Document de validation

gl22

Janvier 2024

1 Introduction

Le fonctionnement du compilateur a été validé au moyen d'un ensemble de tests unitaires et de tests système, qui ont été développés en utilisant la méthodologie du Test Driven Development. Cette approche nous a permis de valider nos implémentations de manière efficace.

1.1 Descriptif des tests:

Notre compilateur se compose de trois parties essentielles à développer : la partie syntaxique et lexical, la partie contextuelle et la partie de génération de code. Ces trois composants ont été testés de manière indépendante et différenciée. Nos tests ont été répartis entre deux outils : la bibliothèque de tests de Java, JUnit, et des scripts écrits en Bash.

La plupart des tests ont été conçus avec des scripts Bash, car ils sont jugés plus fiables et plus faciles pour tester des micro-implémentations. Ces scripts Bash s'exécutent sur des fichiers de code Deca répartis dans des répertoires bien organisés. JUnit a été principalement utilisé pour tester des fonctions censées retourner des types bien précis.

1.2 Partie lexicale et syntaxique

Les tests de la partie syntaxique ont été conçus pour évaluer l'arbre syntaxique abstrait sans décorations (AST). Dans cette phase, nous testons l'application de la grammaire sur l'AST, en nous assurant que chaque fonctionnalité apparaît à la bonne ligne et colonne de l'arbre en utilisant des scripts bash avec la commande grep. Ces tests sont disponibles dans le répertoire:

```
./src/test/script
```

Pour executer ces tests individuellement , vous pouvez passer au repertoire indique et executer ces commandes :

```
cd src/test/script
```

Et puis exécuter l'un des fichiers de test suivants :

```
./basic-lex.sh : // test basique pour la lexicographie
./basic-synt.sh : // test basique sur les branches de l'AST
./mots-reserves-test.sh : // test sur les mots réservés du langage deca
./test-lexer-in-general.sh : // teste la lexicographie d'un grand
programme.
```

1.3 Partie lexicale et syntaxique : Decompilation

Pour tester la décompilation des programmes Deca, nous avons mis en place des tests visant à vérifier que la décompilation d'un fichier déjà décompilé conservait la grammaire et, par conséquent, restait équivalente. Les tests ont été développés en utilisant des scripts Bash. Chaque script de test de décompilation fonctionne de la manière suivante : il décompile un programme Deca, écrit le résultat dans un fichier, puis décompile à nouveau ce fichier et compare le résultat avec le contenu du premier fichier en utilisant la commande Unix:

```
diff
```

Les tests de décompilation ont été organisés en deux catégories distinctes : l'une se concentrant sur la décompilation de la partie sans objet, et l'autre incluant la décompilation des classes, des attributs et des méthodes. Pour lancer ces tests, veuillez rester dans le même répertoire précédent et exécuter les commandes suivantes :

```
./test-decompile-sans-objets.sh: // test de la Partie Sans Objets
./test-class-decompile: // test de la Partie Objets
```

1.4 Partie Contextuelle

Les tests de la partie contextuelle ont pour objectif de vérifier la décoration des différentes branches de l'arbre abstrait après les trois passes Nous utilisons le lanceur de tests:

```
  test_context
```

sur deux types de fichiers de tests Deca : des tests invalides, où nous vérifions qu'une erreur est déclenchée, et des tests valides, où nous vérifions l'apparition des branches sur l'AST au bon emplacement (ligne et colonne). Pour exécuter les tests contextuels individuellement, veuillez vous rendre dans le répertoire mentionné dans la section sur les tests lexicaux et syntaxiques, puis exécuter les commandes suivantes :

```
./basic-context.sh : // tests basique sur la partie non Objets
./test-errors.sh : // tests pour verifier le declenchement d'erreurs
./test-general-ast.sh: // tests qui verifie l'arbre du programme
  inclue dans le polycopié
```

1.5 Partie Génération de Code

Pour la partie de génération de code, nous avons rédigé des tests qui compilent et exécutent des fichiers Deca, puis comparent les résultats d'exécution pour s'assurer du bon fonctionnement du compilateur. De plus, ces tests permettent de vérifier le déclenchement d'erreurs telles que la division par 0 et le débordement de float. Pour exécuter ces tests individuellement, veuillez vous placer dans le répertoire indiqué dans les parties précédentes et exécuter les fichiers suivants :

```
./basic-gencode.sh : // Des tests basique sur la partie Non objets
./test-class-gencode : // Des tests sur la partie objets
```

1.6 Test divers

L'utilisation de la méthodologie Test Driven Development (TDD), qui consiste à développer des tests avant la mise en œuvre des fonctionnalités, a été intégrée dans notre processus de développement pour garantir le bon fonctionnement des différentes fonctionnalités implémentées. Pour assurer l'organisation de nos tests par rapport aux fonctionnalités développées, nous avons conçu des tests ciblant des fonctionnalités spécifiques. Voici les tests :

```
// Test de l'implémentation de if-else sur l'AST :
./test-ifelse.sh

// Test de l'implémentation de New :
./test-new.sh

// Tests des fonctions interactives (readInt() et readFloat) :
./test-interactive.sh

// Test de l'exécution de la partie sans objets entièrement :
./test-general-sans-objet


// Test de la partie contextuelle de la déclaration de classe :
./test-declaration-class
```

Ces commandes permettront d'exécuter des tests spécifiques pour chaque fonctionnalité, assurant ainsi que les implémentations répondent aux attentes. En suivant cette approche TDD, nous avons renforcé la robustesse de notre compilateur en nous assurant que chaque fonctionnalité est accompagnée de tests appropriés.

Il ya aussi le teste *general* qui permet de tester tous les fichiers du le dossier:

```
/src/test/deca/codegen/valid/unprovided/general_tests
```

Est de comparer le resultat avec le fichier correspondant dans

`src/test/deca/codegen/valid/unprovided/expected`

De cette façon nous pouvons integrer tous les fichiers teste sans avoir recours à modifier le script shell.
Ce script nous a permis de facilement ecrire les test pour otim vu que ces derniers prenaient du temps et devaient etre automatisé.

1.7 Test Junit

Au cours du développement, nous avons également conçu des tests JUnit visant à évaluer les types retournés par certaines implémentations spécifiques. Plus précisément, ces tests portent sur les types de retour des opérations arithmétiques telles que l'addition, la soustraction, la multiplication et la division, ainsi que sur l'implémentation de la fonctionnalité convFloat, permettant la conversion d'un entier (Int) en un flottant (float). Les tests JUnit se trouvent dans le répertoire :

`./src/test/java/fr/ensimag/deca/context`

Ces tests implémentent des mocks de la bibliothèque Mockito, lesquels servent à rendre l'exécution abstraite des différentes parties que nous ne souhaitons pas tester, mais qui représentent des dépendances. En utilisant Mockito, nous pouvons isoler les parties spécifiques que nous testons, rendant ainsi nos tests plus ciblés sur les fonctionnalités individuelles tout en assurant une couverture adéquate.

1.8 Jacoco: Couverture des tests

Jacoco a été employé pour surveiller la couverture des tests tout au long du projet et identifier les parties du code non testées. Nous avons réussi à atteindre une couverture de 76 % pour l'ensemble du code. Notre méthode consistait à rédiger des tests ciblant des parties spécifiques du code. Cependant, nous avons rencontré un problème avec Jacoco lors de l'ajout de nouveaux tests, où la couverture du code diminuait pour certains fichiers. En conséquence, nous avons atteint une couverture maximale de 76%, qui même variait d'un membre de l'équipe à l'autre.

1.9 Lancer les tests

Notre projet est structuré de manière à permettre la validation de toutes ses composantes par le biais d'une commande unique, déclenchant l'exécution de chaque bloc de tests avec un arrêt immédiat en cas d'échec. Pour initier ces tests, vous avez à votre disposition deux commandes après vous être positionné dans le répertoire racine avec la commande 'cd ./gl22' :

```
mvn test
```

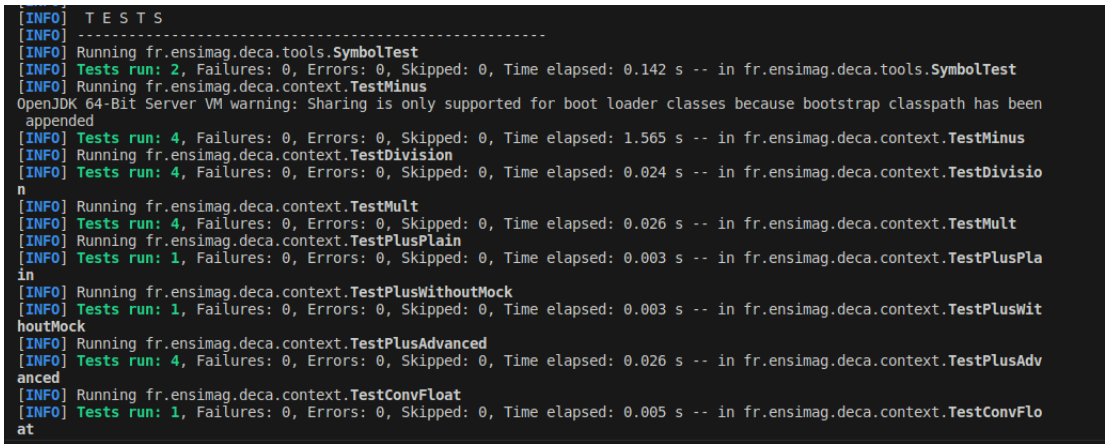

Figure 1: Execution des test Junit.

Cette commande lance les tests sans effectuer le calcul de la couverture des tests.

```
mvn verify
```

Cette commande lance les tests tout en calculant la couverture des tests à l'aide de Jacoco.

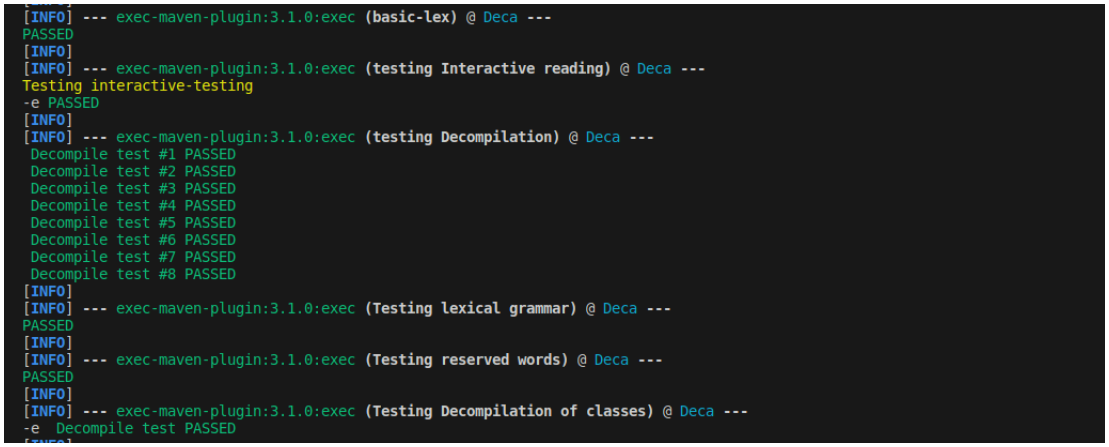

Figure 2: Execution de quelques tests.

Gestion des risques et gestion des rendus

Groupe:	Date:
GI22	22/01/2024

Type de Risque

	Conception de code
	Rendue
	Gestion d'équipe
	développement
	git

PERSONNE(S) IMPACTÉE(S)

	Clients
	Membres de l'équipe

NIVEAU DE GRAVITÉ DU RISQUE

	FAIBLE
	FAIBLE / MOYEN
	MOYEN
	MOYEN / ÉLEVÉ
	ÉLEVÉ

IMPACT DU RISQUE

	Niveau d'impact	DESCRIPTION
	FAIBLE	N'a aucune conséquence sur les autres parties du projet.
	FAIBLE/MOYEN	Possède des conséquences mineures sur le projet.
	MOYEN	A des impacts moyens sur le projet qui peuvent être résolus en moins d'une journée de travail.
	MOYEN / ÉLEVÉ	A des impacts majeurs sur le projet qui peuvent être résolus en moins d'une journée de travail, mais nécessitent l'intervention de plusieurs membres.
	ÉLEVÉ	Difficile à résoudre et impacte le parcours de développement.

Bilan des Risques

RISQUE	Type de Risque	PERSONNE(S) IMPACTÉE(S)	NIVEAU DE GRAVITE
Oublier l'implémentation complète d'un type primitif ou d'une fonctionnalité.	Conception de code	Membres de l'équipe	FAIBLE/MOYEN
Avoir un programme ayant un grand impact énergétique.	développement	Clients	MOYEN
Oublier un document à rendre.	Rendue	Clients	FAIBLE
Oublier un push sur la branche Master.	git	Clients	ÉLEVÉ
Manquer une partie de code non testée.	développement	Membres de l'équipe	MOYEN
Plusieurs membres travaillent sur la même partie sans être informés de l'activité de chacun.	Gestion d'équipe	Membres de l'équipe	MOYEN/ÉLEVÉ
Dépasser les dates de remise.	Rendue	Clients	ÉLEVÉ

ACTIONS À METTRE EN ŒUVRE

RISQUE	ACTION
Oublier l'implémentation complète d'un type primitif ou d'une fonctionnalité.	Assigner cette tâche à un membre de l'équipe qui possède la meilleure connaissance à ce sujet.
Avoir un programme ayant un grand impact énergétique.	Examiner les parties du code susceptibles d'avoir un impact énergétique élevé en utilisant une stratégie d'isolation du code.
Oublier un document à rendre.	Préparer les documents à rendre en parallèle avec le développement du projet.
Oublier un push sur la branche Master.	Commencer à intégrer les parties totalement correctes dès qu'elles sont vérifiées, même si elles ne sont pas complètes.
Manquer une partie de code non testée.	Effectuer un bilan des tests avant le developement .