Thomas CASSAGNES
Angelica FIGUEROA
Antoine FLICHY
Mathieu VALLICCIONI
Joël ZHU

Groupe 48

30/01/20

Documentation de validation

Introduction

Dans ce document de validation, nous allons résumer en quoi consistent nos tests, quels types de test nous avons fait pour chaque étape, leur organisation ainsi que leurs objectifs. Ensuite nous allons détailler les scripts des tests, les erreurs que nous avons pu faire dans notre ensemble de test, notre gestion des risques et des rendus, et enfin les tests que nous avons fait avec cobertura.

I] Descriptif des tests

Un test correspond à un fichier .deca écrit à la main avec pour chaque un fichier .txt associé représentant la sortie attendue et correcte du programme. Ces tests permettent de vérifier le bon fonctionnement des différentes parties du compilateur, en allant de simples programmes qui testent seulement une fonctionnalité (exemple une déclaration ou un ifthenelse) jusqu'à des programmes plus complexes comme un calculateur de la suite de Fibonacci.

Les différents tests créés sont exécutés automatiquement à l'aide d'un script Shell qui sert d'oracle. Ces derniers consistent à parcourir tous les fichiers .deca d'un dossier, exécuter chacun d'eux, stocker leurs sorties dans un fichier temporaire pour ensuite le comparer avec la sortie attendue écrite manuellement dans le fichier .txt.

Pour le cas du test du lexer, il a fallu éroder la sortie en utilisant un script Python pour pouvoir comparer les tokens un à un.

II] Types de tests pour chaque étape/passe

Pour le lexer et le parser, des tests unitaires ont été créés pour tester le bon fonctionnement des différentes fonctionnalités de ceux-ci.

Il a fallu premièrement tester la validité des tokens reconnus par le lexer. Nous avons donc fait une batterie de test correspondant aux capacités du lexer. Pour cela, nous avons remarqué que le programme Java fourni *ManualTestLex.java* dans le paquet fr.ensimag.deca.syntax était utilisable. En revanche, la sortie fourni par ce programme était jugé trop complexe pour pouvoir écrire les sorties attendues manuellement à la main car il inclut par exemple les positions dans le fichier de chaque lexème qui ne sont au final pas utiles pour tester les fonctionnalités propres

au lexer. Nous avons donc réalisé un script Python qui modifie la sortie en éliminant les données superflues pour ne récupérer que les tokens pertinents. Cette suite de tokens est considérée comme suffisante pour tester le lexer et le parser.

Dans la partie B, deux types de tests sont utilisés : des tests unitaires pour tester en détails les fonctionnalités basiques de l'analyse syntaxique afin de s'assurer que la fondation du compilateur est parfaitement fonctionnelle et empêcher toute régression et des tests de vérification contextuelle sur des morceaux de code simples pour garantir la cohésion du programme.

Pour faciliter certains de ces tests le framework Mockito a été utilisé. La vérification contextuelle étant une étape intermédiaire dans la compilation, il faut parfois générer une grande quantité de paramètres pour chaque tests, ainsi Mockito permet de mocker certaines variables et de simuler les retours générés par les appels de certaines fonctions qui demandent en théorie l'instanciation de multiples objets pour produire le résultat voulu. Mockito permet donc de créer des tests réellement unitaires en isolant la partie testée des dépendances externes.

Pour l'étape de génération de code, une base de 338 tests sont proposés. La plupart des tests sont composés de la manière suivante : Une déclaration ou une instruction est écrite, ainsi qu'une instruction d'affichage quand cela est possible (dans le cas des booléens cela est donc impossible de tester les sorties sans les conditions, on teste juste si le programme s'exécute sans erreur). Ainsi les tests sont pratiquement unitaires et couvre tout ce qui a été implémenté.

Plus largement, des tests globaux effectuant une série d'instruction ont été créés, et cela permet de tester si cette suite d'instruction génère ou non des problèmes, de gestion de mémoire notamment.

III] Organisation des tests

Pour chaque étape du compilateur, une base de tests lui est associée composée de tests unitaires qui vérifient le bon fonctionnement de toutes les fonctionnalités basiques implémentées et des tests d'intégration qui permettent de s'assurer que des programmes plus complexes restent tout de même cohérent.

Pour la partie analyse contextuelle, en ce qui concerne les test unitaires, ils sont situés dans le package **fr.ensimag.deca.context** du répertoire **src/test/java**. Ils sont regroupés par types de tests : opérations arithmétiques (binaires, unaires),

comparaisons, classes, méthodes... Les noms de classes de tests unitaires sont suffisamment explicites pour reconnaître les éléments testés dans ces fichiers.
En ce qui concerne les tests d'analyse contextuelle sur des arbres, l'ensemble des fichiers sources sur lesquels ils sont basés se trouvent dans les répertoires **src/test/context/{valid,invalid}/created**. On ne peut pas réellement créer de sous catégories parmis ceux-ci, cependant ils sont nommés suivant les fonctionnalités testées (exemple : comparaisons.deca, include.deca, ifthenelse.deca...).

Pour la partie génération de code, les tests sont organisés de la manière suivante : pour une instruction déca donnée (assign, plus, if, ...), une série de tests numérotés correspondant à l'utilisation de cette instruction dans des conditions différentes est écrite.
Par exemple pour les "and", on teste cette instruction dans plusieurs cas différents :

- avec deux immédiats tels que true && false
- avec deux identifiers
- un mélange entre les deux
- lors d'une association
- plusieurs "and" en série
- avec d'autres opérateurs booléens
- avec une évaluation paresseuse
- etc

Ainsi, lors de l'ajout de nouvelles fonctionnalités, il est très facile de repérer et situer les régressions éventuelles. Avec de nombreux tests existants pour chaque opérateur, nous avons pu rapidement détecter les bogues et dans quels cas ils étaient présents, ce qui s'est avéré être un gain de temps considérable pour la suite.

IV] Objectifs des tests

Les objectifs des tests étaient avant tout de trouver des erreurs dans le code, de se prémunir contre les cas particuliers générant ces erreurs, et de nous aider à comprendre l'origine des erreurs.

De plus, les tests ont été automatisés de sorte à détecter rapidement les régressions et nous permettre de nous concentrer sur le développement de nouvelles fonctionnalités car nous savions que si régression il y avait, cela serait vite repérée et corrigée.

Dans la partie analyse contextuelle, les fonctionnalités testées par les tests unitaires sont principalement des fonctions dans les Type et les Définitions comme par exemple isSameType() ou asClassDefinition(), ainsi que les opérations : par exemple les opérations {+, -, x, /} entre entiers / flottants, {&&, ||} entre booléens...
Ils permettent de vérifier par exemple que le type de retour des opérations en fonction des types des membres passés en paramètre est cohérent (i.e. il faut qu'ils soient du même type si c'est une addition entre deux entiers par exemple).

Pour les tests exécutés sur des portions de code basique, nous produisons du code très simple concernant des fonctionnalités précises et l'objectif est de générer les arbres décorés associés sans erreur. Il peut ainsi y avoir un contrôle visuel du bon fonctionnement du code. Pour pousser les tests plus loin, des fichiers java ont été créés dans le dossier src/test/java/fr/ensimag/deca/syntax/tests_oracle/. Des arbres abstraits correspondants aux programmes deca de test sont créés à la main et servent en tant que résultat attendu pour l'oracle. L'oracle exécute les fichiers deca et compare les arbres produits avec ceux générés manuellement.

Parmis les tests vérifiant la syntaxe contextuelle, la moitié sont des programmes à la syntaxe contextuelle valide et l'autre moitié sont des programmes deca à la syntaxe contextuelle invalide. Le compilateur ne produit pas d'arbre sur les programmes à la syntaxe contextuelle invalide et génère à la place les erreurs adéquates. Ces programmes invalides permettent donc de vérifier que les erreurs contextuelles sont bien levées.

La limite des tests sur des programmes deca est qu'ils nécessitent une partie A fonctionnelle pour être exécutés. C'est pourquoi la base de test pour l'analyse syntaxique est particulièrement vaste.

Dans le cas de la partie sur la génération de code, les tests proposés permettent un parcours assez large du code. De plus, le nombre de test écrit permet de trouver les cas particuliers où il peut potentiellement y avoir des erreurs. Ainsi, avec ces tests, nous avions une idée précise des fichiers dans lesquels l'erreur était présente.

V] Les scripts de tests

Pour la partie analyse contextuelle, l'ensemble des tests unitaires se trouvent dans le package **fr.ensimag.deca.context** du répertoire **src/test/java**. Les autres tests de vérification d'analyse contextuelle sont dans le dossier **src/test/deca/context**. Un script automatise le lancement des tests d'analyse contextuelle sur arbre : **src/test/script/launchers/chain_test_context [files]**.
Une dernière manière de lancer des tests au détail est d'utiliser test_context dont le fonctionnement est décrit dans le polycopié du projet.

Les oracles se trouvent dans le dossier src/test/script/launchers.
Dans le cas de la Partie A et C, les scripts *validor_(cod|lex |synt|synt_objet)* qui testent les fichiers de test avec un code valide, et *invalidor_(lex|synt)* qui testent les fichiers avec un code invalide, permettent d'exécuter les tests pour les différentes parties.

Par exemple, dans la génération de code, le fichier validor_cod qui parcours l'ensemble des fichiers qui se trouvent dans le dossier src/test/deca/codegen/valid/created, les compilent, les exécutent, et compare les sorties obtenues avec les fichiers ayant le même nom (mais l'extension .txt) dans un dossier *expected* (contenu lui aussi dans le dossier valid). Il en est de même pour la partie syntaxique, en remplaçant codegen par syntax dans le chemin de dossier (cependant le dossier expected s'appelle *alexpected* (pour analyseur lexical).

VI] Comment faire passer tous les tests

On peut exécuter la commande suivante : **src/test/script/launchers/chain_test_context src/test/deca/context/{invalid,valid}/created/*** pour lancer les tests sur tous les fichiers deca prévus pour l'analyse contextuelle sur des arbres.

Pour passer les tests du lexer, il faut lancer validor_lex pour les tests valides, et invalidor_lex pour les tests invalides. Pour passer les tests du parser, il faut lancer validor_synt pour les tests valides et invalidor_synt pour les tests invalides. Pour passer les tests de la partie de génération de code, il faut lancer le script validor_cod. Pour lancer tous les tests, l'oracle all-test ou la commande mvn test permettent de le faire.

VII] Les erreurs dans les tests

Nous avons fait quelques erreurs lors de la création de tests. Par exemple nous pouvons remarquer que dans la partie A, des fichiers tests sont écrits sans réponse attendue. Nous pouvons aussi remarquer que le script *validor_synt* effectue de la génération de code. Cela est dû au fait que le fichier java fr.ensimag.deca.syntax/ManuelTestSynt a été modifié pour faire de la génération de code, or ce fichier est appelé dans le script pour tester la partie A. Cela est dû à un manque de communication de notre part.

VIII] Gestion des risques et gestion des rendus

Nous avons identifié plusieurs tâches pour avoir une bonne gestion des risques et des rendus :
Pour nous préparer aux rendus, nous avons effectué :

1. des séances de code en pair programming pour que l'architecture du programme soit vérifiée et confirmée par plusieurs membres. De plus, nous avons fait du code review pour s'assurer que chaque classe et fonction soit bien implémentée.
2. des lancements du compilateur avec toutes les étapes (prettyPrint, checkAllLocations, verifyProgram, decompile, codeGen, IMA) sur des exemples simples. Cela nous a permis de vérifier qu'il n'y a pas d'erreur de compilation sur le code lors du lancement.
3. un lancement des oracles de tests valides pour s'assurer qu'il n'y a pas eu de régression
4. un lancement des oracles de tests invalides
5. des exécutions de la commande decac avec différentes options pour vérifier que la commande est fonctionnelle
6. des lancements de cobertura pour vérifier que la base de tests couvre bien la majorité du code.

IX] Résultats de Cobertura

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	251	78%	3660/4645	59%	826/1387	1.634
fr.ensimag.deca	5	69%	254/368	69%	126/181	2.771
fr.ensimag.deca.codegen	4	81%	57/70	77%	14/18	1.591
fr.ensimag.deca.context	21	89%	200/223	74%	37/50	1.359
fr.ensimag.deca.syntax	50	73%	1536/2083	47%	384/808	2.011
fr.ensimag.deca.tools	4	87%	36/41	83%	5/6	1.357
fr.ensimag.deca.tree	87	87%	1375/1570	80%	245/304	1.448
fr.ensimag.ima.pseudocode	26	77%	138/179	75%	15/20	1.171
fr.ensimag.ima.pseudocode.instructions	54	57%	64/111	N/A	N/A	1

Nous pouvons voir en utilisant le rapport de Cobertura que les tests que nous effectuons nous donnent une couverture moyenne de 78%.

Cependant, nous pouvons remarquer que nous n'avons que 57% sur le package fr.ensimag.ima.pseudocode.instructions. Cela est dû au fait qu'il y a des instructions assembleur que nous n'avons pas utilisées.