des unités compréhensibles pour le parser.

librairie

Ensemble de classes externes réalisant une ou plusieurs fonctionnalités spécifiques.

liste

En Prolog, désigne une structure de la forme `[a,b,c | d]`. Une liste possède possiblement une queue, ou *tail* en anglais, qui désigne l'élément à la fin de celle-ci.

machine virtuelle

Couche logicielle permettant l'interprétation d'un langage de plus haut niveau.

main

Fonction principale d'un programme.

open source

Projet dont le code source est disponible publiquement.

parser

Composant chargé de donner un sens aux jetons créés par le lexer, afin de créer des structures de plus haut niveau.

pattern de développement

Patron donnant une réalisation standard d'un composant.

requête

Question posée à l'interpréteur Prolog.

structure

En Prolog, structure de forme `a(b, c)`.

termes

En Prolog, mot générique désignant les autres unités de données (structures, variables, etc...).

unification

Mécanisme permettant de rendre deux termes logiques identiques.

variables

En Prolog, représente une inconnue logique, capable de prendre une valeur lors de l'unification.

13 Annexes

13.1 Lien du projet

Le projet est disponible sur le GitLab de l'école à l'adresse suivante :
https://gitlab.forge.hefr.ch/frederic.bapst/prolog-truffle
(une demande d'accès à monsieur Bapst est nécessaire)

13.2 Librairies utilisées

Cette section liste les différentes librairies nécessaires au fonctionnement du projet.

JLine3

JLine est une librairie open source permettant de faciliter la gestion du terminal et de l'entrée utilisateur. Elle est utilisée jusqu'à la phase 2.
Lien : https://github.com/jline/jline3

ANTLR

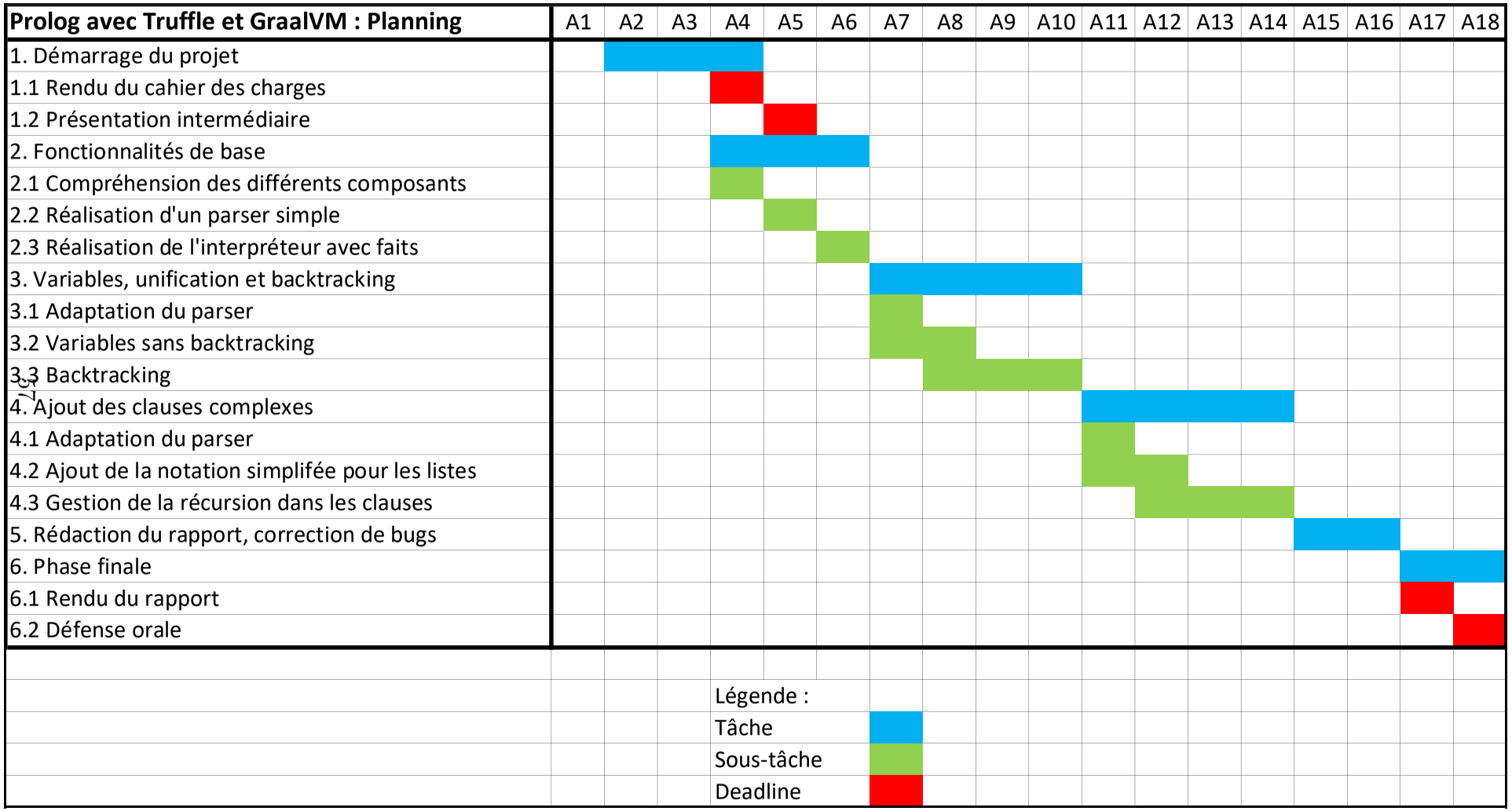
ANTLR est la librairie open source utilisée pour la réalisation du parser et du lexer. Elle a été utilisée durant tout le projet.
Lien : https://github.com/antlr/antlr4

JUnit

JUnit est la librairie open source utilisée pour la réalisation des tests unitaires.
Lien : https://github.com/junit-team/junit5/

Suite de librairies GraalVM et Truffle

Truffle et les autres librairies de GraalVM sont utilisées pour l'implémentation du langage. Elles sont open source.
Lien : https://github.com/oracle/graal


Prolog avec Truffle et GraalVM : Planning
A1
A2
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15
A16
A17
A18
1. Démarrage du projet
1.1 Rendu du cahier des charges
1.2 Présentation intermédiaire
2. Fonctionnalités de base
2.1 Compréhension des différents composants
2.2 Réalisation d'un parser simple
2.3 Réalisation de l'interpréteur avec faits
3. Variables, unification et backtracking
3.1 Adaptation du parser
3.2 Variables sans backtracking
3.3 Backtracking
4. Ajout des clauses complexes
4.1 Adaptation du parser
4.2 Ajout de la notation simplifiée pour les listes
4.3 Gestion de la récursion dans les clauses
5. Rédaction du rapport, correction de bugs
6. Phase finale
6.1 Rendu du rapport
6.2 Défense orale
Légende :
Tâche
Sous-tâche
Deadline

B Manuel d'installation et d'utilisation

Table des matières

1 Introduction **3**

2 Installation des dépendances **3**
2.1 GraalVM 3
2.2 Maven 3

3 Préparation de l'environnement **3**

4 Compilation du projet **3**

5 Lancement du projet **3**

6 Notes pour Windows **4**

1 Introduction

Ce document décrit l'installation et l'utilisation du projet "Prolog avec Truffle et GraalVM", pour l'environnement Linux. Le projet fonctionne également sous Windows, mais les étapes d'installation ne sont pas détaillées.
Ce manuel assume que vous disposez localement d'une copie du projet, et que vous vous trouvez dans le répertoire "code" de celui-ci.

2 Installation des dépendances

Cette section décrit l'installation des différents pré-requis au fonctionnement du projet.

2.1 GraalVM

Le projet a été développé avec GraalVM Community Edition 19.2.0.1.
On peut télécharger cette version ici : https://github.com/oracle/graal/releases/tag/vm-19.2.0.1. Prenez la version qui correspond à votre système d'exploitation, et décompressez l'archive, par exemple dans `/usr/lib/graalvm`. Prenez note du chemin d'accès de l'archive décompressée.

2.2 Maven

Maven est utilisé comme gestionnaire de version. La version 3.6.3 a été utilisée durant le projet, mais les versions proches et futures devraient également fonctionner.
On peut télécharger Maven ici : https://maven.apache.org/download.cgi. Comme pour GraalVM, décompressez l'archive, par exemple dans `/usr/lib/maven`.

3 Préparation de l'environnement

Pour préparer les variable d'environnement nécessaires, un script `envsetup` est fourni. Ce script assume que les chemins de GraalVM et de Maven sont ceux proposés plus haut. Si ce n'est pas le cas, éditez le script avec vos chemins d'installation avant le lancement.
Ce script définit GraalVM comme environnement Java en réglant la variable JAVA_HOME. Il ajoute également les répertoires bin de GraalVM et de Maven au PATH. Les changements de ce script sont temporaires. En tapant `exit`, on peut à tout moment sortir de l'environnement (pratique si vous avez plusieurs installations de Java par exemple).
Le script définit également quelques alias qui sont surtout utiles au développement. Ces alias sont consultables dans le fichier `envaliases`. Toujours pour le développement, le script récupère une copie de la librairie ANTLR utilisée pour générer le parser/lexer (vous aurez besoin pour cela de la commande `wget`). (D'ailleurs, si un jour vous souhaitez faire exactement ça, un script `generate_parser` est également fourni.)
Si tout s'est bien passé, la commande `mvn --version` devrait s'effectuer avec succès, et la version Java devrait être celle de GraalVM.

4 Compilation du projet

Si l'environnement est correctement configuré, alors la compilation s'effectue simplement avec la commande `mvn package`. Cette opération produit un fichier JAR dans launcher/target et dans language/target.

5 Lancement du projet

Si la compilation du projet s'est passée sans encombres, alors on peut lancer le projet à l'aide du script fourni `prolograal`, en lui donnant comme argument un fichier Prolog, par exemple : `./prolograal language/tests/02_variables.pl`, et voilà ! Le projet tourne.

6 Notes pour Windows

Pour Windows, à part que les scripts ne sont pas utilisables, l'installation et l'utilisation se déroulent exactement de la même manière. Il faut donc simplement configurer la variable JAVA_HOME et le PATH correctement.