

# Projet Génie-Logiciel

## Documentation de validation

BECK BALIHAUT-UTARRE LOULA GILLET MAXIME  
GILLIOT MATHIS KRZISCH PIERRE-EMMANUEL TROMPETTE THOMAS  
ÉQUIPE 27

Janvier 2017

## 1 Introduction

Une part importante du projet GL consiste à s’assurer que l’implémentation du compilateur `decac` respecte fidèlement les fonctionnalités attendues. Ainsi, nous avons consacré une grande partie de notre travail à la validation de la qualité et de la robustesse du compilateur.

## 2 Descriptif des tests

Afin d’exhiber le plus grand nombre d’erreurs potentielles et de les corriger le plus rapidement possible, nous avons construit une base de tests conséquente pour chaque étape de l’implémentation. Pour cela, nous avons analysé les fonctionnalités attendues du compilateur ainsi que les erreurs éventuelles dans la réalisation du compilateur.

Nous avons gardé en tête que ces tests ont pour but de vérifier la validité du compilateur `decac` et pas de prouver que notre code fonctionne. Ainsi, pour concevoir nos programmes de test, nous nous sommes appuyés sur les règles de syntaxe du langage Deca et pas sur notre code.

### 2.1 Types de tests pour chaque étape/passe

Pour valider le compilateur `decac`, nous avons réalisé différents types de tests pour chacune des étapes de l’implémentation.

**Tests unitaires** Les tests unitaires permettent de tester une partie précise du code. Ils se sont révélés importants, notamment après la création de nouvelles classes. Ils ont été utiles pendant toute la durée de l’implémentation et ont été facilités par l’utilisation d’un logger.

**Tests système** Les tests système constituent la grande majorité de la base de tests que nous avons fournie. Ils permettent de vérifier que le bon résultat est obtenu ou que le bon message d'erreur est affiché. Ces tests ont été utilisés pour toutes les étapes de l'implémentation du compilateur.

**Gestion des exceptions** La gestion des exceptions a été testée en créant une base de tests invalides et en vérifiant que le programme est rejeté avec le bon message d'erreur.

**Tests de non-régression** Tout au long de nos différentes phases de test, nous avons stocké les résultats obtenus par nos tests système dans un dossier **references**. Cela nous a permis de comparer les résultats obtenus après une modification du code et de s'assurer que l'implémentation du compilateur est toujours conforme aux fonctionnalités attendues.

## 2.2 Organisation des tests

### Étape A

**Lexer** Le lexer sert à découper le code en mots qui seront analysés par la suite. À la sortie du lexer le code peut tout à fait être syntaxiquement faux. En effet, tant qu'il est lexicalement juste, le programme passe les tests du lexer.

Pour tester la reconnaissance de ces tokens, nous avons divisé nos tests en deux catégories : les tests valides et invalides.

Les tests valides vérifient que les tokens utilisés sont bien découpés et interprétés. Ainsi, pour chaque définition lexicale du langage Deca, un test est effectué. La base de tests valides vérifie donc comment le lexer interprète chaque unité lexicale et comment il gère les commentaires, les séparateurs et les inclusions de fichier.

Les tests invalides, eux, vérifient que lorsqu'un token inconnu est utilisé, le compilateur s'arrête et renvoie une erreur.

Ainsi, cette base de tests permet de mettre en évidence les problèmes lexicaux de façon précise et rapide.

**Parser** Le parser doit vérifier que le programme Deca est syntaxiquement correct. Une base de tests système a donc été conçue. Ces programmes sont testés avec le script `test_synt.sh`. Les programmes de tests valides affichent l'arbre de syntaxe abstraite alors que les tests invalides renvoient un message d'erreur correspondant à la règle de grammaire d'arbre qui n'est pas respectée

**Étape B** Lors de l'étape B, le compilateur doit s'assurer si le programme Deca est bien typé, c'est-à-dire s'il est conforme à la syntaxe contextuelle du langage Deca pour chaque grammaire attribuée des différentes passes.

Nous avons construit une base de programmes de tests système conséquente. Ces programmes sont testés avec le script `test_context.sh`. Nous avons donc conçu des programmes de tests valides (dans le répertoire `valid/`) pour lesquels on vérifie que l'arbre enrichi obtenu est conforme aux attentes, particulièrement au niveau des types et des définitions. Nous avons aussi réalisé des programmes de tests invalides (dans le répertoire `invalid/`) pour lesquels on vérifie que le programme Deca est rejeté avec le bon message d'erreur.

Lors de l'implémentation de cette étape, nous avons effectué un nombre important de tests unitaires pour vérifier, par exemple, si les environnements étaient correctement mis à jour.

**Étape C** L'étape C correspond à la génération du code. En principe, un programme qui arrive à cette partie est lexicalement et syntaxiquement correct car il aura passé les tests de l'étape A. Il aura de même passé les tests de l'étape B et aura donc vérifié que le programme est conforme à la syntaxe contextuelle du langage Deca pour chaque grammaire attribuée. Les tests à ce stade vérifient donc la sémantique du langage Deca. Autrement dit, ils vérifient que l'exécution s'arrête et provoque une erreur en cas de dépassement des limites de la machine support de l'exécution ou en cas d'erreur de la part de l'utilisateur lors de la lecture d'une valeur (on s'appuie sur la norme IEEE-754).

La base de tests est organisée de la même façon que pour les étapes précédentes, c'est-à-dire en deux dossiers : `valid/` et `invalid/`.

Les tests valides vérifient la bonne initialisation des variables et des champs, que l'instruction `new` initialise bien les objets après les avoir alloués, que l'instruction `return` arrête bien le corps d'une méthode, que les procédures d'affichage sont bien respectées et que les opérations arithmétiques et booléennes sont bien supportées (commutativité par exemple).

Les tests invalides vérifient que les débordements lors de l'évaluation des expressions sont bien gérés, et que lorsque les variables ne sont pas initialisées on renvoie bien une erreur.

Cette base de tests est très spécifique et prend donc du temps à être effectuée.

## 2.3 Objectifs des tests, comment ces objectifs ont été atteints

Au final, le but de de ces tests est de valider la qualité du compilateur en s'assurant de la robustesse du code. Nous nous sommes assurés de couvrir le plus grand nombre d'erreurs possibles. Cela nous a permis d'analyser et d'exhiber les erreurs qui peuvent apparaître dans notre code et de les éliminer.

Pour atteindre cet objectif, nous avons tâché d'agir le plus méthodiquement possible. Par exemple, pour les étapes A et B, nous avons construit au moins un programme de test valide et un programme de test invalide par règle de grammaire à vérifier. Pour l'étape C, au moins un programme de test valide et un programme de test invalide par opération ont été conçus.

### 3 Les scripts de tests

Notre but a été d'automatiser les tests de façon à pouvoir tout de même tester les parties souhaitées.

Pour chaque test que nous avons effectué, nous avons stocké le résultat obtenu dans un fichier `-lex`, `-synt`, `-context` ou `-codegen` dans un dossier `references/`. Cette référence correspond à ce que le test doit renvoyer et permet de faire des tests de non-régression. Les scripts de tests de non-régression réalisés parcourent le dossier des références, exécutent les programmes correspondant jusqu'au point attendu et vérifient que ce que le fichier `.deca` renvoie bien ce qui est attendu après l'étape souhaitée.

Nous avons implémenté huit scripts de tests.

**Scripts de tests de non-régression** `test_all_lex.sh`, `test_all_synt.sh` et `test_all_context.sh` qui testent tous les fichiers de références `-lex`, `-synt` et `-context` en les comparant au résultat du scripts `test_lex.sh`, `test_synt.sh` et `test_context.sh`. On vérifie ainsi la validité du lexer, du parser et des arbres de syntaxe abstraite décorés.

`test_all_codegen.sh` qui teste tous les fichiers de références `-codegen` en les comparant au résultat de l'exécution de la commande `ima` sur le fichier `.ass` généré par le compilateur `decac`.

Ces scripts de tests de non-régression affichent, pour chaque programme de tests, **Success** si le résultat obtenu est le même que le résultat attendu et **Failure** sinon. Dans ce cas, ils affichent ce qui était attendu et ce qui a été renvoyé.

**Scripts de tests système** `test_all_synt_sans_verif.sh` et `test_all_context_sans_verif.sh` qui exécutent les scripts `test_synt.sh` et `test_context.sh` sur tous les programmes de la base de tests correspondante et affichent le résultat obtenu. Ils permettent donc de visualiser les arbres et les erreurs renvoyées au fur et à mesure.

`test_all_option_decac.sh` qui exécute toutes les options implémentées du compilateur `decac` (sauf `-P`) et qui vérifie que le résultat attendu est le bon. Ce script affiche, pour chaque option valide **pas de problème** si le test renvoie le résultat attendu **Erreur** sinon.

`test_all_decompile.sh` qui teste tous les fichiers `-codegen` en les comparant au résultat du compilateur `decac` avec l'option `-p`. On vérifie ainsi la validité de la décompilation. Ce script affiche pour chaque programme du dossier `codegen/` **success** ou **Erreur**.

## Comment faire passer tous les tests

Nous n'avons pas implémenté de script automatisant l'exécution de tous les tests. L'utilité de ce script aurait été limité car les tests à effectuer sont spécifiques à chaque étape de l'implémentation. De plus, le temps d'exécution d'un tel script aurait été très important.

Ainsi, pour faire passer tous les tests au compilateur `decac`, il faut lancer un à un les scripts de tests ci-dessus.

## 4 Gestion des risques et gestion des rendus

### 4.1 Gestion des risques

Dans tout projet, il est important d'analyser les problèmes que l'on peut rencontrer et de mettre en place des actions pour les limiter ou les éviter. Ainsi, nous avons identifié ces risques selon 4 grandes catégories : les risques techniques, les risques humains, les risques liés aux délais et les risques intrinsèques à la gestion de projet.

Pour chacune de ces sections, nous avons identifié les différents risques qu'on peut rencontrer et les solutions pour les limiter.

Risques possibles	Actions pour limiter ou éviter les risques
Risques techniques	
Spécifications ambiguës	Ne pas hésiter à poser des questions
Rendu d'un programme qui ne compile pas	Tester la dernière version avant le rendu
Régression	Faire un test de régression Utiliser l'historique de Git
Problèmes de tests lors d'une démonstration	Faire les tests dans les mêmes conditions avant la démonstration
Oubli de l'ajout d'un fichier au dépôt	Utiliser git status Demander aux autres si tout fonctionne bien après un git pull
Ajout d'un programme qui ne compile pas sur le dépôt	Vérifier systématiquement au préalable que ça compile
Problèmes de connexion à distance	Installer le matériel sur sa machine personnelle
Risques humains	
Incompétence face à une tâche attribuée	Demander de l'aide aux autres membres ou à un professeur
Conflits au sein du groupe	Échanger calmement Être franc
Maladie, indisponibilité	Ne pas affecter une seule personne à la gestion d'une tâche

Problèmes de transport	Disposer de moyens pour travailler depuis chez soi
Risques sur les délais	
Mauvaise estimation de la durée nécessaire à l'exécution des tâches	Discuter de l'avancement du projet Revoir les priorités
Produit rendu non fonctionnel	Toujours disposer d'une version moins avancée mais fonctionnelle
Oubli d'une date	Utiliser un agenda
Risques intrinsèques à la gestion de projet	
Pas de connexion	Mettre régulièrement le dépôt à jour
Mauvaise affectation des rôles	Avoir une connaissance du projet dans sa globalité
Mauvaise implication dans le projet	Motiver les membres



## 4.2 Gestion des rendus

La gestion des rendus implique une organisation différente de celle de la gestion des risques. En effet, pour éviter tout conflit au sein du groupe nous avons listé quelques règles à respecter avant chaque rendu.

- Instaurer une heure butoire après laquelle on ne touche plus à la branche principale du dépôt,
- `git clone` de la dernière version du dépôt pour faire les tests pour avoir le même code que celui qui sera pull lors du rendu,
- Relancer l'ensemble des tests la dernière version du dépôt,
- Effectuer la check-list suivante avant l'heure butoire








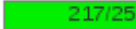












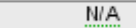
Pour le rendu du projet, la check-list était composée des tâches suivantes :

1. Mettre à jour le planning
2. Supprimer les fichiers avec tilde
3. Nettoyer les classes en trop et les commentaires
4. Faire passer le test `common-tests.sh`
5. Refaire tous les tests
6. Refaire les références des tests si besoin
7. Vérifier Cobertura
8. `mvn test`

## 5 Résultats de Cobertura

Pour vérifier la validité de notre couverture de test, nous avons utilisé l'outil mis à notre disposition : Cobertura.

Cet outil met en évidence les parties du code par lesquelles on ne passe pas lors de l'exécution. En utilisant Cobertura on a ainsi pu optimiser nos tests pour essayer de couvrir au maximum les lignes de code de notre compilateur. Au rendu, nous avons une couverture de test de l'ordre de 70%.

Package 	# Classes	Line Coverage		Branch Coverage		Complexity
<b>All Packages</b>	259	71%		57%		1.696
<a href="#">fr.ensimag.deca</a>	5	49%		19%		2.906
<a href="#">fr.ensimag.deca.codegen</a>	5	83%		75%		1.211
<a href="#">fr.ensimag.deca.context</a>	22	86%		81%		1.286
<a href="#">fr.ensimag.deca.syntax</a>	50	69%		55%		1.961
<a href="#">fr.ensimag.deca.tools</a>	4	44%		50%		2.067
<a href="#">fr.ensimag.deca.tree</a>	87	74%		63%		1.691
<a href="#">fr.ensimag.ima.pseudocode</a>	27	75%		80%		1.16
<a href="#">fr.ensimag.ima.pseudocode.instructions</a>	54	67%		N/A		1
<a href="#">fr.ensimag.ima.pseudocode.instructionsARM</a>	5	0%		N/A		1

Nous sommes cependant conscients de la limite d'utilisation de cet outil. En effet, il ne vérifie que les lignes de code parcourues et ne se pose pas la question de la validité de celles-ci. De plus, si une méthode n'est pas implémentée la couverture de test n'en est que meilleure, ce qui est très problématique.

Nous avons donc utilisé cet outil uniquement pour vérifier que les lignes de code qui nous intéressaient étaient bien utilisées, et donc testées. Le pourcentage de couverture n'est donc pas ce qui compte pour nous mais bien les lignes parcourues.

## 6 Méthodes de validation utilisées autres que le test

Faire des fichiers de tests au fur et à mesure de l'implémentation est nécessaire, cependant, ce n'est pas suffisant. En effet de nombreuses configurations de tests sont à penser et certaines d'entre-elles ne sont pas évidentes. Pour améliorer nos méthodes de validation, nous avons donc découvert l'outil Cobertura, dont les résultats sont explicités plus haut.

Une autre méthode de validation qui peut paraître subjective mais qui nous a apporté beaucoup est l'évaluation par les pairs. Cette méthode, bien que chronophage, apporte énormément pour la validation. En effet, cela permet de limiter le code obscur (noms de variables non explicites, cast, manque de commentaires, etc.) qui est généralement source d'erreurs. Pour mettre en place cette validation, la personne qui a codé doit être en mesure d'expliquer ce qu'il a implémenté et pourquoi. Ceci met souvent en lumière les incompréhensions ou les faiblesse du code.

Nous avons aussi fait la lecture du code assembleur généré pour vérifier que le celui-ci correspondait bien à celui attendu. Cette méthode a été utilisée pour régler des problèmes mineurs de génération de code et a donc permis de déboguer rapidement et efficacement l'étape C. Cette méthode a été utilisée de façon très ponctuelle mais a permis de mettre en lumière des soucis qui auraient été très difficiles à détecter autrement.