

PROJET GÉNIE LOGICIEL



---

## Documentation de Validation

---

Aurélien VILMINOT  
Damien CLAUZON  
Guilherme KLEIN  
Léon ROUSSEL  
Pierre ARVY

26 janvier 2022

## Résumé

Ces quelques pages résument et expliquent la validation de notre compilateur. Il décrit les différents types de tests implémentés, leurs objectifs, les résultats obtenus (à chaque étape, couverture Jacoco...) ainsi que leur organisation. L'automatisation de nos tests, principalement gérée par des scripts shell, est expliquée. Enfin, on retrouve quelques mots la gestion des risques, des rendus et sur les différentes méthodes de validation utilisées.

## Table des matières

<b>1</b>	<b>Organisation des tests</b>	<b>2</b>
<b>2</b>	<b>Validation du compilateur</b>	<b>3</b>
2.1	Types de tests . . . . .	3
2.1.1	Boîte-noire . . . . .	3
2.1.2	Unitaires . . . . .	3
2.1.3	Conformité . . . . .	3
2.2	Résultats . . . . .	4
<b>3</b>	<b>Automatisation</b>	<b>4</b>
3.1	Scripts shell . . . . .	4
3.1.1	Gestion des différents types de tests . . . . .	4
3.1.2	Fonction principale . . . . .	5
3.1.3	Scripts finaux . . . . .	5
3.1.4	Commande decac . . . . .	6
3.2	Maven . . . . .	6
3.3	JUnit . . . . .	6
<b>4</b>	<b>Gestion des risques</b>	<b>6</b>
4.1	Motivations . . . . .	6
4.2	Date de rendu . . . . .	6
4.3	Erreurs triviales . . . . .	7
4.4	Git . . . . .	7
4.5	Tests . . . . .	7
4.6	Méthodes générales . . . . .	7
<b>5</b>	<b>Gestion des rendus</b>	<b>8</b>
5.1	Motivations . . . . .	8
5.2	Mesure de prévention . . . . .	8
5.3	Liste des actions à effectuer . . . . .	8
<b>6</b>	<b>Jacoco</b>	<b>9</b>

# 1 Organisation des tests

L'ensemble des fichiers relatifs à la validation du compilateur se trouvent dans le répertoire `/src/test/`. Jusqu'à la partie 2, on considère que le répertoire courant est ce dernier (figure 1). On distingue alors plusieurs catégories de tests, et qui seront expliquées dans les pages qui suivent. Le nom de chaque fichier de test décrit au maximum l'élément testé. Par exemple, le fichier `binary_op_eq_with_two_class.deca` teste l'égalité (opération binaire) entre deux classes.

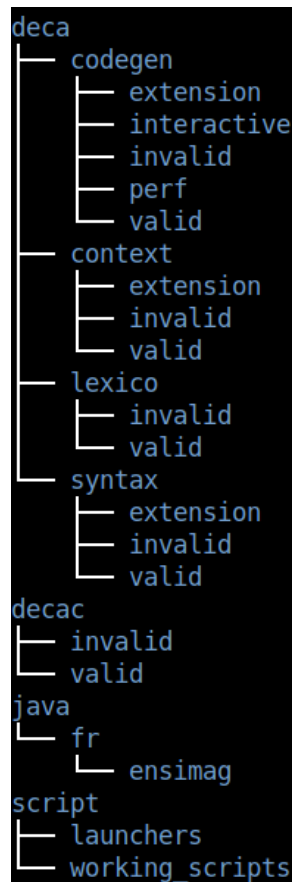


FIGURE 1 – Arborescence des tests.

Dans le répertoire `script` de la figure 1 se trouvent tous les scripts shell utilisés pour automatiser la validation et intégrer les tests à l'outil d'automatisation Maven. Ceux-ci seront décrits partie 3.

Dans `deca` se trouvent tous les tests relatifs aux différentes parties de la compilation : *lexer*, *parser*, *analyse contextuelle* et *génération de code*. Les tests sont donc divisés dans quatre répertoires distincts (`lexico` a été ajouté à l'arborescence initiale). Dans ces derniers, on retrouve une séparation entre les fichiers valides, invalides et relatifs à l'extension. A noter que certains tests se situent dans un répertoire `not_implemented`. Ils correspondent à des parties non développées du compilateur. Ces tests ont été mis à part pour ne pas déranger l'exécution automatique des autres tests. Les scripts `test-lex.sh`, `test-pars.sh`, `test-context.sh` et `test-codegen.sh` permettent de lancer tous les tests relatifs à une certaines parties.

Dans le répertoire `decac` se trouvent quelques tests utilisés pour valider les différentes options du compilateur. Les scripts `test-decac.sh` et `test-decac2.sh` permettent de tester automatiquement l'ensemble du compilateur sur ces tests.

Dans le répertoire `java/fr/ensimag/deca` se trouvent tous les tests unitaires, divisés selon les classes Java concernées (la division est la même que dans le répertoire de développement `main`). Utiliser Jacoco permet de les lancer et d'analyser la couverture de ceux-ci. Cela sera abordé dans la partie 5 de cette documentation.

## 2 Validation du compilateur

L'écriture des tests s'est, pour chaque partie, faite en deux temps. Pour commencer, les testeurs, "candide" vis-à-vis de la partie testée, développaient des tests unitaires, "boîte-noire" et de conformité les larges possibles de manière à tester le fonctionnement globale de la partie concernée. Après développement, et validation des tests précédents, les testeurs écrivaient le plus de tests de conformité (passés en "oracle") possible de manière à tester tous les détails et à détecter le maximum d'erreurs du développement. Juste avant de livrer une fonctionnalité, la dernière étape consistait à écrire des tests systèmes, simulant ce que pourrait être amené à faire un utilisateur, et vérifier la bonne compilation et exécution de ceux-ci. La suite de cette partie explique les différents types de tests mis en place, et leurs résultats.

### 2.1 Types de tests

#### 2.1.1 Boîte-noire

Les tests dits "boîte-noire" étaient, pour chaque partie, les premiers à être écrits. Ceux-ci sont basés sur les spécifications du programme. Chaque test est composé d'un fichier au format `.deca`, testant une certaine fonctionnalité, et d'un fichier au format `.txt` situé dans le même répertoire, contenant le résultat attendu. L'objectif est de comparer la sortie associée au fichier `.deca` avec le résultat attendu (commande `diff` en `shell`), et d'en déduire si le test est validé ou non. Pour tester la *génération de code*, ces tests étaient les plus utilisés. En effet, ils permettent de vérifier la bonne compilation d'un programme Deca, et que l'exécution se déroule comme prévue. Par exemple, le test `class_maj_and_min.deca` teste s'il est possible de définir deux classes avec le même nom qui ne diffère que par une majuscule. Le fichier `class_maj_and_min.txt` comporte le résultat attendu, à savoir l'affichage d'arguments de ces deux classes.

#### 2.1.2 Unitaires

Les tests unitaires ont été écrits pour assurer la couverture du code, principalement sur les méthodes de la partie B, dans les classes des dossiers `tree` et `context`, mais aussi concernant la partie C, dans des classes telles que `DecacCompiler` et `LabelGenerator`. Pour atteindre cet objectif, le rapport Jacoco a été utilisé comme référence pour créer les tests, en indiquant le code pas encore couvert. Chaque fois qu'une fonctionnalité non testée était trouvée, un test était ajouté dans la classe de test correspondante, en comparant le résultat donné par le programme avec celui attendu conformément à la spécification, en utilisant les méthodes d'assertion de JUnit et parfois en utilisant Mockito pour éviter les instances complexes inutiles.

Ces tests ont été utiles pour le développement, en particulier pour trouver des erreurs qui étaient syntaxiquement corrects, mais influent sur le résultat, comme des arguments du même type dans le mauvais ordre dans un appel de méthode.

#### 2.1.3 Conformité

Les tests de conformité sont presque tous testés en "oracle". Les sorties ne sont donc pas étudiées. Il suffit qu'un test de conformité situé dans un dossier `valid` ne génère pas d'erreur pour être considéré comme valide, et inversement pour les tests situés dans les dossiers `invalid`. Les tests de conformité ont été écrits en parallèle du développement de chaque partie. Ceux-ci visent à valider l'ensemble des spécifications du polycopié, et à rejeter toutes celles qui ne le seraient pas. Par exemple, on retrouve énormément de tests de conformité pour tester le parser (partie A du développement). L'objectif des tests valides est de s'assurer que notre compilateur reconnaisse au moins la syntaxe concrète définie dans le polycopié. Les tests invalides visent à s'assurer que le compilateur reconnaisse théoriquement au plus celle-ci. En effet, cela n'est pas possible avec un nombre fini de tests. Ces tests se font en admettant que la syntaxe pouvant être reconnue est syntaxiquement cohérente.

## 2.2 Résultats

Ces différents types de tests ont tous le même but : savoir si le compilateur développé correspond aux spécifications. Les tests systèmes donneront plus d'assurance mais seront plus long à écrire, les tests de conformité sont beaucoup plus courts à écrire, il est possible d'en faire beaucoup, mais chaque test donne qu'une petite information de validité.

Il est tout de même bon de rappeler que chaque type de test a sa place dans le processus de validation, ils ont tous leurs avantages.

En testant la validation, chaque test aide à "resserrer l'étau" autour du compilateur qui correspond à celui attendu par les spécifications. La figure 2 montre ce processus. Les tests valides représentés en vert et les tests invalides représentés en rouge tentent de venir épouser les spécifications du compilateur représentées en noir. Ainsi, plus la zone blanche se réduit, plus le compilateur que nous créons, qui passent effectivement ces tests, est assuré d'être proche du compilateur donné dans les spécifications. Le but des testeurs est donc de réduire au maximum cette zone d'inconnu en blanc. En effet, dans cette zone, il n'y a aucune garantie sur le fonctionnement de notre compilateur.



FIGURE 2 – Principe de "resserrer l'étau" pour avoir un compilateur qui correspond au maximum aux spécifications du cahier des charges

Pour ces raisons, notre base de tests est plutôt fournie et compte plusieurs centaines de tests. Avec une base de test exhaustive, le compilateur répondra aux spécifications. L'enjeu est donc de couvrir le plus de cas différents possibles pour assurer la validité du compilateur. Un compilateur qui n'est pas testé a peu de chance d'être juste. L'un ne va pas sans l'autre, les tests et le développement sont complémentaires.

## 3 Automatisation

Exécuter les tests un par un peut vite s'avérer être chronophage. D'autant plus qu'à chaque ajout de fonctionnalité, pour s'assurer qu'il n'y a pas de régression, il est préférable de relancer un groupe de tests. L'automatisation de tests est donc indispensable pour gagner du temps et connaître la nature des erreurs de développement.

### 3.1 Scripts shell

Afin de lancer des tests sur plusieurs fichiers d'extension `.deca` en même temps, l'utilisation de scripts Shell est une technique efficace. Le langage permet de lancer des commandes très facilement et les redirections à l'entrée et à la sortie sont très simple à effectuer.

#### 3.1.1 Gestion des différents types de tests

Pour tous les tests de conformités, il s'agit de créer un fichier d'extension `.deca` et de lui faire passer différentes étapes du processus de compilation (lexer, parser, vérification contextuelle, ...). Pour savoir si un test est passé ou non, il faut définir des seuils qui définissent la validation d'un test ou non. Il est possible de définir différents seuils, ce qui est détaillé dans la partie sur les différents types de tests (cf. 2.1). C'est à l'aide des scripts Shell que nous

mettons en oeuvre le type de vérification associé à chaque test. Pour les tests qui doivent uniquement ne pas lever d'erreur, une commande Shell s'assure uniquement que la sortie d'erreur ne contient aucun caractère. Pour des tests où l'on attend la réponse exacte, une commande compare le résultat de la sortie standard avec le contenu attendu. Ces fonctions qui traitent différemment la validation d'un test sont écrites en bas du fichier suivant :

```
script/working_scripts/main-functions.sh
```

Ces fonctions aident non-seulement à créer différents types de validation, mais elles permettent aussi d'être réutilisées pour les différentes parties du processus de validation.

### 3.1.2 Fonction principale

La fonction majeure utilisée pour la plupart des tests est également situé dans le fichier suivant :

```
script/working_scripts/main-functions.sh et elle se nomme exec_test_from_dir
```

Elle permet de répertorier tous les cas possibles pour chaque partie. En créant une telle fonction, il devient facile d'ajouter de nouvelles phases de tests.

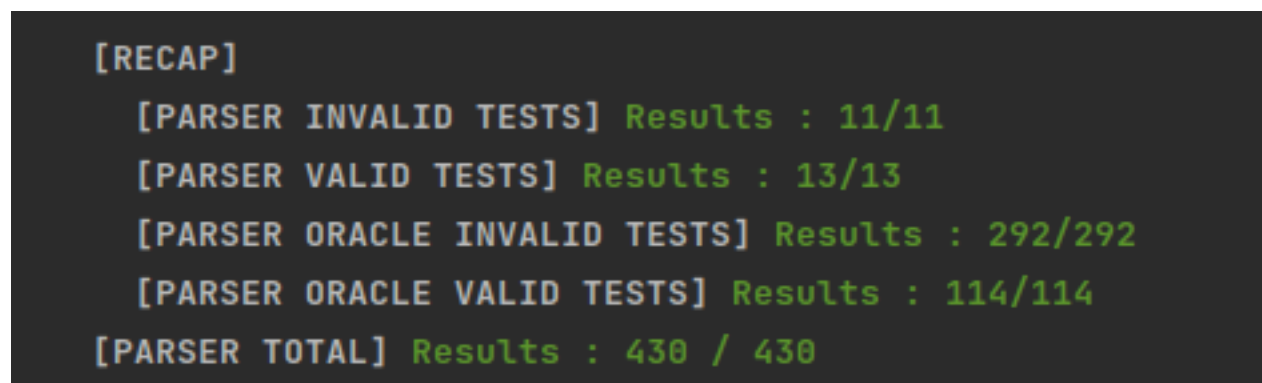
Les scripts dits `launchers` sont utilisés à cet endroit. On y retrouve donc 4 principales parties : LEXER, PARSER, CONTEXT, CODEGEN. Ce sont les principales étapes du processus de compilation.

### 3.1.3 Scripts finaux

Pour que les utilisateurs aient plus d'aisance avec l'exécution des tests et pour des raisons de consommation énergétique, des scripts individuels sont créés : `test-lex.sh`, `test-pars.sh` et `test-context.sh`, `test-codegen.sh`. En lançant ces scripts, 4 catégories de tests sont généralement lancées (valid/invalid et oracle/boîte noire).

Un affichage a été implémenté afin de rendre les résultats plus lisibles. L'ergonomie est bien meilleure et cela permet de repérer les problèmes beaucoup plus rapidement. À l'issue des tests appartenant à une partie, un récapitulatif apparaît. La couleur du texte annonce la réussite ou non de tous les tests.

Pour les tests ayant échoué, selon la nature du test, il se peut qu'un fichier d'extension `.log`, `.lis` ou `.res` se trouve dans l'arborescence à côté du fichier d'extension `.deca`. Ces fichiers contiennent le résultat produit par la compilation ou l'exécution du fichier d'extension `.deca`. Ils donnent donc une information sur la nature de la différence qu'il y a entre le résultat généré et le résultat attendu. Ainsi, les erreurs se repèrent plus facilement. Un testeur peut donc rapidement, et avec précision, avertir le développeur de son erreur.



```
[RECAP]
[PARSER INVALID TESTS] Results : 11/11
[PARSER VALID TESTS] Results : 13/13
[PARSER ORACLE INVALID TESTS] Results : 292/292
[PARSER ORACLE VALID TESTS] Results : 114/114
[PARSER TOTAL] Results : 430 / 430
```

FIGURE 3 – Récapitulatif des tests issus de la partie sur le parser

À l'aide de cette construction fonctions par fonctions, il devient alors aisé de créer rapidement d'autres scripts pour d'autres phases de tests. Un script de tests pour la validation de l'extension se crée donc rapidement, il se nomme `test-ext.sh`.

Cette faculté à créer des programmes de validation rapidement s'est également illustrée avec le test de la partie 'avec objet'. Des fichiers de tests sont créés rapidement puis supprimés pour tester de petites fonctionnalités.

### 3.1.4 Commande `decac`

Le test de la commande `decac` est crucial, car l'oubli d'une option se remarque immédiatement. Afin d'être sûr que les options fonctionnent et pour consolider le test `common-tests.sh`, nous avons créé deux tests supplémentaires (`test-decac.sh` et `test-decac2.sh`).

## 3.2 Maven

L'outil Maven a été très utile pour compiler nos fichiers sources, mais aussi pour réaliser les tests. En effet, l'ajout au fichier `pom.xml` des scripts Shell présentés plus haut permet d'effectuer tous nos tests lorsque les commandes `mvn test` ou `mvn verify` sont lancées. Ainsi, il est possible de s'assurer que tous les tests passent avant de déployer une version finale.

## 3.3 JUnit

Les tests unitaires, tous faits avec JUnit, ont été tous ajoutés aux classes nommées `Test<classe testée>` ou `<classe testée>Test` avec le décorateur `@Test`, ce qui fait que les tests soient exécutés automatiquement avec la commande `mvn test`. Un autre décorateur utile utilisé était `@BeforeEach`, pour appeler une méthode de configuration avant chaque test qui instancie toutes les variables communes pour les entrées de test. De cette façon, nous avons pu éliminer de nombreuses redondances de code, ce qui rend les tests plus propres. Tout ce qui reste à faire dans la méthode de test elle-même c'est configurer les variables particulières et d'appeler une méthode `assert` de la bibliothèque JUnit qui contrôle le succès ou l'échec du test, soit pour s'assurer que le résultat obtenu était le même que prévu (`assertEquals`, `assertTrue`), soit pour s'assurer qu'une entrée invalide lance une exception (`assertThrows`).

# 4 Gestion des risques

## 4.1 Motivations

Lors du développement d'un projet logiciel, la gestion des risques est cruciale car ces derniers sont présents à chaque étape, de la conception à l'implémentation. Par exemple, l'outil de versionnage permet de prévenir des risques potentiels de perte de données. Son utilisation doit néanmoins être maîtrisée. La gestion des risques vise à prévoir ces potentielles erreurs, et, en conséquence, s'appropriier les outils nécessaires pour les résoudre.

## 4.2 Date de rendu

Le client exige des dates de rendu. Il faut donc être sûr de pouvoir rendre le travail à temps. Pour cela, nous avons mis en place une liste des tâches qui nous permet de savoir quelles parties nous devons implémenter et tester avant la fin d'un sprint (et donc avant un rendu dans le cadre du projet).

Pour avoir davantage de sécurité, des alarmes sont déclenchées depuis Trello pour nous alerter qu'une tâche est à finir avant une certaine date. Par exemple, avant un suivi ou un rendu, une notification est envoyée à chaque membre de l'équipe pour l'informer de l'événement. L'équipe peut ainsi s'assurer que tous les documents sont rendus à temps.

Nous choisissons de diviser les fonctionnalités du compilateur. Cela est, en effet, un des objectifs de la méthode SCRUM. En développant les fonctionnalités les unes après les autres, par exemple après une rencontre avec le client, nous pouvons lui présenter de nouvelles fonctions à chaque fois. Le client peut ainsi constater les avancements, et mieux se projeter sur la réalisation du produit. Cela diminue aussi les risques concernant les attentes du client. Si une partie du projet ne convient pas, nous pouvons réagir en conséquence dans le sprint suivant.

Ce qui est décrit précédemment est un des principaux avantages de la méthode SCRUM. En travaillant avec une méthode plus classique, le risque est de passer beaucoup de temps à développer un ensemble de fonctionnalités en parallèle, et de se rendre compte à la fin que le rendu ne correspond pas aux attentes du client.

### 4.3 Erreurs triviales

Quotidiennement, tout programmeur est exposé à des erreurs triviales. Par exemple, un point-virgule manquant peut empêcher la compilation d'un programme entier. Ces petites fautes, qui ont de grandes conséquences, sont donc à bannir.

Pour cela, nous avons pris plusieurs décisions afin de ne pas faire face à ces erreurs. Tout d'abord, nous utilisons tous un environnement de développement intelligent (Intellij par JetBrains). L'utilisation d'un IDE a certes des inconvénients, mais sur ce point, c'est un atout de taille. En effet, notre IDE nous prévient lors d'une erreur de syntaxe évidente. Il permet aussi de faciliter l'écriture du code. Pour un projet de cette envergure, l'IDE nous permet de développer avec plus de sécurité.

Nous avons également mis en place des revues de code. À l'aide de Git, nous pouvons assigner des revues de codes aux différents membres de l'équipe. Cela a pour effet de corriger les erreurs évidentes, et par la même occasion, les différents membres prennent connaissance du code des autres. Sur les quelques revues de code que nous avons fait, des erreurs ont été retrouvées, la mise en place de ce fonctionnement est donc un succès.

Afin de garantir un minimum de fonctionnalités, des tests de bases sont développés durant le projet pour assurer que la version courante valide des fonctionnalités de base. Ces tests portent sur la base de chacune des parties des différents sous-langages de Deca. Des scripts comme `common-test.sh` entrent dans ce principe. En cas de succès de ces derniers, nous nous assurons que les fonctionnalités minimales sont couvertes.

### 4.4 Git

Deux types de risques peuvent advenir sur Git. Le premier est le non fonctionnement de GitLab, la plateforme sur laquelle est stockée tout notre travail. Si GitLab cesse de fonctionner, nous pouvons toujours reprendre une version sauvegarder dans les dernières 24 heures. En effet, tous les jours, nous faisons une copie de l'entièreté du dépôt GitLab. Pour continuer le projet, nous possédons tous un compte GitHub. Il nous est donc possible de continuer le projet sur cette plateforme.

Un autre problème est la gestion du Git. Toutes les personnes du groupe n'ayant pas un niveau équivalent, il se peut que certaines personnes en viennent à endommager le dépôt Git. Pour palier à cela, un responsable Git est désigné. Il aide donc les autres membres du groupe pour effectuer les opérations de `commit`, `push`, `merge`, `rebase`, si ces derniers ne sont pas familiers avec l'environnement.

### 4.5 Tests

La gestion des risques existe également au sein-même de la gestion des tests. Comme le précise cette documentation liée à la validation, il y a plusieurs types de tests. Les tests dits *boîte-noire* sont les plus sûrs. Nous vérifions le contenu exact de la réponse. Les tests oracles sont moins sûrs. Nous vérifions uniquement si l'exécution s'est déroulée avec une erreur ou non. Il faut donc essayer de cibler le but de chaque test. Voulons-nous tester qu'un programme s'exécute correctement? Voulons-nous vérifier qu'un programme donne une bonne réponse? Voulons-nous vérifier l'erreur d'un programme invalide?

Prenons l'exemple du test du parser. Nous réalisons des tests sur l'arbre produit. Sur ces tests, nous répliquons exactement les arbres attendus, et nous les comparons avec les résultats. Cette méthode de test est très précise mais elle demande beaucoup de temps pour produire les tests. C'est pourquoi les tests oracles permettent de créer une base de tests beaucoup plus grande, et plus exhaustive. Il faut donc juger du nombre de test à effectuer dans chaque catégorie.

### 4.6 Méthodes générales

Pour limiter les biais de compréhension et d'interprétation, les testeurs ne doivent pas être en contact avec le développement. Ainsi, les testeurs n'ont pas de supposition quant au code source, notamment sur ses faiblesses et ses forces.



Pour l'extension, notre groupe utilise une méthode de développement dirigé par les tests. Avec cette technique de travail, les tests sont créés avant même le début du développement. Ainsi, les développeurs doivent passer tous les tests créés pour affirmer qu'une fonctionnalité est implémentée. Cette méthode de travail permet d'augmenter la fiabilité du code produit.

Pour ce qui est des documentations, nous les rédigeons au fur et à mesure. Cela nous permet d'être tous au courant des avancements du projet. Les clients peuvent également comprendre à tout moment ce qui est en train d'être développé, et ils peuvent aussi comprendre les méthodes de travail de l'équipe.

## 5 Gestion des rendus

### 5.1 Motivations

En génie logiciel, la gestion des mises en production d'un rendu est une technique utilisée pour planifier, gérer et contrôler un déploiement ou une mise à jour logicielle. En effectuant les étapes adéquates, la gestion des rendus permet à l'équipe d'obtenir des informations importantes au bon moment afin de favoriser le succès du déploiement. Dans la vie d'un projet informatique, cette étape est cruciale car c'est à ce moment-là que le client va véritablement découvrir le contenu du produit attendu. La gestion des rendus est un processus de concrétisation du concept original du logiciel et consiste à le placer dans un environnement réel.

### 5.2 Mesure de prévention

A l'approche d'un rendu important comme le rendu final, une bonne pratique est de créer un pré-rendu sur la branche de rendu. Ce pré-rendu ne comporte pas forcément toutes les fonctionnalités du rendu final, mais il sert de sécurité. En cas de problème, cette version fonctionnelle mais non-complète pourra tout de même être livrée. Il s'agit donc de fixer une date limite fictive avant la date limite réelle (par exemple un jour ou une demi-journée avant).

### 5.3 Liste des actions à effectuer

Afin de limiter les risques au moment de la livraison d'une version, nous proposons d'effectuer une liste d'actions avant chaque rendu. À chaque étape, il est nécessaire d'effectuer l'action demandée puis, selon le résultat, nous passons à la suivante ou nous revenons sur une étape précédente.

1. Développement et correction de bugs.
2. Réunion pour discuter de la version à rendre. Vérification de l'implémentation des fonctionnalités. Effectuer ou modifier la liste des points forts/faibles de la version.
3. Obtenir l'accord des testeurs pour passer à l'action suivante. Il est nécessaire d'avoir assez de tests pour effectuer le rendu. Un calcul de la couverture de tests est effectué avec l'outil Jacoco pour vérifier cela. Les testeurs doivent également vérifier que les spécifications du compilateur Deca mentionnées dans le polycopié sont toutes implémentées. *Tant que la couverture n'est pas assez haute, ou que les testeurs pensent que les tests ne sont pas exhaustifs, rester à l'étape 2). Si un test ne passe pas, revenir à l'étape 0).*
4. Sur une machine personnelle, après s'être assuré qu'il n'y a plus de fichier à `commit`, vérifier que l'exécution de la commande `mvn test` s'effectue correctement. Si des fichiers ne sont pas encore `commit`, vérifier leur contenu, comprendre leurs impacts, et `push` ces fichiers. *Si l'ensemble des tests passent, continuer à l'étape suivante, sinon, revenir à l'étape 0).*
5. Effectuer un `merge` de la branche courante sur la branche `master`. Les tests et le code source doivent se retrouver sur la branche `master` à l'issue de cette opération.
6. Sur une machine candide de l'école, s'assurer que le fichier `.bashrc` ajoute bien les chemins requis pour utiliser les différents outils (Maven, IMA...). Faire un `git clone` du projet à partir du dernier `commit` de la branche `master`. Puis, effectuer la commande `mvn verify`. S'assurer que tous les tests passent et que la couverture Jacoco reste la même que celle de l'étape 2). *Si certains tests échouent, retourner à l'étape 3), il se peut que certains fichiers n'aient pas été commit sur cette version.*

7. Supprimer le projet localement.
8. Livrer la version.

## 6 Jacoco

Jacoco est l'outil utilisé pour mesurer la couverture de l'ensemble de nos tests. Cela nous permet de vérifier quels parties du code sont exécutées par les tests intégrés à Maven.

Les parties concernant les entrées utilisateurs n'ont pas été testées automatiquement, et n'apparaissent donc pas dans la couverture. Elles ont néanmoins été validées manuellement par des tests systèmes.

### Deca Compiler
















Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">fr.ensimag.deca</a>		82%		76%
<a href="#">fr.ensimag.deca.codegen</a>		93%		83%
<a href="#">fr.ensimag.deca.context</a>		88%		86%
<a href="#">fr.ensimag.deca.syntax</a>		75%		55%
<a href="#">fr.ensimag.deca.tools</a>		94%		100%
<a href="#">fr.ensimag.deca.tree</a>		94%		89%
<a href="#">fr.ensimag.ima.pseudocode</a>		78%		75%
<a href="#">fr.ensimag.ima.pseudocode.instructions</a>		69%		n/a
Total	3,822 of 24,816	84%	475 of 1,579	69%

FIGURE 4 – Couverture globale du compilateur

### fr.ensimag.deca.tree



















Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">AbstractReadExpr.java</a>		7%		0%
<a href="#">Null.java</a>		65%		n/a
<a href="#">NoOperation.java</a>		71%		n/a
<a href="#">InstanceOf.java</a>		74%		100%
<a href="#">ArrayAccess.java</a>		83%		85%
<a href="#">ReadInt.java</a>		84%		n/a
<a href="#">ReadFloat.java</a>		84%		n/a
<a href="#">LocationException.java</a>		85%		60%
<a href="#">MethodAsmBody.java</a>		86%		100%
<a href="#">Selection.java</a>		87%		87%
<a href="#">NewArray.java</a>		88%		75%

FIGURE 5 – Couverture des classes issues de la grammaire (inférieure à 80%)

La classe `AbstractReadExpr` concerne des entrées utilisateurs, la partie n'ayant pas été testé est sa méthode concernant la génération de code.

La faible couverture des autres classes est majoritairement dû à la décompilation qui n'a pas été testée dans l'automatisation des tests.

**fr.ensimag.deca.context**



















Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">Definition</a>		63%		100%
<a href="#">ParamDefinition</a>		63%		n/a
<a href="#">Signature</a>		76%		50%
<a href="#">ClassDefinition</a>		77%		70%
<a href="#">VariableDefinition</a>		77%		n/a
<a href="#">Type</a>		79%		n/a
<a href="#">MatrixFloatType</a>		85%		n/a
<a href="#">MatrixIntType</a>		85%		n/a
<a href="#">FieldDefinition</a>		87%		n/a
<a href="#">MethodDefinition</a>		92%		50%
<a href="#">EnvironmentExp</a>		96%		92%
<a href="#">EnvironmentTypes</a>		98%		94%

FIGURE 6 – Couverture des classes issues de la partie contextuelle (inférieure à 100%)

Les méthodes des classes spécifiques à l'analyse contextuelle qui n'ont pas été redéfinies n'ont pas toutes été testées, ce qui explique la faible couverture sur ces classes.

**fr.ensimag.deca.codegen**



Element	Missed Instructions	Cov.	Missed Branches	Cov.
<a href="#">LabelGenerator</a>		93%		83%
Total	16 of 259	93%	1 of 6	83%

FIGURE 7 – Couverture des classes issues de la génération de code

Une seule classe spécifique à la génération de code a été rajoutée, et testée dans sa quasi-intégralité.