

Documentation de Validation du Compilateur Deca

Roboam Guillaume Raballand Cyprien Talbi El Mehdi
Charles-Mennier Matéo Altieri Aubin

GL1, GL01, Équipe 1
19 janvier 2025

Contents

1	Introduction	2
2	Descriptif des Tests	2
2.1	Types de Tests	2
2.2	Organisation des Tests	2
3	Tests Decac	3
3.1	Syntax	3
3.1.1	Lexer	3
3.1.2	Parser	3
3.2	Context	3
3.3	Codegen	3
3.3.1	Compilateur	3
3.3.2	Bytecode	4
3.4	Options Decac	4
4	Scripts de Tests	4
4.1	Présentation	4
4.2	Exemples	5
5	Gestion des rendus et des risques	5
5.1	Assurer la qualité technique du projet	5
5.2	Gestion des risques spécifiques	6
6	Coverage	6
6.1	SonarQube	6
6.2	JaCoCo	8
7	Méthodes de validation utilisées autres que le test	8
7.1	Tentatives	8
7.1.1	Fuzzing	8
7.1.2	Générateur de tests par IA	9

1 Introduction

Cette documentation de validation a pour objectif de détailler les processus et les méthodes mises en place pour assurer la qualité et la fiabilité du compilateur Deca. Elle a été conçue pour permettre à un utilisateur externe de comprendre et de reproduire les étapes de validation, ainsi qu'identifier des opportunités d'amélioration et d'extension des tests existants.

La conception d'un compilateur nécessite une validation rigoureuse pour garantir que chaque instruction fonctionne comme prévu. Cette documentation couvre les différents types de tests appliqués, l'organisation des tests, les objectifs poursuivis, et les moyens mis en œuvre pour les atteindre. Elle inclut également des instructions sur l'exécution des scripts de tests et des résultats obtenus via l'outil de couverture de code Jacoco et/ou SonarQube.

Enfin, cette documentation aborde la gestion des risques et des rendus durant la conception de ce compilateur et décrit les méthodes de validation complémentaires aux tests, utilisées pour renforcer la robustesse du compilateur.

2 Descriptif des Tests

2.1 Types de Tests

Nous avons utilisé deux types de tests pour notre compilateur :

- fichiers de test .deca
- Tests unitaires en JUnit

Les fichiers .deca sont majoritairement utilisés pour tester les différentes parties du compilateur, car ils permettent d'imaginer et de réaliser des tests complexes pour vérifier la robustesse du compilateur. Les tests doivent respecter une nomenclature spécifique vérifiée automatiquement selon l'étape.

Quelques tests avec JUnit ont été réalisés pour des comportements plus complexes à reproduire avec un fichier .deca. Cependant, JUnit est principalement utilisé pour le coverage.

2.2 Organisation des Tests

Les fichiers de test .deca constituent la grande majorité de notre base de tests. Ils sont répartis dans des dossiers organisés à partir du chemin `./src/test/deca` avec les sous-dossiers :

- `syntax`
- `context`
- `codegen`

Ces dossiers correspondent à chaque partie de l'exécution lors de la compilation d'un fichier (vérification syntaxique, contextuelle, puis génération de code).

3 Tests Decac

3.1 Syntax

3.1.1 Lexer

Les tests pour le lexeur sont situés dans `./src/test/deca/syntax/[valid | invalid]/created/lexer` et permettent de vérifier que le lexeur reconnaît les bons tokens et les associe correctement selon le fichier d'entrée.

Le script de test est `./src/test/script/basic-lex.sh`.

Peu de temps a été alloué pour le test de cette partie, vérifiant simplement la bonne compilation (ou non selon le type de test) du fichier `.deca` testé avec l'exécutable `test_lex` (`./src/test/script/launchers/test_lex`). Les tokens reconnus ou le message d'erreur retourné ne sont pas vérifiés.

3.1.2 Parser

Les tests du parseur sont situés dans `./src/test/deca/syntax/[valid | invalid]/created/parser` et visent à créer un premier arbre abstrait.

Le script de test du parseur est situé à `./src/test/script/basic-synt.sh`.

Comme le lexeur, le choix a été fait de se concentrer plus sur les tests end-to-end avec la partie Codegen. Les tests réalisés vérifient simplement la bonne exécution du launcher `test_synt` (`./src/test/script/launchers/test_synt`)

3.2 Context

Les tests de la vérification contextuelle sont situés dans `./src/test/deca/context/[valid | invalid]`. Ces tests vérifient la bonne exécution du launcher `test_context` sur nos fichiers de test.

En cas d'erreur, on vient vérifier ici que le message d'erreur retourné correspond à celui fourni dans l'en-tête du fichier test.

3.3 Codegen

Cette partie teste la compilation du fichier `.deca` et les sorties attendues. Elle sert de test end-to-end pour notre compilateur. Les fichiers de test pour la partie Codegen se trouvent dans `./src/test/deca/codegen/[valid | invalid | interactive | perf]`.

3.3.1 Compilateur

Les fichiers relatifs aux tests de la partie Codegen du compilateur sont dans `./src/test/deca/codegen[valid | invalid | interactive | perf]`

Les dossiers `valid` et `invalid` sont relatifs aux tests dont on attend (ou non) le succès de la compilation et les bons résultats en sortie. Le dossier `interactive` est garni de tests où l'on aura besoin d'input de la part de l'utilisateur.

Enfin, le dossier `perf` sert de benchmark pour mesurer les performances lors de l'exécution des

programmes avec la machine abstraite (`ima -s fichier.ass`)

On retrouve le script d'exécution ici : `./src/test/script/basic-gencode.sh`

3.3.2 Bytecode

On trouve un dossier `extension` dans `codegen`. Ce sont les mêmes tests que pour le compilateur mais la sortie attendue y est parfois changée pour s'adapter aux sorties de Java sur nos tests, différentes de celles d'IMA.

On lance ces tests via les tests de la commande `decac` présentée ci-après.

3.4 Options Decac

L'exécutable `decac` possède de nombreuses options (disponibles dans le Manuel-Utilisateur) que nous testons toutes via le script `./src/test/script/basic-decac.sh`

Afin de réaliser les tests, on se base en général sur les programmes `codegen` valides (pour tester notamment l'idempotence de la décompilation) Le script est séparé en beaucoup de sous-fonctions pour faciliter la maintenabilité et l'amélioration de ce dernier.

Ce dernier est aussi testé via le script `./src/test/script/common-test.sh` qui assure le bon fonctionnement de fonctionnalités de base du compilateur et de ses options.

4 Scripts de Tests

4.1 Présentation

Pour exécuter tous les tests, Maven est utilisé via le fichier `pom.xml`. Dedans, chaque script de test est renseigné par un bloc :

```
<execution>
  <id>SCRIPT</id>
  <configuration>
    <executable>./src/test/script/SCRIPT.sh</executable>
  </configuration>
  <phase>test</phase>
  <goals>
    <goal>exec</goal>
  </goals>
</execution>
```

Pour lancer tous les tests, la commande suivante est utilisée :

```
mvn clean verify
```

```

[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.825 s -- in fr.ensimag.deca.context.TestPlusPlain
[INFO] Running fr.ensimag.deca.context.TestPlusAdvanced
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.025 s -- in fr.ensimag.deca.context.TestPlusAdvanced
[INFO] Results:
[INFO] Tests run: 28, Failures: 0, Errors: 0, Skipped: 0
[INFO] --- exec-maven-plugin:3.1.0:exec (basic-lex) @ Deca ---

```

Figure 1: Tests JUnit

```

[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/IncompatibleUnaryMinusType.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/ReadFloatToInt.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/WrongTypeAssign.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/InvalidTypeWhile.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/InvalidTypeInIf.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/InvalidModulo.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/NoTypeVar.deca
[invalid] Test passed for src/test/deca/context/invalid/created/SansObjet/WrongTypeArithmeticExp.deca
[invalid] Test passed for src/test/deca/context/invalid/provided/affect-incompatible.deca
[INFO] --- exec-maven-plugin:3.1.0:exec (basic-gencode) @ Deca ---
Running basic-gencode tests...
[valid] Test passed for ./src/test/deca/codegen/valid/created/DoubleNegation.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/MultipleIfElse.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/LowerOrEquals.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/printCarriageReturn.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/LongArithmeticExpr.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/EmbeddedAssignment.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/bool.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/ExcessiveParenthesis.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/while5.deca
[valid] Test passed for ./src/test/deca/codegen/valid/created/AssignmentAndLogicalOp.deca

```

Figure 2: Test de context et de codegen avec les scripts sh

4.2 Exemples

5 Gestion des rendus et des risques

Pour garantir la réussite des livraisons et limiter les risques associés, les actions suivantes sont prévues :

- S'assurer que les échéances sont respectées et que les livrables sont remis à temps (des retards ont été observés précédemment).
- Effectuer un suivi rigoureux sur Trello, en cochant les tâches au fur et à mesure de leur réalisation.
- Organiser des revues d'équipe régulières pour discuter des sprints et des progrès réalisés.

5.1 Assurer la qualité technique du projet

- Vérifier que `mvn test` passe entièrement avant toute intégration.

- S'assurer du bon fonctionnement de la branche main, en vérifiant que toutes les branches ont bien été fusionnées d'abord dans develop, puis dans main.
- Maintenir un taux de couverture de code acceptable : 100% dans l'idéal, avec un minimum de 60%.

5.2 Gestion des risques spécifiques

- Risque de retard dans la livraison de l'extension :
Ce risque a été identifié car l'extension a été un peu délaissée au début du projet.
Une personne est désormais dédiée à plein temps à cette tâche pour rattraper le retard.
- Risque de confusion avec la multiplication des branches et le passage sur Git-Lab :
La gestion de nombreuses branches peut entraîner des erreurs ou des oublis lors des fusions.
Une documentation claire et un suivi rigoureux des branches sont nécessaires pour éviter ce problème

6 Coverage

6.1 SonarQube

Pour réaliser notre coverage et notre analyse de code afin de respecter au maximum le Clean Code, l'outil SonarQube a été utilisé.

Actuellement hébergé sur un Raspberry Pi model 4 chez Guillaume ROBOAM, membre du groupe, et disponible à l'URL : <http://82.65.7.115:9000/dashboard?id=ProjetGL> avec les identifiants suivants:

login : admin

mot de passe : ProjetGL38000!

Initialement sur GitHub, nous avons mis en place une Pipeline qui analysait notre qualité de code et notre coverage a chaque push sur les branches principales (master et develop).

Puis une fois repassé sur GitLab, suite à des problèmes avec les runners de l'Ensimag, nous avons abandonné cette pipeline pour privilégier un lancement manuel via la commande :

```
mvn clean &&
mvn -Djacoco.skip=false verify sonar:sonar
-Dsonar.projectKey=ProjetGL
-Dsonar.host.url=http://82.65.7.115:9000/
-Dsonar.token=squ_9cee7cad1bb29095d66a9b7dac00c7d1b902765b
```

Afin d'obtenir un coverage réaliste, le dossier ./src/main/java/fr/ensimag/ima a été volontairement ignoré sur SonarQube pour le calcul du coverage.

Le compilateur sans l'extension a pu être couvert à hauteur de **89.3%** en date du 18 Janvier.

Suite à l'ajout de l'extension (encore pas totalement fonctionnelle sur les tests initiaux du compilateur), nous arrivons pour l'instant à un coverage de **86.6%**.








Uncovered Lines 457 See history		New Code: Since January 9, 2025	
		Uncovered Lines	Line Coverage
 <code>src/main/java/fr/ensimag/deca/DecacCompiler.java</code>		45	65.6%
 <code>src/main/java/fr/ensimag/deca/tree/Identifier.java</code>		30	77.8%
 <code>src/main/java/fr/ensimag/deca/tree/MethodAsmBody.java</code>		30	30.2%
 <code>src/main/java/fr/ensimag/deca/tree/Assign.java</code>		29	66.7%
 <code>src/main/java/fr/ensimag/deca/tree/Cast.java</code>		24	61.9%
 <code>src/main/java/fr/ensimag/deca/tree/Instanceof.java</code>		20	81.3%
 <code>src/main/java/fr/ensimag/deca/tree/DeclField.java</code>		16	85.2%

Figure 3: Exemple d'analyse des fichiers les moins couverts

6.2 JaCoCo

JaCoCo est utilisé pour le rapport de couverture. La commande suivante est exécutée pour générer le rapport :

```
mvn -Djacoco.skip=false verify jacoco:report
```

7 Méthodes de validation utilisées autres que le test

En plus des tests unitaires et end-to-end, nous utilisons plusieurs outils pour valider la qualité du code dans notre projet. SonarQube est utilisé pour analyser statiquement le code, détecter les problèmes de complexité, de duplication et de sécurité, et appliquer des règles de Clean Code.

Les linters sont employés pour garantir la cohérence du code en imposant des conventions de style et en repérant les erreurs de syntaxe ou les mauvaises pratiques.

Ces outils permettent de maintenir un code propre, lisible et maintenable, tout en évitant les erreurs avant même qu'elles ne soient détectées par les tests.

7.1 Tentatives

7.1.1 Fuzzing

Afin d'assurer une robustesse sans égal de notre compilateur, nous nous sommes attachés à générer le plus de tests possibles afin de dénicher des bugs insoupçonnés. Une des méthodes utilisées dans l'industrie est le **fuzzing**. Cela consiste à générer aléatoirement des programmes (pendant une longue durée dû à la génération aléatoire) puis vérifier leur validité et leur compilation / exécution.

Afin de réduire le temps de calcul, nous avons utilisé un outil nommé [grammarinator](#). Cet outil est un générateur de tests basé sur la grammaire ANTLRv4, parfait pour notre compilateur. Pour que l'outil fonctionne correctement, il faut commenter la section **options** en haut des fichiers .g4. Après lui avoir passé les règles de notre langage Deca via la commande :

```
grammarinator-process DecaLexer.g4 DecaParser.g4 --no-actions
```

Nous obtenons un fichier `DecaGenerator.py` qui permet de générer des programmes Deca conformément à la grammaire fournie par la commande (10 tests):

```
grammarinator-generate DecaGenerator.DecaGenerator
-n 10 -d 10 -o tests/test_%d.deca
-s grammarinator.runtime.simple_space_serializer
--sys-path .
```

On peut régler la complexité du programme avec l'option `-d`. On obtient des programmes similaires à celui ci :

```
class _ { }
class $_d { }
```

Ou plus complexe comme :


```

class p extends $ {
  _ _ = readInt() == readInt() - !readFloat(),
  _ = readInt() instanceof _$h = readInt(),
  __ = readInt() || readFloat() == readInt() <= readFloat() - readInt()._
    = readInt(),
  __, $;
}
{
  _ c = readFloat() || readInt() && readInt() == readFloat() + readFloat
    () % readInt()._9 = readInt();
  _$$ A = readFloat() && readInt() != -readFloat() = readFloat();
}

```

Nous n'avons pas exploité plus l'outil car manquant de temps et trouvant difficile de générer des programmes syntaxiquement, contextuellement correctes, qui compilent bien et qui révèle des erreurs dans le code.

7.1.2 Générateur de tests par IA

Afin de résoudre le problème du fuzzing, qui avait du mal à générer des programmes contextuellement correct n'ayant que la grammaire du programme à disposition, après avoir lu ce résumé de conférence [ALPHAPROG: Reinforcement Generation of Valid Programs for Compiler Fuzzing](#) nous avons essayé d'entraîner une simple IA comme décrite dans le papier.

L'IA est composée d'un premier layer LSTM avec 128 neurones suivi de 2 couches cachées de 100 et 512 neurones. nous n'avons pas réussi à générer des programmes plus complexes avec cette solution

En lui spécifiant certains tokens valides en entrée comme le vocabulaire que l'IA pouvait utiliser, nous avons essayé de générer des programmes très simples comme

```
{ int x; }
```

Mais sans succès, par manque de temps de développement et temps d'entraînement. Nous pensons que cela reste une très bonne solution pour générer des jeux de tests automatiquement.