Documentation sur la Validation

Malo Nicolas - Mathéo Dupiat - Baptiste Le Duc - Ryan El Aroud - Théo Giovinazzi

January 23, 2025

Contents

1 Introduction

L'objectif de ce document est d'expliquer notre démarche sur la mise en place de tests lors de ce projet. Les tests sont centraux pour deux raisons principales. Premièrement, ils permettent un bon développement du logiciel. En effet, ils garantissent que chaque composant du projet fonctionne conformément aux spécifications et que l'ensemble respecte les spécifications. Deuxièmement, les tests générés servent de base de tests servant à la validation globale du compilateur Deca. Cet objectif fixé, il est nécessaire de se focaliser sur des fonctionnalité opérationnelles, tout en répertoriant le maximum d'erreurs possibles. Ce document a pour objectif de détailler les méthodes, outils et résultats utilisés pour assurer la qualité du compilateur développé.

Précisément, ce document permet :

- La reproduction du processus de test dans sa globalité en détaillant la structure des tests, leur forme et leur rôle
- la compréhension des différents scripts de tests, leur rôle, et comment incorporer de nouveaux tests à ces scripts
- l'assimilation des outils utilisés pour valider le fonctionnement du compilateur

2 Description des tests

2.1 Types de tests pour chaque étape/passe

Les tests sont classé selon quatre catégories suivant la structure du projet :

- Partie A sur la syntaxe hors-contexte
- Partie B sur la syntaxe contextuelle
- Partie C sur la génération du code machine
- Partie sur la décompilation

Tous nos tests sont des fichiers **.deca**. Chaque partie suit une logique spécifique et se focalise sur des aspects particuliers. Il est possible de retrouver plusieurs tests similaires, mais l'objectif était de rendre chaque partie indépendante afin de minimiser le risque d'oublis.

Une particularité concerne les tests interactifs, qui nécessitent l'intervention de l'utilisateur pour être exécutés, comme pour les fonctionnalités `ReadInt()` ou `ReadFloat()`. Les tests associés à ces fonctionnalités sont placés dans un répertoire dédié.

2.2 Organisation des tests

L'organisation des tests s'appuie sur la structure définie en début du projet, complétée en fonction de nos besoins.

```
../src/test/deca
 codegen
    invalid
    perf
       provided
    valid
        interactive
        provided
           not_done
        test_arithmetic
        test_cast
        test_class
        test_if
        test_while
            infinite_loop
 context
    invalid
       provided
       test_class
    valid
        provided
        test_class
 decompile
    valid
 syntax
     invalid
```

```
    lexer
    provided
valid
    provided
```

On trouve des sous-dossiers permettant d'organiser les batteries de tests dans les dossiers `valid/` et `invalid/`.

2.3 Objectifs des tests

Afin de vérifier différents aspects d'un compilateur **deca**, les tests ont des objectifs variés, et les commandes éxecutées varient pour chaque dossier. Tous les tests sont formatés de la même façon. Pour chaque fichier **.deca**, on retrouve :

- un nom de fichier décrivant l'objectif du test
- un en-tête au fichier de la forme:

// Description:
// <Description of the test>
//
// Results:
// <For tests that produce terminal output, include the expected result for comparison or "Error" if the test produce an error message>
//
// History:
// Created on <date>

Petite note au sujete des test JUnit : Nous avons très peu utilisé les tests unitaires pour chaque fonctions. Cela est dû à la difficulté de créer des arbres syntaxiques à la main. Nous nous sommes donc concentrés sur des test en **.deca**. Cela a également permis de se rendre compte d'erreurs d'implémentation lors des étapes ultérieures.

2.3.1 Tests sur la syntaxe

Pour les tests de la **Partie A**, les tests sont exécutés avec la commande `test_lex` et la commande `test_synt`. Ce sont des tests qui s'arrêtent après l'étape du lexer, respectivement parser dans le processus de compilation.

1. Valid: L'objectif des tests valid est de vérifier d'un que le lexer reconnait bien les différents type de token, spécifiquement pour pour vérifier le bon fonctionnement des commentaires, des nombres, de l'import,... Chacun de ces tests vont donc deux sorties différentes :

    - une pour la reconnaissance des tokens après l'étape du lexer
    - une pour la construction de l'arbre syntaxique non décoré on viendra stocker ces résultats dans 2 fichiers différents. Ce point est détaillé ici.

2. Invalid: On va principalement venir tester les messages d'erreur renvoyés par le parser concernat un **integer** ou un **flottant** incorrect.

2.3.2 Tests sur la syntaxe contextuelle

Pour la **Partie B**, les tests sont exécutés avec la commande `test_contex`. Ce sont des tests qui arrêtent le processus de compilation après la décoration de l'arbre syntaxique.

1. Valid: On retrouve dans ce dossier un ensemble de test dans le but de produire des arbes syntaxiques décorés. Cela permet de vérifier que les features du langage deca sont bien fonctionelles en suivant les règles de la syntaxe contextuelle. Ils permettent aussi de voir si les définitions et les types dont les bons. Les résultats des tests sont stockés dans un fichier, ce point est détaillé ici.

2. Invalid: Ce dossier recensie le maximum d'erreur syntaxique que nous avons détecté durant tout le projet.

2.3.3 Tests sur la décompilation

1. Valid: Les tests sur la décompilation vérifient l'opion -p du compilateur qui, pour rappelle, arrête le processus de compilation après la construction de l'arbre non décoré, puis affiche un programme **deca** obtenu après la décompilation de cet arbre. Ces tests sont donc à utiliser avec la commande `decac -p`. L'objectif des tests de décompilation était de vérifier que le programme Deca produit après la décompilation correspond bien au programme initial. Nous avons conçu divers tests inspirés des cas mentionnés dans le polycopié, notamment :

    - Les instructions conditionnelles

- La visibilité des variables
- Les structures complexes de code
- Les classes et méthodes

2.3.4 Tests sur la génération de code

Pour tester la **Partie C** qui se concentre sur la génération de code machine, on utilise donc des programmes **deca** qu'on exécute avec la commande `decac` pour générer le fichier assembleur, puis on exécute ce fichier avec la commande `ima`. On a ici tout le processus de compilation, par conséquence l'ensemble des parties est testé.

1. Valid: Comme decrit précédemment, chaque test valid va générer un fichier assembleur en **.ass** qui produit un résultat lorsque exécuté par **ima**. On rappelle que si ce résultat est affiché sur la sortie standard, il doit être mentionné dans l'en-tête du fichier test dans le champ Results. Les tests avaient pour but de détecter des erreurs de :

   - Branchement (conditions if et while)
   - Overflow
   - Entrée et sortie
   - Stack overflow
   - Division par zéro
   - Erreurs arithmétiques

   Pour ce faire, nous avons généré plusieurs fichiers de tests Deca très similaire en faisant varier le moins de paramètre possible, par exemple pour vérifier les erreurs de branchement (pour while et if), nous avons utilisé les même script en variant systématiquement les valeurs (0,-1,4) et les opérateurs de comparaison (`==`, `<`, $>=$, $<$, $>$, $!=$), ainsi que les opérateurs logiques (|| et &&), totalisant 36 programmes pour if et de même pour while.

2. Invalid: Ce dossier recensie les erreurs liées à l'exéction du programme.

3 Les scripts de tests

L'ensemble des scripts de tests peuvent être exécutés dans le terminal en local mais sont forcemment lancés lors de tout push lors de l'ensemble du projet

par la pipeline gitlab. Notons qu'il était nécessaire d'avoir une pipeline qui ne fail pas pour pourvoir push dans la branch **developp** ou **main**. Pour exécuter les scripts en local, il faut se placer dans le dossier racine du projet.

3.1 Description des scripts de tests

Un ensemble de scripts **bash** est disponible dans le dossier `src/test/script`. Afin de lancer une batterie de test sur un élément en particulier du compilateur:

- `non-regression.sh` lance une batterie de test pour vérifier la non régression du code lors de l'implémentation d'une nouvelle feature. Il va venir exécuter les fichiers test donnés, puis comparer le résultat obtenu avec un fichier résultat obtenu lors d'une ancienne exécution du test dont on est sûre que le resultat est celui attendu. Cela aura été vérifié manuellement. Voir ci-dessous pour avoir une vision plus détaillé de ces fichiers résultat. Il est possible de ne pas effectuer l'ensemble des tests à l'aide d'options: `-t [testlex | testsynt | testcontext | invalidsyntax | invalidcontext | decompile | codegen]`.

- `decac-verif.sh` est un script qui exécute `decac -v` sur un ensemble de fichiers **.deca** et qui renvoie une erreur si l'exéction renvoie un résultat. Cela signifie que soit le programme **deca** n'est pas syntaxiquement correct. Soit qu'il y a un problème au niveau de l'implémentation du compilateur car l'arbre syntaxique n'est pas correctement décoré. L'objectif de ce script était principalement de vérifier qu'une décoration n'a pas été oublié lors du développement. Dans cet objectif, les tests devaient être syntaxiquement correct. Ainsi tout les tests concernant la syntax contextuelle et la génération de code sont concernés par ce script.

- `verify-ima-output.sh` ce script exécute des tests pour valider les phases de génération de code **decac** et d'exécution **ima** pour tous les fichiers **.deca** situés dans le répertoire spécifié. Il compare les sorties réelles produites par le compilateur aux résultats attendus spécifiés dans la section **Results:** de chaque fichier **.deca**. Ces tests nous on était très utile pour la dernière phase de vérification. Ils nous ont permis de nous rendre compte des derniers problèmes de notre compilateur autant sur la partie contextuelle que sur la partie génération de code.

3.2 Comment faire passer tout les tests

L'ensemble des scripts se trouvant dans le dossier `src/test/script` se lancent en effectuant la commande `mvn test.`

3.3 Ajouter un test à un des scripts

3.3.1 verify-ima-output:

On ajoute un test en ajoutant un fichier **.deca** dans le dossier `src/test/deca/codegen/valid/`. **Attention** le champ **Result** doit contenir le résultat attendu.

3.3.2 decac-verify:

Même pocédé, l'ajout de fichier **.deca** doit se faire dans le dossier `src/test/deca/context/valid/` ou dans le dossier `src/test/deca/codegen/valid/`.

3.3.3 non-regression

Le script des tests de non régression vient comparer le résultat avec un fichier contenant un résultat validé à la main par l'utilisateur. Ces fichiers résultats se trouvent dans l'arborescence ci-dessous:

```
../src/test/results/
 deca
    codegen
    context
       invalid
    decompile
    syntax
        invalid
        lex
        synt
 tmp
```

Le dossier `/tmp/` contient le résultat des tests pour le comparer avec le résultat valide. Il est ensuite supprimé. On retrouve la même arborescence que pour les tests et les fichiers résultats doivent se trouver dans le même sous-dossier. Notons qu'on ne retrouve pas les dossiers qui permettent d'organiser les fichiers test et ne doivent pas être ajoutés.

Type de test	Valid/Invalid	Extention du fichier résultat	Information
Syntax [Lexer]	Valid	**.lex**	Contient le résultat de `test_lex`
Syntax [Parser]	Valid	**.synt**	Contient le résultat de `test_synt`
	Invalid	**.err**	Contient l'erreur renvoyé par `test_synt`
Context	Valid	**.synt**	Contient le résultat de `test_context`
	Invalid	**.err**	Contient l'erreur renvoyé par `test_context`
Decompilation	Valid	**.deca**	Contient le programme **.deca** renvoyé par `decac -p`
CodeGen	Valid	**.ass**	Contient le code assembleur renvoyé par `decac`
	Invalid	**.err**	Contient l'erreur renvoyé par `decac`

4 Gestion des risques et gestion des rendus

4.1 Objectifs et importance

La gestion des risques et des rendus a pour but d'assurer que le projet progresse sans interruptions majeures, que les erreurs critiques sont évitées, et que les livrables respectent toutes les exigences spécifiées. Il s'agit de minimiser les impacts des imprévus tout en garantissant un produit final de qualité.

4.2 Analyse des risques critiques

Exemple : Oubli de fichiers critiques lors de commits Git (comme des tests ou des fichiers sources nécessaires).

Impact : Le compilateur ne fonctionne pas sur certaines étapes critiques (par exemple, la génération de code).

Stratégie d'atténuation :

- Mettre en place un processus strict de revue de commits avant chaque push (git status, git diff).
- Automatiser la vérification avec un CI/CD pipeline via GitLab qui exécute systématiquement les tests avant d'accepter un commit.

Exemple : Messages d'erreur non conformes aux spécifications (par exemple, message non explicite pour une erreur contextuelle).

Impact : Confusion chez les utilisateurs et échec des validations par les enseignants.

Stratégie d'atténuation :

- Centraliser les messages d'erreur dans une classe dédiée pour standardiser leur gestion.
- Prévoir une étape de validation manuelle dédiée à la vérification des messages d'erreur avec un programme testeur.

4.3 Gestion de la qualité des rendus

Exemple : Tests incomplets ou mal classifiés (par exemple, tests de syntaxe placés dans le répertoire context).

Impact : Difficulté à démontrer la couverture complète des tests et perte de points dans l'évaluation.

Stratégie d'atténuation :

- Audit périodique des répertoires : Avant chaque rendu, un membre de l'équipe est désigné pour vérifier que tous les tests sont bien organisés.
- Utilisation d'outils d'analyse de couverture comme Jacoco pour identifier les zones de code non testées.

Exemple : Fichiers inutiles ou obsolètes dans le dépôt Git.

Impact : Augmentation des erreurs possibles lors de l'exécution ou confusion.

Stratégie d'atténuation :

- Ajout d'une étape automatisée de nettoyage (mvn clean) dans le pipeline CI/CD.
- Documentation claire de l'arborescence dans un fichier README.

4.4 Risques liés aux échéances

Exemple : Livraison en retard de certaines fonctionnalités majeures (comme la passe de vérification contextuelle).

Impact : Impossibilité de respecter les délais des rendus intermédiaires et finaux.

Stratégie d'atténuation :

- Utiliser un tableau de gestion de tâches (GitLab Issues, Planner) pour suivre les échéances.
- Planifier des jalons intermédiaires, avec une priorité absolue donnée aux fonctionnalités de base (analyse syntaxique, génération d'arbre abstrait).

4.5 Gestion des rendus intermédiaires

Le rendu intermédiaire étant une étape clé, voici les éléments spécifiques à considérer pour cette phase :

- **Exigences minimales** :
  - Implémentation complète et fonctionnelle de la partie sans objet.
  - Base de tests exhaustive pour cette partie, avec une organisation conforme.
- **Méthodologie suggérée** :
  - Un jour avant le rendu : Effectuer une revue finale des tests et des messages d'erreur.

4.6 Risques spécifiques au rendu final

Pour le rendu final, certains risques supplémentaires apparaissent :

- Erreur dans les documentations (utilisateur, validation, extension) :
  - **Solution** : Révision croisée entre membres de l'équipe.
- Divergences de branches Git :
  - **Solution** : Geler les développements non critiques quelques jours avant la date limite et effectuer une fusion finale supervisée.

5 Résultats de Jacoco

Afin d'avoir une information plus précise sur la couverture de nos tests, l'utilisation de Jacoco via la commande `mvn verify` nous donne ce résultat :

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75%		56%	455	663	478	2,004	246	373	2	50
fr.ensimag.deca.tree		91%		87%	125	728	158	1,899	83	549	0	86
fr.ensimag.deca		69%		73%	35	116	91	308	14	70	1	5
fr.ensimag.deca.codegen		87%		82%	19	98	30	297	14	78	0	6
fr.ensimag.ima.pseudocode		84%		83%	26	106	34	215	22	94	2	26
fr.ensimag.ima.pseudocode.instructions		66%		n/a	23	62	40	111	23	62	19	54
fr.ensimag.deca.context		93%		81%	22	141	26	266	15	119	0	22
fr.ensimag.deca.tools		94%		100%	1	16	3	39	1	13	0	3
Total	3,960 of 22,725	82%	314 of 1,085	71%	706	1,930	860	5,139	418	1,358	24	252

Cet outil nous a permis de nous rendre compte des messages d'erreur qui nous manquaient dans nos tests invalides. À la suite de cet état des lieux, nous avons pu améliorer notre couverture de quelques pourcentages. En ce qui concerne le pourcentage de non-couverture, il est principalement dû à des fonctionnalités utilisées pour le débogage et pendant le codage du compilateur. Mais on retrouve aussi de la présence de code mort, en raison d'un mauvais codage de notre part, mais aussi d'un manque de temps pour nettoyer notre rendu.

6 Méthodes de validation utilisées autres que les tests

En plus des méthodes de validation expliquées précédemment, nous avons mis en place quelques autres outils pour compléter. L'élément principal est l'utilisation de la revue de code. En effet, nous avons mis en place l'utilisation de merge requests sur GitLab, ce qui était nécessaire pour pouvoir push sur la branche main ou develop. Plus précisément, il était nécessaire d'obtenir la validation d'un autre membre du groupe, qui devait revenir sur les changements appliqués. Ce procédé nous a permis de corriger des incompréhensions dans l'implémentation du compilateur, notamment dans la partie syntaxique contextuelle. Cette partie possède quelques passages assez délicats qui demandaient un gros effort de compréhension.

L'autre outil mis en place est l'organisation de l'environnement de test sur GitLab. Pour chaque nouvelle fonctionnalité, nous créions une nouvelle branche depuis la branche **developp**. Pour chaque push effectué sur une branche, cette nouvelle version passait par une pipeline de tests qui vérifiait que le build du code était possible. Nous lancions aussi les tests de non-régression. Enfin, nous vérifiions également le formatage du code à l'aide de `mvn spotless:check`. Cette pipeline ne devait pas nécessairement être validée pour push dans une branche (même si nous essayions au maximum de faire en sorte que ce soit le cas), sauf pour la branche **developp** ou la branche **main**, qui nous sert pour les releases. Cette procédure mise en place au début nous forçait à garder un code propre et accessible pour les autres membres du groupe.

Ajouté à cela, afin d'aider cette relecture, nos messages de commit devaient être dans un format particulier, sinon le push n'était tout simplement pas accepté par GitLab. Ce format est le suivant :

- un message de commit doit commencer par un préfixe:

Préfixes possibles pour ce projet :

- `feat:` : Pour une nouvelle fonctionnalité ajoutée au compilateur (e.g., une étape de compilation comme la vérification contextuelle ou la génération de code).
- `fix:` : Pour corriger un bug ou un problème identifié.
- `test:` : Pour ajouter ou améliorer des tests (unitaires ou d'intégration).
- `doc:` : Pour des modifications ou ajouts à la documentation (e.g., manuel utilisateur, documentation de conception).
- `refactor:` : Pour des modifications visant à améliorer le code sans changer son comportement.
- `build:` : Pour des modifications liées à la configuration ou aux outils de build (e.g., Maven, GitLab CI/CD).
- `perf:` : Pour optimiser les performances du compilateur ou des tests.
- `chore:` : Pour des tâches diverses comme le nettoyage de code ou l'ajout de commentaires.

Validation manuelle : Nous avons exécuté certains scénarios complexes manuellement pour vérifier des cas limites non couverts par les tests.

Analyse statique du code : Outils de linting et de formatage pour garantir la conformité aux bonnes pratiques.