# Validation Compilateur

*Projet Génie Logiciel*

**Fabien Galzi, Abdelmlek Mrad, Lisa Cornu,**

**Matéo Bedel, Eliot Clerc**

Groupe GL 31

20/01/20025

# Sommaire

# 1. Introduction

Ce document de validation a pour but de présenter les méthodes et outils mis en place pour garantir la fiabilité et la robustesse du compilateur. Il est destiné à toute personne souhaitant reproduire les tests, analyser la validation effectuée sur le projet et de trouver des faiblesses. Nous y décrivons l'ensemble des stratégies adoptées pour tester le compilateur, en expliquant leur organisation, leurs objectifs, et la manière dont nous les avons fait. Les méthodes de validation autres que les tests sont également abordées, afin de fournir une vision complète de la démarche mise en œuvre. Ce document illustre également l'utilisation des outils nécessaires pour exécuter les tests, notamment les scripts, ainsi que les résultats obtenus, notamment ceux produits par l'analyse de couverture par Jacoco.

# 2. Description des tests

## 2.1 Les types de tests

- Les tests unitaires :

Pour la majeure partie de nos tests, nous avons fait le choix des tests unitaires, en effet ils sont assez rapides à exécuter et nous fixent sur l'endroit d'un éventuel problème. Tout au long du projet, nous avons utilisé ces tests pour être certain du bon fonctionnement de chacune des parties. Les tests unitaires ont été majoritaires pour les parties A et B. A chaque règle, nous avons essayé de faire un test valide et autant de tests invalides nécessaires pour tester toutes les conditions.

- Les tests d'intégration :

Ils concernent principalement la partie C, ces tests sont souvent plutôt complets et nécessitent l'implémentation de toutes les parties pour pouvoir les exécuter. Nous avons souvent fait ces tests pour savoir si nous avions finalement fait des erreurs dans la partie A et B car nous avons principalement regardé si la compilation se faisait mais sans en regarder le résultat au départ.

- Les tests fonctionnels :

Tout comme les tests d'intégration, ils ont été appliqués dans la partie de génération de code. Cette fois-ci nous avons vérifié si le résultat retourné était celui attendu. Ce qui une nouvelle fois pouvait faire parvenir des erreurs dans les parties A et B, par exemple des erreurs dans la définition de certaines variables.

- Les tests d'acceptation :

Il existe quelques tests qui n'ont pas de but particulier mais étant simplement des programmes en deca qui doivent pouvoir être exécutés, ils permettent de voir si un utilisateur qui ne cherche pas nécessairement à trouver des bugs peut faire fonctionner le compilateur.

## 2.2 Organisation des tests

Nous avons établi une arborescence particulière concernant les tests. Ils sont répartis dans quatre dossiers, *codegen*, *context*, *others* et *syntax*. Chaque dossier est composé de plusieurs dossiers : les dossiers *valid* et *invalid* contenant les tests valides ou invalides des parties à vérifier, sachant qu'un test valide peut toujours être utilisé dans la partie suivante (valide ou non). Dans les dossiers *valid* et *invalid*, nous avons décidé de les répartir selon les tests personnels (*personnal*) que nous avons fait nous même et *provided* pour ceux qui nous ont été donnés. S'il n'y a pas de sous dossier c'est que tous les tests ont été créés par nous-même. Vous pouvez aussi trouver dans les dossiers *expected*, les résultats attendus des divers programmes de test. Il existe aussi quelques tests de performance qui nous ont été donnés.

Il nous semble important de mentionner que la majorité des tests ont été réalisés par une personne différente de celle s'occupait du code, bien que quelques-uns aient été faits par les mêmes personnes car il est plus simple de trouver les limites du code si nous l'avons fait nous même. Il nous semblait important de procéder de cette manière pour avoir deux points de vue sur un même aspect du développement, et ainsi pouvoir discuter et relever les éventuelles erreurs que l'un ou l'autre ferait. Enfin, nous avons, dans certains cas, trouvé très compliqué de tester l'invalidité de certains programmes.

### 2.3 Objectifs des tests

Tous ces tests ont pour objectifs de rendre notre compilateur le plus robuste et le plus juste possible afin qu'il réponde aux exigences de la meilleure des manières possibles. Ils nous ont permis de trouver beaucoup d'erreurs durant tout le projet. Mais nous avons aussi donc permis d'en corriger un grand nombre. Tous les noms des tests sont écrits en snake_case et se doivent d'être assez clair sur le contenu du code.

# 3. Les scripts de tests

Nous avons fait deux scripts permettant de lancer automatiquement nos fichiers de test et d'en analyser les résultats.

Le premier est très succinct et concerne les tests de la syntaxe contextuelle. Il lance les tests dans chaque section, puis vérifie qu'il n'y ait pas d'erreur dans les tests valides et qu'au contraire lorsqu'une erreur est levée, ce soit la bonne et qu'elle soit levée par un test invalide. Nous n'avons pas fait de programmes *expected* pour la partie contexte car pour de petits arbres, il est simple et rapide de le faire, mais pour des programmes beaucoup plus complets, nous aurions probablement fait des erreurs dans l'écriture de l'arbre, entraînant ainsi des tests non concluants pour de mauvaises raisons. Nous avons donc pensé que la vérification du résultat de la partie B serait surtout perçue dans la partie de génération de code et dès lors qu'une erreur apparaissait dans la partie

C et ayant comme source la partie B, nous l'avons corrigée.

Le second script quant à lui concerne la génération de code et la vérification des tests est beaucoup plus complète. En effet le script *test_codegen.sh* fait une comparaison entre le résultat après l'exécution du code assembleur avec un résultat attendu. Pour certains tests, il n'existe tout de même pas de fichier expected car les fonctionnalités que l'on test ne peuvent pas produire de sortie, ou bien nécessite un input de l'utilisateur.

Pour lancer ces tests, il faut se mettre à la racine du projet et faire la commande suivante ./src/test/script/[nom du script].sh. Si vous le souhaitez, le script concernant les tests de génération de code peut nettoyer et compiler le compilateur (code à décommenter dans le script).

Nous avons également écrit un test : *test_decompile.sh* pour valider l'idempotence de la décompilation, une caractéristique essentielle pour notre compilateur. Ce script compare, pour chaque fichier *.deca*, le résultat de la commande *decac -p* à celui obtenu en appliquant à nouveau cette même commande sur le résultat précédent. Cela permet de vérifier que la décompilation produit toujours le même résultat. Nous avons choisi cette méthode de comparaison plutôt que de comparer directement au fichier source, car la décompilation ajoute certaines parenthèses non obligatoires qui n'apparaissent pas dans le fichier original ce qui aurait rendu la comparaison des résultats plus complexe.

# 4. Gestion des risques

**1. Gestion des tests incorrects**

**Problème** : La personne réalisant les tests peut produire des tests erronés.
**Solution** : Une autre personne du groupe (autre que celle ayant codé la partie testée) doit examiner le test pour éviter des biais ou des erreurs.

**2. Tests incomplets ou non ciblés**

**Problème** : Les tests peuvent ne pas couvrir des cas critiques. Par exemple, au début, les tests peuvent uniquement détecter des erreurs évidentes (comme l'absence d'arbre abstrait ou de tokens) sans traiter les cas subtils.
**Solution** : Mettre en place une stratégie de tests progressifs en suivant les étapes A, B, et C décrites ci-dessous.

**Étape A : Construction de l'arbre abstrait**

1. **Tests des non-terminaux** :

   **Problème** : Une mauvaise gestion des non-terminaux impacte toutes les étapes suivantes, notamment les vérifications contextuelles et la génération de code.
   **Exemples critiques** : Déclarations de classes et méthodes, instructions conditionnelles.
   **Risque** : Moyen | **Difficulté** : Facile à moyen.
2. **Tests des listes** :

   **Problème** : Une gestion incorrecte des listes peut générer des erreurs dans les structures répétitives ou la manipulation des blocs complexes.
   **Cas à tester** : Liste vide, liste avec un seul élément, ou liste contenant plusieurs éléments.
   **Risque** : Modéré | **Difficulté** : Facile à moyen.
3. **Tests lexicographiques** :

   **Séparateurs (;)** : Une mauvaise gestion peut désorganiser toute l'analyse lexicale et syntaxique.
   **Risque** : Très élevé | **Difficulté** : Facile.
   **Mots réservés** : Leur confusion avec des identificateurs peut compromettre l'analyse syntaxique.
   **Risque** : Modéré | **Difficulté** : Moyen.
   **Littéraux numériques (entiers et flottants)** :
   **Problèmes** : Overflows, erreurs de précision, arrondis incorrects.
   **Risque** : Modéré à élevé | **Difficulté** : Moyen à difficile.
   **Chaînes de caractères** : Les erreurs dans les chaînes non valides (caractères interdits, fins de ligne) peuvent entraîner des plantages.
   **Risque** : Modéré | **Difficulté** : Moyen.
   **Commentaires** : L'absence de fermeture des commentaires peut perturber tout le fichier source.
   **Risque** : Faible | **Difficulté** : Facile.

**Étape B : Vérifications contextuelles**

1. **Tests de couverture sémantique** :

   **Types dans les expressions** : Les erreurs de typage affectent la logique du programme.
   **Risque** : Très élevé | **Difficulté** : Moyen.
   **Décorations des nœuds** : Des décorations incorrectes compromettent la génération de code final.
   **Risque** : Moyen | **Difficulté** : Facile.
   **Enrichissement de l'arbre** : Un arbre incomplet devient inutilisable pour les étapes suivantes.

**Risque** : Élevé | **Difficulté** : Facile.

2. **Tests pour les environnements** :

   **Variables locales** : Les erreurs de portée ou de réutilisation incorrecte peuvent perturber le programme.
   **Risque** : Élevé | **Difficulté** : Difficile.
   **Attributs de classe** : Leur mauvaise gestion impacte les relations entre les classes.
   **Risque** : Modéré à élevé | **Difficulté** : Difficile.
   **Méthodes héritées** : Les erreurs dans les environnements des méthodes héritées affectent la cohérence.
   **Risque** : Modéré | **Difficulté** : Difficile.

3. **Tests d'erreurs contextuelles** :

   **Variables non déclarées** : Une utilisation non déclarée affecte la cohérence du programme.
   **Risque** : Très élevé | **Difficulté** : Facile.
   **Types incompatibles** : Des types mal gérés dans les affectations rendent le programme inutilisable.
   **Risque** : Très élevé | **Difficulté** : Facile.

**Étape C : Validation finale**

Si les étapes A et B produisent des résultats corrects, les erreurs dans cette étape sont plus faciles à identifier et à corriger grâce à des tests bien ciblés.

# 5. Jacoco

Nous n'avons pas utilisé Jacoco tout au long du projet cependant il nous a été très utile à la fin du projet. Nous avons essayé d'avoir le maximum de couverture et grâce à cela nous avons trouvé de nouveaux tests, souvent invalides dont nous n'avions pas eu l'idée. De plus, il nous a permis de retrouver quelques parties de code mort (vérification déjà faites auparavant).

Au final nous avons une très bonne couverture des instructions (77%) et 64% des branches, sachant que la quasi-totalité des branches non couvertes correspondent à de l'affichage ou bien la décompilation.

Le script *test_decompile.sh* avait pour but de couvrir ces branches en testant l'idempotence de la décompilation cependant nous l'avons commencé trop tard et nous nous sommes rendu compte en lançant ce test que notre décompilation comporte quelques bugs. Toutefois, les parties ne concernant que la compilation sont bien mieux couvertes (possiblement >90%).

Pour utiliser les tests automatisés par jacoco, vous devez d'abord nettoyer l'environnement : mvn clean. Puis vous devez commenter la partie qui compile toute seule dans le *test_codegen.sh* et enfin faire mvn -Djacoco.skip=false verify et finalement ouvrir dans internet le fichier index.html qui se trouvera dans target/site/jacoco.

# 6. Autres méthodes de validation hors tests

Nous avons refait le jeu de la vie qui nous permet indirectement de tester presque toutes les fonctionnalités de notre compilateur. Nous avons aussi essayé de relire le code avec la documentation à côté à la fin pour vérifier si ça la respectait bien.