Document de validation

Equipe 10

Janvier 2021

Contents

1 Introduction

A travers ce document, nous allons décrire nos processus de validation de codes, la logique de nos tests, leur organisation et leur exécution. Les tests et leurs scripts de lancement ont été un pilier de notre organisation tout le long du projet. Ils sont nombreux donc pour faciliter leurs utilisations, nous les avons soigneusement organisés et nommés et avons écrit de nombreux scripts spécifiques. Notre objectif a été très rapidement d'avoir une couverture maximale de manière à laisser passer un nombre minimal de *bugs*.

2 Descriptif des tests

2.1 Types de tests

Dans notre projet, il faut distinguer trois types de tests : les tests de grammaire, de génération de code assembleur et de performance. Tous ces tests sont des programmes Deca. Ils sont rangés dans src/test/deca en partant de la racine du projet. Dans chaque type de tests, nous avons séparé les programmes deca n'appelant aucun objet (SansObjets) et les programmes Deca utilisant des objets (AvecObjets). Cette séparation est liée à notre gestion incrémentale du projet : dans un premier temps nous avons géré le langage Deca sans objet puis une fois cette partie validée, nous avons développé la programmation objet.
Le jour du rendu, nous comptions 566 tests. Ce nombre relativement élevé se justifie par le fait que chaque test vérifie un point précis de la grammaire ou de la génération de code assembleur. Cette volonté de multiplier les programmes très courts et précis a été entretenue par la volonté de savoir précisément quelle partie de notre code nous fait défault, quelle règle de grammaire n'est pas gérée ou a minima mal gérée. Ainsi, dans chaque type de tests, nous rangeons les tests dans différents dossiers comme par exemple les tests de comparaisons, d'opérations arithmétiques, de "while" ou "if" *etc.* Enfin, chaque test porte un nom que nous voulions le plus explicite possible. Nous n'avons pas de nomenclature formelle mais en en créant une, nous pourrons gagner en efficacité lors du *debuging* en sachant exactement quelle fonctionnalité testée n'est pas passée.

2.1.1 Tests de grammaire

Ce type de tests comprend les tests lexicaux, les tests syntaxiques et les tests contextuels. La logique de ces tests est très simple. Les tests lexicaux vérifient que notre compilateur génèrent bien tous les tokens du programme. Les tests syntaxiques vérifient que la syntaxe du programme est respectée sachant qu'il n'y pas de problème au niveau lexical. De même pour les tests contextes, la vérification grammaticale des programmes se fait sur la partie contextuelle sachant que, syntaxiquement, le programme est valide.
Tous ces tests sont séparés en deux parties : les tests dits valides et invalides. Malgré ce que laissent sous-entendre leur nom, tous ces tests lors de l'exécution de nos scipts doivent passer. Il s'agit en réalité de différencier les programmes

grammaticalement corrects et les programmes volontairement incorrects. En effet, notre compilateur doit pouvoir lever des exceptions à l'utilisateur en cas d'erreurs grammaticales pour assurer l'intégrité de la compilation et l'exécution du programme. Nous allons détailler la méthode de validation des tests invalides dans la partie suivante "Script de tests". L'organisation des tests invalides est la même que l'organisation des tests valides c'est-à-dire que chaque tests est regroupé dans un dossier portant le nom de l'opération levant une erreur. Avec le recul, il aurait été plus pertinent et efficace de regrouper ces tests par rapport à la règle de grammaire non respectée.

2.1.2 Tests de génération de code

Ces tests sont des programmes Deca grammaticalement correctes, leur objectif étant de vérifier que la génération de code assembleur est correcte et renvoie le résultat voulu lors de son exécution plutot que de vérifier le respect des règles de grammaire. La validation automatisée de ces tests se fait donc sur la sortie de l'exécution du fichier d'extension .ass. La raison de valider le test sur la sortie de l'exécution est simple : ça aurait été bien plus fastidieux de vérifier le code assembleur et surtout ça aurait été très peu pertinent compte tenu du nombre de possibilités de traduire le programme deca en assembleur.
Cependant, il était très fréquent que nous regardions directement le contenu du fichier assembleur généré par *decac*. Cela avait pour but de vérifier précisément que la génération d'assembleur s'est déroulée comme prévue. Toute la partie *debuging* de l'étape C se faisait aussi via l'analyse du code assembleur.

2.1.3 Tests de performances

Les tests de performances sont extrêmement minoritaires dons notre batterie de tests complète. Ces tests ont trois objectifs : tester notre compilateur sur des tests volontairement complets et complexes (récursivité, liste chaînée), analyser le nombre de cycles nécessaires à l'exécution de programmes Deca et tester l'option -n de *decac*. Ces tests sont le support de notre étude énergétique.

3 Script de tests

Le nombre de nos tests ne nous permet pas d'en faire leur vérification "à la main". Nous avons donc créer plusieurs scripts de tests que nous avons voulu spécialisés. Chaque script itère sur leurs tests attribués et lance soit le compilateur, soit les exécutables fournis. Si le test est validé, le script passe au prochain test. Sinon, il affiche le résultat obtenu et s'arrête. Ainsi, nous pouvons directement voir quel test n'est pas validé et pourquoi il ne l'est pas.

3.1 Scripts de grammaire

Nous avons trois scripts différents pour lancer les tests grammaticaux : un script pour chaque partie de la grammaire. Ainsi, le script **Lexical.sh** lance *test_lex*

sur les tests lexicaux du répertoire src/test/deca/lexical, le script **Syntax.sh** lance *test_synt* sur les tests syntaxiques et **Context.sh** lance *test_context* sur les testes contextuels. A leur lancement, le programme vous propose trois options : tester seulement les tests valides (V), seulement les tests invalides (I) ou les deux (2). Nous avons ajouté cette option car tester notre compilateur sur des tests valides et sur des tests invalides sont deux choses très différentes. Dans le premier cas, on teste que la génération des tokens ou des arbres alors que dans le deuxième on test que toutes les règles de grammaire sont bien implémentées et que le compilateur renvoie une exception. Ces scripts sont efficaces et permettent d'avoir une bonne idée des capacités du compilateur mais possède quelques limites.

La première est dans la validation des tests. Un test valide sera validé par le script si aucune erreur n'est levée. Cette idée fonctionne mais n'est pas totalement satisfaisante. En effet, pour un programme exemple.deca testé par Context.sh, la commande test_context peut générer un arbre abstrait décoré faux. Comme aucune exception n'est levée, le script validera tout de même ce test. Pour les tests invalides, ils seront validés si une exception est levée par notre compilateur. La condition est que, si dans la sortie de l'exécution du test, on trouve le nom du fichier test alors le script validera ce test. Il est donc impossible de vérifier que l'exception levée est bien celle attendue ou le test n'affiche pas volontairement le nom du document par pure malice de son créateur.
Pour les tests valides, le problème est géré par la compilation par *decac* du fichier test puisque si une erreur s'est glissée dans l'arbre, la génération du code assembleur sera impossible. Cependant, la seule solution utilisée actuellement pour les tests invalides est de les lancer une première fois sans script, d'analyser l'exception et vérifier qu'elle correspond bien à celle attendue.

La deuxième est la factorisation du code. Ces trois scripts font presque la même chose et leur mode de validation est la même. Il aurait été interessant de regrouper ces scripts en en seul avec pour option le type de grammaire testée. Actuellement, si l'on veut changer de mode de validation, il faut le faire trois fois.

3.2 Scripts de compilation

Nous avons trois scripts de compilation. Un (**Compilateur.sh**) qui lance *decac* et *ima* sur tous les tests dans src/test/deca/codegen, un (**CompilateurDecompile.sh**) qui lance *decac -p* puis *decac* et *ima* sur le programme deca décompilé généré et un dernier (**testOption.sh**) qui lance différentes options de *decac* sur des tests sans intérêts.
Les deux premiers tests fonctionnent sensiblement de la même manière. Ils compilent et exécutent tous les tests de codegen et comparent la sortie avec le contenu d'un fichier d'extension .ans de même nom que le test. Ces fichiers .ans sont créés par les membres de l'équipe en même temps que l'écriture des tests et contiennent seulement la sortie attendue. Il est obligatoire que dans tout le

répertoire src/test/deca/codegen le nombre de fichiers .deca soit égal au nombre de fichier .ans sinon le script renvoie un message d'erreur donnant le nombre de fichiers .ans manquants ainsi que leur nom. La spécificité de **CompilateurDecompile.sh** est qu'il compare la sortie de l'exécution du programme décompilé avec le fichiers .ans. La logique derrière ce choix de validation est que deux programmes sont équivalents si pour une même entrée, ils renvoient la même sortie. Ne pouvant pas tester les programmes sur une infinité d'entrées, nous avons fait le choix de tester sur une seule entrée.
Ces scripts ont aussi la capacité de sauter des tests. Pour cela il suffit de rajouter dans la description des tests "Skipped test" pour Compilateur.sh et "Skipped decompile" pour **CompilateurDecompile.sh**. Aussi, tous les tests comportant les fonctions **readFloat()** ou **readInt()** sont sautés. La raison d'avoir ajouté cette option est de sauter les tests dont on ne peut pas connaître la sortie en avance. Il est tout de même nécessaire de créer un fichier .ans associé mais son contenu est inutile.

Finalement, le test **testOptions.sh** est très rudimentaire. Il lance différentes options de *decac* sur les trois tests de src/test/deca/OptionsDecac. Aucune validation n'est faite par le script. Elle doit se faire par un membre directement. Le script affiche la sortie pour faciliter la validation.

4 Gestion des risques

Risques	Gravité (1 à 4)	Actions préventives ou correctrices
Retard sur la date de rendu	4	Communication quotidienne avec l'équipe sur l'avancement de chacun et suivi et actualisation du diagramme Gantt
Script de tests bugé	4	Tester le script sur des exemples connus
Oubli de tests	4	Analyse de la couverture Jacoco
Burnout d'un membre de l'équipe	4	Communication quotidienne et respect des temps de repos de chacun
Mauvaise compréhension d'une spécification	4	Echanger avec les enseignents nous encadrant et les autres membres de l'équipe
Apporter un bug quelques heures avant le rendu	3	Lancer tous les scripts de test avant de commit
Plantage de l'ordinateur d'un membre	2	Commit après chaque modification après l'avoir testée
Courte indisponibilité d'un membre	1	Prévenir les membres du groupe en avance
Perte de connexion Internet sur le lieu de travail	1	Partage de connexion si possible ou changement de lieu de travail
Mésentente entre membres de l'équipes	1	Communiquer sur ses problèmes à toute l'équipe

5 Jacoco

L'outil Jacoco nous a été très utile sur la fin du projet pour nous assurer que notre couverture de tests était suffisante et pour écrire les tests manquants pour déceler des éventuels bugs. L'objectif de l'équipe était d'atteindre 85% de couverture. Ce nombre a été atteint le jour du rendu. Notre base de test couvre toutes les classes et méthodes que nous avons implémentés. Les éventuels bugs restant apparaissent sur des cas que nous n'avons pas traités et pas sur des cas

mal traités. Pour cela, nous considérons que nous avons réussi à délivrer un produit fiable et performant. Les classes qui nous font perdre de la couverture sont les classes des fichiers Java générés par les fichier .g4

6 Autres méthodes de validation

Les tests n'ont pas été la seule méthode de validation. Nous avons souvent travaillé par *peer review*. Les deux membres de chaque étape relisaient systématiquement le code de chacun mais les membres des autres étapes regardaient aussi ce que les autres codaient. Ainsi, tous les membres ont une bonne vision de comment est implémenté le compilateur et amènent un regard critique sur le travail de chacun. Cette méthode de validation nous a permis de *debug* de nombreux problèmes rapidement.
De plus, pour chaque changement de structure du compilateur (changement de signature d'une méthode, changement de nom d'une classe, suppression ou ajout d'une classe), le membre l'amenant demandait l'approbation de tous les membres de l'équipe afin de s'assurer qu'il n'existe pas d'autres solutions moins risquées, que cela demande peu de changement, que ça n'apporte pas de conflits *etc.*