

Projet Génie Logiciel

Compilateur Decac

Manuel de validation

Équipe GL2

**Étudiants : Élina Houdé, Julien Pinto Da Fonseca,
Léa Solo Kwan, Yin Xu**

Sommaire

Introduction	3
1. Descriptif des tests	4
1.1. Les différents types de tests	4
1.2. Organisation des tests unitaires et de précondition	4
1.3. Organisation des tests système	4
1.2.1. Génération du code (dossier codegen)	5
1.2.2. Analyse contextuelle (dossier context)	5
1.2.3. Analyse syntaxique (dossier syntax)	6
2. Résultats de couverture du code (Cobertura)	6
3. Scripts de tests automatisés	8
4. Gestion des risques et gestion des rendus	8
5. Processus d'intégration continue	10

Introduction

Dans ce document, vous trouverez les différents tests et systèmes de validation mis en place.

1. Descriptif des tests

1.1. Les différents types de tests

Au sein du projet, nous pouvons retrouver plusieurs types de test :

- les **tests de précondition**, afin de lever l'exception *IllegalArgumentException* lorsqu'un argument d'une méthode publique ou d'un constructeur ne vérifie pas une précondition ;
- les **tests unitaires**, afin de valider la non-régression du code. Ceux-ci sont mis en place dans le but de couvrir à minima 80% du code nouveau, c'est-à-dire du code ne provenant pas du commit initial ;
- les **tests système**, afin de valider la non-régression des fonctionnalités. Il s'agit de vérifier que lors de la mise en place d'une fonctionnalité, les programmes deca qui doivent s'exécuter correctement génèrent toujours les mêmes résultats pour chacune des étapes du compilateur et que les programmes deca devant provoquer une erreur continuent de provoquer la même erreur.

1.2. Organisation des tests unitaires et de précondition

Les tests unitaires font des vérifications sur les méthodes *decompile*, *verifyXXX* et *codeGenXXX* en regardant que les sorties sont correctes et qu'il n'y pas de modifications sur les noms des attributs dans le cas des méthodes *codeGenXXX*.

Les tests unitaires sont réalisés grâce aux frameworks de test JUnit et Mockito et sont essentiellement centrés sur les classes concernant la programmation sans objet.

1.3. Organisation des tests système

Tous les programmes deca de test sont répartis en trois catégories dans le dossier **src/test/deca**.

1.2.1. Génération du code (dossier codegen)

L'étape de génération de code est traitée de par une base de tests systèmes interactif et une base de tests valides. Les programmes de tests interactifs se situent dans le sous-dossier **interactive** et sont généralement testés manuellement. Les programmes de test valides se situent dans le sous-dossier **valid**.

Chemin général depuis la racine du projet :

src/test/deca/codegen/

Chacun des tests possède :

- un fichier **.expected** contenant le résultat attendu en sortie de l'étape de vérification syntaxique. Ce fichier est placé dans le dossier **src/test/deca/syntax/valid**.
- un fichier **.expected** contenant le résultat attendu en sortie de l'étape de vérification contextuelle. Ce fichier est placé dans le dossier **src/test/deca/context/valid**.

et, dans le cas spécifique des programmes deca du sous-dossier **codegen/valid** :

- un fichier **.expected** contenant le résultat attendu lors de l'exécution du code assembleur généré. Ce fichier est placé dans le dossier **src/test/deca/codegen/valid**.

1.2.2. Analyse contextuelle (dossier context)

Dans ce dossier se situent tous les fichiers de test invalides pour l'étape de vérification contextuelle du code. Chacun de ces tests possède :

- un fichier **.expected** contenant le résultat attendu en sortie de l'étape de vérification syntaxique. Ce fichier est placé dans le dossier **src/test/deca/syntax/valid**.

- un fichier **.expected** contenant l'erreur attendu en sortie de l'étape de vérification contextuelle. Ce fichier est placé dans le dossier **src/test/deca/context/invalid**.



















1.2.3. Analyse syntaxique (dossier syntax)

Dans ce dossier se situent tous les fichiers de test invalides pour l'étape de vérification syntaxique du code. Chacun de ces tests possède :

- un fichier **.expected** contenant l'erreur attendu en sortie de l'étape de vérification syntaxique. Ce fichier est placé dans le dossier **src/test/deca/syntax/invalid**.

2. Résultats de couverture du code (Cobertura)

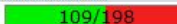
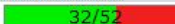
Coverage Report - All Packages




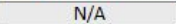
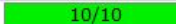
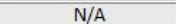
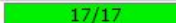
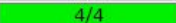
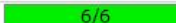
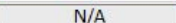
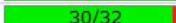


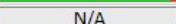

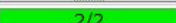
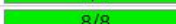
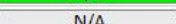
Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	258	19%  954/4960	12%  172/1418	1.688
fr.ensimag.deca	6	13%  32/246	0%  0/105	2.811
fr.ensimag.deca.codegen	9	55%  109/198	61%  32/52	2.162
fr.ensimag.deca.context	22	39%  89/226	2%  1/46	1.322
fr.ensimag.deca.syntax	50	0%  0/2044	0%  0/761	1.906
fr.ensimag.deca.tools	4	90%  36/40	83%  5/6	1.417
fr.ensimag.deca.tree	87	27%  524/1916	27%  119/428	1.64
fr.ensimag.ima.pseudocode	26	66%  118/177	75%  15/20	1.174
fr.ensimag.ima.pseudocode.instructions	54	40%  46/113	N/A  N/A	1

La couverture Cobertura ne prend actuellement en compte que les tests unitaires.

Actuellement, les tests unitaires couvrent pratiquement tous les classes Codegen ajoutées et, pour le package tree, ils couvrent environ 85% des classes concernant la programmation sans objet.

Coverage Report - fr.ensimag.deca.codegen

Package ^	# Classes	Line Coverage	Branch Coverage	Complexity
fr.ensimag.deca.codegen	9	55%  109/198	61%  32/52	2.162

Classes in this Package ^	Line Coverage	Branch Coverage	Complexity
ErrorLabelManager	15%  15/100	26%  7/26	6.167
ErrorLabelManager\$1	100%  1/1	N/A  N/A	6.167
ErrorLabelType	100%  10/10	N/A  N/A	0
LabelManager	100%  17/17	100%  4/4	1.5
LabelType	100%  6/6	N/A  N/A	0
RegisterManager	93%  30/32	95%  19/20	1.889
StackManager	86%  13/15	N/A  N/A	1
TSTOManager	100%  9/9	100%  2/2	1.333
VTable	100%  8/8	N/A  N/A	1

Concernant les tests systèmes (non pris en compte par Cobertura), nous avons écrit un test pour relever chacune des erreurs de la classe ErrorMessage. De plus, nous avons essayé de couvrir toutes les branches du code en effectuant à chaque fois un test spécifique.

3. Scripts de tests automatisés

Afin d'exécuter de manière automatique les tests système et éviter toute régression fonctionnelle, nous avons mis en place plusieurs scripts Shell :

- **basic-lex.sh** : exécute les tests touchant à l'analyse lexicale ;
- **basic-synt.sh** : exécute les tests touchant à l'analyse syntaxique ;
- **basic-context.sh** : exécute les tests de résultat de l'analyse contextuelle ;
- **basic-codegen.sh** : exécute les tests de l'étape de génération de code assembleur ;
- **basic-decac.sh** : vérifie la bonne implémentation des options du compilateur.

Chemin depuis la racine du projet :

src/test/script/

Ces différents scripts sont lancés de façon systématique lors de l'exécution de la commande ***mvn test***, commande permettant également d'exécuter l'ensemble des tests unitaires.

4. Gestion des risques et gestion des rendus

Lors de la réalisation du projet, il est nécessaire de faire attention aux risques suivants :

- **Régression du code** : afin de limiter ce risque, de tests systèmes et des tests unitaires sont mis en place, et exécuter grâce à des scripts Bash. Ces tests sont effectués de façon systématique via un pipeline d'intégration continue, il est également possible de les lancer manuellement ;
- **Non implémentation d'une spécification** : la partie n°1 de la procédure de gestion des mises en production permet de limiter ce risque ;

-
- **Commit incomplet** : lors de l'ajout d'une fonctionnalité, la fonctionnalité est testée une première fois localement par le développeur, puis une seconde fois à partir de la branche *develop*, une fois la fonctionnalité partagée ;
 - **Mauvaise gestion du planning** : pour éviter ce risque, il est nécessaire de faire des réunions régulières sur l'avancement du projet afin de, si nécessaire, adapter le planning.

Un dispositif d'intégration continue a été mis en place sur le Gitlab du projet, celui-ci a été configuré pour lancer systématiquement, lors de chaque commit sans contrainte de branche, une pipeline comprenant 3 jobs spécifiques:

- **Build** : effectue la commande *mvn clean compile* pour vérifier que la phase de build du projet se déroule sans erreur, dans le cas contraire, le job échoue.
- **Test** : effectue la commande *mvn test* pour vérifier que l'ensemble des tests systèmes et unitaires se déroulent correctement ; la vérification du score des tests systèmes s'effectue manuellement en vérifiant les logs. Si une erreur est lancée ou qu'un test unitaire ne passe pas, le job échoue.
- **Cobertura** : effectue un rapport cobertura récapitulant le taux de couverture des tests unitaires. Celui-ci peut être visible à tout moment via les logs sur le commit correspondant.

Avant chaque mise en production, nous réalisons la procédure suivante de gestion des mises en production afin de limiter les risques précédents :

1. **Relecture rapide des spécifications** : vérification qu'une règle n'a pas oublié d'être implémentée ;
 2. **Vérification via intégration continue** : on vérifie qu'aucun job n'a échoué et que les tests système et les tests unitaires ne provoquent pas d'erreur critique sur la branche *develop* ;
 3. **Vérification des tests systèmes** : afin d'éviter la régression du code, il est nécessaire de vérifier que les tests systèmes qui passaient lors du rendu précédent fonctionnent toujours. De plus, on vérifie que les options du compilateur fonctionnent correctement (lecture des logs du job *Test*) ;
-

-
4. **Vérification de la branche *Master*** : une fois la branche *develop* est fusionnée sur la branche *Master*, on vérifie que la procédure s'est bien déroulée en répétant les étapes 2 et 3 sur la branche *Master*.

5. Processus d'intégration continue

Le processus d'intégration continue est divisée en trois étapes :

- **Build** : vérifie que la compilation du code ne provoque pas d'erreur ;
- **Tests** : vérifie que les tests unitaires ne provoquent pas d'erreur et exécute tous les tests systèmes. Lors de cet étape, seuls les tests unitaires peuvent générer une erreur critique. Concernant les tests systèmes, les erreurs sont signalées mais n'empêche pas de valider le bon déroulé de cet étape ;
- **Cobertura** : vérifie que l'exécution de la commande de couverture par Cobertura ne provoque pas d'erreur.

Ce processus est exécutée à chaque fois qu'un commit est réalisé ou qu'une branche est "merge" sur les branches *develop* ou *master*.