GRENOBLE-INP ENSIMAG

INGÉNIERIE DES SYSTÈMES D'INFORMATION

2A

GRENOBLE-INP ENSIMAG

INFORMATION SYSTEMS ENGINEERING

2A

ÉCOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES DE GRENOBLE

PROJET GL

VALIDATION DU PROJET

Réalisé par :

DIAKITE Alpha OUSMANE
DREUILLE NATHAN
FLISS YASSINE
KOSSI DAVID
OUDRHIRI IDRISSI SAFWANE

CLIENT :

FREIRE MARCO

Année académique : 2025–2026

Table des matières

1 Descriptif des tests

1.1 Types de tests

Notre stratégie de validation repose sur une combinaison de tests unitaires et de tests système pour assurer la qualité et la robustesse de chaque composant du compilateur.

Tests Unitaires

Les tests unitaires sont écrits en Java avec JUnit 5 et Mockito. Ils se trouvent dans le répertoire `src/test/java/fr/ensimag/deca/`. Ces tests valident le comportement de classes individuelles de manière isolée. Par exemple, ils vérifient la logique de bas niveau des symboles, des types, et des structures de données internes.

Tests Système

Les tests système, également appelés tests de bout-en-bout, valident le comportement global du compilateur. Ils sont constitués de programmes Deca (`.deca`) situés dans le répertoire `src/test/deca/`. Ces tests sont exécutés par des scripts shell qui invoquent le compilateur `decac`, puis comparent la sortie avec un résultat attendu.

Ces tests système sont catégorisés comme suit :

- **Tests de syntaxe** : Vérifient que l'analyseur lexical et syntaxique accepte les programmes valides et rejette les programmes invalides.
- **Tests contextuels** : S'assurent que l'analyse sémantique détecte correctement les erreurs de types, les variables non déclarées, etc.
- **Tests de génération de code** : Compilent des programmes Deca, exécutent le code assembleur IMA résultant, et vérifient que la sortie est conforme aux attentes. Cette catégorie inclut :
  - **Tests valides** : Programmes corrects dont on vérifie l'exécution.
  - **Tests invalides** : Programmes qui doivent provoquer une erreur à l'exécution (ex : division par zéro).
  - **Tests interactifs** : Programmes nécessitant une entrée de l'utilisateur.
  - **Tests de performance** : Évaluent le comportement du compilateur sur des programmes plus complexes.
- **Tests d'options du compilateur** : Valident le comportement des différentes options du compilateur (ex : `-n`, `-p`, `-r`).

1.2 Organisation des tests

L'organisation des tests dans le dépôt Git est conçue pour être claire et extensible. Le répertoire principal `src/test/` est divisé en sous-répertoires qui reflètent les différentes étapes de la validation.

La structure est la suivante :

```
src/test/
|-- deca/
|   |-- codegen/
|   |   |-- interactive/
|   |   |-- invalid/
|   |   |-- perf/
|   |   `-- valid/
|   |-- compiler_option/
|   |-- context/
|   |-- syntax/
|-- java/
|   `-- fr/ensimag/deca/  (Tests unitaires)
`-- script/ (Scripts de lancement des tests)
```

Chaque répertoire de tests système (`syntax`, `context`, `codegen`) contient des sous-répertoires `valid/` et `invalid/` pour séparer les cas de test qui doivent réussir de ceux qui doivent échouer.

1.3 Objectifs des tests

L'objectif principal est de couvrir chaque fonctionnalité du langage Deca et chaque passe du compilateur.

- **Analyse Lexicale et Syntaxique** : L'objectif est de garantir que le compilateur reconnaît correctement la grammaire Deca. Les tests valides assurent que tous les aspects de la syntaxe sont pris en charge, tandis que les tests invalides vérifient que les erreurs de syntaxe sont bien détectées et signalées.
- **Analyse Contextuelle** : Ces tests visent à valider la correction sémantique des programmes Deca. Ils s'assurent que les règles de portée, de type, et d'héritage sont correctement appliquées.
- **Génération de Code** : L'objectif est de s'assurer que le code assembleur généré est correct et s'exécute comme prévu. Cela inclut la gestion de la mémoire, les opérations arithmétiques, les structures de contrôle et les appels de méthode. Les

tests d'erreurs à l'exécution (comme la division par zéro) sont également cruciaux pour valider la robustesse du code généré.

Ces objectifs sont atteints en créant une base de tests exhaustive pour chaque fonctionnalité, en s'assurant que les cas nominaux et les cas limites sont couverts.

2 Les scripts de tests

2.1 Comment faire passer tous les tests

L'ensemble de la suite de tests peut être exécuté de manière centralisée grâce à Maven. La commande suivante, lancée à la racine du projet, compile le projet et exécute tous les tests (unitaires et système) :

```
mvn test
```

Le fichier `pom.xml` est configuré pour utiliser le plugin `exec-maven-plugin` afin de lancer les différents scripts de test shell pendant la phase de test de Maven.

Il est également possible de lancer les tests pour une passe spécifique en utilisant directement les scripts shell situés dans `src/test/script/` :

- `./src/test/script/test-synt.sh` : Lance les tests de l'analyseur syntaxique.
- `./src/test/script/test-context.sh` : Lance les tests de l'analyseur contextuel.
- `./src/test/script/gencode-tests-exaustifs.sh` : Lance les tests de génération de code.

3 Résultats de Jacoco

Nous utilisons Jacoco pour mesurer la couverture de notre code par les tests. Le plugin Maven de Jacoco est configuré dans le `pom.xml`.

Pour générer le rapport de couverture, il faut exécuter la commande Maven suivante :

```
mvn jacoco:report
```

Oubien :

```
mvn verify -Djacoco.skip=false
```

Le rapport est ensuite disponible dans le répertoire `target/site/jacoco/index.html`. Ce rapport nous permet d'identifier les parties du code qui ne sont pas suffisamment testées et d'ajouter de nouveaux tests pour améliorer la couverture.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		76 %		57 %	466	677	472	2 107	241	369	0	49
fr.ensimag.deca.tree		73 %		62 %	299	834	516	2 091	133	547	1	86
fr.ensimag.ima.gencodeobjet		51 %		38 %	27	36	56	131	0	6	0	2
fr.ensimag.deca		78 %		71 %	40	126	69	316	17	80	0	5
fr.ensimag.deca.context		81 %		61 %	40	151	43	271	27	130	0	21
fr.ensimag.ima.pseudocode.instructions		57 %		0 %	28	64	51	118	27	63	20	54
fr.ensimag.ima.pseudocode		81 %		77 %	25	87	36	184	20	76	2	26
fr.ensimag.deca.codegen		67 %		44 %	12	34	19	74	4	25	0	3
fr.ensimag.deca.tools		98 %		90 %	1	22	0	45	0	17	0	3
Total	5 643 of 22 646	75 %	551 of 1 377	59 %	938	2 031	1 262	5 337	469	1 313	23	249

FIGURE 1 – Rapport de couverture de code Jacoco. On observe une bonne couverture des passes principales du compilateur, notamment la génération de code, l'analyse contextuelle et la syntaxe. Les parties moins couvertes correspondent souvent à du code généré par ANTLR dont nous ne pouvons tester même avec JUnit.

4 Démarche d'Optimisation et Corrections de Bugs

Un effort particulier a été consacré à l'optimisation du code généré, notamment sur l'utilisation des registres. Par exemple, pour éviter de charger inutilement une valeur immédiate dans un registre lorsqu'elle est utilisée comme seconde opérande, nous avons modifié la génération de code pour utiliser directement les instructions avec opérandes immédiates (par exemple `ADD valeur, Ri`).

Cette démarche d'optimisation a eu un impact significatif sur la base de code existante. Elle a nécessité de nombreuses corrections, modifications et adaptations dans la passe de génération de code. Un audit complet du code a été entrepris pour garantir la robustesse de ces nouvelles implémentations.

Cependant, le rendu final du projet a été effectué avant la fin de cet audit exhaustif. De ce fait, certains bugs ont été identifiés après le rendu, puis corrigés. Cette démarche de transparence et de correction continue est essentielle à nos yeux. Parmi les bugs notables, on peut citer :

- **Écrasement du registre R0** : Lors de calculs complexes impliquant des appels de méthode, comme `int x = A.getX() / 3;`, le registre `R0` (utilisé pour les valeurs de retour) pouvait être écrasé prématurément. Ce bug a été corrigé en s'assurant de sauvegarder la valeur de retour avant d'effectuer d'autres opérations.
- **Réinitialisation du gestionnaire de registres** : Le gestionnaire de registres n'était pas réinitialisé après la génération de code pour la déclaration des variables dans le `main`. Cela provoquait des erreurs dans l'allocation des registres pour les instructions suivantes.
- **Estimation de la taille de la pile** : Notre estimation initiale de la taille de la pile pour les appels de fonction était incorrecte, car nous avions sous-estimé le comportement de l'instruction `BSR`, qui effectue deux empilements (adresse de

retour et base du tas). L'estimation a été mise à jour pour refléter ce comportement, garantissant ainsi une allocation de pile correcte.

— **Appel de méthode sans `this`** : Initialement, notre compilateur exigeait l'utilisation explicite du mot-clé `this` pour appeler une méthode à l'intérieur de sa propre classe (ex : `this.maMethode()`). Cette contrainte a été levée ; le compilateur gère désormais correctement les appels implicites (ex : `maMethode()`), ce qui rend le code plus intuitif et aligné avec les standards de la programmation orientée objet.

5 Gestion des risques et Gestion des rendus

5.1 Analyse et gestion des risques

La complexité du projet Deca impose une vigilance constante. Nous avons identifié plusieurs risques critiques, classés par impact, et défini des mesures d'atténuation concrètes.

— **Régression fonctionnelle :** Lors de l'ajout de nouvelles fonctionnalités (notamment le passage à la partie Objet), le risque de briser l'existant est élevé. *Action :* Nous prévoyons une mise en place d'une **Intégration Continue (CI)** via GitLab CI. Chaque *commit* déclenchera une *pipeline* de tests automatisés (runners) vérifiant que le compilateur passe toujours la base de tests de non-régression.

— **Erreurs d'interprétation de la grammaire :** Une mauvaise lecture du sujet peut mener à autoriser des syntaxes interdites ou inversement. *Action :* Organisation de **sessions de transfert de connaissances** hebdomadaires. L'objectif est de confronter les implémentations de chaque binôme à l'esprit critique du reste du groupe et d'enrichir mutuellement nos batteries de tests.

— **Incohérence des fichiers de référence (`.res`) :** Un risque majeur est de valider un comportement erroné en écrivant un fichier de sortie attendue incorrect. *Action :* **Double vérification (Peer-review)** : les fichiers de résultats attendus pour les tests complexes doivent être validés par un membre n'ayant pas codé la fonctionnalité.

— **Négligence de la documentation technique :** Le retard dans la documentation peut mener à une perte de connaissance interne ou à l'oubli de détails d'implémentation cruciaux pour le rapport final. *Action :* La documentation est utilisée comme **support des transferts de connaissances**. Une fonctionnalité n'est considérée comme "terminée" que si son paragraphe dédié dans la documentation est à jour.

— **Défaut de synchronisation Git (Oubli de `git add`) :** Travailler sur un code qui compile localement mais qui est incomplet sur le dépôt distant. *Action :* Utilisation de la procédure du **"Clone de vérification"** avant chaque rendu majeur (cloner le projet dans un répertoire neutre et lancer un `mvn test`).

— **Conflits de périmètre entre étapes (A, B et C) :** Risque de "zone grise" où chaque binôme pense que la responsabilité d'un contrôle (ex : `ConvFloat`) incombe à une autre étape. *Action :* **Réunions de synchronisation d'interface** lors des transferts de connaissances pour figer explicitement les responsabilités de chaque nœud de l'arbre entre l'étape B et C.

— **Manquement aux échéances administratives :** Oubli de l'envoi des documents après un suivi ou retard de rendu. *Action :* Programmation systématique d'une **réunion de bouclage** 24h avant chaque échéance pour valider l'intégralité du package de rendu (code, tests et documents).

— **Défaillance du matériel de projection :** Risque d'impossibilité de projeter la démonstration ou de panne technique durant la soutenance. *Action :* Mise en place d'un **dispositif de secours à deux postes** : utilisation de deux PC synchronisés, l'un pour le présentateur et l'autre pour le client, afin de garantir la visibilité de la démonstration même en l'absence de projecteur.

— **Répercussions de modifications mineures :** Une modification simple peut introduire des régressions subtiles dans la génération de code. *Action :* Tout changement, même mineur, doit entraîner une **révision complète du code associé** et une analyse rigoureuse de l'assembleur généré pour s'assurer de la stabilité du comportement du compilateur.

— **Non-conformité des tests de validation :** Risque que les tests de non-régression ne reflètent plus l'état attendu du compilateur final. *Action :* **Vérification croisée de la validité** : les membres de l'équipe responsables des différentes étapes (A, B et C) doivent valider systématiquement que tous les tests présents dans les répertoires `valid` passent correctement avec le compilateur final.

5.2 Gestion des rendus et procédure de mise en production

Pour garantir la qualité de nos livrables et éviter les erreurs critiques de dernière minute, nous avons instauré une procédure rigoureuse de *Release Management*. Chaque rendu est conditionné par la validation des étapes suivantes :

1. **Nettoyage et revue du code :** Nous procédons à une suppression systématique des commentaires de débuggage (*print*) et des portions de code mortes. Parallèlement, nous enrichissons la *Javadoc* et ajoutons des commentaires explicatifs sur les algorithmes complexes pour faciliter la maintenance future.
2. **Gel du code (*Code Freeze*) :** Une heure de "gel" est fixée avant l'échéance. Durant cette période, plus aucune fonctionnalité n'est ajoutée. L'objectif est de stabiliser la version et de ne réaliser que des tests de bon fonctionnement.

3. **Cycle Maven complet :** Nous exécutons systématiquement la commande `mvn clean compile package`. Le `clean` garantit que nous ne travaillons pas sur des résidus d'anciennes compilations, tandis que le `package` vérifie que le binaire final est prêt à l'emploi.
4. **Procédure du "Clone Neutre" :** Pour s'assurer que le dépôt Git est complet, nous effectuons un `git clone` du projet dans un répertoire temporaire, totalement indépendant de notre espace de travail habituel. Nous y relançons les tests pour valider qu'aucun fichier n'a été oublié (`git add`).
5. **Validation sur l'environnement de référence :** Le projet est testé sur les machines de l'Ensimag. Cette étape est cruciale pour garantir la portabilité du compilateur et s'assurer qu'il se comporte de la même manière sur l'environnement de correction.
6. **Audit de la documentation et des manuels :** Nous vérifions que chaque fonctionnalité implémentée possède sa documentation de conception associée. Les manuels utilisateur et technique sont mis à jour et les versions PDF sont générées et placées dans le répertoire `/docs`.
7. **Vérification des options d'interface :** Nous testons manuellement la réaction du binaire `decac` face aux différentes options (`-b`, `-p`, `-v`, etc.) pour vérifier la conformité avec le sujet.
8. **Commit de clôture :** Une fois la checklist validée, nous effectuons un dernier *commit* dont le message explicite indique que la version est prête pour le rendu. Cela marque la fin officielle du cycle de développement pour le jalon concerné.