

Documentation de validation Projet GL

Groupe 44

COLIN-SOHY Mathéo

DIMITRIOU Tristan

EL HIT Oumayma

GAUX Antoine

HARRAUD Paul-Louis

Table des matières

1	Introduction	2
2	Tests	2
2.1	Arborescence	2
2.2	Stratégie d'écriture des tests Deca	2
2.3	Scripts	3
3	Gestion des Risques & Rendus	5
4	Couverture des tests via Jacoco	5
4.1	Couverture effective	5
4.2	Améliorer la couverture de test	7

1 Introduction

Ce document a pour but de présenter la stratégie de validation adoptée lors du projet GL. Il reviendra sur les différents tests mis en place pour vérifier la robustesse de notre compilateur. Il comportera les points suivants :

- Les tests réalisés
- L'organisation des tests
- La gestion des risques & rendus
- La couverture des tests (Jacoco)

Le but de ce projet GL est d'atteindre un compilateur parfait, en réduisant au maximum le nombre d'erreur. L'étape de validation est alors cruciale dans ce projet. L'objectif est alors de construire une base de test suffisamment importante pour essayer de simuler tout les cas possibles que l'utilisateur pourrait saisir. Il en va alors de soi que la base de test doit être construite de manière logique et efficace.

2 Tests

2.1 Arborescence

Les tests Deca ont été répartis selon l'étape testée.

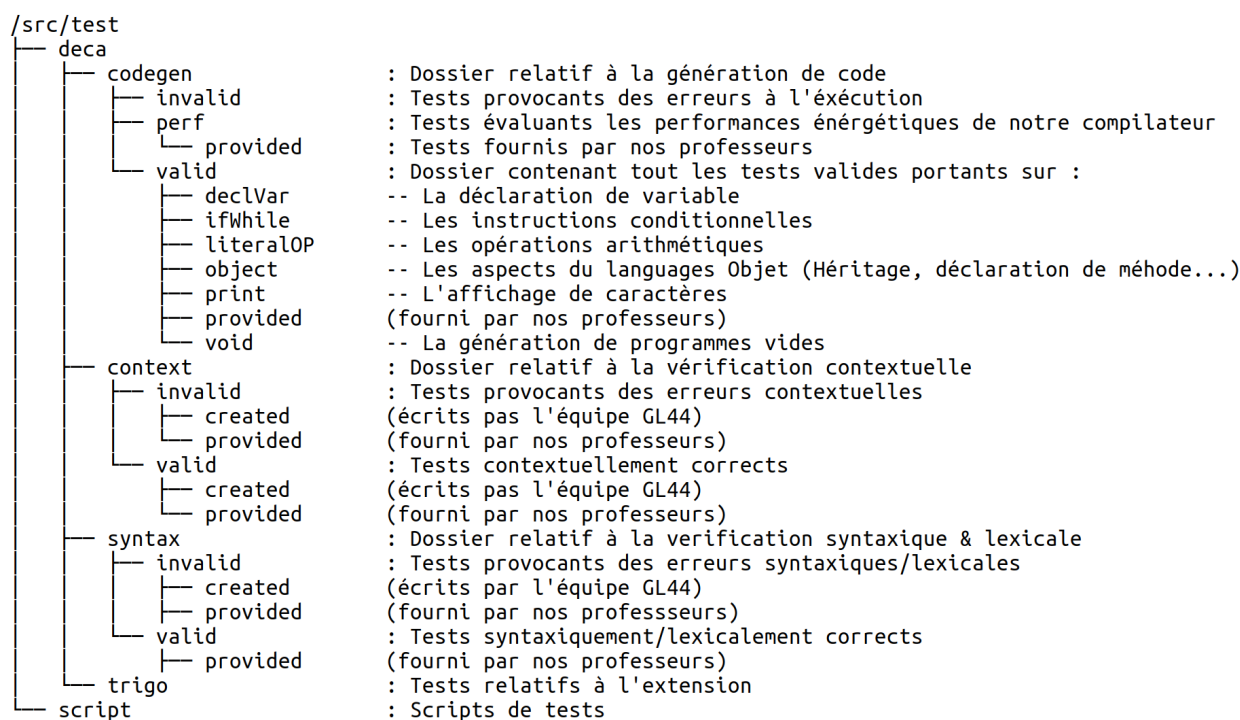


FIGURE 1 – Arborescence des tests

2.2 Stratégie d'écriture des tests Deca

Au total, la base de test comptabilise 446 tests écrits en Deca. Ils sont repartis différemment dans les dossiers valid & invalid de chaque partie.

Les tests de la partie A et B (context & syntax) ont des entêtes présentés comme ceci :

```

// Description :
// <Description du Test>
//
// Resultat :
// <Erreur> ou <Ok> ou <Resultat attendu>
  
```

Les tests de la partie C ont un en-tête un peu différent :

```
// <Resultat>
```

```
// <Description du test>
```

Cette modification est justifiée pour simplifier les lectures des résultats des programmes exécutables. En effet, on doit s'assurer que l'exécution via ima nous donne le bon résultat.

NB : Suite à un problème de merge, certains tests n'ont pas le bon format d'en-tête. Les tests sont alors considérés "faux" par le script qui les exécute alors que le résultat est bon.

Chaque tests erronés visent à vérifier un type d'erreur : Il ne possède qu'une erreur à chaque fois (Lexicale, syntaxique ou contextuelle). Les tests corrects cherchent à assurer la robustesse du compilateur en testant le plus de combinaison de code Deca possible. La création des tests était d'autant plus efficace en présentiel, car chaque membre de l'équipe posait des questions de manière naturelle "Et si...?". Ce genre de réflexions ont alors donner lieu à des tests ambigus, mais nécessaire pour s'assurer que notre compilateur respectait bien le langage Deca décrit par le cahier des charges.

Certains tests ont aussi été créés avec une complexité grandissante. Par exemple, les tests de la partie C de la série decl_classX permettaient de juger le travail restant sur la partie Objet. Le fichier decl_class1 déclare simplement une classe, et se complexifie au fur et à mesure des fichiers.

Exécuter ces tests un à un serai chronophage, c'est pourquoi nous avons écrit des scripts automatisant la tâche.

2.3 Scripts

Nous allons revenir sur les 5 principaux scripts de validation des tests. On estime que ces tests ont un aspect anti-régressif : A chaque fois que l'un d'eux est exécuté, il opère sur tout les tests concernés depuis le début du projet. Ainsi, si une modification viens perturber la sortie de l'un des tests, elle peut être identifiée puis résolue.

decompile_eq.sh :

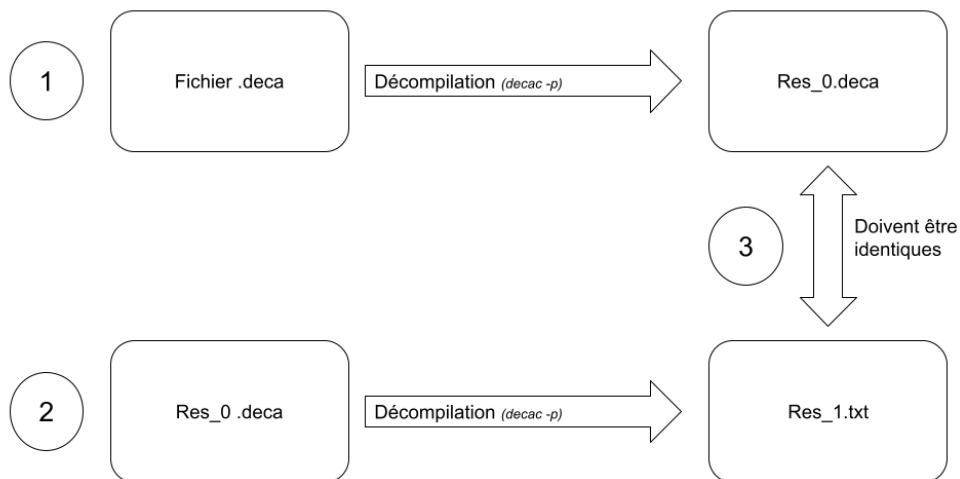


FIGURE 2 – Synoptique du script decompile_eq.sh

Le but de ce script est de vérifier l'étape A via la décompilation. On décompile et conserve une première fois un programme deca (reconstruction du programme à partir de l'arbre abstrait non décoré). Ce fichier est alors ensuite de nouveau décompilé, et sauvegardé dans un autre fichier. On compare ensuite les deux fichiers créés pour s'assurer que l'analyse syntaxique construit bien le même arbre abstrait pour deux programmes identiques. Cependant, le test de décompilation peut être comparé à la preuve par 9 : Un test avec un résultat faux implique forcément un programme faux (ou du moins une analyse syntaxique), mais un test avec un résultat juste ne

certifie pas l'absence d'erreur. Une erreur qui n'est pas reconnue lors de la première décompilation peut venir à disparaître dans le code généré dans `Res_0.deca`, et donc par défaut ne pas apparaître dans `Res_1.txt`. L'oracle de ce test n'est pas suffisant pour valider à 100% l'analyse lexico-syntaxique, mais permet de confirmer l'idempotence de la décompilation, et ainsi avoir une première idée sur la robustesse de la partie A.

gen_auto_context.sh & gen_auto_syntax.sh :

Ces deux scripts sont basés sur les scripts fournis par les professeurs *basic_synt.sh* & *basic_context.sh*, eux même reposant sur les launchers *test_synt* et *test_lex*. Ces deux scripts vérifient alors que les tests situés dans les dossiers `invalid` renvoient bien une erreur sous la forme suivante :

```
<nom_du_fichier>:ligne:colonne:<msg_erreur>
```

Ainsi, le script peut détecter les erreurs Java (et donc internes au compilateur, celles que l'on est censé éliminer). Cependant, ce script ne vérifie pas si les messages d'erreur sont conforme à ce qui est attendu. Pour ceci, il faudrait indiquer dans l'en-tête le message d'erreur attendu, et le comparer à la sortie.

A l'inverse, si un test situé dans un fichier `valid` renvoie une erreur, alors le test s'arrête et indique quelle test pose problème.

NB : Certains scripts possèdent des variantes. C'est le cas de `decac_v_context` qui est une variante de `gen_auto_context.sh`, qui ne s'arrête pas à la rencontre d'un test erroné, et qui test également contextuellement les tests de la partie C.

codegen-all.sh :

Ce script se charge de compiler les fichiers de la partie `codegen` et d'exécuter le fichier `ima` généré. Le résultat de l'exécution est alors ensuite comparé au résultat attendu indiqué dans le test. Si le résultat est non conforme à ce qui est attendu, le script affiche un message d'erreur. Une vérification des résultats un à un par un tiers est recommandé car certains formats de résultats peuvent différer de ce qui est attendu, sans pour autant être erronés.

basic-decac.sh :

Le rôle de ce script est de s'assurer du bon fonctionnement de `decac`, et de toutes les options que l'on peut lui fournir. Il vérifie alors que le passage d'options au compilateur ne produit pas d'erreur, ou bien au contraire qu'une levée d'erreur est bien présente dans le cas où ces options seraient mal utilisées.

Si l'on regarde de plus près le dossier *script*, on s'aperçoit qu'il y a encore d'autres scripts. Ce sont des versions moins optimisées des précédents, ou alors des variantes selon la manière de travailler adoptée par le membre de l'équipe. On peut également trouver des scripts écrits en Python qui servent à lire les fichiers plus facilement pour la comparaison des résultats.

On retrouve également les scripts basiques fournis avec le squelette initial, testant les différentes fonctionnalités de bases du compilateur, également incorporés dans la vérification automatique réalisée par Maven.

Finalement, tout les scripts présentés jusqu'à maintenant (mise à part *codegen-all.sh*) renvoient la valeur 1 lors de la détection d'une anomalie (Succès inattendu ou échec inattendu). La valeur 1 en shell a pour signification d'une erreur. Nous avons alors intégré ces scripts, via le fichier *pom.xml*, à la vérification automatique utilisée par Maven. Ainsi, l'utilisation de la commande *mvn verify* permet de lancer tout les tests, et stoppera en cas d'erreur dans l'un des tests.

3 Gestion des Risques & Rendus

Les risques à considérer avant/pendant un rendu ont été identifiés. Si l'équipe ne peut se réunir en présentiel lors du rendu, un responsable de rendu est nommé : Antoine. Nous avons incorporé au fichier *pom.xml* différents script de tests qui seront lancés via la commande (depuis la racine du projet) :

```
mvn verify
```

On surveille alors que les tests renvoient des résultats cohérents et qu'aucun script de test ne renvoie d'erreur. Notre stratégie est alors la suivante :

Risque	Conséquence	Action	Gravité (/5)
Louper la date de rendu	Branche Master avec une version/vidé	Chaque date de rendu à été notée dans le Planning. De plus, un événement à l'heure du rendu est mit en place sur Discord	5
Erreur(s) bloquant la compilation	Compilateur inutilisable	Le responsable de rendu effectue un clone sur un dossier différent de son répertoire de travail, afin de simuler un clonage du côté professeur/client. Il vérifie alors que la compilation fonctionne sans erreur et que le script de test général se déroule sans problème	5
Branche GIT de rendu qui n'est pas la dernière version	Manque de fonctionnalités ou Compilateur inutilisable	La branche master est réservée pour les rendus : Un seule fusion de branche par rendu. Si une fusion à eu lieu, c'est que la branche master possède la dernière version. Il suffira alors de d'assurer que cette fusion à eu lieu.	4
Dépendance(s) par rapport à des fichiers locaux	Certaines fonctionnalités peuvent manquer	Le responsable de rendu effectue un clone sur un dossier différent de son répertoire de travail, afin de simuler un clonage du côté professeur/client. Il vérifie alors que la compilation fonctionne sans erreur et que le script de test général se déroule sans problème	3
Print de debug qui n'ont pas été effacés	Messages intempestifs	On interdit l'utilisation du "System.out.println()" pour utiliser le Logger. Via le logger, on peut désactiver les messages affichés via la config du fichier log4j.properties. On est alors assuré qu'aucun message de debug ne sera affiché lors des différents tests/exécutions.	2
Non réussite d'un programme fourni	Fonctionnalité exigée non implémentée	Couverture par le script de test général qui est censé couvrir tous les tests.	2

FIGURE 3 – Gestion des risques

La vérification de ces différents points sera effectuée au minimum 2H avant le rendu final. Cependant, cette stratégie de repose sur une bonne base de test qui couvre le plus de code possible. Il faudra alors faire preuve de rigueur dans l'écriture des tests et leurs interprétations à l'écran.

NB : Suite a notre rendu final, où le problème est situé au niveau de l'extension, nous nous sommes rendu compte du manque de rigueur dans les tests. C'est un aspect à améliorer dans notre gestion de risques (cf. Bilan de projet 5.2) : Les tests doivent renvoyer des résultats plus facilement interprétable dans ce flot de donnée important.

4 Couverture des tests via Jacoco

4.1 Couverture effective

Via le squelette du projet GL, il nous a été fourni l'outil Jacoco qui permet d'évaluer la quantité de code couvert par les tests (en terme d'instructions, classes, méthodes, branches...). Le but fixé initialement par nos professeurs était une couverture de code de 75% (en terme d'instruction).

L'utilisation de Jacoco, une fois maîtrisée et comprise, s'avère être un outil très efficace pour construire une base de test. Pour le premier rendu intermédiaire, notre pourcentage de couverture de code était de 38%. Grâce à cette statistique, nous nous sommes rendu compte que la base de test n'était pas suffisante. Grâce à Jacoco, nous avons pu déterminer quelles fonctions des différents fichiers sources n'étaient pas exécutées, et donc construire des tests en conséquences.

L'utilisation de Jacoco via Maven est décrite selon le procédé suivant : (via la commande *mvn verify*)

- Etape 1 : Lancement du "monitoring" de Jacoco (via le plugin *jacoco-maven-plugin* :0.8.7 :instrument)
- Etape 2 : Déclenchement des tests .deca à l'aide des scripts de tests incorporés
- Etape 3 : Construction du rapport par Jacoco

Finalement, Jacoco nous fournit le rapport suivant :









Element	Missed Instructions	Cov.
fr.ensimag.deca.syntax		75 %
fr.ensimag.deca.tree		86 %
fr.ensimag.deca		81 %
fr.ensimag.ima.pseudocode		79 %
fr.ensimag.ima.pseudocode.instructions		63 %
fr.ensimag.deca.tools		58 %
fr.ensimag.deca.context		90 %
fr.ensimag.deca.codegen		86 %
Total	4 437 of 22 631	80 %

FIGURE 4 – Rapport général de Jacoco (Instructions)

On obtient une couverture générale de code de 80%. La génération de code et l'analyse contextuelle sont plutôt bien testés, alors que l'analyse syntaxique pêche un peu. Cependant, la couverture de code relative à l'analyse syntaxique s'appuie sur le code utilisé dans les fichiers *DecaParser.java* et *DecaLexer.java*. Ces fichiers sont générés à partir des fichiers antlr correspondant écrits en ANTLR4, et sont relativement imposants en terme de nombre de lignes. Le nombre d'instructions non exécutées peut alors rapidement augmenter. Par exemple, notre compilateur implémente le langage essentiel et non le langage complet. Il manque alors les tests relatifs aux instructions *instanceof* et *cast*, dont une partie est implémentée dans le Parser et Lexer.







Element	Missed Branches	Cov.
fr.ensimag.deca.syntax		53 %
fr.ensimag.deca.tree		84 %
fr.ensimag.deca		77 %
fr.ensimag.ima.pseudocode		75 %
fr.ensimag.ima.pseudocode.instructions		n/a
fr.ensimag.deca.tools		75 %
fr.ensimag.deca.context		89 %
fr.ensimag.deca.codegen		66 %
Total	527 of 1 479	64 %

FIGURE 5 – Rapport général de Jacoco (Branches)

Cette partie du rapport de Jacoco indique les branches (en fonctions des conditions booléennes présentes dans les tests) qui sont exécutées. On remarque que la moitié des branches syntaxiques ne sont pas testées. Ceci vient principalement des assertions du Parser qui vérifient si les non-terminaux des axiomes des grammaires ne sont pas null. Il est compliqué de générer des tests vérifiant la contradiction de ces assertions. Seulement la branche où le non-terminal n'est pas null est donc testé, d'où les 53% renvoyés par Jacoco (1 branche sur 2).

4.2 Améliorer la couverture de test

Nous sommes loin d'avoir suffisamment de tests dans notre projet. Il faudrait rajouter des cas supplémentaires pour l'appel de méthode, qui n'a pu être suffisamment testé. L'extension également qui ne possédait pas vraiment de tests incorporés comme les autres parties. Il s'agirait alors d'écrire des fichiers deca combinant les opérations trigonométrique. On pense également à un script qui ferait tourner en parallèle un programme python calculant les fonctions trigonométriques, et un programme deca avec une lecture sur stdin (via `readFloat()`) avec ces mêmes fonctions, et enfin comparer les résultats de manière automatique.

L'une des faiblesses de notre base de test est l'absence de test Java sur les méthodes "internes" au compilateur (par exemples celle liées à la table des symboles). Une utilisation de JUnit aurait alors pu être intéressante à la création de tests unitaires très spécifique. Les tests réalisés jusqu'à maintenant porte sur une partie entière du compilateur, voir sa globalité. Il est alors plus compliqué d'identifier un problème.