Projet Génie Logiciel

Documentation de validation

ENSIMAG 2ème année - 2022-2023

Groupe 1 - équipe 3

Jorge Luri Vañó
Nils Depuille
Vianney Vouters
Virgile Henry
Logan Willem

I. Descriptif des tests

A. Types de tests pour chaque étape

Pour l'étape A on distingue deux types de tests différents, des tests unitaires, concernant les tests du lexer, et des tests d'intégration pour les tests du parser.

L'étape B comporte aussi des tests unitaires (83 pour être précis) utilisant à la fois JUnit pour les tests unitaires et l'outil Mockito permettant d'isoler la classe que l'on souhaite tester en créant *mocks*. Ces tests ne concernent que la partie sans-objet et permettent de tester des instructions simples comme les comparaisons, les opérations sur booléens, les opérations arithmétiques et les opérations unaires, les différentes fonctions `print` et les fonctions `readInt` et `readFloat`. Par la suite, comme le lexer et le parser étaient bien implémentés et testés tôt dans le projet, seuls des tests d'intégration ont été créés que ce soit pour la partie avec et sans objet. Chacun de ces tests avaient pour but précis de tester une règle de la grammaire, en s'appuyant à chaque fois sur l'arbre créé par le parser. Ces tests pourraient aussi être considérés comme des tests boîte noire.

La partie C s'appuie quant à elle sur des tests système puisqu'ils utilisent l'entièreté du compilateur, toutes ses composantes.

Il en est de même pour les fichiers de tests pour la partie optimisation.

Notons que tous ces tests peuvent être considérés comme des tests de non régression puisque notre but était de conserver le caractère correct de notre compilateur sur des tests qui ont réussis à passer une première fois.

B. Organisation des tests

Voici l'arborescence de notre dossier de tests:

```
src/test/deca
├── codegen
│   ├── interactive
│   │   └── demo-suivi-3
│   ├── invalid
│   ├── perf
│   │   └── provided
│   └── valid
│       ├── objet
│       └── provided
├── context
│   ├── invalid
│   │   ├── objet
│   │   └── provided
│   └── valid
│       ├── objet
│       └── provided
├── optim
│   ├── collapse
│   ├── facto
│   ├── Redondant
│   │   └── valid
│   └── unused_var
└── syntax
    ├── invalid
    │   ├── lexer
    │   │   └── include_tests
    │   ├── provided
    │   └── synt
    └── valid
        ├── provided
        └── synt
```

Chaque partie du projet est représentée par un dossier de test. La partie A a donc pour dossier **syntax**, la partie B **context** et enfin la partie C **codegen**. L'extension OPTIM a elle aussi droit à son dossier: **optim**.

Chacun de ces dossiers - excepté **optim** - est divisé en deux sous dossiers: **valid** et **invalid**. Un test situé dans **syntax/invalid** (resp. **context/invalid**) signifie qu'il est syntaxiquement (resp. contextuellement) invalide, tandis que les tests situés dans **codegen/invalid** génèrent une erreur à l'exécution et non à la compilation.

Le dossier **codegen** comporte d'autres sous dossiers: **interactive** regroupant des tests valides, comme les démonstrations utilisées aux deux derniers suivis entre autres, mais nécessitant une entrée de la part de l'utilisateur dû à un readInt ou readFloat et le dossier **perf** regroupant les tests fournis et des tests personnels utiles pour tester les performances du code assembleur généré par notre compilateur.

Le dossier **optim** contient un dossier par optimisation de notre compilateur. On y retrouve donc l'optimisation de suppression de variables inutilisées avec le dossier **`unused_var`**, celle de suppression des calculs redondants dans **`Redondant`**, celle de factorisation puis décomposition en somme de facteur de puissance de deux : **`facto`**, et enfin celle de simplification des calculs évidents **`collapse`**.

Enfin, le dossier **syntax** est un peu plus complexe: le dossier **valid** possède en plus du dossier **provided** un dossier **synt** des tests syntaxiquement valides tandis que le dossier **invalid** a un dossier **synt** pour les tests syntaxiquement incorrects et **lex** pour les tests donnant un DUMMYTOKEN après passage du lexer.

Il est à noter que les trois dossiers **`objet`** sont des restes d'une organisation antérieure du dossier de tests. Même s'il reste quelques tests dans ceux-ci, ils ne sont pas exécutés par nos scripts de test lors de l'exécution de la commande `mvn test`.

C. Objectifs des tests

L'utilité des tests est multiple:

- Ils permettent avant tout de tester chaque fonctionnalité implémentée et vérifier que les cas d'erreur renvoient bien une erreur tandis que les cas valides génèrent bien le résultat attendu. Nous avons essayé tout au long du projet de suivre la méthode de développement dirigé par les tests (ou Test-Driven Development - TDD) ce qui nous a permis de travailler par petits impléments que l'on pouvait directement tester une fois que l'on considérait que l'implémentation était terminée.
- Une fois qu'un test passe une première fois, que le résultat renvoyé est correct, il s'agit de conserver cet état. Ainsi, les tests vont aussi avoir un rôle de maintenance. Il est donc utile de lancer une batterie de tests simples et peu coûteux lorsqu'un implément nécessite une modification plus ou moins conséquente d'une partie du code écrite en amont.
- Certains tests ont aussi un rôle d'évaluation des performances (cf. dossier **perf**) permettant de monitorer les performances de notre compilateur, en particulier lorsque les impléments ont pour but d'améliorer les performances.
- Enfin, les tests, conjugués avec l'outil de couverture de code JaCoCo, permettent d'avoir une vision plus ou moins concrète de la suffisance de nos tests. Le but n'étant pas de dresser une liste exhaustive de tests permettant de traiter tous les cas imaginables mais au moins pouvoir tester toutes les instructions. Ainsi, cette association test-JaCoCo permet de ne pas omettre certains branchements permis par notre code.

Pour résumer, les objectifs des tests sont de pouvoir s'assurer de la correction de nos fonctionnalités et de maintenir cette correction dans le temps. Afin d'avoir une batterie de tests des plus utiles, l'association à l'outil JaCoCo permet d'établir la couverture de nos tests et d'en rajouter en cas d'oublis. Enfin, certains tests permettent aussi un suivi des performances.

II. Scripts de tests

A. Scripts de tests

Partie A	Partie B	Partie C	Extension	Autre
basic-lex.sh basic-synt.sh full-lex.sh full-lex-diff.sh full-synt.sh	basic-context.sh full-context.sh	basic-gencode.sh full-codegen.sh	diff-optim.sh optim-compare.sh full-optim.sh	basic-decac.sh common-tests.sh JaCoCo-report.sh compiler-options.sh

Liste des scripts de tests présents dans le dossier **script**

Les scripts en rouge dans le tableau sont ceux fournis à l'origine. Rien ne sera donc expliqué à leur sujet puisqu'ils restent inchangés. Les autres sont donc des scripts créés par nos soins.

Pour la partie A, trois scripts ont été ajoutés:

- `full-lex.sh` n'est plus utilisé désormais mais permettait à l'origine de générer les fichiers de correction des tests du lexer. Ces fichiers étaient bien entendu vérifiés un par un par la personne chargée de l'implémentation du lexer.
- `full-lex-diff.sh` quant à lui est bien utilisé et permet de comparer les `.lis` générés avec les fichiers de nom `<nom_de_fichier>-lex.corr` correspondant à la correction. Ce script comme le précédant s'appuient sur le launcher fourni du nom de `test_lex`. Ce script teste les fichiers présents dans les dossiers **`provided`** et **`lexer`** du dossier **`syntax`**.
- `full-synt.sh` s'appuie quant à lui sur le launcher fourni `test_synt` et test ainsi les fichiers présents dans les dossiers **`provided`** et **`synt`** du dossier **`syntax`**. Pour les tests valides, on test à ce qu'aucune erreur n'est renvoyée, tandis que pour les tests invalides on cherche à trouver le nom du test, le mot "java" ou le terme "mismatched" dans la sortie de la commande `test_synt` sur le fichier en question.

Pour la partie B, seul le script `full-context.sh` a été créé puisque celui-ci permet dans un premier temps d'exécuter le launcher fourni `test_context` sur les fichiers du dossier **valid** en vérifiant que le code de retour de la commande est bien 0. Puis, ce même launcher est utilisé sur les tests du dossier **invalid** et vérifie que la sortie de la commande contient l'expression régulière suivante: "`<nom_du_test>:*:*`" qui est le format attendu des erreurs de la partie contextuelle.

La partie C n'a elle aussi été complétée que d'un script: `full-codegen.sh`. Celui-ci ne se base sur aucun launcher puisque cela revient à simplement exécuter la commande `decac`. Ce script est un peu plus complet et test chaque fichier valide en deux temps. Tout d'abord, on vérifie que chaque test valide ne renvoie pas ni son nom, ni l'expression "Exception in thread" lors de la compilation. Une fois que ce test passe, le fichier assembleur généré est exécuté avec la

commande `ima` et le résultat est comparé au fichier de correction ayant pour nom `<nom_du_fichier>.res`. Si aucune différence n'est trouvée, le test passe. Cette seconde étape est omise pour les tests présents dans le dossier **`interactive`**. Enfin, pour les tests dans le dossier **`invalid`**, l'exécution renvoyant un code différent de 0 est considéré comme un test passé.

Il est à noter que les quatre derniers scripts présentés renvoient un pourcentage de succès des tests passés et permettent ainsi d'un coup d'œil de vérifier que tout est en ordre.

Pour l'extension, trois scripts ont été créé:

- `diff-optim.sh` n'est plus utilisé mais permettait à l'origine de générer un fichier `.decomp` permettant de visualiser l'effet de l'optimisation sur le code. Ce script a été substitué à la commande `decac -o -p` permettant directement de renvoyer le code optimisé puis décompilé dans la console.
- `optim-compare.sh` permet de visualiser l'effet de l'optimisation sur tous les scripts du dossier **`optim`**. On y voit ainsi facilement la différence de nombre d'instructions et de nombre de cycles entre une exécution sur un code fourni et un code optimisé.
- `full-optim.sh` quant à lui fonctionne en plusieurs étapes. Pour chaque test présent dans le dossier **`optim`** on vérifie que la compilation sans l'option `-o` fonctionne, puis avec. Ensuite, on vérifie que l'exécution du code optimisé ne renvoie pas d'erreur et enfin on regarde si le fichier assembleur optimisé comporte moins de lignes que celui non optimisé.

`compiler-options.sh` teste toutes les options du compilateur. Ce script permet l'exécution de tous les scripts présents dans le dossier **`script/options-scripts`**. On y retrouve un script par option du compilateur auquel on a ajouté le script `other-cases` permettant de tester des cas annexes du compilateur: lancer la commande `decac` sans rien spécifier, avec une option inconnue, avec un fichier n'étant pas un `.deca` ou encore un fichier qui n'existe pas. Chacun de ces scripts teste donc les options telles qu'elles sont demandées dans le cahier des charges en testant aussi leurs cas d'échec (option `-v` avec l'option `-p` par exemple). Les messages renvoyés par ces scripts sont assez explicites et permettent de bien savoir ce qui est testé.

Comme dernière remarque, on peut ajouter que chaque script de tests supprime si nécessaire, les fichiers temporaires créés pour faire des vérifications. Cela évite d'être encombré de fichiers assembleurs ou autre.

En ce qui concerne les scripts de tests donc, il n'est normalement pas nécessaire d'en rajouter puisque ceux-ci permettent de tester une partie, ou tout le compilateur directement.

B. Comment lancer les scripts ?

Il existe plusieurs façons de lancer les scripts de test:

1. Individuellement: si le dossier `src/test/script` est dans votre variable système `PATH` alors écrire le nom du fichier script permet de l'exécuter. Si le dossier n'est pas présent dans la variable, le script peut tout de même être lancé en écrivant le chemin complet dans la console.
2. Sinon, tout les scripts de tests peuvent être exécutés à la fois avec l'une des deux commandes suivantes: `mvn test` ou `mvn -DJaCoCo.skip=false verify`, la dernière ayant comme avantage de générer la couverture de tous les tests exécutés grâce à l'outil JaCoCo. Bien sûr, ces commandes ne peuvent être lancées que depuis la racine du projet, là où est situé le fichier `pom.xml` de configuration de Maven.

Il est bon de relever que dans un souci de consommation énergétique, il est très préférable de n'exécuter qu'un test grâce à l'un des quatre launcher mis à disposition ou directement avec la commande `decac` plutôt que d'exécuter un script le faisant sur tous les fichiers tests disponibles. Ces scripts ont été, et doivent être utilisés avec parcimonie. Cette remarque est d'autant plus valable pour les commandes du point numéro 2 évoqué ci-dessus.

C. Configuration de maven

Tous les scripts de tests présentés ou fournis ne sont pas inclus dans le fichier de configuration `pom.xml` de Maven. Tous les scripts fournis (en rouge dans le tableau) sont toujours présents, à l'exception du script `JaCoCo-report.sh`. À ces scripts ont été ajoutés les scripts suivants:

- `full-lex-diff.sh`
- `full-synt.sh`
- `full-context.sh`
- `full-codegen.sh`
- `full-optim.sh`
- `compiler-options.sh`

Notons que chacun de ces 6 scripts est exécuté avec l'argument `--maven` permettant de spécifier aux scripts d'exécuter la commande `exit 1` à la moindre erreur. Cette commande n'est pas exécutée lors d'un test manuel afin de pouvoir avoir un compte rendu global sous forme de pourcentage de réussite.

III. Gestion des risques et gestion des rendus

1. Introduction

Risque: Oubli des dates de rendus.
Actions: Faire un point sur les prochaines deadlines à chaque daily meeting.

2. Exemple sans objet

Risque: Ne pas avoir le même résultat que l'exemple fourni.
Actions: Créer un test reprenant exactement les mêmes instructions et l'exécuter une fois les parties A et B implémentées pour la partie sans objet.

3. Rendu intermédiaire

Risque: La banque de tests est insuffisante
Actions: Avoir un document recueillant l'entièreté des tests réalisés et relevant quelles règles n'ont pas encore été testées.

Risque: Baisse de qualité contre l'avancement du projet.
Actions: Avant de passer à l'étape suivante (exemple: de sans objet à objet), s'assurer que la qualité de la phase terminée est satisfaisante et que la banque de tests est complète.

4. A rendre

Risque: Oublier un fichier du projet
Actions: S'assurer que la branche develop a bien été push sur master

Risque: Lors de la mise à jour de la branche master, une erreur apparaît dans master
Actions: Garder la dernière journée pour réaliser tous les tests possibles afin de nous assurer que le code de master fonctionne comme il devrait.

5. Soutenance

Risque: Oublier certaines spécifications ou aspect du projet
Actions: Réaliser un bilan après chaque sprint en notant les points majeurs réalisés pour pouvoir synthétiser le livrable.

Risque: Retard le jour de la soutenance
Actions: S'assurer de l'heure de la soutenance, mettre différents réveils et nous donner rendez-vous deux heures avant pour prévoir les retards et éventuellement pratiquer la soutenance à l'avance.

Risque: Inégalité des temps de parole.
Actions: Préparer la présentation à l'avance en faisant attention à bien distribuer les temps de parole et pratiquer plusieurs fois la soutenance.

Risque: La machine utilisée pour réaliser les démonstrations présente un problème durant la soutenance.
Actions: Avant la soutenance, s'assurer que démonstrations prévues sont possibles et correctes dans au moins deux machines. Le jour-J, apporter les deux machines chargées afin d'avoir une machine principale et une machine de secours.

6. **Communication**

Risque: Ne pas communiquer sur l'avancement de chacun. Ou bien sur la répartition du travail.
Actions: Daily meeting tous les matins pour savoir comment avance le projet et comment se répartir le travail de la journée.

Risque: Ne pas comprendre les règles de syntaxe de la même manière entre 2 étapes
Actions: Tester et analyser le comportement des programmes à l'exécution.

Risque: Il y a un conflit au sein du groupe.
Actions: Essayer de régler le problème diplomatiquement, et demander à quelqu'un de médier si le problème est toujours présent (c.f. Charte d'équipe).

Risque: Ne pas avoir bien compris ce qu'un autre membre a demandé de faire.
Actions: Demander des précisions et des explications tant que le récepteur de l'ordre a toujours des doutes de ce qu'il faut faire.

7. **Suivis**

Risque: Négliger le suivi et perdre des points.
Actions: Préparer à l'avance nos diapos et les sujets évoqués pendant le suivi.

Risque: Ne pas adopter les conseils donnés par les professeurs.
Actions: Noter les conseils proposés et les évoquer au prochain Daily Meeting afin de changer notre fonctionnement de travail.

Risque: La machine utilisée pour réaliser les démonstrations présente un problème durant le suivi.
Actions: C.f. Soutenance.

8. **Consignes**

Risque: Mauvaise compréhension des consignes.
Actions: Demander aux professeurs.

Risque: Doutes sur une consigne.
Actions: Demander aux professeurs.

9. **Partie A**

Risque: Mauvaise implémentation d'une règle lexicale ou syntaxique
Actions: Faire des "code review" et faire des tests de validation sur une base de test exhaustive mise en place par un membre différent de celui qui a codé.

10. **Décompilation**

Risque: Mauvaise transcription des règles de décompilation.
Actions: Le membre réalisera des tests pour vérifier que le code résultant de la décompilation est le même que le code originel fourni en test. Les tests chercheront à couvrir l'entièreté des classes de l'arbre.

Risque: Erreur de compréhension
Actions: Suivre le sujet et, si une loi n'est pas bien comprise, demander aux autres membres.

Risque: Le code ne réalise pas ce qui a été attendu et on ne trouve pas l'erreur.
Actions: Réaliser des tests pour trouver l'erreur. Si l'erreur n'est pas repérée, demander aux autres camarades de réviser l'erreur et proposer des tests à faire. Si l'erreur n'est pas encore repérée, demander à un professeur.

11. **Syntaxe contextuelle**

Risque: Oublier d'implémenter certaines règles, ou oublier d'implémenter toutes les conditions d'une règle.
Actions: Création d'un Google Sheet, répertoriant toutes les règles et toutes les conditions et en notant à coté s'il elles ont été réalisées ou non. Cela nous permet, pour chaque règle de réellement nous concentrer pour étudier tous les cas possible avant de passer à la règle suivante. De plus, une grande diversité des tests essayant de couvrir tous les cas permet de trouver certaines erreurs (décors non implémentés, règles oubliées, conditions non vérifiées, ...)

12. **Sémantique**

Risque: Erreur de compréhension
Actions: Lire le sujet plusieurs fois et, si une loi n'est pas bien comprise, demander aux autres membres. Noter aussi toutes les notions à prendre en compte dans des fiches pour nous assurer de rien oublier.

13. **Decac**

Risque: Oublier d'implémenter les options du compilateur.
Actions: Assignation de la tâche dès le début à un membre du groupe qui est en tout temps conscient du travail à fournir.

Risque: Non respect des attendus de chacune des options.
Actions: Vérification de la bonne exécution de chaque option par un autre membre du groupe.

Risque: Ne pas respecter la syntaxe des messages d'erreur imposée permettant l'évaluation automatique.
Actions: Vérification par un autre membre du groupe que les messages sont correctement écrits, en accord avec la partie résultat des tests invalides pour la partie contextuelle par exemple.

14. **Machine abstraite**

Risque: mauvaise organisation de la génération du code, qui mène à des complications et erreurs du code généré.
Actions: S'imposer des conventions de génération du code, et penser des méthodes pour simplifier la génération.

15. **Convention liaison**

Risque: Ne pas respecter les conventions de liaisons.
Actions: Relire plusieurs fois la doc : écrire clairement toutes les conventions, et les relire après chaque implémentation en s'assurant qu'elles sont respectées.

16. **Risques sur Git**

Risque: Membre ne peut pas pull à cause d'un conflit dans les commits
Actions: Le membre réalisera une résolution de conflits à la main, en vérifiant quelles lignes de quels fichiers garder et lesquels modifier.

Risque: Membre code sur une branche qui n'est pas la sienne et s'aperçoit trop tard.
Actions: Le membre devra push sur la branche où il désirait de pusher et, après avoir pushé, il devra refuser les modifications réalisées dans la branche où il se trouve : la branche où il se trouve n'aura pas de modifications mais les commits seront bien présents dans la branche désirée. Si git ne laisse pas push dans l'autre branche à cause d'un merge conflit, le membre devra manuellement copier et coller les codes dans l'autre branche. Une option pourra être de copier les fichiers en dehors du dossier du projet, refuser les modifications, changer à la branche désirée et coller les fichiers dans la nouvelle branche.

Risque: Membre a push un commit qui ne devait pas push
Actions: Si l'erreur est petite, le membre résoudra l'erreur en priorité et pushera la solution pour ne pas affecter le travail d'autres camarades. Si l'erreur n'est pas rapide à résoudre, le membre

révertira le commit (fera le push du commit précédent au commit problématique) et résoudra le problème en local.

IV. Résultats avec JaCoCo

Lors de l'exécution de la commande `mvn -DJaCoCo.skip=false verify` et de l'ouverture du fichier `index.html` généré dans `target/site/` nous voyons de prime abord que nous avons une couverture de 80%. Lors du suivi n°3, nous avions affirmé pouvoir atteindre une couverture entre 80% à 82%: notre objectif a donc été atteint.

Voyons tout cela maintenant plus en détail. Pour les package `fr.ensimag.deca.context` et `fr.ensimag.deca.pseudocode` les couvertures sont respectivement de 93% et 83% et les instructions manquantes sont principalement dûes à des getters, des setters ou des vérification de statut (`isClass`, `isParam`, ...) fournis par le code de base mais jamais utilisés. Le package fr.ensimag.deca quant à lui est couvert à 81%. Les instructions manquantes sont principalement des cas d'erreurs qui ne sont jamais soulevés, ou encore le fait qu'aucun test pour l'option parallèle n'est vraiment implémenté.

Les cas intéressants restent `fr.ensimag.deca.syntax` regroupant les fichiers DecaParser et DecaLexer respectivement couverts à 82% et 97% et enfin et surtout `fr.ensimag.deca.tree` contenant chaque nœud de l'arbre. Ce package est couvert à 86% et pourrait l'être encore plus si d'avantages de tests sont ajoutés pour l'extension. De plus certains getters ne sont jamais utilisés et certains cas des fonctions de codegen non plus. Cependant, les différentes fonctions de vérification de la partie B, elles, sont entièrement testées.

V. Méthodes de validation autres que les tests

1. Code review

Nous avons essayé tout au long du projet de revoir le code écrit par une personne s'occupant de la même partie. C'est une technique que nous avons réussi à mettre en place pour les 3 premiers sprints sur la partie A ainsi que sur la partie B. La partie C n'étant gérée que par une seule personne, les codes reviews étaient moins présents quoique quelques fois effectués par un des membres s'occupant de la partie A. En ce qui concerne le dernier sprint, où l'effort était globalement concentré sur l'extension, l'organisation était plus erratique et les codes review se faisaient beaucoup plus rares.

Cette technique a tout de même eu un impact largement bénéfique sur la propreté et la qualité de notre code et a permis de très nombreuses fois de relever des oublis ou des fautes pouvant facilement passer inaperçu.

2. Pair Programming

La partie B a souvent été travaillée en Pair Programming avec l'outil LiveShare de VSCode. Cela nous a permis de travailler de manière efficace tout en faisant un code review instantané de ce que l'autre membre faisait. Les autres membres n'ont globalement jamais travaillé de la sorte pour les autres parties.

3. Vulgarisation du code

Quelques pauses dans la conception ont pu être prises tout au long du projet afin que chaque membre puisse vulgariser, quand il en avait besoin, le code qu'il voulait écrire ou qu'il était en train d'écrire. Bien que cela ne permette pas de vérifier le caractère correct du code écrit, cela permettait à chaque fois de poser ses idées au clair et de faire valider par le reste de l'équipe le raisonnement effectué.

4. Dossier de suivi de validation

Pour la partie B seulement, nous avons mis en place un tableur permettant de rester méthodique quant à l'implémentation des trois passes de cette partie. Chaque règle y était inscrite, avec ses potentielles erreurs, ses dépendances, et l'état de l'avancement de chaque règle: tests créés, règle implémentée, règle testée de manière la plus exhaustive possible (cas de validité et d'invalidité).