Documentation de validation

GL28

Janvier 2020

Table des matières

1 Descriptif des tests

1.1 Étape A

1.1.1 Lexicographie

Les tests valides sur la lexicographie ont eu pour but de valider le fait que les tokens devant être reconnus le sont effectivement, par exemple. Parmi les tests invalides, on trouve des tests qui permettent de vérifier la non reconnaissance de tokens non définis dans la lexicographie, ainsi que le renvoi d'erreur en cas de fichier à inclure qui existe pas ou d'inclusion circulaire. On a souhaité développer un maximum de tests unitaires pour tester les cas souhaités 1 par 1. Ci dessous se trouvent les différentes thématiques des tests valides et invalides :

- **Valides**
  - Tokens
- **Invalides**
  - Tokens
  - Include

L'analyse lexicographique est conforme vis-à-vis de la base de test, tous les tests valides sont acceptés et tous les tests invalides sont refusés. Il serait intéressant à l'avenir de classer dans le répertoire les tests suivant ces thématiques, car actuellement tous les tests sont regroupés dans le même dossier sans tenir compte de leurs distinctions.

1.1.2 Syntaxe abstraite

L'effort de rédaction de tests pour la partie syntaxe abstraite a porté principalement sur la couverture de l'arbre abstrait. Pour pouvoir couvrir l'arbre abstrait, un parcours de chaque noeud avec rédaction de tests a été réalisé. Il faut noter que les entiers et flottants non codables, ainsi que l'arrondi à 0 d'un flottant sont gérés dans la partie invalide.
Par ailleurs, le développement de cette partie s'est faite par programmation défensive : les pré-conditions sont vérifiées au moyen d'assertions dans le code du parseur.

- **Valides**
  - Noeuds de l'arbre abstrait
- **Invalides**
  - Variantes des noeuds de l'arbre abstrait
  - Entiers et flottants trop grands ou trop petits
  - Entiers et flottants mal formés (int x = 010)

Ici également, l'analyse syntaxique est conforme à la base de test.

1.2 Étape B

La rédaction des tests à l'étape B s'est faite indépendamment du développement, pour que la personne rédigeant les tests puisse se baser uniquement sur la syntaxe contextuelle

sans tenir compte de l'implémentation logicielle. Une étude approfondie de la syntaxe contextuelle a été faite, en rédigeant des tests valides et invalides sur chaque point significatif. Il s'agit de tests unitaires portants chacun sur un point particulier, et une attention particulière a été portée à la description en commentaire dans le test du point testé.

- **Valides**
  - Passe 1
    - Respect de chaque règle
    - Éléments dans les environnements synthétisés
    - Éléments dans les environnements hérités
  - Passe 2
    - Respect de chaque règle
    - Éléments dans les environnements synthétisés
    - Éléments dans les environnements hérités
  - Passe 3
    - Respect de chaque règle
    - Éléments dans les environnements synthétisés
    - Éléments dans les environnements hérités
- **Invalides**
  - Passe 1
    - Domaines et opérations sur les attributs
    - Conditions sur chaque règle
    - Type des attributs synthétisés
    - Type des attributs hérités
  - Passe 2
    - Domaines et opérations sur les attributs
    - Conditions sur chaque règle
    - Type des attributs synthétisés
    - Type des attributs hérités
  - Passe 3
    - Domaines et opérations sur les attributs
    - Conditions sur chaque règle
    - Type des attributs synthétisés
    - Type des attributs hérités

Les tests contextuellement valides sont tous acceptés par le compilateur, et les tests invalides sont tous refusés par le compilateur. Il aurait été intéressant d'utiliser notre script de test (dont le fonctionnement sera explicité plus loin) pour faire passer la base de tests de l'étape A sur l'étape B, et de voir quels tests valides en étape A mais invalides en étape B sont acceptés ou refusés en étape B, pour renforcer davantage notre compilateur. Il est ici également important de noter que les tests ne sont pas classés selon cette thématique dans les dossiers, c'est un point qui aurai pu être fait pour qu'il soit plus facile de voir la thématique de chaque test.

Au-delà de l'aspect valide / invalide, une vérification visuelle de l'arbre généré a été réalisée, pour la partie sans objet l'arbre généré a été utilisé pour effectuer des tests de

non-régression, mais par manque de praticité de notre script de test, cette démarche a été abandonnée pour la partie objet. Dans la partie objet, une validation manuelle de chaque arbre a été réalisée au moins une fois, et les tests de non-régression s'appuient sur le passage ou pas du test par l'étape B, pas sur d'éventuelles modifications dans l'arbre généré.

Pour identifier de nouvelles erreurs dans le code, une bonne piste est de faire des légères modifications dans les tests présents, car il y a le risque que certains tests soient bien gérés par notre compilateur, mais seulement dans le cas du test unitaire, pas dans le cas général.

1.3 Étape C

La recherche de tests pour l'étape C a nécessité une recherche d'informations dans plusieurs parties du polycopié ([Semantique], [MachineAbstraite], [ConventionsLiaison], [Gencode]...). L'identification des cas invalides a été en grande majorité faite par une personne indépendante du développement, les cas valides ont étés fait avec l'appui de la personne chargée du développement.

- **Valides**
  - Cas valides concernant les entiers et flottants
  - Comportement des fonctions d'affichage
  - Evaluation des expressions booléennes
  - Méthodes
  - Classes
  - Attributs de classe
- **Invalides**
  - Cas invalides concernant les entiers et flottants
  - Evaluation des expressions booléennes
  - Méthodes
  - Attributs de classe
  - Lexicographie de l'assembleur
- **Interactifs**
  - readInt
  - readFloat

L'ensemble des tests valides sont acceptés et l'ensemble des tests invalides sont refusés, pour pouvoir identifier des cas non traités dans le compilateur, il faudra donc écrire de nouveaux tests. Le script de test permet d'effectuer des tests de régression, en prenant en compte une annotation dans les fichiers de tests.

Quelques tests sur des programmes longs sont présents, ils servent principalement à valider notre implémentation à l'aide de notre script de tests, mais la grande majorité des tests sont des tests unitaires permettant de valider toute implémentation du compilateur.

Les tests interactifs sont bien commentés, pour que l'utilisateur puisse identifier une

erreur en fonction de l'entrée qu'il aura fournie.

Le test "non_object_print_long_string.deca" est mal placé, c'est un test valide dans le cas de notre implémentation mais pas dans le cas général.

1.4 Informations communes à toutes les étapes

Il figure quelques tests qui portent sur la version complète de Deca pour l'étape B et C, ce sont des tests qui sont refusés par l'implémentation actuelle du compilateur. Pour débuter le développement de la version complète de Deca, il faudra tout d'abord désactiver les exceptions lancées par le parseur sur l'opérateur instanceof et le cast.

2 Scripts de tests

Au cours de notre projet, plusieurs scripts ont vu le jour, nous ne parlerons que du dernier en date qui est le plus efficace.

2.1 Mode d'emploi

Dans *./src/test/script/* il y a un fichier *v2-test.sh*. On peut l'appeler depuis n'importe où avec la commande :

v2-test.sh [args]

Les arguments sont les suivants :
- -l pour tester le lexer
- -s pour tester le parser
- -ct pour tester l'analyse contextuelle
- -cg pour tester la génération de code
- -pgm pour afficher le programme .deca
- -err pour afficher l'erreur causée par decac
- -log pour renseigner les erreurs dans un fichier
- -ex1 pour arrêter le script dès qu'il y a une erreur
- -lO pour afficher la sortie du lexer (i.e. les tokens et leur position)
- -sO pour afficher la sortie du parser (i.e. l'arbre simple)
- -ctO pour afficher la sortie de l'analyse contextuelle (i.e. l'arbre décorée)
- -cgO pour afficher la sortie de ima sur le programme assembleur
- -ass pour afficher le programme assembleur

On peut aussi utiliser le script pour utiliser un seul fichier .deca, pour celà il faut renseigner :
- le fichier .deca (depuis la racine du projet /src/test/...)
- la catégorie du fichier (-l, -s, -ct ou -cg)
- sa validité (-v pour valid -nv pour invalid)

Les autres arguments (-pgm, -err, -log, -ex1, -lO, -sO, -ctO, -cgO, -ass) restent valables.

2.2 Exemples

2.2.1 Exemple 1

`v2-test.sh -ct -ctO -pgm`

execute tous les tests d'analyse contextuelle, affiche l'arbre et le programme .deca

2.2.2 Exemple 2

`v2-test.sh -cg -cgO -pgm -ass`

execute tous les tests de génération de code, affiche le programme .deca, le programme .ass et la sortie du .ass exécutée par ima.

2.2.3 Exemple 3

`v2-test.sh -l -s -ct -cg -log`

execute tous les tests, n'affiche rien et écrit les erreurs dans toFix.log

2.2.4 Exemple 4

`v2-test.sh src/test/syntax/valid/synt/whileSimple.deca -s -v`

test le fichier whileSimple.deca. Il est impératif de préciser (-s pour syntax et -v pour valid)

2.3 Critères de validation d'un fichier

2.3.1 Critères

Un programme est acceptée (passed) si :
— il est *valide* et ne produit aucun message d'erreur lors de la compilation
— il est *invalide* et il produit un message d'erreur lors de la compilation
Un programme est refusée s'il n'est pas acceptée.

Exceptionnellement pour la partie *génération de code*, il est possible de mettre au début du programme *deca* une entête de la forme :

```
// @result ok
// @result ok
...
```

Et dans ce cas le programme sera acceptée si la sortie de *ima* est

```
ok
ok
```

et refusé sinon.

2.3.2 Limites

On voit tout de suite les limites de cette approche, en effet si en partie B, pour un programme *valide* la construction de l'arbre se fait mais est fausse alors le programme va quand même être accepté. Même chose pour le lexing et le parsing. C'est pour ça qu'il est impératif de vérifier "à la main" la construction de l'arbre pour chacun des tests. En d'autres termes il est malheureusement impératif qu'un humain vérifie une fois chaque programme afin de s'assurer que la sortie attendue est la bonne (i.e. que les arbres soient bien construits et que les tokens soient bien identifiés au bon endroit).

3 Gestion des risques et gestion des rendus

3.1 Introduction

Nous avons pu actualiser notre documentation sur la gestion des risques et la gestion des rendus établie pour le suivi 2, en prenant en compte les retours des enseignants ainsi que le résultat de notre expérience.

3.2 Gestion des risques

Nous avons identifié 3 grandes catégories de risques à prendre en compte dans le cadre du projet génie logiciel, qui sont explicités ci-dessous :

3.2.1 Risques internes

— Perdre du temps à travailler sur une partie qu'on a du mal à concevoir
— Perdre trop de temps sur une fonctionnalité non capitale
— Favoriser le développement rapide mais de moindre qualité
— Ne pas oser demander de l'aide sur un sujet qu'on ne maîtrise pas
— Trop se focaliser sur le développement au détriment des tests
— Problèmes de compilation lors de la mise en commun de nos travaux

Nous avons eu des difficultés en première semaine à pallier à ces risques, car on a eu du mal à estimer le temps nécessaire pour compléter les tâches. Mais à partir de la seconde semaine, ces risques ont étés pleinement pris en compte, et nous avons changé notre organisation pour affecter environ 2 personnes à la gestion des tests, cela a eu aussi pour avantage que les personnes qui travaillaient sur les tests pouvaient conseiller les développeurs pour comprendre ce qui est attendu.

Par ailleurs, on a souhaité que chacun se spécialise dans un domaine particulier (rédaction de tests, développement...). Nous avons pris cette décision car on a rencontré des problèmes en première semaine, nos rôles étaient mal définis et on a perdu beaucoup de temps à s'adapter à chaque changement d'étape.

3.2.2 Risques techniques

— S'y prendre trop tard pour un rendu ou suivi

— Ne pas tester les démonstrations avant un suivi
— Introduire de la régression dans le code

Ces risques peuvent être limités en gardant à l'esprit que notre objectif est de rendre un produit qui fonctionne le mieux possible sur un sous-ensemble de ce qui est attendu. Nous avons surmonté ces risques en prenant soin de préparer une version fonctionnelle de notre présentation pour un suivi, ou de notre code pour un rendu, plusieurs heures avant la date limite.

L'importance de la fiabilité et de la robustesse a été un point-clé de notre organisation, et le but des développeurs ainsi que des testeurs a été d'assurer le fonctionnement de la plus grande partie possible du langage Deca.

3.2.3 Risques externes

— Maladie
— Contrainte extérieure
— Problème matériel
— Indisponibilité de *pcserveur*

Nous n'avons heureusement pas rencontré de problèmes liés à une absence prolongée d'un membre de l'équipe, mais nous aurions pu réagir à de tels problèmes car sur chaque étape, il y a au moins une personne chargée de la conception et une autre chargée des tests. De ce fait, en cas d'absence prolongée d'une personne, la personne chargée de la conception ou des tests aurait pu se former rapidement pour prendre en main le travail de la personne absente.

En revanche, nous avons eu à faire face à une indisponibilité de pcserveur, le fait que notre travail était aussi bien disponible sur nos machines que sur le dépôt Git nous a permis de travailler tout de même.

3.3 Gestion des rendus

Nous nous sommes mis d'accord pour éviter au maximum de modifier les jours de rendus des parties critiques du code, et à se limiter à des tests et corrections mineures. Cela nous a semblé essentiel pour éviter le risque d'une erreur introduite sous la pression, qui engendrerait des problèmes dans la version rendue.

Le travail qui précède chaque rendu peut se résumer ainsi :
— Être sûr des documents et / ou du code qui devra être rendu
— S'assurer que le compilateur passe la base de tests sans régréssions, et surtout le script "common-tests.sh"
— Publier sur la branche master de git plusieurs heures avant l'heure limite une version fonctionnelle, pour anticiper d'éventuels problèmes
— Limiter au maximum les modifications dans le code du compilateur, et rédiger de nouveaux tests

— Vérification de la base de tests
— Vérifier que la branch master contient bien tous les fichiers, et faire des essais sur plusieurs ordinateurs

Cette procédure a été un succès, chaque rendu s'est très bien passé et nous étions tous concentrés avant les rendus à tester et renforcer l'existant et non pas à développer de nouvelles fonctionnalités hasardeuses.

4 Résultats de Cobertura

4.1 Introduction

La couverture totale est de 3426 lignes de code. Une analyse du rapport de couverture a mis en évidence qu'une grande partie des lignes de code non couverte viennent du paquetage "fr.ensimag.deca.syntax".

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	247	73% 3426/4655	54% 790/1456	1.685
fr.ensimag.deca	5	60% 105/174	36% 30/83	2.821
fr.ensimag.deca.codegen	2	91% 61/67	73% 22/30	1.545
fr.ensimag.deca.context	20	84% 191/227	67% 27/40	1.312
fr.ensimag.deca.syntax	52	63% 1328/2100	43% 365/847	2.035
fr.ensimag.deca.tools	4	61% 24/39	83% 5/6	1.308
fr.ensimag.deca.tree	84	85% 1471/1721	76% 326/426	1.606
fr.ensimag.ima.pseudocode	26	78% 169/216	62% 15/24	1.163
fr.ensimag.ima.pseudocode.instructions	54	69% 77/111	N/A N/A	1

FIGURE 1 – Couverture des tests dans chaque paquetage

4.2 Étape A

Au sein de ce paquetage, la classe "DecaParser" regroupe l'essentiel des lignes non couvertes. La moitié des lignes non couvertes dans cette classe viennent des asserts utilisés dans le cadre de la programmation défensive. L'autre moitié provient de code effectivement non couvert, qui gagnerait à être couvert par de nouveaux tests dans la partie A.

Coverage Report - fr.ensimag.deca.syntax

Package	# Classes	Line Coverage	Branch Coverage	Complexity
fr.ensimag.deca.syntax	52	63% 1328/2100	43% 365/847	2.035

Classes in this Package	Line Coverage	Branch Coverage	Complexity
AbstractDecaLexer	69% 57/82	56% 18/32	3.5
AbstractDecaLexer$1	N/A N/A	N/A N/A	3.5
AbstractDecaLexer$IncludeSaveStruct	100% 5/5	N/A N/A	3.5
AbstractDecaLexer$SkipANTLRPostAction	100% 1/1	N/A N/A	3.5
AbstractDecaParser	96% 25/26	50% 3/6	2
AbstractDecaParser$1	70% 7/10	75% 3/4	2
CircularInclude	100% 4/4	N/A N/A	1
DecaLexer	84% 55/65	64% 11/17	1.611
DecaParser	67% 1046/1559	41% 322/774	2.042

FIGURE 2 – Couverture des tests dans le paquetage "fr.ensimag.deca.syntax"

Les exceptions levées en étape A sont totalement couvertes.

4.3 Étape B

Les lignes non couvertes en étape B proviennent en très grande majorité de la décompilation. Les tests unitaires n'utilisant pas la décompilation, ces lignes ne peuvent être couvertes dans le rapport Cobertura.

4.4 Étape C

On constate en analysant les classes qui relèvent de l'étape C que les lignes non couvertes proviennent d'instructions en assembleur non utilisées, ainsi que de constructeurs dans certaines instructions en comportant plusieurs, qui ne sont pas utilisés.

5 Méthodes de validation utilisées autres que le test

La relecture de code, en complément des jeux de tests, a permis d'identifier des erreurs dans le compilateur. L'utilisation de l'outil Git permet de voir facilement les modifications qui ont étés apportées au code, et ainsi pouvoir corriger d'éventuelles erreurs introduites.

Le développement du parseur s'est fait suivant la méthode de programmation défensive, au moyen d'assertions disposées dans le code, pour vérifier que les pré-conditions sont toujours remplies. Un point qui a été relevé durant la phase de test est l'apparition d'une erreur récurrente sur l'opérateur d'union disjointe sur plusieurs règles, mais la correction d'une règle n'entraînait pas la correction de toutes les autres erreurs du même type. Cela semble mettre en exergue une erreur de conception, il aurait fallu par exemple créer une méthode chargée de faire l'union disjointe en respectant les conditions associées. Il aurai été préférable d'adopter le mode de programmation défensive pour les étapes B et C, afin d'augmenter la robustesse de notre compilateur, en complément des tests.