PHELMA GRENOBLE INP
ENSIMAG GRENOBLE INP

Validation du compilateur Deca

DIJS Thomas FALGAYRAC Loic HO-SUN Jules
NOIRY Sylvain WANG Caroline
groupe GL 54

Phelma SEOC/ Ensimag ISI-MMIS
28 janvier 2022

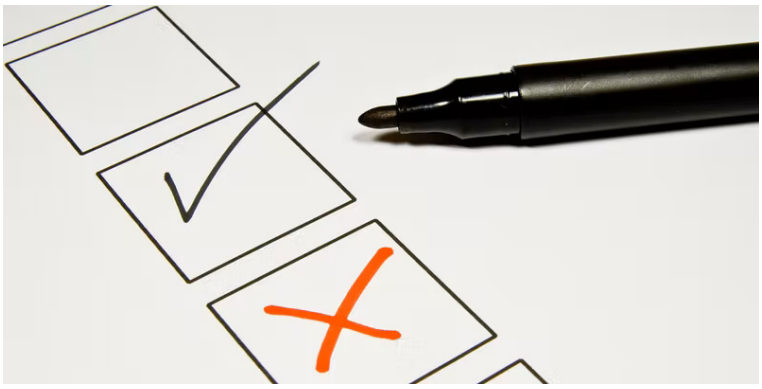

Table des matières

1 Introduction

Dans le développement de tout logiciel, la validation occupe une place majeure. En effet, elle permet d'éviter tous les bugs au cours de son utilisation future. La validation se fait par des tests, car c'est le seul moyen de vérifier qu'un programme qui compile fonctionne. L'importance des tests réside dans leur diversité : ils doivent simuler tous les codes possibles qu'un utilisateur pourrait faire, mais aussi toutes les erreurs possibles et imaginables. Pour cela, la diversité des tests est primordiale, et l'objectif est d'arriver à une couverture aussi importante que possible (une couverture de 75% est demandée pour le projet).

2 Automatisation de tests

Avant de commencer le projet, nous avons décidé d'automatiser les tests, ainsi, le lancement de tests est devenu bien plus rapide et nous n'avions plus à nous occuper de cet aspect *lancement répétitif*.

2.1 Script d'automatisation

Le script de lancement de tests se nomme **text_res_gen.sh** et se situe dans le répertoire */src/test/deca/*. La commande se lance de la manière suivante:

```
./test_res_gen.sh <chemin_du_fichier>
```

Cette commande va également permettre de différencier les tests suivant le répertoire dans lequel ils sont, et va générer des fichiers *.lis* ou *.ass* suivant le répertoire sélectionné. Lest tests qui ne sont pas correctement positionnés dans un des répertoires *valid* ou *invalid* ne sont donc pas repérés ni testés.

2.2 Pipeline sur git

Pour s'assurer de l'intégrité en tout temps de notre dépôt git, nous avons décidé de mettre en place un *Pipeline* afin d'être certain qu'à chaque push, la branche en question reste intègre et compile correctement. C'est encore plus le cas sur la branche **Master**, qui doit continuellement fonctionner, afin de ne pas prendre le risque de la bloquer et de rendre un projet non compilable.

3 Tests pour les options de compilation

La vérification pour les options de compilation a été facile à mettre en place car elle ne nécessite pas énormément de tests différents. En effet, dès que ces tests fonctionnent avec quelques programmes un minimum complexes, ils fonctionnent avec tous les autres programmes. Les tests dans cette partie ont surtout permis de montrer que les options *-p* et *-v* sont incompatibles ou à renvoyer des erreurs lorsque l'option *-r X* est mal utilisée. Nous n'avons pas fait

de tests particuliers pour les options de compilation mais avons plutôt repris des tests justes et faux déjà implémentés pour les parties suivantes.

4 Tests pour le lexer et le parser

Le lexer et le parser sont l'entrée du compilateur, ils vont devoir gérer les erreurs grossières et renvoyer les messages d'erreurs en conséquence. Le parser va créer un arbre qui permettra la vérification contextuelle ainsi que la génération de code. De cette étape va dépendre les suivantes, il est donc primordial de la tester correctement.

4.1 La compilation

4.1.1 Le lexer

Les tests pour le lexer sont les plus simples à réaliser. En effet, le lexer ne fait que de la lecture: il reconnaît certains symboles et n'en reconnaît pas d'autres. Le lexer et le parser ayant été développés en même temps, l'immense majorité des tests de la vérification syntaxique ont été réalisés dans le parser. Les tests du parser se limitaient aux symboles qui n'existent pas, comme le symbole #.

4.1.2 Le parser

Le parser a pour objectif d'effectuer la vérification syntaxique du code et de générer l'arbre. Parmi les tests invalides pour le parser on retrouve:

- l'oubli d'un caractère : un *;* , une opération binaire avec une seule opérande...
- un caractère qui n'a pas sa place : deux opérateurs qui se suivent (*a*/b*), une phrase non commentée qui aurait dû l'être...
- l'utilisation incorrecte de mots réservés : *int while = 2*, *boolean if = true*...
- un appel de constructeur avec des arguments interdit en déca : *A a = new A(3)* ...
- un appel de méthode ou un constructeur sans parenthèses: *a.method;*, *A a = new A;*...

Lorsqu'un test de la syntaxe est correct et vérifié, un fichier *.lis* est généré. Celui-ci contient l'arbre non décoré réalisé par le parser. Nous avons pu comparer les arbres réalisés aux exemples fournis dans le polycopié et s'assurer ainsi que pour des exemples valides simples, le parser fonctionne correctement.

4.2 La décompilation

La lecture d'un arbre est quelque chose d'assez inhabituel et compliqué pour une personne. Il est donc assez difficile de vérifier qu'un arbre est correct. Le meilleur moyen pour vérifier l'exactitude de l'arbre reste donc la décompilation, qui permet de transformer un arbre en code deca. Il est alors beaucoup plus

facile de comparer ce code généré avec celui saisi en entrée. Si les deux codes sont similaires, il y a de très fortes chances que l'arbre soit correct.

5 Tests pour le contexte

Une fois que l'arbre est généré et que la syntaxe est correcte, il faut faire la vérification contextuelle et la décoration de l'arbre. Ces opérations se font dans le répertoire *src/main/java/fr/ensimag/deca/context*. Il s'agit d'abord de réaliser une dernière vérification avant la génération du code assembleur, mais aussi de préparer celle-ci avec des décorations sur l'arbre. Ces tests vont donc être beaucoup plus nombreux et poussés. En effet, 95% des tests invalides restants le sont à cause du contexte.

5.1 Vérification contextuelle

Avant de commencer la décoration, il faut donc effectuer la vérification contextuelle de l'arbre. Il s'agit de tester toutes les erreurs contextuelles possibles et imaginables, mais aussi de vérifier que toutes les subtilités du langage deca soient acceptées par le compilateur.

Parmi les erreurs testées, on trouve:

- Un appel à un identificateur non initialisé : *a = 2*
- Une mauvaise affectation : *int a = 3.5*, *boolean a = 4*
- Une mauvaise utilisation des conditions : *if (3)*
- Des opérations avec des mauvaises opérandes : des opérations booléennes avec des nombres, des opérations arithmétiques de booléens ...
- Une affectation d'une variable vide : *void a = ...*
- Une double définition de classe : *Class A{} A{}*
- Une utilisation de méthode non implémentée
- Une utilisation de méthode avec des arguments incorrects : appel d'une méthode sans argument alors qu'il en faut ...
- L'absence de *new* dans les constructeur s: *A a = A()* ...
- Un mauvais appel à une méthode : *A.method()*
- Une réassignation d'un identificateur : *float f = 4.6, int f = 2*

5.2 Vérification de la décoration de l'arbre

Il a été assez difficile de vérifier que la décoration de l'arbre était correcte ; nous nous sommes beaucoup aidés des arbres décorés fournis dans le polycopié. Après les avoir rapidement comparés avec ceux fournis, nous sommes directement passés aux tests de la génération de code: si le code est généré correctement, alors l'arbre est très certainement décoré correctement. Ainsi, tous les tests de la partie *context* se concentrent uniquement sur la vérification contextuelle : il est très dur pour une personne de vérifier visuellement qu'un arbre est correct, alors que le générateur de code est supposé parfaitement le lire.

6 Tests pour la génération de code

Lorsque le contexte est correct, il ne reste plus qu'à générer le code. Néanmoins, il reste une dernière vérification. Celle-ci permet d'éviter l'allocation de ressources non disponibles ou des opérations impossibles. Les tests implémentés permettent donc de vérifier que les points suivants soient bien respectés :

- Le non dépassement des ressources mémoires. La bonne implémentation de cette exception d'exécution est vérifiée grâce à une boucle infinie *while (true)*
- L'absence de divisions par 0 ou par 0.0 qui renvoie une erreur avant de générer le code assembleur.

7 Extension optimisation

Nous avons choisi l'extension optimisation. Il a donc fallu logiquement tester que, d'une part, le nouveau code engendré était correct, et de l'autre, que les modifications apportées diminuaient bien le nombre de cycles d'horloge nécessaires à l'exécution du programme.

7.1 Code généré

Pour la vérification du code généré, il est difficile de comparer avec les codes assembleurs générés précédemment (il est supposé être optimisé). Il s'agit donc d'utiliser les mêmes tests avec les options d'optimisation *-O1* ou *-O2* dans notre cas et de comparer les résultats attendus avec ceux espérés. Il faut donc là aussi passer par de nombreux tests.

Il est cependant possible d'aller plus loin dans l'analyse du code optimisé grâce à l'option de compilation *-O2g*. Cela va générer un fichier *.dot* qui contient un graphe. Ce graphe peut être lu via la bibliothèque *graphviz*. On peut donc comparer facilement les différentes boucles de ce graphe avec les conditions *while, if, else, ...* et les différentes branches créées.

7.2 Gain en terme de cycles

Maintenant que le code généré est vérifié et qu'il correspond aux attentes en terme de validité, il faut vérifier que l'optimisation permette aux programmes de s'exécuter plus rapidement, en d'autres termes, il faut vérifier que l'optimisation ne soit pas inutile.

Pour cela, on va majoritairement utiliser les trois tests fournis dès le début du projet, dans le répertoire *src/tests/deca/codegen/perf/provided*. Ces tests que nous n'avons jamais touché, en plus de nous permettre de constater notre avancement, nous permettent de nous comparer aux autres groupes via l'onglet *Palmarès 2022* du site internet du Projet Génie Logiciel. En se basant sur ces tests, nous avons réussi à optimiser suffisamment le compilateur pour finir premier au palmarès, avec plus de 7000 cycles d'avance sur le deuxième groupe.

8 Perspective d'améliorations

La couverture de tests de notre compilateur est de 77% ; c'est plus que le minimum demandé, mais cela reste assez loin des 100%. Ici, nous allons présenter quelques perspectives d'amélioration de notre code.

8.1 Tests avec objet

L'implémentation du compilateur avec objet a eu lieu assez tardivement, et nous n'avons pas commencé les tests avec objet tant que le compilateur n'était pas suffisamment avancé. La réalisation de tests avec objet a donc été assez courte et s'est effectuée en partie en parallèle de la rédaction de la documentation. Une grande partie de la non couverture des tests provient donc sûrement de l'absence de tests avec objet. Une perspective d'amélioration de la couverture de tests serait donc de développer plus de tests avec objet, notamment au niveau de l'appel des méthodes et de l'hérédité.

8.2 Plus de tests

Il est évident que les tests sur les classes ne sont pas les seuls à manquer. Les 23% de couverture de test restants concernent également en partie le langage sans objet. Nous avons fortement sous-estimé l'importance des tests, surtout au début du projet, et avons préféré foncer *tête baissée* dans l'implémentation de ce compilateur plutôt que de commencer par déployer une couverture de tests. Nous avons tout de même réussi à implémenter plus de 250 tests, ce qui reste une base assez solide pour un compilateur.

8.3 Une politique de tests différente

Jusqu'à aujourd'hui, nous faisions les projets seuls, en binômes voire grand maximum en trinômes. Cela nous limitait dans la répartition des tâches, et donc chacun s'assurait que sa partie fonctionnait. Pour un projet tel que le compilateur, nous avons à disposition une équipe de 5 personnes, ce qui nous laisse une plus grande liberté dans la répartition des tâches. Nous avons donc choisi dès le début du projet que pour éviter des problèmes de biais cognitif, les personnes qui développent une partie ne la testent pas. Personne dans le groupe n'avait jamais essayé cette manière de tester. Les tests sont fait par des membres de l'équipe qui n'y ont pas ou peu participé. C'est d'un côté une excellente solution car cela empêche un membre de l'équipe de bâcler sa partie, mais la personne qui écrit le test n'est pas forcément au courant de toutes les subtilités de la partie, ce qui l'empêche de la tester de manière totale. La personne qui écrit les tests n'est pas non plus tout le temps au courant de ce que le compilateur doit renvoyer.

9 La qualité de notre compilateur

Nous sommes conscients que notre compilateur n'est pas efficace à 100%, mais il n'en est pas pour autant très éloigné. Sur la fin du projet, nous avons en effet décidé de nous concentrer sur l'extension, quitte à limiter un peu le développement sur le reste du compilateur.

Nous estimons le temps nécessaire à la finalisation totale du compilateur à 2 jours maximum (en prenant en compte le développement d'une couverture de test bien plus importante) et à 8h si nous nous occupons seulement des bugs que nous avons pu recenser jusqu'à aujourd'hui.