



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

Plugin IntelliJ pour GNU Prolog

PROJET DE SEMESTRE 5

2022 - 2023

Rapport

STURZENEGGER ERWAN

ISC-IL-3

Mandant :
Bapst Frédéric

Superviseur :
Bapst Frédéric

2 février 2023

Table des matières

1	Introduction	1
1.1	Contexte	1
1.2	Motivations	1
1.3	Objectifs	1
1.3.1	Mise à jour du plugin	1
1.3.2	Ajout de fonctionnalités	2
1.4	Structure du rapport	2
2	Mise à jour du Plugin	3
2.1	Correction des bugs	3
2.1.1	Bug 1 : Run dans la console d'IntelliJ Idea sous Windows	3
2.1.2	Bug 2 : Opérateur non reconnu lors de la PLC	4
2.2	Mise à jour du SDK	5
2.2.1	Passage du projet en Gradle	5
2.2.2	Le problème du GrammarKit	6
2.3	Mise en place de la CI	7
3	Tests unitaires	9
3.1	Mise en place des tests unitaires	9
3.2	Exécution des tests unitaires	9
4	Ajout des fonctionnalités	11
4.1	Navigation	11
4.1.1	Goto Declaration	11
4.1.2	Find usage	13
4.2	Refactoring	15
4.2.1	Renommer un prédicat	15
4.2.2	Renommer une variable	16
4.2.3	Fonctionnement du renommage	16
4.3	Affichage des erreurs et warnings	17
4.3.1	L'idée	17
4.3.2	La mise en pratique	17
4.4	Auto-complétion	18
4.4.1	Fonctionnement de l'auto-complétion	19
4.4.2	Gestion des erreurs en temps réel	19
4.4.3	Exemple d'utilisation	20
5	Déploiement	21
5.1	Choix de la licence	21
5.2	Déploiement sur le marketplace	21
5.3	Déploiement sur Github	22

6 Conclusion	23
Glossaire	24
Table des illustrations	25
Liste des codes	26
Bibliographie	27

1 | Introduction

Ce document est le rapport du projet de semestre 5 : *Plugin IntelliJ pour GNU-Prolog*. Il a pour but de permettre la compréhension des différentes étapes de la réalisation du projet et du fonctionnement du plugin.

1.1 Contexte

Dans le cadre d'un projet de semestre (printemps 2018), un plugin IntelliJ-IDEA a été réalisé, permettant ainsi d'aider les étudiants lors du développement de programmes Gnu-Prolog. Actuellement, le plugin est employé en 3ème année pour le cours de programmation logique. Bien que fonctionnel, il n'est pas parfait et n'offre pas certaines fonctions qui seraient très utiles. En outre, un bug empêche les utilisateurs du plugin travaillant sur Windows de lancer l'exécution du programme directement dans le terminal de l'IDE.

1.2 Motivations

Mes motivations à prendre ce projet sont multiples :

- **Apprentissage** : J'ai très envie de découvrir le fonctionnement d'un plugin IntelliJ, et de voir comment il est possible de créer un plugin pour un IDE.
- **Utilité** : Le plugin sera utilisé par les étudiants de 3ème année de la filière ISC, spécialisés dans l'informatique logicielle.
- **Projet** : Le projet étant déjà existant, je devrai m'adapter à son fonctionnement et le faire évoluer.

1.3 Objectifs

1.3.1 Mise à jour du plugin

Le premier objectif est de mettre à jour le plugin afin qu'il soit compatible avec les dernières versions d'IntelliJ. En effet, le plugin ayant été développé en 2018, il est nécessaire de mettre à jour le plugin afin d'éviter une perte de compatibilité avec les futures versions d'IntelliJ.

Ceci comprendra :

- **Passage du projet sur Gradle** : Le plugin a été développé sans aucune gestion de dépendances, et utilise un système de librairies externes. Il est donc nécessaire de passer le projet sur Gradle afin de gérer les dépendances.
- **Mise à jour des dépendances** : Changement de la version du SDK d'IntelliJ, mise à jour des dépendances du plugin.
- **Mise à jour du code** : Mise à jour des classes et méthodes dépréciées.
- **Corrections des bugs** : Correction des bugs qui empêchent le plugin de fonctionner correctement.
- **Implémentation des tests unitaires** : Implémentation des tests unitaires pour le plugin.
- **Mise en place d'une pipeline** : Mise en place d'une pipeline pour effectuer les tests unitaires et la compilation du plugin.

1.3.2 Ajout de fonctionnalités

Le second objectif est d'ajouter des fonctionnalités au plugin. En effet, le plugin actuel ne permet pas de faire certaines actions qui seraient très utiles pour les étudiants.

Ces fonctionnalités sont :

- **Navigation** : Ajout de la navigation vers les définitions des prédicats et vers leur usage dans le projet.
- **Refactoring** : Ajout de la possibilité de renommer les prédicats et les variables.
- **Autocomplétion** : Ajout de l'autocomplétion des prédicats et des variables.
- **Affichage des erreurs et des warnings** : Ajout de l'affichage des erreurs et des warnings dans le code.
- **Déploiement du plugin** : Déploiement du plugin sur le marketplace d'IntelliJ et sur le site de GNU-Prolog.

1.4 Structure du rapport

Ce rapport est divisé en 4 grandes parties :

- **Introduction** : Présentation du projet et de ses objectifs.
- **Mise à jour du plugin** : Présentation de la mise à jour du plugin, de la correction des bugs et de l'ajout des tests unitaires.
- **Ajout des fonctionnalités** : Présentation de l'ajout des fonctionnalités au plugin.
- **Conclusion** : Conclusion du projet.

2 | Mise à jour du Plugin

Ce chapitre traite de la correction des bugs existants dans le plugin, de la mise à jour du SDK ainsi que le remplacement des méthodes dépréciées par des méthodes plus récentes. Enfin, un pipeline a été mis en place sur GitLab afin de tester le plugin à chaque commit.

2.1 Correction des bugs

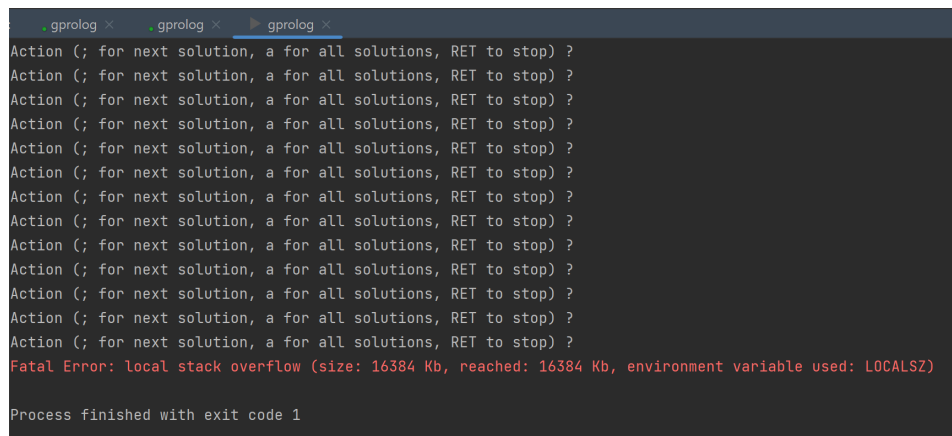
Les bugs existants identifiés dans le plugin sont les suivants :

- **Bug 1** : Le plugin ne permet pas de lancer le script prolog dans la console d'IntelliJ Idea sous Windows.
- **Bug 2** : Le plugin ne reconnaît pas un opérateur lors de la programmation logique par contrainte.

2.1.1 Bug 1 : Run dans la console d'IntelliJ Idea sous Windows

Le plugin ne permet pas de lancer le script prolog dans la console d'IntelliJ Idea sous Windows.

Description du bug : Une fois le script prolog compilé et lancé, l'interpréteur prolog boucle sur l'entrée de la console d'IntelliJ Idea, ce qui empêche l'utilisateur de saisir des commandes.



```
gprolog x gprolog x gprolog x
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Action (; for next solution, a for all solutions, RET to stop) ?
Fatal Error: local stack overflow (size: 16384 Kb, reached: 16384 Kb, environment variable used: LOCALSZ)
Process finished with exit code 1
```

FIGURE 1 – Image du bug en action dans la console d'IntelliJ Idea

Cause du bug : La cause n'a pas été clairement identifiée. Cependant, il semble que le problème vienne du fait que le programme prolog s'attend à recevoir des entrées de la console d'IntelliJ Idea et que celle-ci lui envoie des retours à la ligne sans que l'utilisateur ne les ait saisis. Malgré plusieurs tests, je n'ai pas réussi à trouver la cause exacte du problème.

Comme le problème ne se pose pas sous Linux et macOS, il est possible que celui-ci vienne de la façon dont IntelliJ Idea gère les retours à la ligne sous Windows.

Un test à envisager serait de faire notre propre programme qui attend des entrées de la console et de voir si le problème se pose également. Si c'est le cas, alors le problème ne vient pas du plugin mais d'IntelliJ Idea. Si le problème ne se pose pas, alors il faudrait en chercher la cause dans le plugin ou dans

l'interpréteur prolog.

Solution du bug : Durant l'implémentation de la fonctionnalité permettant d'afficher les erreurs de compilation dans le code, j'ai trouvé une solution fonctionnelle.

En lançant une invite de commande invisible, il est possible de lancer des commandes via les InputStream et OutputStream de Java. Voici le code permettant de lancer le script prolog :

```

1 private Process createWindowsProcess(Path interpreterPath) throws IOException {
2     Process p;
3     BufferedWriter writer;
4     command = "cmd.exe /min";
5
6     if (isInExternalWindow) {
7         p = Runtime.getRuntime().exec("cmd.exe /min");
8         writer = new BufferedWriter(new OutputStreamWriter(p.getOutputStream()));
9         writer.write("cd " + Path.of(getWorkingDir())); //Go to the working
        directory
10        writer.newLine();
11        writer.write("set LINEDIT=gui=yes"); //Prevent windows from opening a
        console
12        writer.newLine();
13        String query = " --query-goal \"consult('" + Path.of(filePath).getFileName
        () + "')\"";
14        writer.write("start " + interpreterPath.toString() + query); //Launch the
        compiler
15        writer.newLine();
16    } else {
17        p = Runtime.getRuntime().exec("cmd.exe /min");
18        writer = new BufferedWriter(new OutputStreamWriter(p.getOutputStream()));
19        writer.write("set LINEDIT=gui=no"); //Prevent windows from opening a
        console
20        writer.newLine();
21        writer.write("cd " + Path.of(getWorkingDir())); //Go to the working
        directory
22        writer.newLine();
23        writer.write(interpreterPath.toString()); //Launch the compiler
24        writer.newLine();
25        writer.write("consult('" + Path.of(filePath).getFileName() + "')");
26        writer.newLine();
27    }
28
29    writer.flush();
30
31    return p;
32 }

```

Code 1 – Code permettant de lancer le script prolog

2.1.2 Bug 2 : Opérateur non reconnu lors de la PLC

Le plugin ne reconnaît pas un opérateur lors de la programmation logique par contrainte.

Description du bug : L'opérateur "#=<" n'est pas reconnu en temps que réel opérateur.

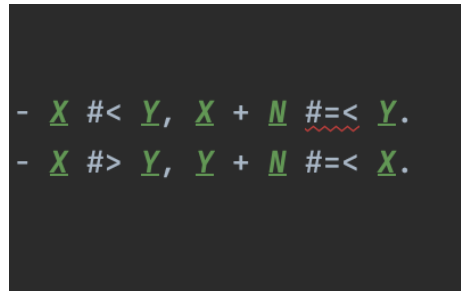


FIGURE 2 – Image du bug en action dans l'éditeur

Cause du bug : Dans le fichier de grammaire BNF, l'opérateur "#=<" n'est pas présent mais, à la place, il y a l'opérateur "#<=" qui, lui, n'existe pas en Prolog. Il s'agit certainement d'une faute de frappe lors de la rédaction de celui-ci.

Solution du bug : L'opérateur a été ajouté dans la grammaire.

2.2 Mise à jour du SDK

Le SDK utilisé pour le développement du plugin est la version 173.0 (2017.3.xxx) de IntelliJ Idea. Actuellement, la version la plus récente est la 222.4345.14 (2022.2.3) et de nombreuses méthodes utilisées dans le plugin sont dépréciées.

Le but est de mettre à jour le SDK vers une version plus récente afin de pouvoir utiliser les méthodes les plus récentes.

2.2.1 Passage du projet en Gradle

Le projet existant n'avait aucun gestionnaire de dépendances et les dépendances étaient gérées manuellement. Pour lancer le plugin, il fallait donc installer un IDE (par exemple IntelliJ Idea Community) et ajouter les dépendances dans les paramètres du projet.

Afin de se conformer à la documentation de JetBrains ainsi que de faciliter grandement le développement du plugin, le projet a été migré vers Gradle.

Pour cela, il a fallu ajouter un fichier build.gradle.kts à la racine du projet et configurer les dépendances. Afin de pouvoir générer le parser et le lexer, il a fallu ajouter des goals Gradle pour lancer les commandes suivantes :

```
1 gradle generateParser
2 gradle generateLexer
```

Code 2 – Génération du parser et du lexer

Ces commandes sont définies dans le fichier build.gradle.kts et permettent de générer le parser et le lexer à partir des fichiers .flex et .bnf.

```
1 generateLexer {
2     source.set("src/main/java/ch/heiafr/intelliprolog/Prolog.flex")
3     targetDir.set("src/gen/java/ch/heiafr/intelliprolog/")
4     targetClass.set("PrologLexer")
5     skeleton.set(file("src/main/java/ch/heiafr/intelliprolog/Prolog.skeleton"))
6     purgeOldFiles.set(true)
7 }
8
```



```

9
10 generateParser {
11
12     try {
13         val compiledFilesSources =
14             files("build/classes/java/main/")
15             classpath.from(compiledFilesSources)
16     } catch (e: Exception) {
17         // Ignore => no compiled files when running the task for the first time
18     }
19
20     source.set("src/main/java/ch/heiafr/intelliprolog/Prolog.bnf")
21     targetRoot.set("src/gen/java/")
22     pathToParser.set("PrologParser.java")
23     pathToPsiRoot.set("psi")
24     purgeOldFiles.set(false)
25 }

```

Code 3 – Gradle pour la génération du parser et du lexer

Il a aussi fallu configurer le GrammarKit afin de spécifier la version de JFlex.

```

1 grammarKit {
2     jflexRelease.set("1.7.0-2")
3 }

```

Code 4 – Utilisation de JFlex

Un fichier settings.properties a également été ajouté afin de spécifier :

- Le nom du plugin
- Le groupe du plugin
- La version du plugin
- La version minimum d'IntelliJ Idea requise pour lancer le plugin
- La plateforme cible (Ultimate, Education, Community) pour lancer le plugin
- La version de la plateforme cible pour lancer le plugin
- La dépendance de la plateforme cible
- La version de Java utilisée
- La version cible de Java pour le Kotlin
- La version de Gradle utilisée

Le fichier settings.properties est lu par le fichier build.gradle.kts afin de configurer le plugin lors de la compilation ainsi que lors de la création du ZIP du plugin.

2.2.2 Le problème du GrammarKit

Le plugin utilise le GrammarKit afin de générer le parser et le lexer à partir des fichiers .flex et .bnf. Le problème est que pour pouvoir générer le parser et le lexer, il faut que le plugin soit compilé et pour cela, il faut que le parser et le lexer soient générés. . .

Pour résoudre ce problème, j'ai cherché une solution sur de nombreux forums et malgré plusieurs tentatives, je n'ai pas réussi à trouver de solution fonctionnelle aussi bien en local que sur la CI.

J'ai donc décidé d'écrire mes propres "goals" Gradle afin de générer le parser et le lexer en 3 étapes :

1. Génération du parser et du lexer
2. Compilation du plugin
3. Génération du parser et du lexer

```

1 register("compileAndRegenerate") {
2     dependsOn("compileJava")
3     finalizedBy("generateParser")
4 }
5
6 register("initProject") {
7
8     doFirst {
9         generateParser.get().generateParser()
10        generateLexer.get().generateLexer()
11        println("Classes generated")
12    }
13    finalizedBy("compileAndRegenerate")
14 }

```

Code 5 – Code Gradle pour l'initialisation du projet

Attention : lorsque le projet est cloné, il faut lancer la commande suivante afin de générer le parser et le lexer :

```

1 gradle initProject

```

Code 6 – Initialisation du projet

J'ai dû implémenter certaines méthodes dans un fichier "PrologNamedElementHelperImpl.java" afin de pouvoir générer le parser et le lexer sans erreur de compilation.

```

1 public class PrologNamedElementHelperImpl extends ASTWrapperPsiElement implements
    PrologNamedElement {
2
3     public PrologNamedElementHelperImpl(@NotNull ASTNode node) {
4         super(node);
5     }
6
7     @Override
8     public @Nullable PsiElement getNameIdentifier() {
9         return null;
10    }
11
12    @Override
13    public PsiElement setName(@NlsSafe @NotNull String name) throws
        IncorrectOperationException {
14        return null;
15    }
16 }

```

Code 7 – Code permettant de supprimer les erreurs de compilation

2.3 Mise en place de la CI

La CI a été mise en place sur GitLab afin de tester le plugin à chaque commit. Celle-ci permet aussi de créer une release entièrement automatisée lors de la création d'un tag sur le dépôt GitLab.

La pipeline est définie dans le fichier .gitlab-ci.yml à la racine du projet.

Le pipeline est composé de 3 jobs :

- **Build** : Génération du parser et du lexer, compilation du plugin
- **Test** : Lancement des tests unitaires
- **Release** : Création d'une release du plugin (uniquement sur les tags)

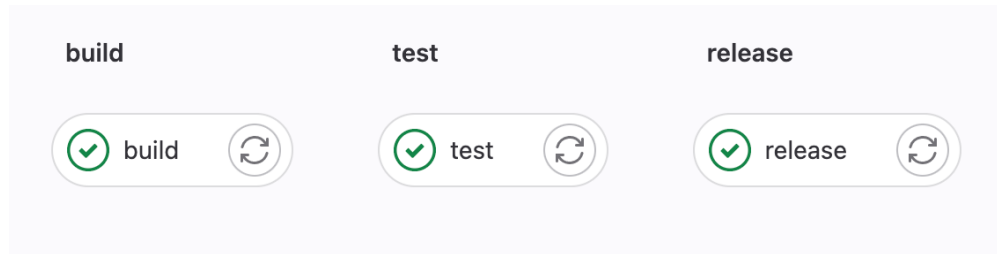


FIGURE 3 – Pipeline de la CI

3 | Tests unitaires

Les tests unitaires sont spécifiques à IntelliJ et permettent de tester les fonctionnalités du plugin. Ils sont écrits en Java et sont exécutés à chaque commit sur la CI.

3.1 Mise en place des tests unitaires

Les tests unitaires sont écrits dans le dossier "test" à la racine du projet.

Ils sont séparés en 2 classes de tests :

- **PrologCodeInsightTest** qui teste les fonctionnalités telles que :
 - les annotations
 - les commentaires (Line comment, Block comment)
 - le renommage par refactoring
 - la fonction "Find usages"
 - l'auto-complétion
- **PrologParsingTest** qui teste si le parser et le lexer sont correctement générés et fonctionnels.

La classe "PrologCodeInsightTest" étend la classe "LightJavaCodeInsightFixtureTestCase" qui permet de tester les fonctionnalités du plugin de manière plus simple.

Cette classe permet de configurer des tests automatiques avec des fichiers d'entrée et des fichiers de sortie attendus. Voici un exemple de test :

```
1 public void testRenameRefactor() {
2     myFixture.configureByFiles("RenameTestData1.pl", "RenameTestData2.pl", "
   RenameTestData3.pl");
3     myFixture.renameElementAtCaretUsingHandler("fifo_new_renamed");
4     myFixture.checkResultByFile("RenameTestData1.pl", "RenameTestData1Renamed.pl",
   false);
5     myFixture.checkResultByFile("RenameTestData2.pl", "RenameTestData2Renamed.pl",
   false);
6     myFixture.checkResultByFile("RenameTestData3.pl", "RenameTestData3Renamed.pl",
   false);
7 }
```

Code 8 – Tests unitaires pour le refactoring

La méthode "configureByFiles" permet de charger les fichiers d'entrée.

La méthode "renameElementAtCaretUsingHandler" permet de renommer l'élément sélectionné dans le fichier d'entrée.

La méthode "checkResultByFile" permet de comparer le fichier d'entrée avec le fichier de sortie attendu.

3.2 Exécution des tests unitaires

Les tests unitaires sont exécutés à chaque commit sur la CI.

Pour les exécuter en local, il faut lancer la commande suivante :

1 `gradle test`

Code 9 – Lancement des tests

4 | Ajout des fonctionnalités

Ce chapitre liste les fonctionnalités qui seront implémentées.

La documentation de JetBrains a été grandement utilisée¹.

4.1 Navigation

La navigation est une fonctionnalité très utilisée par les développeurs. Elle permet de se déplacer dans le code source d'un projet. Dans le cas d'un projet Prolog, il est important de pouvoir naviguer entre les différents prédicats, leurs usages et leur définitions.

Dans le cas d'un langage comme Prolog, il est possible de définir plusieurs prédicats avec le même nom mais avec une arité (nombre d'arguments) différente :

```
1  predicat(A).  
2  predicat(A, B).  
3  predicat(A, B, C).
```

Code 10 – Exemple de prédicats nommés pareillement

La navigation devra donc être consciente de l'arité du prédicat afin de pouvoir naviguer entre les différentes définitions du prédicat mais aussi lors de la recherche de l'utilisation de celui-ci dans le code.

Ceci sera également valable plus tard, lors du refactoring.

4.1.1 Goto Declaration

Cette partie plus spécifique parlera de la fonctionnalité Goto Declaration. Cette fonctionnalité permet de naviguer vers la définition d'un prédicat. Pour cela, il faut sélectionner le prédicat et appuyer sur Ctrl+B (ou Cmd+B sur Mac). La définition du prédicat s'ouvrira alors dans un nouvel onglet s'il existe. Si le prédicat est défini à plusieurs endroits, un menu sous forme de popup s'ouvrira et permettra de choisir la définition à ouvrir.

Réalisation dans le code

La réalisation n'a pas été de tout repos : en effet, le peu de documentation et d'exemples sur le sujet a rendu la tâche difficile.

Cheminement :

1. Recherche d'exemples sur GitHub et dans la documentation de JetBrains
2. Création d'une classe "ch.heiafr.intelliprolog.reference.PrologGotoDeclarationHandler"
3. Méthode de recherche d'inclusion de fichiers
4. Méthode de recherche des définitions
5. Méthode qui permet de "matcher" un prédicat avec une définition

Méthode permettant d'extraire le prédicat défini dans la "PrologSentence"

```

1 public static PsiElement findDefinition(PrologSentence sentence) {
2
3     if (sentence == null) {
4         return null;
5     }
6
7     //Multiple cases
8     Class<?>[][] searchPatterns = new Class[][]{{
9         // 1. test :- test2. => atom used as predicate name
10        PrologSentence.class, PrologOperation.class,
11        PrologNativeBinaryOperation.class, PrologBasicTerm.class, PrologAtom.class}, {
12        // 2. test(A) :- test2. => compound used as predicate name
13        PrologSentence.class, PrologOperation.class,
14        PrologNativeBinaryOperation.class, PrologBasicTerm.class, PrologCompound.class
15    }, {
16        // 3. test(X). => compound without definition
17        PrologSentence.class, PrologCompound.class}, {
18        // 4. test. => atom without definition
19        PrologSentence.class, PrologAtom.class}};
20
21    for (Class<?>[] searchPattern : searchPatterns) {
22        var definition = patternFitPsiElement(sentence, searchPattern);
23
24        if (definition != null) {
25            return definition; //Found a definition
26        }
27    }
28
29    //If not found, return null
30    return null;
31 }

```

Code 11 – Méthode permettant d'extraire le prédicat défini dans la "PrologSentence"

Le fonctionnement de cette méthode est le suivant :

1. Pour une certaine "PrologSentence" en entrée, on va tester plusieurs cas possibles :
 - (a) test :- test2. => c'est un simple atom qui est défini
 - (b) test(A) :- test2. => c'est un compound qui est défini
 - (c) test(X). => c'est un compound qui est défini mais énoncé en tant que fait
 - (d) test. => c'est un atom qui est défini mais énoncé en tant que fait
2. Si le cas est trouvé, on récupère le prédicat correspondant.
3. Sinon, on retourne null.

Méthode permettant de trouver tous les fichiers inclus récursivement

```

1 public static Collection<PsiElement> findEveryImportedFile(PsiElement elt,
2     Collection<String> files) {
3     if (elt == null) {
4         return new ArrayList<>();
5     }
6
7     Collection<String> paths = PsiTreeUtil.collectElementsOfType(elt.
8         getContainingFile(), PrologSentence.class).stream()
9         .map(ReferenceHelper::findIncludeStatement) // Find the first compound
10        name which is the predicate name

```

```

8      .filter(Objects::nonNull) //Prevent null values
9      .map(ReferenceHelper::extractQuotedString)//Extract the quoted string
10     .filter(Objects::nonNull)//Prevent null values
11     .filter(s -> !files.contains(s)) //Filter out already visited files
12     .collect(Collectors.toList()); //Collect to list
13
14     files.addAll(paths); //Add the new paths to the list of visited files to
    prevent infinite recursion
15
16     Collection<PsiElement> psiFiles = new ArrayList<>(); //Create a new list of
    psi files
17     for (String path : paths) {
18         PsiElement rootElt = pathToPsi(elt, path); //Get the psi element from the
    path
19         psiFiles.add(rootElt); //Add the psi element to the list
20         psiFiles.addAll(findEveryImportedFile(rootElt, files)); //Find imported
    files recursively
21     }
22     return psiFiles;
23 }

```

Code 12 – Méthode permettant de trouver tous les fichiers inclus récursivement

Le fonctionnement de cette méthode est le suivant :

1. On recherche toutes les "PrologSentence" du fichier.
2. Pour chaque "PrologSentence", on va chercher le prédicat "include".
3. Si le prédicat est trouvé, on va chercher le chemin du fichier inclus.
4. Si le chemin n'a pas déjà été visité, on va chercher le fichier correspondant.
5. Si le fichier est trouvé, on va chercher tous les fichiers inclus dans ce fichier de manière récursive.
6. On retourne la liste des fichiers trouvés.

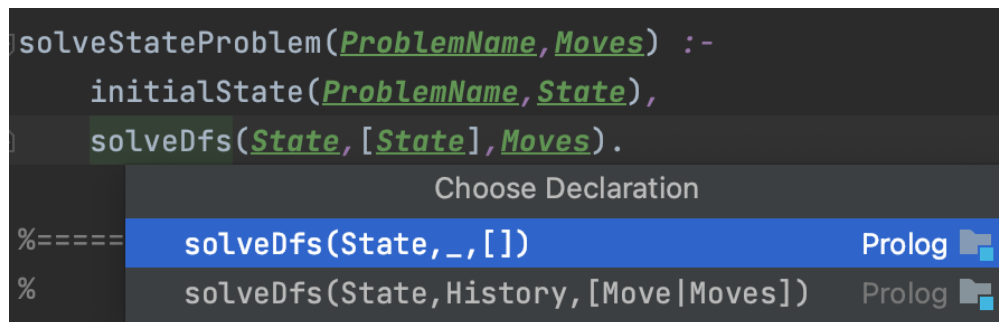


FIGURE 4 – Déclaration d'un prédicat

4.1.2 Find usage

Pour la recherche d'utilisation, c'est un peu plus simple car certaines méthodes sont déjà implémentées lors de la réalisation du "Goto declaration".

En revanche, la recherche doit être exécutée dans un thread séparé pour éviter de bloquer l'IDE pendant la recherche. Si ce n'est pas le cas, une exception est levée automatiquement par l'IDE afin d'interrompre la méthode.

Le fonctionnement est le suivant :

1. On récupère le prédicat à rechercher.
2. On lance la recherche dans un thread séparé afin de ne pas bloquer l'IDE.
3. On récupère tous les fichiers .pl.
4. On filtre en fonction de la portée désirée (choisi par l'utilisateur).
5. On filtre et on retourne les résultats sous forme de UsageInfo.
6. Le Thread s'arrête et les résultats sont affichés.

Lancement d'un thread séparé

```
1 public class PrologCustomUsageSearcher extends CustomUsageSearcher {
2
3     @Override
4     public void processElementUsages(@NotNull PsiElement element,
5                                     @NotNull Processor<? super Usage> processor, @NotNull
6                                     FindUsagesOptions options) {
7
8         Application app = ApplicationManager.getApplication(); // get the
9         application
10
11         app.runReadAction(new FindPrologCompoundNameRunnable(element, processor,
12         options));
13     }
14
15     private static class FindPrologCompoundNameRunnable implements Runnable {
16         private final PsiElement element;
17         private final Processor<? super Usage> processor;
18         private final FindUsagesOptions options;
19
20         public FindPrologCompoundNameRunnable(PsiElement elt, Processor<? super
21         Usage> processor, FindUsagesOptions options) {
22             //Code here
23         }
24
25         @Override
26         public void run() {
27             //Code here
28         }
29     }
30 }
```

Code 13 – Lancement d'un thread séparé

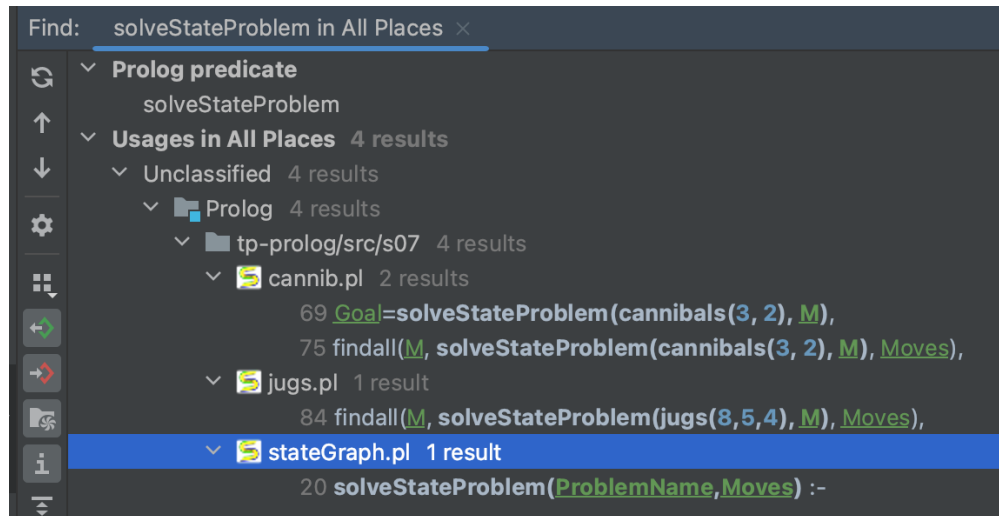


FIGURE 5 – Recherche d'utilisation d'un prédicat

4.2 Refactoring

Le refactoring est une fonctionnalité qui permet de modifier le code de manière automatique comme par exemple renommer une variable, une méthode, etc. C'est une fonctionnalité très appréciée des développeurs car elle permet de gagner du temps et d'éviter les erreurs de frappe ou simplement les oublis lors de la modification du code.

Dans notre cas, seul la partie renommage d'un prédicat, d'un atome ou d'une variable nous intéresse. En effet, il est fréquent de renommer un prédicat ou une variable dans un fichier Prolog et il est important de renommer toutes les occurrences de ce prédicat ou de cette variable dans tous les fichiers inclus.

4.2.1 Renommer un prédicat

Pour renommer un prédicat, il faut d'abord récupérer le prédicat à renommer. Ensuite, il faut récupérer tous les fichiers inclus dans le fichier courant ainsi que tous les fichiers inclus dans ces fichiers inclus et ainsi de suite.

Il faut ensuite remplacer toutes les occurrences du prédicat par le nouveau nom.

Exemple d'un renommage de prédicat :

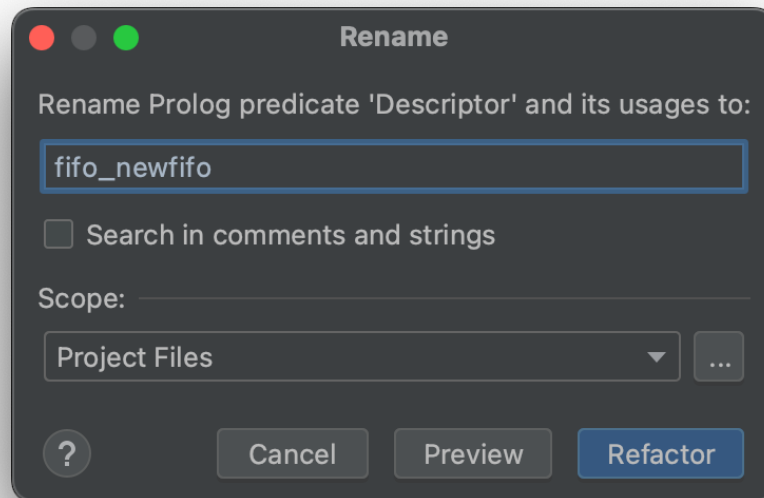
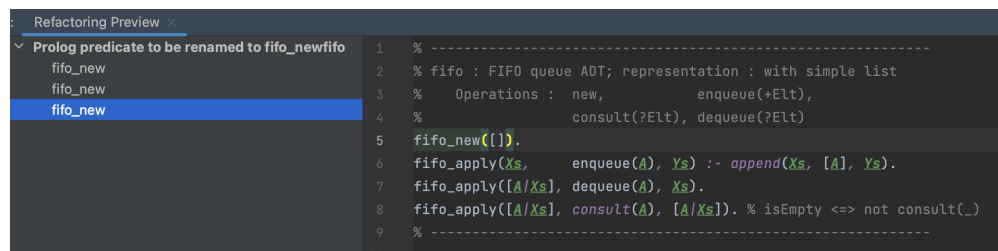


FIGURE 6 – Fenêtre de renommage d'un prédicat



4.2.2 Renommer une variable

Pour ce qui est du renommage d'une variable, la portée du refactoring est limitée à la phrase Prolog dans laquelle se trouve la variable.

Le fonctionnement est similaire à celui du renommage d'un prédicat, à la différence que l'on ne va pas chercher en dehors de la phrase Prolog dans laquelle se trouve la variable.

4.2.3 Fonctionnement du renommage

Le fonctionnement du renommage est le suivant :

1. On récupère le prédicat ou la variable à renommer.
2. On récupère tous les fichiers inclus dans le fichier courant.
3. On récupère tous les fichiers inclus dans ces fichiers inclus et ainsi de suite.

4. On affiche un aperçu des modifications.
5. On remplace toutes les occurrences du prédicat ou de la variable par le nouveau nom.

La classe qui gère le renommage est **PrologRenamePsiElementProcessor**. Voici une brève description des méthodes de cette classe héritée de **RenamePsiElementProcessor** :

1. **canProcessElement** : Cette méthode permet de vérifier si l'élément peut être renommé. Dans notre cas, on vérifie si l'élément est un prédicat ou une variable.
2. **prepareRenaming** : Cette méthode permet de préparer le renommage. Dans notre cas, on récupère tous les prédicats de tous les fichiers touchés par le renommage.
3. **renameElement** : Cette méthode permet de renommer chaque prédicat/variable.

4.3 Affichage des erreurs et warnings

Lors de la compilation d'un fichier Prolog, il est possible que des erreurs ou des warnings apparaissent qui ne sont pas détectés plus tôt car ce ne sont pas des erreurs de syntaxe.

Par exemple, il est possible d'avoir une erreur de prédicats non définis ou une erreur de variables non définies ou simplement des avertissements par rapport à des variables "Singletons".

4.3.1 L'idée

L'idée est de récupérer les erreurs et les warnings de la compilation du fichier Prolog et de les afficher dans l'éditeur Prolog.

Ceci implique différentes contraintes :

1. Il faut prendre en compte que la compilation doit avoir lieu en temps réel et en arrière-plan.
2. La compilation doit avoir lieu sur macOS, Linux et Windows. Ce qui implique de mettre en place un système multi-plateforme.
3. La compilation doit aussi être dans un thread séparé pour ne pas bloquer l'interface et ne pas ralentir l'éditeur.

4.3.2 La mise en pratique

Pour la compilation, nous utiliserons le SDK relatif au projet ouvert.

Pour l'aspect compilation en temps réel, JetBrains propose une classe nommée "ExternalAnnotator" qui permet de faire des annotations externes.

Cette classe permet de faire des annotations externes, notamment à l'aide d'un thread séparé. La classe se présente comme suit :

1. **collectInformation** : Cette méthode permet de récupérer les informations afin de générer les annotations. C'est dans cette partie que le lancement de la compilation aura lieu.
2. **doAnnotate** : Cette méthode permet de générer les annotations pour chaque ligne.

Voici le code permettant de récupérer un processus de compilation Prolog :

```

1  public static Process getProcess(Path compiler, Path filePath) throws
    IOException, CantRunException {
2      Process p;
3      BufferedWriter writer;
4
5      if (SystemInfo.isWindows) {
6          p = Runtime.getRuntime().exec("cmd.exe /min");
7          writer = new BufferedWriter(new java.io.OutputStreamWriter(p.
getOutputStream()));
8          writer.write("set LINEDIT=gui=no"); //Prevent windows from opening a
console
9          writer.newLine();
10         writer.write(compiler.toString()); //Launch the compiler
11         writer.newLine();
12         writer.flush(); //Flush the stream
13     } else {
14         p = Runtime.getRuntime().exec(compiler.toString());
15         writer = new BufferedWriter(new java.io.OutputStreamWriter(p.
getOutputStream()));
16     }
17
18     writer = new BufferedWriter(new java.io.OutputStreamWriter(p.
getOutputStream()));
19     String normalizedFilePath = filePath.toString().replace("\\", "/"); //
Mandatory for windows
20     String goal = "consult('" + normalizedFilePath + "').";
21     writer.write(goal);
22     writer.newLine();
23     writer.flush();
24     writer.close();
25     return p;
26 }

```

Code 14 – Méthode de créer un processus de compilation Prolog

Au final, voici le résultat en action dans l'éditeur :



FIGURE 8 – Affichage d'avertissement grâce à la compilation

4.4 Auto-complétion

L'auto-complétion est une fonctionnalité très importante pour un IDE. Elle permet de faciliter la saisie de code en proposant des suggestions d'auto-complétion et en évitant de faire des fautes de frappe.

L'auto-complétion est implémentée dans le plugin grâce à la classe "PrologCompletionContributor" qui étend la classe "CompletionContributor".

4.4.1 Fonctionnement de l'auto-complétion

L'auto-complétion est déclenchée lors de la frappe d'un caractère. Lors de l'écriture, l'auto-complétion propose les prédicats définis dans le fichier courant ainsi que dans les fichiers inclus de manière récursive. L'arité du prédicat est également affichée dans la liste des propositions d'auto-complétion, permettant ainsi de savoir combien d'arguments le prédicat attend.

Dépendamment du contexte, l'auto-complétion propose des propositions différentes. Par exemple, si l'utilisateur écrit actuellement le corps d'une règle, l'auto-complétion proposera, en plus du reste, les variables de la règle.

4.4.2 Gestion des erreurs en temps réel

Le problème que j'ai rencontré lors de l'implémentation de l'auto-complétion est qu'il y a forcément des erreurs de syntaxe dans le code, étant donné que l'utilisateur n'a pas encore fini de taper son code. Cela pose problème car l'auto-complétion ne détecte pas les prédicats situés sous le niveau de l'erreur de syntaxe.

Pour résoudre ce souci, j'ai implémenté une méthode qui permet de détecter les erreurs de syntaxe et de les corriger en temps réel. Cette méthode est appelée à chaque fois que l'utilisateur tape un caractère dans le fichier. Elle parcourt le fichier et détecte les erreurs de syntaxe. Si une erreur est détectée, elle est corrigée temporairement et l'auto-complétion est relancée.

Toutefois, cette méthode n'est pas parfaite car elle se limite à une seule erreur de syntaxe.

```
1 private static PsiElement removeErrorForIndexing(PsiElement rootFile) {
2     PsiElement copy = rootFile.copy();
3     var error = PsiTreeUtil.findChildOfType(copy, PsiErrorElement.class);
4
5     if (error == null) {
6         return copy;
7     }
8
9
10    var parent = error.getParent();
11
12    boolean deleteParent = false;
13    //Find all children of the parent of the error
14    PsiElement lastUsable = null;
15    for (var child : parent.getChildren()) {
16        //If the child is an error, delete it
17        if (child instanceof PsiWhiteSpace || child instanceof PsiComment) {
18            continue;
19        }
20        if (child instanceof PsiErrorElement) {
21            if (lastUsable != null) {
22                lastUsable.delete();
23            } else {
24                deleteParent = true;
25            }
26            break;
27        }
28        lastUsable = child;
29    }
```

```
30     if (deleteParent) {
31         if (Objects.equals(parent, copy)) {
32             return null; //Impossible to delete the root file
33         }
34         parent.delete();
35     }
36
37     try {
38         copy = PrologElementFactory.rebuildTree(copy);
39         return copy;
40     } catch (Exception e) {
41         //More than one error in the file => impossible to rebuild the tree
42         return null;
43     }
44 }
```

Code 15 – Méthode de correction des erreurs de syntaxe

Si plus d'une erreur de syntaxe est détectée, la méthode retourne null et l'auto-complétion ne se lance pas.

4.4.3 Exemple d'utilisation

Voici un exemple d'utilisation de l'auto-complétion. L'utilisateur écrit le prédicat "pred" et l'auto-complétion propose les prédicats "predicat/1" et "prediction/2".

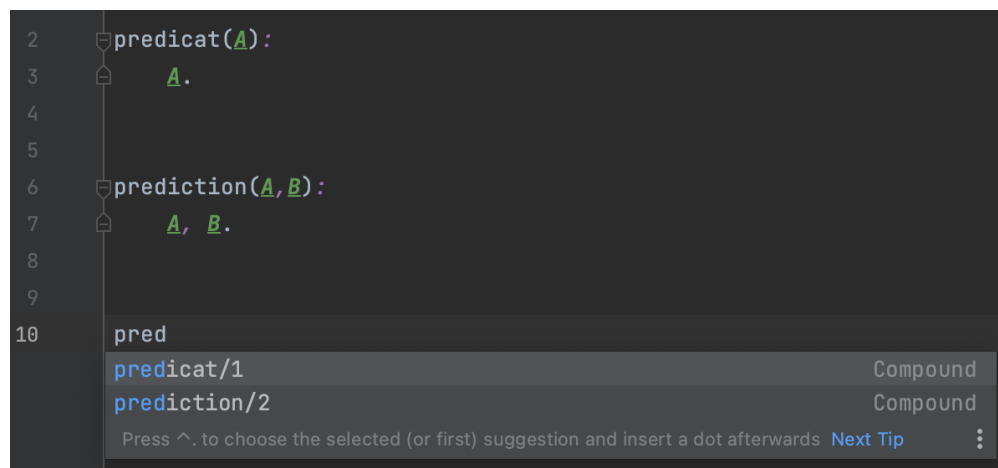


FIGURE 9 – Exemple d'utilisation de l'auto-complétion

5 | Déploiement

Nous avons décidé de déployer le plugin sur le marketplace de JetBrains. Afin de permettre à tout le monde de pouvoir modifier le plugin, nous avons aussi décidé de le mettre sur Github.

5.1 Choix de la licence

Nous avons choisi de mettre le plugin sous la licence MIT car elle est très permissive et permet à tout le monde de pouvoir modifier le plugin. Une licence moins permissive aurait pu être la GNU GPLv3, mais elle impose de mettre le code source en open source et de mettre la licence dans le plugin.

Voici un tableau comparatif des différentes licences que nous avons étudiées :

Permissions	MIT	GNU GPLv3	GNU LGPLv3	Mozilla Public 2.0	Apache 2.0
Utilisation commerciale	x	x	x	x	x
Utilisation privée	x	x	x	x	x
Distribution	x	x	x	x	x
Modification	x	x	x	x	x
Conditions					
Indiquer la source		x	x	x	
Même licence		x	x	x	
Indiquer la licence et le copyright	x	x	x	x	x

5.2 Déploiement sur le marketplace

Pour déployer le plugin sur le marketplace, il faut créer un compte sur le site de JetBrains. Une fois le compte créé, il suffit de se rendre sur la page de déploiement du plugin, de choisir le plugin à déployer et de remplir les informations demandées.

Voici un exemple de ce que l'on peut voir sur la page de déploiement du plugin :

Upload Plugin for IDEs ▾

Plugin file

Aucun fichier choisi

jar or .zip format

License

Select license ▾

https://

Your license depends on several aspects, like how you use personal data. We recommend using [Open Source licenses](#) or independent [EULA generators](#) (not affiliated with JetBrains).

Tags

No tags selected ▾

Tags will help users to find your plugin

Channel (optional)

Stable

By default, the 'Stable' plugin repository channel, available to all JetBrains plugin repository users, will be used. If you do not want your plugin to be generally available, choose a custom channel. [Learn more about custom release channels](#)

Upload Plugin

FIGURE 10 – Premier déploiement du plugin

Lors des déploiements suivants, il suffit de cliquer sur le bouton « Update » et de choisir le fichier .zip du plugin.

La page du plugin sur le marketplace est la suivante : <https://plugins.jetbrains.com/plugin/20982-prologcode>

Au moment où ce rapport est écrit, le plugin n'a pas encore été approuvé en raison du nom (IntelliProlog). Le nom ne devant pas faire référence à JetBrains ou l'un de ses produits, le plugin va être renommé et le plugin devrait être approuvé.

5.3 Déploiement sur Github

Une organisation Github a été créée pour le projet. Le projet a été publié à l'adresse suivante : <https://github.com/IntelliProlog/IntelliProlog>

6 | Conclusion

En conclusion, ce projet de semestre a été très intéressant et m'a permis d'apprendre à créer des composants d'un plugin pour IntelliJ IDEA. J'ai pu découvrir le fonctionnement de ceux-ci et, malgré quelques difficultés en raison du manque de documentation, j'ai pu ajouter toutes les fonctionnalités que je souhaitais.

Je pense que le plugin constitue à présent une bonne aide pour les étudiants qui suivent le cours de programmation logique. Il manque encore quelques fonctionnalités clés, comme la possibilité de formater le code ou de déboguer les programmes, mais il est déjà très fonctionnel.

Nous avons également pu mettre en ligne le plugin sur le marketplace d'IntelliJ IDEA, ce qui permettra aux étudiants de l'installer facilement et offre la possibilité à d'autres personnes de l'utiliser.

Déclaration d'honneur

Je, soussigné, Erwan Sturzenegger, 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 toutes autres formes de fraudes. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

A handwritten signature in dark ink, reading "Sturzenegger". The script is cursive and fluid, with the first letter 'S' being large and prominent. The signature is centered on the page.

Glossaire

Forme de Backus-Naur (BNF) Notation permettant d'écrire des règles syntaxiques.

GitLab Système libre de versionning basé sur Git.

GNU-Prolog Connu aussi sous le nom "gprolog". Compilateur prolog développé par Daniel Diaz.

Gradle Environnement de développement édité par JetBrains.

IDE Environnement de développement intégré.

IntelliJ IDEA Environnement de développement édité par JetBrains.

JetBrains Éditeur de logiciels destinés aux développeurs.

Lexer Produit des "tokens" pour le Parser.

Parser Analyseur syntaxique.

PLC Programmation logique par contrainte.

Plugin Extension d'un logiciel.

SDK Software Development Kit - Ensemble d'outils destinés aux développeurs.

Table des illustrations

Figure 1	Image du bug en action dans la console d'IntelliJ Idea	3
Figure 2	Image du bug en action dans l'éditeur	5
Figure 3	Pipeline de la CI	8
Figure 4	Déclaration d'un prédicat	13
Figure 5	Recherche d'utilisation d'un prédicat	15
Figure 6	Fenêtre de renommage d'un prédicat	16
Figure 7	Aperçu du refactoring	16
Figure 8	Affichage d'avertissement grâce à la compilation	18
Figure 9	Exemple d'utilisation de l'auto-complétion	20
Figure 10	Premier déploiement du plugin	22

Liste des codes

1	Code permettant de lancer le script prolog	4
2	Génération du parser et du lexer	5
3	Gradle pour la génération du parser et du lexer	5
4	Utilisation de JFlex	6
5	Code Gradle pour l'initilisation du projet	7
6	Initialisation du projet	7
7	Code permettant de supprimer les erreurs de compilation	7
8	Tests unitaires pour le refactoring	9
9	Lancement des tests	10
10	Exemple de prédicats nommés pareillement	11
11	Méthode permettant d'extraire le prédicat défini dans la "PrologSentence"	12
12	Méthode permettant de trouver tous les fichiers inclus récursivement	12
13	Lancement d'un thread séparé	14
14	Méthode de créer un processus de compilation Prolog	18
15	Méthode de correction des erreurs de syntaxe	19

Bibliographie

- [1] **IntelliJ Platform Plugin SDK.** <https://plugins.jetbrains.com/docs/intellij/welcome.html>.