Documentation de validation — Compilateur Deca

Projet Génie Logiciel (Grenoble-INP Ensimag)

Équipe gl43 – Année 2025–2026

22 janvier 2026

Table des matières

1 Préambule

Dans le développement d'un compilateur quelconque, la validation constitue une étape cruciale qui doit être faite de manière méticuleuse.

En effet, valider un compilateur ne se borne pas à exécuter des programmes de test et vérifier les sorties attendues. Il s'agit d'implanter un degré de confiance raisonné et raisonnable quant au strict respect d'un ensemble de propriétés fondamentales : correction syntaxique et contextuelle, préservation de la sémantique lors de la génération de code, conformité des conventions d'exécution, et robustesse face à des cas limites.

Cette documentation présente la démarche de validation mise en œuvre pour le compilateur du langage Deca, développé dans le cadre du projet de Génie Logiciel à Grenoble-INP Ensimag. Elle couvre l'ensemble des étapes du compilateur (analyse lexico-syntaxique, vérifications contextuelles, génération de code), dans le périmètre fonctionnel effectivement implémenté.

La stratégie de validation adoptée repose sur une combinaison structurée de tests automatisés, de tests négatifs ciblés, de tests de non-régression et de méthodes de validation complémentaires : raisonnement par invariants, relectures croisées, analyse de couverture. Chaque famille de tests est conçue pour réduire des risques identifiés propres à chaque étape du compilateur.

Le principe directeur de cette stratégie est le suivant : la validation vise à établir des garanties sur des propriétés du compilateur et non à accumuler machinalement des cas de test isolés. Cette documentation permet ainsi à un tiers de comprendre, reproduire et prolonger la validation du compilateur, tout en identifiant clairement les garanties obtenues et les limites actuelles.

2 Périmètre, hypothèses et garanties

2.1 Périmètre fonctionnel validé

Cette sous-section a pour objectif de délimiter le périmètre de ce qui est pris en charge par la démarche de validation, de manière à éviter toute ambiguïté quant à l'interprétation des résultats présentés ultérieurement.

Le compilateur validé dans le cadre de ce projet couvre les sous-langages suivants du langage Deca :

- le langage hello-world,
- le langage sans objet,
- le langage essentiel,
- ainsi que le langage complet.

La validation concerne l'ensemble de la chaîne de compilation, à savoir :

- l'étape A : analyse lexico-syntaxique et construction de l'arbre abstrait ;
- l'étape B : vérifications contextuelles et décoration de l'arbre abstrait ;
- l'étape C : génération de code assembleur pour la machine abstraite IMA.

Les tests ont été réalisés dans diverses configurations, notamment :

- sur des programmes valides et invalides ;
- sur des programmes de taille et complexité variées ;
- avec différentes valeurs du nombre maximal de registres autorisés (option -r) ;
- avec l'extension d'optimisation activée (option -o).

Il est à prendre en compte que la validation ne couvre que les fonctionnalités effectivement implémentées dans le compilateur au moment du rendu, conformément aux spécifications données dans le cahier des charges du projet.

2.2 Hypothèses de validation

Il est manifeste que toute démarche de validation repose sur un ensemble d'hypothèses préétablies. Celles-ci sont explicitées ci-dessous afin de préciser le contexte dans lequel les garanties présentées sont valables dans le cadre défini.

Les tests ont été exécutés dans le même environnement que celui du développement du compilateur, en particulier :

- Système GNU / Linux ;
- Machine virtuelle Java ;
- Interprète de la machine abstraite IMA fourni dans le cadre du projet.

Il est cependant fortement conseillé au lecteur désirant reproduire voire continuer la validation de se référer à la section *Séance Machine* du cahier des charges du projet.

On suppose par ailleurs que l'outil IMA est correct et conforme à sa spécification : la validation du compilateur ne cherche pas à remettre en cause l'implémentation de la machine abstraite. Au contraire, elle s'y appuie comme oracle d'exécution du code généré.

2.3 Garanties établies

Sur la base du périmètre et des hypothèses précédemment définis, l'approche de validation choisie garantit, avec un degré de confiance suffisamment élevé, que :

- les programmes Deca syntaxiquement et contextuellement corrects sont acceptés par le compilateur ;
- les programmes incorrects sont rejetés à l'étape appropriée, avec un message d'erreur cohérent et ciblé ;
- le code assembleur généré respecte les conventions de la machine abstraite IMA ;
- la sémantique des programmes Deca valides est préservée lors de la génération de code, dans les cas testés ;
- les mécanismes fondamentaux de gestion de la pile, des registres sont correctement implémentés.

En contrepartie, la validation ne garantit pas :

- l'absence totale de défauts dans le compilateur ;
- la correction du compilateur pour l'ensemble des programmes Deca possibles ;
- la performance et efficacité énergétique optimales du code généré ;
- la robustesse du compilateur face à des environnements d'exécution ne se conformant pas aux hypothèses préalablement énoncées.

3 Modèle de risques du compilateur

3.1 Identification des risques majeurs

L'espace des programmes sources possibles étant infini, la validation d'un compilateur ne peut aucunement être exhaustive. Une démarche orientée risques doit donc être élaborée et suivie. Celle-ci consiste à identifier les défaillances critiques susceptibles d'affecter la correction du compilateur, et à concevoir par suite des tests spécifiquement destinés à les réduire.

Les risques relevés sont classés par étape du compilateur :

3.1.1 Étape A — Analyse lexico-syntaxique et construction de l'AST

Risques sémantiques :

- Mauvaise reconnaissance de certaines constructions du langage, conduisant à l'acceptation de programmes syntaxiquement incorrects ou au rejet de programmes syntaxiquement valides.
- Construction d'un arbre abstrait infidèle à la structure du programme source.

Risques structurels :

- Ambiguïtés lexicales ou syntaxiques mal gérées (priorité des règles, associativité).
- Mauvaise gestion des cas limites (programmes vides, commentaires imbriqués, etc).

Risques d'exécution :

- Comportement non déterministe lors de l'analyse de programmes incorrects.

3.1.2 Étape B — Vérifications contextuelles

Risques sémantiques :

- Non-détection d'erreurs de typage ou d'utilisation erronée d'identificateurs.
- Mauvaise résolution des liens entre identificateurs et définitions.

Risques structurels :

- Incohérences dans les environnements de symboles.
- Mauvaise gestion des mécanismes d'héritage et de redéfinition de méthodes.

Risques d'exécution :

- Détection tardive d'erreurs qui auraient dû être levées plus tôt.
- Messages d'erreur imprécis ou incohérents, compromettant l'utilisabilité du compilateur.

3.1.3 Étape C — Génération de code

Risques sémantiques :

- Non-préservation de la sémantique du programme source lors de la génération du code cible.
- Mauvaise implémentation des mécanismes d'orienté objet (tables des méthodes, accès aux champs).

Risques structurels :

- Mauvaise gestion de la pile et des registres (écrasement de valeurs, déséquilibre de pile).
- Non-respect des conventions d'appel et de retour définies par la machine abstraite IMA.

Risques d'exécution :

- Plantage à l'exécution du programme généré.
- Comportements indéfinis ou erronés apparaissant uniquement dans des cas complexes (expressions profondes, appels imbriqués, forte pression sur les registres).

3.2 Hiérarchisation des risques

Les risques identifiés sont hiérarchisés selon trois critères : leur probabilité d'occurrence, leur gravité, et leur détectabilité.

En l'occurence :

- les risques de l'étape C présentent une gravité élevée, dans la mesure où ils peuvent produire des programmes incorrects sans erreur apparente à la compilation ;
- les risques de l'étape B sont critiques sur le plan sémantique, d'autant plus qu'ils conditionnent la validité des hypothèses de l'étape C ;
- les risques de l'étape A sont souvent plus facilement détectables, mais peuvent bloquer l'ensemble de la chaîne de compilation.

Cette hiérarchisation a ainsi aiguillé l'effort de validation en concentrant les tests les plus exigeants sur les risques à forte gravité, faible détectabilité, et davantage susceptibles d'être rencontrés.

3.3 Stratégie globale de réduction des risques

L'approche de validation adoptée cherche à réduire systématiquement les risques identifiés, en associant à chaque catégorie de risques une ou plusieurs familles de tests.

Ainsi, les risques lexico-syntaxiques sont minimisés via :

- des tests invalides ciblant des violations précises de la grammaire ;
- des tests limites sur les constructions minimales du langage.

Quant aux risques contextuels, leur réduction est assurée par :

- des programmes presque corrects ne violant qu'une seule règle contextuelle ;
- des tests spécifiques sur l'héritage, le typage et la résolution des identificateurs.

Enfin, les risques en lien avec la génération de code sont réduits via :

- des tests de pression sur les registres ;
- des programmes comportant des appels imbriqués et des boucles profondes ;
- des tests de non-régression comparant le comportement attendu et le comportement observé à l'exécution ;
- des tests métamorphiques.

Il est à noter que certains risques demeurent partiellement couverts et constituent par conséquent des axes d'amélioration pour une éventuelle validation ultérieure, en particulier : les erreurs apparaissant uniquement sur des programmes de taille ou de complexité extrêmes.

4 Architecture globale de la validation

Cette section décrit l'organisation générale de la validation du compilateur Deca : les types de tests mis en œuvre, la structuration des jeux de tests et les mécanismes assurant la reproductibilité.

4.1 Typologie des validations mises en œuvre

La validation repose sur une combinaison de méthodes complémentaires :

- **Tests unitaires Java (JUnit).** Exécutés via `mvn test`, ils valident certains composants internes du compilateur. Leur rôle est volontairement limité : l'essentiel de la confiance est apporté par les tests système de bout en bout.
- **Tests système par scripts.** Ils valident la chaîne complète de compilation, depuis le programme Deca jusqu'à l'exécution sous IMA, ainsi que la détection d'erreurs aux étapes appropriées. Ces tests constituent le cœur de la validation.
- **Tests *valid* / *invalid* par étape.** Pour chaque étape du compilateur (analyse lexicale, syntaxique, contextuelle et génération de code), les tests sont explicitement séparés entre :
  - des programmes valides devant être acceptés ;
  - des programmes invalides devant être rejetés avec un diagnostic cohérent.
- **Tests interactifs.** Les tests du répertoire `codegen/interactif` valident la génération de code pour des programmes nécessitant des entrées utilisateur (`readInt()`, `readFloat()`). Ils permettent de vérifier la bonne interaction entre le code généré et l'environnement d'exécution IMA.

- **Tests de performance.** Le répertoire `codegen/perf` contient un nombre limité de programmes destinés à observer le comportement du compilateur et du code généré sur des cas plus coûteux, sans constituer une évaluation exhaustive des performances.
- **Tests métamorphiques.** Ces tests sont organisés par paires de programmes (`_1.deca`, `_2.deca`) qui doivent produire exactement la même sortie à l'exécution. Ils permettent de vérifier des propriétés d'invariance sémantique, indépendamment d'un oracle de sortie figé.
- **Tests différentiels.** Certains scripts exécutent les mêmes programmes avec différentes configurations du compilateur, notamment en faisant varier le nombre maximal de registres autorisés (`-r X`, avec `X` allant de 4 à 16). Ces tests visent à vérifier que les choix d'allocation de registres n'affectent pas la correction fonctionnelle des programmes générés.

4.2 Organisation des jeux de tests

Les tests Deca sont organisés par étape du compilateur dans `src/test/deca/`, avec une séparation systématique entre programmes `valid` et `invalid` :

- `lex/` pour l'analyse lexicale ;
- `syntax/` pour l'analyse syntaxique et la construction de l'AST ;
- `context/` pour les vérifications contextuelles ;
- `codegen/` pour la génération de code et l'exécution sous IMA.

Pour la génération de code, des sous-répertoires spécifiques dans `valid` permettent de distinguer clairement les objectifs des tests. Cette organisation facilite l'ajout de nouveaux tests, l'analyse des échecs et l'identification des zones de fragilité du compilateur.

5 Validation par étape du compilateur

Cette section décrit, pour chaque étape du compilateur, les objectifs de validation poursuivis, les stratégies de test mises en œuvre, les propriétés effectivement validées et les limites identifiées.

5.1 Étape A — Analyse lexico-syntaxique et construction de l'AST

5.1.1 Objectifs de validation

L'objectif de la validation de l'étape A est de garantir que :

- tout programme conforme à la grammaire officielle du langage Deca est accepté ;
- toute violation lexicale ou syntaxique est correctement détectée et signalée ;
- l'AST construit reflète de manière fidèle la structure syntaxique du programme source.

5.1.2 Stratégies de test

La validation repose sur :

- des tests `lex/valid` et `lex/invalid` ciblant la reconnaissance des unités lexicales et les cas limites ;
- des tests `syntax/valid` et `syntax/invalid` visant des constructions correctes, incorrectes ou ambiguës de la grammaire ;
- des programmes minimaux ou volontairement dégénérés afin de tester la robustesse de l'analyse.

5.1.3 Propriétés validées

Les tests permettent d'établir que, dans les cas couverts :

- l'analyse lexicale reconnaît correctement l'ensemble des tokens attendus ;

- l'analyse syntaxique respecte la grammaire définie et rejette les programmes incorrects ;
- la structure de l'AST correspond à la structure syntaxique du programme source.

5.1.4 Limites identifiées

La validation ne permet pas de garantir l'absence d'erreurs dans des cas pathologiques très complexes ou non testés, notamment en présence de constructions syntaxiques artificielles ou peu naturelles.

5.2 Étape B — Vérifications contextuelles

5.2.1 Objectifs de validation

L'étape B vise à garantir que :

- les règles de typage du langage Deca sont respectées ;
- les mécanismes d'orienté objet (classes, héritage, redéfinition) sont correctement vérifiés.

5.2.2 Stratégies de test

La validation s'appuie sur :

- des tests `context/valid` couvrant des programmes bien typés et sémantiquement corrects ;
- des tests `context/invalid` ciblant des violations fines des règles contextuelles (typage, héritage, redéfinitions) ;
- des programmes presque corrects, ne violant qu'une seule règle contextuelle à la fois.

5.2.3 Propriétés validées

Les tests permettent d'affirmer que :

- les erreurs contextuelles sont détectées à l'étape appropriée ;
- les environnements de symboles sont construits de manière cohérente ;
- les règles fondamentales du typage et de l'héritage sont respectées pour les cas testés.

5.2.4 Limites identifiées

La validation ne garantit pas l'exhaustivité de la détection d'erreurs contextuelles dans des scénarios très complexes ou combinant un grand nombre de mécanismes simultanément.

5.3 Étape C — Génération de code

5.3.1 Objectifs de validation

L'objectif de la validation de l'étape C est de garantir que :

- le code assembleur généré est correct et exécutable sur la machine abstraite IMA ;
- la sémantique des programmes Deca valides est préservée à l'exécution ;
- les conventions d'appel, la gestion de la pile et des registres sont respectées.

5.3.2 Stratégies de test

La validation repose sur :

- des tests `codegen/valid` et `codegen/invalid` validant la génération de code et la détection d'erreurs ;

- des tests interactifs (`codegen/interactif`) vérifiant la gestion des entrées utilisateur (`readInt`, `readFloat`) ;
- des tests métamorphiques, comparant l'exécution de paires de programmes syntaxiquement différents mais sémantiquement équivalents ;
- des tests différentiels faisant varier le nombre maximal de registres autorisés (`-r X`, $4 \leq \texttt{X} \leq 16$) afin de vérifier l'indépendance fonctionnelle vis-à-vis de l'allocation de registres ;
- quelques tests ciblés de performance (`codegen/perf`) destinés à observer le comportement du compilateur sur des cas plus coûteux.

5.3.3 Propriétés validées

Les tests permettent d'établir que :

- les programmes générés s'exécutent correctement sous IMA ;
- la sémantique observée est stable face aux transformations métamorphiques et aux variations du nombre de registres ;
- les mécanismes fondamentaux de génération de code sont robustes pour les cas couverts.

5.3.4 Limites identifiées

La validation ne garantit pas :

- l'absence totale de défauts dans des programmes de taille ou de complexité extrêmes ;
- l'optimalité du code généré en termes de performance.

6 Scripts de tests et automatisation

Cette section décrit l'infrastructure d'automatisation de la validation : scripts de tests, orchestration des campagnes et mécanismes de relance conditionnelle. L'objectif est d'assurer une validation fiable et reproductible, tout en évitant des exécutions inutiles lors des itérations de développement.

6.1 Familles de scripts de validation

Les scripts de validation sont regroupés dans `src/test/script/` et se répartissent en trois familles complémentaires.

Scripts `not_basic-` (campagnes exhaustives). Les scripts `not_basic-` correspondent aux campagnes complètes par famille de tests (lexicale, syntaxique, contextuelle, génération de code, différentiel). Ils exécutent *l'ensemble* des fichiers de test pertinents, en distinguant explicitement :

- les tests `valid`, qui doivent être acceptés sans erreur ;
- les tests `invalid`, qui doivent produire un diagnostic approprié ;

Ces scripts constituent la référence pour établir un état global de non-régression.

Scripts `*_energy.sh` (relance conditionnelle). Les scripts suffixés par `_energy` implémentent une stratégie de validation incrémentale. Ils analysent les fichiers modifiés depuis la dernière référence Git afin de déterminer si une relance des tests est nécessaire, et si oui, sur quel périmètre.

Scripts spécialisés. Certains scripts ciblent des propriétés spécifiques :

- tests différentiels multi-registres (`-r X`) ;
- tests métamorphiques par paires de programmes ;

Script `test_all.sh` (point d'entrée de la validation complète) Le script `test_all.sh` constitue le point d'entrée principal de la validation complète. Il enchaîne la compilation du projet, l'exécution des tests unitaires Java, le lancement des campagnes de tests système (scripts `not_basic-*`, tests métamorphiques et différentiels), puis génère le rapport de couverture JaCoCo.

6.2 Relance conditionnelle et sobriété des tests

Les scripts `*_energy.sh` reposent sur une analyse systématique des différences Git entre deux états du dépôt (local ou CI).

Détection des changements. Pour chaque exécution, les scripts identifient :

- les fichiers modifiés entre deux commits ;
- les nouveaux fichiers non suivis ;
- en excluant explicitement les fichiers générés automatiquement (par exemple ceux issus d'ANTLR).

Filtrage des changements non fonctionnels. Les modifications portant uniquement sur des commentaires ou des lignes vides sont considérées comme *non fonctionnelles*. Dans ce cas, les tests correspondants ne sont pas relancés, afin d'éviter des calculs inutiles.

Politique de relance. La stratégie de relance est la suivante :

- en l'absence de changement pertinent, aucun test n'est exécuté ;
- en cas de modification fonctionnelle du code source d'une étape (lex, syntax, context, codegen), l'ensemble des tests de cette étape est relancé via les scripts `not_basic-` ;
- si seules des entrées de test ont été modifiées, seuls les tests concernés sont ré-exécutés.

Cette approche permet de conserver une validation fiable tout en réduisant significativement le coût des campagnes répétées.

6.3 Tests différentiels multi-registres

Les tests différentiels vérifient que la correction fonctionnelle du code généré est indépendante du nombre maximal de registres autorisés. Les programmes sont compilés et exécutés avec différentes valeurs de `-r X` ($4 \leq \texttt{X} \leq 16$), et chaque configuration est vérifiée individuellement.

6.4 Tests métamorphiques

Les tests métamorphiques sont organisés par paires de programmes `_1.deca` et `_2.deca`. Chaque paire représente deux programmes distincts mais sémantiquement équivalents.

Pour chaque paire :

- les deux programmes sont compilés indépendamment ;
- les programmes générés sont exécutés sous IMA ;
- les sorties sont comparées de manière stricte.

Ces tests permettent de valider des invariants sémantiques sans dépendre d'un oracle de sortie prédéfini, et sont particulièrement adaptés à la validation de la génération de code.

6.5 Synthèse et traçabilité des résultats

Chaque script produit une synthèse structurée des résultats (fichiers `.json`) indiquant le nombre de tests réussis et échoués. Ces synthèses sont utilisées pour générer des badges et faciliter :

- l'identification rapide des régressions ;

- l'analyse du périmètre effectivement testé ;
- la lecture globale de l'état de la validation.

L'ensemble de ces mécanismes constitue une infrastructure de validation automatisée, incrémentale et reproductible, adaptée à un projet de compilateur de taille significative.

7 Couverture de tests et analyse critique (JaCoCo)

Cette section présente les résultats de la couverture de tests mesurée à l'aide de JaCoCo, ainsi qu'une analyse critique de leur interprétation. L'objectif n'est pas uniquement de fournir des pourcentages, mais d'évaluer la pertinence de la couverture obtenue au regard de la nature du projet et des méthodes de validation mises en place.

7.1 Résultats globaux

La couverture a été mesurée après l'exécution des tests unitaires Java et des campagnes de tests système, avec génération du rapport JaCoCo.

Globalement, le projet atteint :

- **75% de couverture d'instructions** ;
- **62% de couverture de branches** ;
- **environ 1 450 méthodes** et **260 classes** analysées.

La couverture varie selon les packages, comme illustré ci-dessous :

- `fr.ensimag.deca.syntax` : couverture correcte des instructions, mais couverture de branches plus limitée ;
- `fr.ensimag.deca.context` : couverture élevée, reflétant un fort volume de tests invalides et de cas d'erreur ;
- `fr.ensimag.deca.codegen` : couverture satisfaisante compte tenu de la complexité et de la nature combinatoire du code ;

Ces résultats traduisent une couverture globalement homogène, avec des disparités explicables par la nature des packages.

7.2 Analyse qualitative de la couverture

Classes critiques bien couvertes. Les parties centrales du compilateur — analyse syntaxique, analyse contextuelle et génération de code — bénéficient d'une couverture significative. Cela s'explique par :

- la présence de nombreux tests invalides ciblant les chemins d'erreur ;
- l'utilisation de tests système couvrant des scénarios complets de compilation ;
- l'existence de tests différentiels et métamorphiques sollicitant indirectement une grande variété de chemins d'exécution.

Classes peu couvertes : interprétation. Certaines classes présentent une couverture plus faible, en particulier :

- du code défensif ou des cas d'erreur rares, difficiles à déclencher de manière contrôlée ;
- des chemins dépendant de configurations spécifiques (par exemple, compilation avec les différentes options).

Dans ces cas, une faible couverture ne reflète pas nécessairement un manque de validation, mais parfois un choix conscient de ne pas spécifier des comportements trop marginaux.

Il est important de souligner que la couverture JaCoCo mesure l'exécution du code, et non sa correction. Un taux élevé ne garantit pas l'absence de défauts, tout comme une couverture partielle peut être compensée par :

- des tests système couvrant des comportements globaux ;
- des tests métamorphiques validant des invariants sémantiques ;
- des tests différentiels vérifiant la robustesse face à des variations de paramètres.

La couverture doit donc être interprétée comme un indicateur parmi d'autres, et non comme un objectif en soi.

7.3 Axes d'amélioration

Des gains de couverture pourraient être envisagés via des tests unitaires JUnit ciblés en analysant les portions de code pertinentes non exécutées, dans la limite du possible et du raisonnable, sans pour autant forcer la maximisation artificielle des pourcentages.

8 Méthodes de validation autres que le test

En complément des campagnes de tests automatisés, plusieurs méthodes de validation non instrumentées ont été mises en œuvre tout au long du projet. Ces approches ont permis de détecter des défauts difficiles à mettre en évidence par des tests seuls, et de renforcer la robustesse globale du compilateur.

8.1 Relecture de code et validation croisée

Une relecture de code régulière a été pratiquée entre sous-équipes. Chaque étape du compilateur (analyse lexicale, syntaxique, contextuelle, génération de code, optimisations) étant prise en charge par un ou plusieurs membres dédiés, les relectures croisées ont permis :

- d'identifier des incohérences d'interface entre étapes successives ;
- de vérifier la conformité aux conventions du projet ;
- de détecter des erreurs de raisonnement indépendamment des tests.

Cette validation humaine s'est révélée particulièrement utile pour les parties à forte complexité structurelle, comme la gestion des environnements, des registres ou des conventions d'appel.

8.2 Communication inter-sous-équipes

Le projet a été organisé en sous-équipes responsables de différentes étapes du compilateur. Des échanges réguliers ont été nécessaires afin d'assurer la cohérence du passage d'une étape à l'autre. Ces communications ont permis d'anticiper des erreurs d'intégration qui ne se manifestent pas toujours par des échecs immédiats de tests, mais peuvent produire des comportements incorrects ou instables à plus long terme.

8.3 Diagnostic bas niveau du code généré

En cas de comportement incorrect ou inattendu à l'exécution, une analyse directe du code assembleur IMA (`.ass`) généré a été utilisée comme outil de diagnostic. Cette analyse a permis notamment :

- de vérifier la cohérence des conventions d'appel ;
- d'identifier des erreurs de gestion de pile ou de registres ;
- de comprendre des divergences entre la sémantique attendue et le code effectivement produit.

Ce travail de validation bas niveau s'est avéré indispensable pour la génération de code et les optimisations, où les erreurs ne sont pas toujours détectables par des messages d'erreur explicites.

8.4 Intégration continue

Une chaîne d'intégration continue a été mise en place afin d'automatiser la validation à chaque évolution du code. Chaque modification significative déclenche :

- la compilation du projet ;
- l'exécution des tests unitaires Java ;
- le lancement des campagnes de tests système pertinentes ;
- la génération des indicateurs de couverture.

Cette validation continue permet de détecter rapidement les régressions, de sécuriser les intégrations successives et de maintenir un état du projet cohérent tout au long du développement.

8.5 Apport complémentaire de ces méthodes

Les méthodes décrites dans cette section complètent les tests automatisés :

- la relecture de code permet de valider des propriétés structurelles et des invariants implicites ;
- l'analyse du code généré apporte une compréhension fine des erreurs de bas niveau ;
- l'intégration continue garantit une validation régulière et reproductible.

9 Limites, dette de validation et perspectives

Cette section présente les principales limites de la validation actuelle du compilateur, identifie la dette de validation accumulée au cours du projet, et propose des perspectives d'amélioration réalistes.

9.1 Limites de la validation actuelle

Malgré une infrastructure de tests complète et automatisée, certaines limites subsistent.

Couverture partielle de certains chemins rares. Certains chemins d'exécution, en particulier ceux liés à des cas d'erreur très spécifiques ou à des configurations extrêmes, sont peu couverts. Ces chemins sont difficiles à déclencher de manière contrôlée et ne correspondent pas nécessairement à des scénarios d'usage courant.

Dépendance aux tests système. Une part significative de la validation repose sur des tests système (scripts shell, exécution complète du compilateur), ce qui permet de couvrir des comportements globaux mais rend plus difficile l'isolement précis de certaines fautes internes, comparativement à des tests unitaires très fins.

Tests interactifs. Les tests impliquant des entrées utilisateur (`readInt`, `readFloat`) nécessitent une interaction explicite et sont donc moins facilement automatisables. Leur validation repose principalement sur des scénarios contrôlés et des vérifications manuelles.

9.2 Dette de validation

La dette de validation correspond aux aspects du système dont la validation pourrait être approfondie, mais qui n'ont pas été priorisés au regard des contraintes, surtout temporelles, du projet.

Tests unitaires ciblés. Certaines classes ou structures internes pourraient bénéficier de tests unitaires JUnit supplémentaires afin d'augmenter la couverture locale et de faciliter le diagnostic d'erreurs plus subtiles.

Généralisation des tests métamorphiques. Les tests métamorphiques sont actuellement concentrés sur des cas représentatifs. Leur extension à un plus grand nombre de transformations sémantiques permettrait de renforcer la robustesse de la validation sans dépendre d'oracles explicites.

Validation fine des optimisations. Les passes d'optimisation pourraient faire l'objet de tests dédiés vérifiant explicitement les invariants attendus (préservation de la sémantique, réduction effective du code, absence de régressions), au-delà des tests système actuels.

9.3 Perspectives d'amélioration

Plusieurs axes d'amélioration peuvent être envisagés pour prolonger le travail réalisé.

Renforcement de l'automatisation. L'automatisation pourrait être étendue aux tests interactifs via des mécanismes de simulation d'entrée, permettant une intégration plus complète dans la chaîne de validation continue.

Tests basés sur des propriétés. L'introduction de tests basés sur des propriétés (property-based testing) permettrait de générer automatiquement des familles de programmes respectant certaines contraintes syntaxiques et sémantiques, et d'explorer plus systématiquement l'espace des comportements possibles.

9.4 Bilan

Les limites et la dette de validation identifiées ne remettent pas en cause la validité globale du compilateur, mais reflètent des choix raisonnés effectués dans le cadre d'un projet contraint en temps et en périmètre. La stratégie adoptée privilégie une validation robuste des composants critiques, tout en laissant des perspectives claires pour des améliorations futures.

10 Conclusion

La validation du compilateur Deca s'est appuyée sur une combinaison cohérente de tests automatisés, de campagnes différentielles et métamorphiques, ainsi que de méthodes de validation complémentaires telles que la relecture de code, l'analyse du code généré et l'intégration continue. Cette approche a permis de couvrir efficacement les composants critiques du compilateur tout en assurant une bonne reproductibilité des résultats.

Les limites identifiées relèvent principalement de choix raisonnés liés aux contraintes du projet et à la complexité intrinsèque de certaines parties du système. Elles définissent une dette de validation clairement identifiée, ouvrant des perspectives d'amélioration réalistes.

Dans l'ensemble, la stratégie de validation mise en œuvre fournit un niveau de confiance satisfaisant et maîtrisé dans le fonctionnement du compilateur.

Références

[1] M. E. Delahaye, S. Souyris, et al., *Compiler Testing : A Systematic Review*, ACM Computing Surveys (CSUR), 2019. https://www.software-lab.org/publications/csur2019_compiler_testing.pdf

[2] L. Inozemtseva, R. Holmes, *Coverage is Not Strongly Correlated with Test Suite Effectiveness*, Proceedings of the International Conference on Software Engineering (ICSE), 2014. https://www.cs.ubc.ca/~rtholmes/papers/icse_2014_inozemtseva.pdf