Projet Génie Logiciel

Documentation de conception

Equipe: Equipe GL23

Professeurs: **Patrick Reignier** & **Tarik Larja**

Date : 26 Janvier 2022

Contents

1 Introduction

Les langages de programmation de haut niveau posent toujours des questions cruciales sur ce qui se passe exactement lorsque nos programmes s'exécutent et comment les machines comprennent le code source écrit en langage humain et le transforment en un exécutable. C'est le rôle du compilateur que nous avons programmé dans ce projet. Ce dernier donne une vision plus concrète des langages de programmation que nous connaissons déjà et révèle ainsi le mystère derrière l'exécution d'un programme. Dans le but d'une maintenance ou d'une éventuelle amélioration de notre compilateur nous expliquerons dans ce document le concept et l'architecture de notre code ainsi que nos choix de structure.

2 L'architecture du code et les spécifications ajoutées autre que celles fournies dans le cahier de charge et leurs justifications

2.1 Analyse lexicale et syntaxique :

L'implémentation de l'analyse Lexicale et Syntaxique des programmes par le compilateur développé a été en général guidée aux petits détails par les consignes données par le prof et l'environnement déjà implémenté. Ainsi on expliquera brièvement la conception adoptée:

- Les Fichiers sources en ANTLR4 "DecaLexer.g4" et "DecaParser.g4" servent pour générer des reconnaisseurs de langages, en répartissant le travail. Le Lexer reconnaît les Tokens ou les "mots" utilisés dans le programme, alors que le Parser reconnaît plus l'organisation de ces mots comme implémentation de la grammaire concernée, et donc cherche plus à donner un sens syntaxique aux "phrases" du programme déca.... Ces deux fichiers servent par la suite à la génération du DecaParser et DecaLexer.
- Afin de l'implémentation réelle de l'arbre abstrait se fait à travers les classes du package "Tree", qui

essaie de maximiser l'héritage afin d'éviter les redondances, et regrouper les éléments de même unité de sens. Ainsi, on a essayé de garder la même logique même dans les classes qu'on a dû rajouter pour compléter ce dont on a besoin pour la partie Objet.

En ce qui concerne la décompilation, afin de l'implémenter, on n'a fait que de compléter les classes déjà existantes, afin de préciser à chacune entre elles comment elle devra se décompiler afin de donner du code deca correct et équivalent au programme précédent.

2.2 Analyse contextuelle

2.2.1 L'architecture du code :

A l'issue de la partie analyse lexicale et syntaxique, on obtient l'arbre abstrait correspondant au code source d'entrée. La partie de l'analyse contextuelle consiste à vérifier la sémantique de chaque nœud de l'arbre.

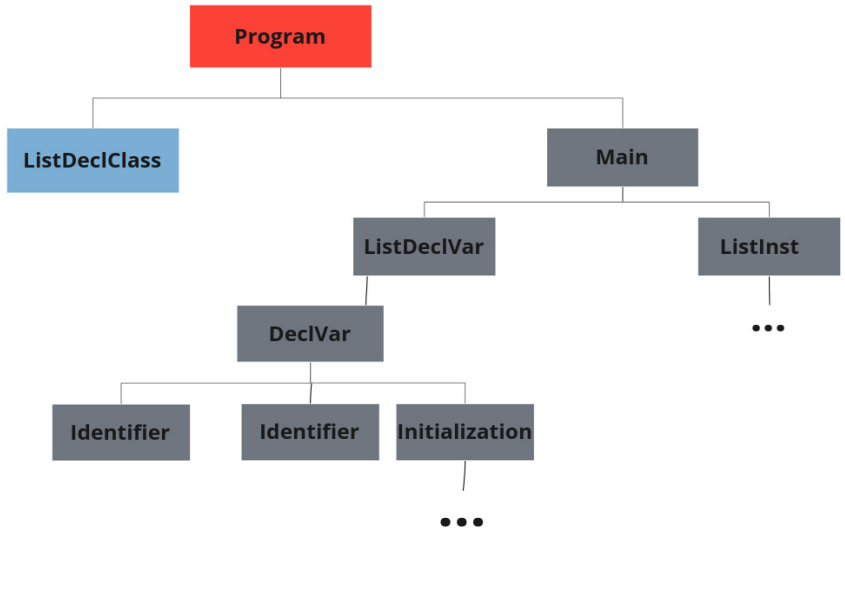

Figure 1: Exemple d'un arbre abstrait

Puisqu'il faut vérifier la sémantique pour chaque nœud possible qui existe dans un arbre d'un code syntaxiquement correct, alors pour être exhaustif il ne faut pas partir des arbres abstraits possibles mais des règles construisant ces arbres (grammaire abstraite).
Afin de permettre une future modification ou correction de notre code. Nous expliquerons l'architecture des

classes qui correspondent aux nœuds ainsi que la dépendance entre elles.

- Chaque non terminal de la grammaire abstraite correspond une classe abstraite par exemple :

  Program ⇒ AbstractProgram.java
  Main ⇒ AbstractMain
  DeclClass ⇒ AbstractDeclClass.java

- Si un non terminal dérive d'un autre non terminal alors la classe abstraite du premier doit étendre la classe abstraite du second :
  Exemple :
  INST →EXPR ≡ AbstractExpr.java étends AbstractInst.java

- Chaque terminal de la grammaire abstraite correspond à un nœud de l'arbre et à une classe concrète qui étend une classe abstraite parmi celles précédemment citées : Program.java , Main.java ... et on a codé la condition de la sémantique dans ces classes (pour une future modification) .

Ainsi l'architecture globale des fichiers est expliquée pour toute future modification ou amélioration .

2.2.1.1 EnvironmentTypes.java :

A ces classes nous avons ajouté la classe EnvironmentTypes.java dans laquelle nous avons défini tous les types prédéfinis : int, float, void, boolean, Object et à laquelle nous pouvons ajouter d'autres objets prédéfinis pour enrichir le langage Deca comme l'objet String qui permettra la déclaration de Strings. Cependant, cette classe contient également les classes déclarées par un programmeur Deca.

2.2.2 les choix de conception du code

Comme précisé dans le cahier des charges de la partie objet (sans rentrer dans les détails théoriques) nous avons dû empiler les environnements d'expression pour éviter les redéclarations et assurer l'appartenance d'un champ ou d'une méthode pour une classe donnée ...
Pour ce faire on avait deux approches différents :

2.2.2.1 La première approche :

La première approche consiste à appliquer l'empilement des environnement sur tous les champs et les méthodes d'une classe en une fois c'est à dire dès la création d'une classe on ajoute tous les champs et les méthodes de sa classe parent qui ne sont pas redéfinis et pour ces derniers on conserve la nouvelle définition mais avec l'ancien index pour les méthodes et on remet à jour les index des champs . A cet environnement on ajoute les méthodes et les champs qui sont définis pour la première fois dans la classe qu'on est en train de déclarer .

Par cette conception chaque classe déclarée doit avoir un environnement qui contient toutes les définitions de tous les champs et les méthodes de tous ses supers ainsi que les siennes .

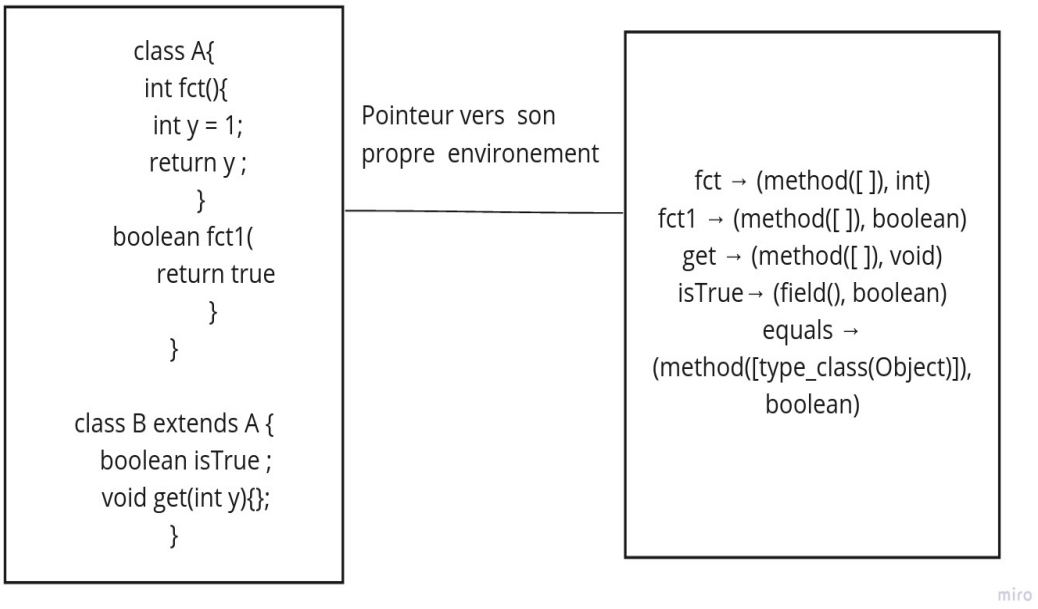

Figure 2: Exemple la première approche

Ce qui conduira à une mauvaise utilisation de la mémoire (trop de définitions à stocker) . Ceci nous a poussé à revoir ce choix d'implémentation .

2.2.2.2 La deuxième approche :

La deuxième approche part d'une exploitation intelligente de l'attribut parentEnvironment de la classe EnvironmentExp.java.

En fait, dès sa définition chaque classe possède un pointeur vers l'environnement de son parent . Ainsi, au lieu de définir son environnement en empilant son environnement avec l'environnement de son parent.
On ajoute dans son environnement juste les définitions des champs et des méthodes qui sont définies pour la première fois dans la classe (avec l'ancien index pour les méthodes).

Pour cela, on applique l'empilement à un seul élément en parcourant les pointeurs des environnements parents et on s'arrête au premier parent qui contient une définition du champ ou de la méthode en cas de redéfinition.

On du faire ce parcours aussi pour vérifier l'appartenance d'un champ ou d'une méthode à une classe donnée .

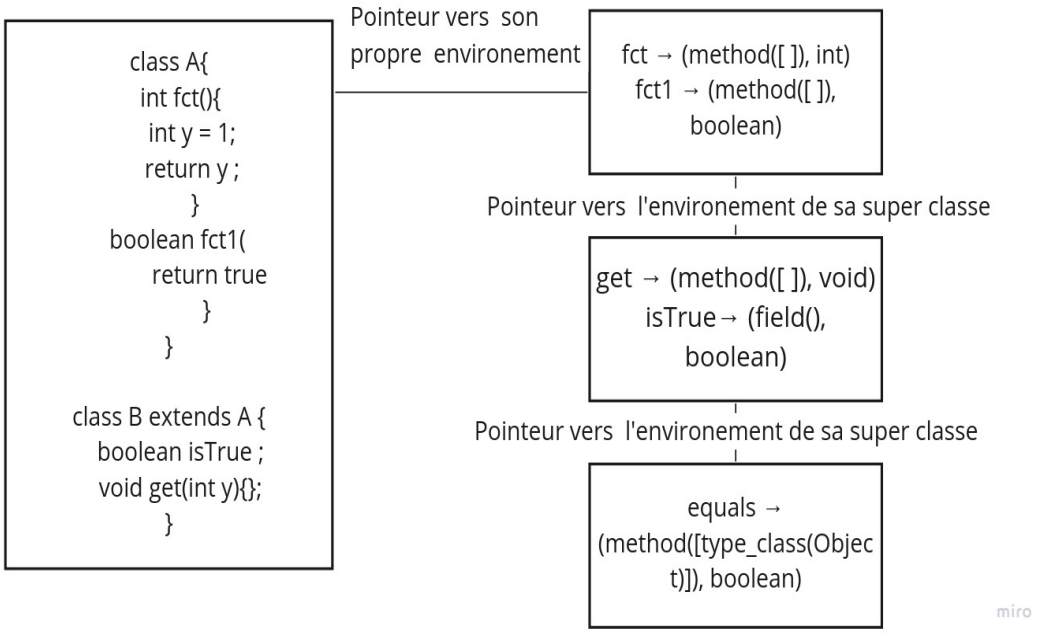

Figure 3: Exemple de la deuxième approche

2.2.2.3 La mise à jour des index : :

Pour éviter un parcours inutile afin de récupérer l'index de la dernière méthode déclarée dans la classe parent on a pensé à ajouter un attribut lastIndexMethod dans la définition de chaque classe . On fait de même pour l'indexation des champs (lastIndexField).

2.3 Génération du code assembleur

Lors de cette partie qui porte sur la génération du code qui sera par la suite écrit dans le fichier assembleur, on a dû faire quelques choix de conception qui nous ont permis d'établir une logique qui a fait office de pilier pour le code par la suite. Pour le commencement, lors de nos premier pas vers la génération des première ligne de code, on a remarqué que les instructions comme LOAD et STORE demandait souvent des attributs de Type DVal et Register, c'est ainsi qu'on a jugé judicieux d'introduire deux attributs de ces même types pour Les AbstractExpr afin de permettre à l'expression d'utiliser ce registre en cas de besoin et ensuite stocker le résultat dans sa DVal qui sera utiliser par la suite par d'autres instruction. Cependant, cela nous a confronté à une première problématique qui pose les questions suivantes: "qui donne les registres à qui? et qui a le droit de s'offrir un registre ?" . Répondre à ces deux questions semblait primordiale pour assurer la qualité du code assembleur. C'est ainsi qu'on a décidé d'appliquer cette règle qui s'est avérée fondamentale par la suite qui dit que c'est généralement l'instruction appelante qui donne un registre à ces fils et dans certains cas elle peut s'offrir un registre grâce au RegManager qui sera le registre descendant dans le nœud. Mais aussi, l'instruction qui s'alloue un registre libre, c.à.d qu'elle a appelé la fonction getFreeReg() du Regmanager, doit absolument le libérer à l'aide de la fonction freeReg() .

2.3.1 Implémentation du sans objet

2.3.1.1 Déclaration des variables :

Une variable déclarée fait appel à la fonction codeGenNewId(DecaCompiler compiler, boolean isLocal) pour s'allouer un registre global ou local selon la valeur du booléen isLocal et le mettre dans l'Opérande de la définition de son Identifier dont on stockera potentiellement par la suite une valeur calculer dans un registre selon son initialisation. Cette dernière, lorsqu'elle a une expression non nulle, alloue un registre à celle-ci dont lequel on va calculer sa valeur et par la suite servira de premier argument pour STORE.

2.3.1.2 Assign :

Assign est une instruction qui peut être appelé par d'autre instruction comme le if par exemple. C'est pour cela qu'elle ne s'offre un registre que si elle n'en a pas un. Ce registre servira par la suite au calcule de l'expression rightOperand et sera stocker dans la DAddr de l'expression leftOperand.

2.3.1.3 Opérations arithmétiques:

Dans un premier temps, On a décidé d'implémenter la partie commune du code aux opérations arithmétiques dans la fonction codeGenInst() du fichier AbstractOpArith.java et puis appeler celle-ci à partie de la fonction codeGenInst() de définit dans le fichier de l'opération souhaitée (Plus, Minus ...) grâce au mot clé super. Mais cette méthode s'est avéré non conforme lors de l'introduction de l'utilisation de la pile puisqu'elle était suceptible de génerer beaucoup de code redondant. Alors on a décidé de définir la fonction codeGen() dans chacun des fichiers des opérations élémentaires qui se charge de générer l'instruction adéquate vis à vis de l'opération souhaitée et l'introduire au corps de codeGenInst() chose qui s'est montrée plus pertinente pendant l'évolution de notre projet puisque les classes définies ne devront pas faire plus de ce que leur définition laisse penser.
La méthode codeGenInst() défini dans le fichier AbstractOpArith est la fonction principale qui permet de stocker dans le registre de l'expression la valeur calculée suivant l'algorithme détaillé dans le poly. Le corps de cette méthode contient alors une référence à la fonction codeGen().
On commence alors par calculer la valeur de l'expression leftOperand dans le registre alloué déjà à la AbstractBinaryExpr car on est sur que celle-ci a été appelé par une autre instruction qui lui a attribué ce registre, et donc il ne reste plus qu'a calculer la valeur et la stocker dans ce registre. Ensuite on vérifie si la DVal de l'expression rightOperand est null auquel cas où on est sûr que celle-ci à besoin d'un registre pour calculer sa valeur sinon cette dernière est connue et l'attribution d'un registre n'est pas nécessaire. Si on a besoin de donner un registrer à la rightOperand alors on vérifie si on a des registres disponibles dans le RegManager et pour lui en demander un et le libérer après utilisation, sinon l'utilisation de la pile est alors nécessaire.

2.3.1.4 Opération booléennes et structures de controles:

Une première approche pour implémenter les expressions booléennes et d'ailleurs celle qui figure dans le premier rendu ne s'est pas avérée efficace face aux tests plus compliqués, et on a pas tardé à comprendre que le problème était la structure d'implémentation même qui était défectueuse. Les premiers pas vers l'implémentation de la partie C ont été compliqués à coordonner. Ce qui a causé ce désordonnément du code qui faisait apparaître des répercussions non voulues sur le reste du code. C'est ainsi qu'on a décidé de repenser la structure des expressions booléennes vers une structure plus appropriée. Afin d'y remédier, on a choisi de reprendre le même raisonnement du poly et de le concrétiser dans un code en introduisant:

- La méthode codeGen (DecacCompiler compiler, boolean branch, Label label) définie pour les opérations booléennes a pour but de générer le code correspondant au branchement selon la valeur de branch à l'étiquette label.

- La fonction branchInst(boolean branch, Label label) définit dans les fichiers des opérations de comparaisons sert à générer l'instruction voulue selon la valeur de branch et est appelée par codeGen pour compléter le code correspondant à l'opération.
- la fonction codeGenInst définie dans les fichiers qui concerne les opération booléennes et est appelé seulement lors d'une opération de stockage d'un booléen dans une variable en faisant appelle à la fonction codeGenBool(DecacCompiler compiler, boolean branch, String label) servant à générer un branchement en utilisant codeGen() afin de calculer la valeur d'un booléen et la stocker par la suite dans le registre attribué à l'expression.

L'introduction de ces fonctions à permis une écriture beaucoup plus naturelle et similaire des algorithmes concernant le codage des structures de contrôles.

2.3.2 Implémentation de la partie avec objet

2.3.2.1 Construction de la table des méthodes

Pour le commencement, il était nécessaire de définir l'implémentation concrète de la Classe Object dans notre génération de code. On a alors décidé après multiples réflexions de ne pas introduire cette classe dans la liste de déclarations de classes mais la traiter telle une classe virtuelle dont on connaît la définition introduite comme attribut static dans la classe ClassDefiniton et définit pendant l'étape B pour établir les liaisons entre les classes et par la suite rendre cette définition accessible presque partout dans le projet. Quant aux définitions concrètes de cette classe et la méthode equals dans les fichiers assembleurs, ils ont été rajoutés manuellement pendant l'appel à la racine de codeGenInst() dans Program.java .
Pour les autres classes on a eu besoin d'introduire la fonction getMethods() dans EnvironmentExp qui renvoie une liste ordonnée des définitions des méthodes de la hiérarchie de cette classe selon les indexes réglés à l'étape B. Ensuite on attribuera des adresses à ces méthodes à partir de l'attribut GB du RegManager dans le corps de la fonction codeGenVtable() .

2.3.2.2 Définitions des méthodes

Pour arriver à implémenter correctement la vérification de la pile avec TSTO d et faire les sauvegardes pertinentes des registres qui seront utilisés par la suite dans le corps de la méthode, On a du faire un choix de conception très important qui s'impose sur le fait d'ajouter une instruction dans la liste chaînée des instructions du programme par l'intermédiaire d'une fonction insertInstruction(Instruction i, int j) introduite dans IMAProgram.java . Ensuite, pour calculer les registres utilisées on a rajouté un attribut regCounter qui nous permet de savoir le dernier registre utilisé lors des opérations effectués dans une méthode qu'on

pourra utilisé par la suite combiné avec la fonction d'insertion pour mettre les instruction assembleur TSTO, PUSH et POP dans leurs lignes adéquates.
Le processus des définitions des méthodes passe par l'enchaînement suivant :

- Appel de la fonction codeGenMethods() implémentée dans DeclClass.java qui aura pour but de définir les méthodes dans l'ordre correspondant .
- Appel de codeGenDeclMethod() qui saura exploiter l'insertion des instruction pour déclarer la méthode en sauvegardant les bons registres et génère les instructions du corps de la méthode avec codeGenMethodBody().

2.3.2.3 Déclaration des champs:

Pour la déclaration des champs, la génération de code se fait comme vous pouvez le prévoir, dans la fonction CodeGenDeclField de la Classe DeclField. On distingue dans cette génération de code entre deux cas principaux: quand la classe hérite directement de la classe Objet, et le cas où elle aurait un autre parent. Et parmi ce dernier cas, on distingue encore une fois entre deux situations vu qu'on est supposé déclarer les variables (surtout celles héritées) en deux passes ; une en prenant des valeurs par défaut, et l'autre pour leur assigner leurs vraies valeurs. Les valeurs par défaut sont bien sûr les mêmes que celles données dans les spécifications au poly (0 pour les entiers et les réels, false pour les booléens et null pour toute autre classe, et elles sont fournies par la fonction getDefaultValue.
On a aussi remarqué l'importance de faire la distinction entre les champs de types prédéfinis et ceux de types qu'on a pu définir à travers des classes qu'on a écrit dans le programme, dans le sens il fallait s'assurer de sauvegarder leur adresse spécifiquement dans la place mémoire correspondante à leur index.

2.3.2.4 Appels de méthodes :

Pour la génération du code de l'appel des méthodes, une problématique s'est imposée lors de l 'implémentation du codeGenInst() du Return. En effet, le return est appelé lorsque la méthode doit renvoyer une expression de type différent de void, mais alors celui-ci est supposé de faire un branchement à la fin de la méthode qui l'a appelé sachant que la classe Return.java ne dispose d'aucun indicateur sur la méthode appelantes et on ne pouvait pas introduire un attribut qui saura remplir cette tâche sans avoir à modifier à un nombre non négligeable de fonction. On a alors pensé à définir un attribut dans le LabelManager qui sauvegarde le dernier label de fin de méthode et ainsi on a résolu le problème.

2.3.2.5 Implémentation de Instanceof :

Afin d'implémenter le codeGen() de cette expression qui viendra compléter les expressions booléennes. On a pensé à l'implémenter comme une boucle de parcours des classes qui finira par faire un branchement vers une étiquette qui définit soit le succès soit l'échec de la recherche d'un lien de parenté entre les deux classes. On remarquera alors que le codage de cette de cette instruction est très inspiré du codage de la structure de contrôle While à quelques détails près afin de l'adapter à notre besoin.

2.3.3 Les Options du compilateur :

Les options du compilateur sont bien sûr intrinsèquement liées au comportement interne du compilateur, et donc elles était dans la plus grande partie implémentées dans les classes DecacCompiler, DecacMain et CompilerOptions.

- Dans le CompilerOptions, on s'est assuré de reconnaitre d'abord les options que l'utilisateur demande, de maintenir les conditions données par les spécifications du compilateur (par exemple empêcher l'utilisation simultanée de -p et -v , interdire l'utilisation d'autres options avec -b ou de lui donner un fichier comme paramètre....)
- Ensuite, dans DecacMain, on gère d'abord les cas où on n'aurait point à lancer la compilation, i.e. le comportement de l'option -b et de l'utilisation decac sans fichiers en paramètres. On gère aussi l'option du parallélisme. ⇒ Pour expliquer brièvement le travail fait pour l'option -P, on avait implémenté un "Callable" : TaskCompile dont la tâche principale est de lancer la compilation d'un fichier qui lui a été donné en paramètres. Et ces tâches sont données à un Executor Service (exc) qu'on avait créé dans la classe DecacMain qui prend n*7 threads, n étant le nombre de processeurs disponibles. (On a choisi de le multiplier par 7 car en travaillant sur une machine virtuelle, on avait en testant qu'un seul processeur disponible à chaque fois ce qui ne permettait pas d'avoir des résultats concrets).
- En outre, dans DecacCompiler, on avait contrôlé tout le comportement des options qui influence directement la compilation, comme par exemple les options -p et -v.
- Enfin, pour ce qui est de l'option -n, on l'avait plus géré dans toutes les parties de génération du code assembleur afin d'arrêter la vérification des erreurs spécifiées.

2.3.4 Implémentation des scripts de tests :

Afin d'assurer le bon fonctionnement de notre compilateur, on a pensé à écrire des scripts de tests qui servent à automatiser le processus de validation des différentes parties du projet. Ces scripts sont écrits

pour que chacun puisse tester une étape du projet. Ainsi, on garde une trace de l'avancement de notre compilateur.
En effet, on a implémenté les scripts lex-test.sh et synt-test.sh qui lancent que les tests vérifiant le fonctionnement de l'étape A qui concerne l'analyse lexicale et syntaxique. Après, pour l'analyse contextuelle on a réalisé un autre script context-test.sh et finalement gencode-test.sh qui lance les tests de la génération du code et les exécutent.
L'implémentation de ces scripts est réalisée dans un premier abord par l'écriture des tests dans leurs répertoires associés. Ces tests seront l'objet de l'examen que font les scripts. Ces derniers lancent les commandes shell désirées pour le test et en fonction de la sortie ils les valident (ou pas).
De plus, dans le fichier pom.xml, des lignes de codes ont été ajoutées pour lancer tous les scripts par une seule commande afin de vérifier le fonctionnement total des parties du projet.

2.3.5 Conclusion:

Le but de ce document reste de donner une vision claire et concrète sur la structure de notre code ainsi qu'une justification des choix de conception qu'on a fait .Ceci est dans le but de faciliter la lecture du code par un futur développeur visant une amélioration ou une éventuelle modification.
Donner ce regard sur notre code fait partie d'un exercice de professionnalisation qui nous a permis d'acquérir une compétence cruciale pour notre futur professionnel.