Grenoble INP - Ensimag

Projet Génie Logiciel (GL)

Documentation de Validation

Compilateur Deca – Janvier 2026

ÉQUIPE GL56

Stratégie Test Système & Intégration

20 janvier 2026

Table des matières

1 Stratégie de Validation

Notre stratégie de validation repose intégralement sur des **tests système (fonctionnels)** et des **tests d'intégration**.

L'objectif est de valider le compilateur comme une **« boîte noire »** : nous lui fournissons des fichiers sources `.deca` et nous vérifions que le comportement en sortie (Code de retour, Sortie standard, Code Assembleur généré) correspond strictement aux attentes des spécifications.

Cette approche se décline en trois phases correspondant aux étapes du compilateur :

1. **Validité Syntaxique** (Parser / Étape A)
2. **Validité Contextuelle** (Vérificateur de types / Étape B)
3. **Validité Sémantique** (Exécution du code généré / Étape C)

2 Organisation des Tests

Nous avons conservé l'arborescence standard fournie par le squelette du projet, tout en l'enrichissant massivement avec nos propres cas de tests (dossiers `created`). Tous les tests fonctionnels sont situés dans `src/test/deca/`. Chaque étape dispose de deux sous-dossiers : `valid` (doit passer) et `invalid` (doit échouer).

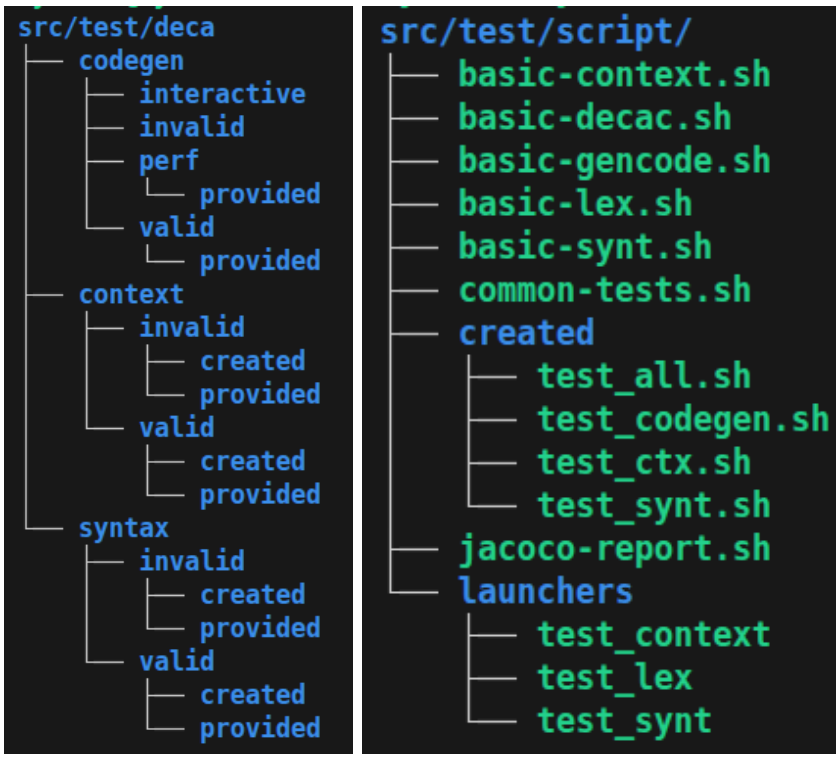

FIGURE 1 – Arborescence des tests et scripts d'automatisation

2.1 Gestion des Tests Interactifs

Les tests nécessitant une intervention utilisateur (utilisant instruction `readInt()` ou `readFloat()`) sont traités manuellement. Contrairement aux tests automatisés, ils ne sont pas lancés dans les

boucles d'intégration continue pour ne pas bloquer l'exécution des scripts, mais sont vérifiés ponctuellement avant les rendus majeurs.

3 Automatisation et Scripts

L'automatisation est le cœur de notre processus de validation. Nous avons distingué les scripts fournis par les enseignants de ceux que nous avons développés nous-mêmes pour améliorer notre productivité et la lisibilité des erreurs.

3.1 Le Script Global : `test_all.sh`

- **Emplacement :** `src/test/deca/test_all.sh`
- **Rôle :** C'est notre filet de sécurité principal. Il lance séquentiellement tous les niveaux de tests (Lexer, Syntaxe, Contexte, Gencode).
- **Politique d'équipe :** Ce script doit impérativement être lancé et réussir (**100% de succès**) avant tout `git push` sur la branche principale. Il affiche un rapport global résumant l'état de santé du projet.

3.2 Nos Scripts Personnalisés

Pour cibler les bugs plus efficacement, nous avons développé trois scripts spécifiques, situés dans `src/test/script/created/`. Ils offrent un affichage coloré et détaillé pour identifier rapidement la cause des échecs.

1. Script de Syntaxe : `synt_test`

Ce script automatise la vérification de l'option `decac -p`.

- **Entrée :** Parcourt les dossiers `syntax/valid` et `syntax/invalid`.
- **Validation :**
  - → Pour `valid` : Attend un code de retour `0` (Succès).
  - → Pour `invalid` : Attend un code de retour `1` (Erreur détectée).

2. Script Contextuel : `context_test`

Ce script automatise la vérification de l'option `decac -v`.

- **Fonctionnement :** Lance le compilateur en mode vérification seule.
- **Validation :** Il s'assure que la décoration de l'arbre (types, définitions) s'effectue sans lever d'exception interne pour les fichiers valides, et que les erreurs contextuelles (`ContextualError`) sont bien levées pour les fichiers invalides.

3. Script de Génération de Code : `gencode_test`

C'est le script le plus complet pour la chaîne de compilation.

1. Compile le fichier `.deca` en assembleur `.ass` via `decac`.
2. Lance le simulateur `ima` sur le fichier généré.

Validation : Vérifie que `ima` ne plante pas (exit code 0) et, pour les tests automatisables, compare la sortie standard avec un résultat attendu.

4 Méthodologie de Travail et Qualité

4.1 La Culture du « Test Croisé »

Afin d'assurer une couverture maximale et d'éviter les biais de confirmation (où un développeur ne teste que les cas qu'il sait gérer), nous avons adopté une règle stricte au sein de l'équipe :

> **« Chaque membre de l'équipe doit écrire des tests pour les étapes dont il n'est PAS responsable. »**

Exemple : Le responsable de l'Analyse Syntaxique (Ayoub) a écrit des tests piégeux pour la Génération de Code (Hamza/Mouad), et inversement.

Cela nous a permis de découvrir des cas limites (*Corner Cases*) inattendus que le développeur de la fonctionnalité n'avait pas envisagés initialement.

4.2 Analyse de la Couverture (Jacoco)

Bien que nous n'utilisions pas de tests unitaires Java, nous utilisons l'outil **Jacoco** (intégré via Maven) pour mesurer la couverture de code de nos tests fonctionnels `.deca`.

Cela nous permet d'identifier les portions du compilateur Java (branches , gestion d'erreurs spécifiques, exceptions) qui ne sont jamais sollicitées par nos scripts et d'ajouter les tests manquants en conséquence dans les dossiers `created`.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75 %		57 %	446	648	488	2 009	250	366	2	48
fr.ensimag.deca.tree		82 %		71 %	213	751	325	2 102	77	472	1	87
fr.ensimag.deca		48 %		50 %	57	118	160	354	19	66	2	5
fr.ensimag.deca.context		83 %		80 %	46	160	53	281	33	125	1	23
fr.ensimag.ima.pseudocode.instructions		65 %		n/a	21	62	36	111	21	62	18	54
fr.ensimag.ima.pseudocode		86 %		75 %	23	91	28	196	16	77	2	27
fr.ensimag.deca.codegen		84 %		75 %	11	29	10	59	6	19	0	2
fr.ensimag.deca.tools		94 %		100 %	1	16	3	39	1	13	0	3
Total	5 068 of 22 543	77 %	447 of 1 283	65 %	818	1 875	1 103	5 151	423	1 200	26	249

FIGURE 2 – Résultats de Jacoco

5 Gestion des risques et gestion des rendus

(Regardez le document **risk_management.pdf** sur GitLab dans le répertoire **docs**.)

6 Bilan de la Validation

Grâce à l'automatisation via `test_all.sh` et à la rigueur de nos scripts personnalisés (`synt_test`, `context_test`, `gencode_test`), nous avons pu maintenir une base de code stable tout au long du projet.

La séparation claire entre les scripts fournis (infrastructure de base) et nos propres outils (scripts colorés et précis) nous a permis d'adapter l'environnement de test à nos besoins spécifiques sans modifier l'architecture du projet.