

ENSIMAG
PROJET GL
GL 59

Documentation de validation



WALID BAKIR

WALID ABIDI

ABDELOUAHAB MESTASSI

EL MEHDI SALAH-EDDINE

YASSIN EL KHALDI AHANNACH

1 Description des tests

1.1 Types de tests

Pour s'assurer de la qualité de notre compilateur tout au long du développement, on a mis en place plusieurs types de tests qui sont écrits sous la forme de programmes.deca. Nos tests sont très diversifiés pour recouvrir chaque partie du compilateur, en commençant par des tests simples qui permettent de vérifier que le compilateur respecte la grammaire attribué et son bon fonctionnement sur des programmes plutôt basiques et d'autres plus complexes pour tester la robustesse de notre compilateur. On retrouve :

- Tests lexicographiques
- Tests contextuelles
- Tests de génération du code
- Tests de l'interface de commande du compilateur et de ses options

On a bien veiller à respecter les conventions de nomination des résultats attendus pour les tests contextuels ainsi que ceux du code assembleur.

1.2 Organisation des tests

On ce qui concerne l'organisation des tests, on a réparti les tests dans des sous-répertoires du répertoire **src/test/Deca**, on retrouve :

- les tests de syntaxes valides qui devraient passer le test de syntaxes sans problèmes dans **syntax/valid**
- les tests de syntaxes invalides qui doivent lever une erreur de syntaxe par le compilateur dans **syntax/invalid**
- Les tests contextuels valides avec et sans objets dans **context/valid**
- Les tests contextuels invalides dans **context/invalid**
- Les tests de la partie C avec et sans objets valides dans **codegen/valid/provided**
- Les tests de la partie C avec et sans objets invalides dans **codegen/invalid/provided**
- Les tests de l'extension TRIGO dans **trigo**

1.3 Objectifs des tests

La partie test joue un rôle primordial pour s'assurer de la qualité d'un projet et de la validité du code des différentes parties indépendamment des autres pour relever

leurs limites et palier aux problèmes détectés avant d'être utilisée par la suite dans l'ensemble du projet .

2 Les scripts de tests

Afin d'automatiser la procédure de tests, on fournit différents scripts dans le répertoire `src/test/script`. Ces scripts sont intégrés dans le fichier de configuration `pom.xml`. Ainsi, `mvn test` passe sur l'ensemble de ces scripts.

`valid-syntax.sh` passe sur l'ensemble des tests valides de syntax.

`valid-context.sh` passe sur l'ensemble des tests contextuelles valides.

`valid-codgen.sh` passe sur l'ensemble des tests valides de génération de code.

`invalid-syntax.sh` passe sur l'ensemble des tests invalides de syntax.

`invalid-context.sh` passe sur l'ensemble des tests invalides de context.

`invalid-codgen.sh` passe sur l'ensemble des tests invalides de génération de code.

`Error-messages-context.sh` affiche l'ensemble des erreurs contextuelles possibles.

`decompile.sh` teste la décompilation des fichiers de tests deca.

3 Gestion des risques et gestion des rendus

Ce projet GL nécessite un grand volume de travail, donc une bonne gestion des rendus et des risques s'impose pour permettre aux membres du groupe de bien s'organiser, de se répartir les missions et d'atteindre l'objectif final.

3.1 Les différentes dates de rendus :

Au cours de ce projet nous serons amenés à rendre plusieurs rendus qui ont des dates limites différentes ,et donc y a un risque d'oublier une de ces dates .Pour cela l'équipe à prévu des dates de travail accéléré en équipe pour les taches critique (tel que le rendu intermédiaire ,le compilateur et les documentations du rendu) pour que ça soit prêt avant sa date limite .Ces dates importantes ainsi que les rendus ont bien été marqués sur le planning prévisionnel afin de les garder en vue et d'avoir une vue globale plus claire sur le projet .

3.2 Compétence communication écrit :

L'équipe s'assure toujours d'avoir une bonne communication entre les membres, afin d'échanger plus d'information et avancer plus rapidement cependant le choix de répartition des taches n'était pas parfaitement judicieux, et il fallait de temps en temps que plusieurs membres contribuent dans une seule partie .L'équipe n'a pas opter pour la préparation des documentations dès le début au fur et à mesure du travail sur le compilateur.

3.3 Non respect des spécifications :

Il est primordial de s'assurer que le compilateur (logiciel) soit conforme aux exigences du client, tel que les spécifications de la ligne de commande de deca doivent être respecté. Pour cela l'équipe fait des relectures à chaque fois du cahier des charges qui est notre poly du projet GL et pour chaque partie afin de s'assurer que chaque détail est respecté.

3.4 Répartition des tâches non équilibré :

Puisque on a opté pour la méthode d'agile, le volume de travail pendant chaque étape critique peut varier fortement selon chaque partie (étape A, B et C) ce qui peut engendrer une prise de retard sur le planning prévisionnel posé dès le début .Ainsi on fait des réunion régulières pour faire le point l'avancement et ce qui reste à faire pour ensuite répartir les taches de manière à atteindre le but avant sa date limite .

3.5 Les tests non fiables :

Les tests de la partie logicielle rédigée par le développeur risque de ne pas relever une erreur (possiblement grave), dans ce cas on aura de test biaisés .Pour cela, on essaye de faire relire les tests par d'autres membres, ou les rédiger par un autre que celui qui l'a implémenté.

3.6 L'implantation de l'extension TRIGO :

Il existe plusieurs approches pour implémenter les fonctions trigo, comme le développement illimité (DL) ou l'algorithme CORDIC qui s'avère une meilleur méthode pour calculer l'approximation qu'aux séries de Taylor .Cependant on est contraint par l'analyse énergétique qui est une partie importante du projet ,donc le but optimal est de trouver un équilibre entre la précision de calcul et le cout énergétique du produit.

4 Résultats de Jacoco

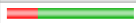

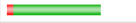










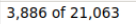

Deca Compiler												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75%		54%	542	751	492	2,017	249	375	2	52
fr.ensimag.deca.tree		92%		85%	82	655	116	1,749	32	462	0	86
fr.ensimag.deca		65%		56%	50	99	115	286	16	52	4	7
fr.ensimag.ima.pseudocode		76%		75%	27	84	43	180	22	74	2	26
fr.ensimag.deca.context		83%		72%	35	139	35	208	22	110	1	22
fr.ensimag.ima.pseudocode.instructions		70%	n/a	n/a	20	63	35	113	20	63	15	54
fr.ensimag.deca.codegen		80%		100%	10	28	15	55	10	22	1	2
fr.ensimag.deca.tools		93%		87%	2	20	4	44	1	16	0	3
Total	3,886 of 21,063	81%	435 of 1,272	65%	768	1,839	855	4,652	372	1,174	25	252

FIGURE 1 – Résultats de Jacoco