



# DOCUMENTATION DE VALIDATION

*Projet Génie Logiciel - Equipe GL 23*

**Boukhriss Nada / Dakhil Yahya / Elhjouji Salah  
/Et-tarraf Zineb / Tamouh Alae**

*JANVIER 2022*

## SOMMAIRE

SOMMAIRE	2
INTRODUCTION	3
DESCRIPTIF DES TESTS	3
LE FORMAT DES TESTS	3
LES TESTS VALIDES	
LES TESTS INVALIDES	
LE CONTENU DES TESTS	4
TESTS DE L'ETAPE A	
TESTS DE L'ETAPE B	
TESTS DE L'ETAPE C	
SCRIPTS DE TESTS	
SYNT-TEST.SH	
CONTEXT-TEST.SH	
GENCODE-TEST.SH	
OPTIONS-DECAC.SH	
AUTOMATISATION DES TESTS	
COUVERTURE DES TESTS	7
GESTION DES RISQUES	8
GESTION DES RENDUS	9
CONCLUSION	10

## INTRODUCTION

Le but de ce projet était de concevoir un compilateur fonctionnel qui respecte les spécifications exigées par le client. Dans ce sens, l'étape de la validation était nécessaire pour vérifier que les consignes ont été respectées avant le rendu final du projet.

Dans ce document, on détaillera les différents procédés adoptés pour tester chaque étape du compilateur . En plus, on entamera les démarches adoptées pour assurer l'avancement du projet ainsi que la qualité des rendus.

## DESCRIPTIF DES TESTS

### I. LE FORMAT DES TESTS

#### A. LES TESTS VALIDES

Les tests valides sont tout simplement les tests qui ne devront pas lever des erreurs lorsque le code du test est lancé. Ils sont nécessaires dans toutes les parties du projet de l'analyse lexicale jusqu'à la compilation et l'exécution. Ils ont tous une partie commentée en tête du fichier qui contient une description de l'objet testé, le résultat pour les tests valides de la dernière partie de la génération du code qui contient le résultat désiré puisque c'est qu'à ce moment qu'on peut exécuter le programme vu que les parties précédentes devront être terminées et validées et finalement une date du jour de l'écriture du test pour garder une historique du travail. Après cette partie commentée, on trouve le code à tester.

#### B. LES TESTS INVALIDES

Les tests invalides ont aussi une importance égale aux tests valides. En effet, si les tests valides examinent ce que le compilateur doit faire, les tests invalides mettent le doigt sur les erreurs à lever si une des spécifications n'est pas respectée.

Le format des tests invalides est très similaire à celui des tests valides. On y trouve toujours une description de ce qu'on désire tester après on trouve le résultat qui

correspond au message d'erreur attendu précédé du numéro de la ligne et la colonne de l'erreur. Ensuite, comme pour les tests valides, il y a la date de l'écriture du test et finalement le code du test.

## II. LE CONTENU DES TESTS

### A. TESTS DE L'ETAPE A

Les tests de cette partie sont écrits pour tester l'analyseur lexical et syntaxique du compilateur. Pour les tests valides, on a écrit des tests correspondant à la première phase du projet, c'est-à-dire la phase sans objet et pour la deuxième phase avec objet, les tests sont mis dans un sous-dossier **Objet**. Cette distinction nous a permis de subdiviser le travail de validation de cette étape en morceaux afin de s'assurer de son bon fonctionnement. Pour examiner un de ces tests, on lance tout d'abord la commande **test\_lex <nom\_du\_fichier>** afin de voir tout le lexique utilisé dans le code. Après on lance la commande **test\_synt <nom\_du\_fichier>** pour dessiner l'arbre abstrait et non décoré du code.

Quant aux tests invalides, l'emplacement des tests est un peu différent puisque cette fois on a distingué entre les tests invalides pour l'analyseur lexical et pour l'analyseur syntaxique. Ainsi, on trouve les dossiers **lexique**, **syntax** et **Objet**.

Cette structure a pour but de montrer séparément au client des exemples de code qui ne doivent pas marcher selon les spécifications au niveau lexical et au niveau syntaxique.

### B. TESTS DE L'ETAPE B

La logique de l'organisation des ces tests est très similaire à l'étape A. Soit pour les tests valides ou bien les tests invalides, on a mis les tests de l'analyseur contextuel sans objet en premier et dans les dossiers **objet** de chaque répertoire on trouve les tests du contexte avec **Objet**. Le but de ces tests est l'examen des règles prédéfinies dans la syntaxe contextuelle et aussi l'illustration de l'arbre abstrait enrichi. Pour ce faire, on lance la commande **test\_context <nom\_du\_fichier>**. Pour les tests valides, on attend en sortie un arbre abstrait décoré après avoir vérifié les règles contextuelles du compilateur. Et concernant les tests invalides, on attend que le compilateur lève le message d'erreur qui correspond à l'erreur testé dans le code.

## C. TESTS DE L'ÉTAPE C :

Finalement, on trouve les tests de l'étape de la génération du code et l'exécution. La structure des dossiers des tests de cette étape est la même que celle de l'étape B sauf qu'ici on ajoute des tests dits **interactifs** autres que valides/invalides. Dans ces tests, la sortie de l'exécution dépend de l'entrée que met l'utilisateur ainsi on ne peut pas parler d'un résultat prévu. Or, ces tests ont été examinés et validés par l'équipe qui a affirmé que ces tests ont bien le comportement désirés.

L'examen des tests dans cette partie se fait en deux parties. La première est la compilation qui génère le code assembleur grâce à la commande **decac <nom.deca>** et après on exécute le code assembleur par la commande **ima <nom.ass>** et on compare finalement la sortie avec le résultat attendu.

## III. SCRIPTS DE TESTS

Afin d'automatiser les tests des différentes parties, on a implémenté des scripts shell associés à chaque étape. Et dans ce qui suit, on citera chaque script et son fonctionnement.

### A. SYNT-TEST.SH

Ce script est spécifié pour lancer les tests de l'analyseur lexical et syntaxique. Il parcourt d'abord les tests valides de cette étape et il lance les commandes **test\_lex <nom\_du\_fichier>** et **test\_synt <nom\_du\_fichier>**. Il vérifie ensuite la sortie; s'il y a une erreur ou une exception qui est levée, il renvoie en rouge un message indiquant que ce test a renvoyé une erreur imprévue. Sinon, on voit un message en vert qui valide le test concerné. Quant aux tests invalides, il les parcourt tous aussi et lance les commandes précédentes; si la sortie contient le message d'erreur écrit dans le fichier **.des** portant le nom du fichier test, donc on a bien le résultat attendu et on voit un message de validation en vert. Sinon, il renvoie un warning en rouge.

## B. CONTEXT-TEST.SH

Dans ce script on trouve exactement le même fonctionnement que dans le script précédent. Il lance la commande **test\_context** <nom\_du\_fichier> et examine la sortie. Les fichiers **.des** dans les répertoires des tests invalides soient du contexte ou du syntax contiennent les messages d'erreur attendu. Ceci nous a permis de valider finement notre compilateur, dans le sens où on s'assure que l'erreur renvoyer et bien l'erreur qu'on attend.

## C. GENCODE-TEST.SH

La manière dont fonctionne ce script est un peu différente des scripts qui précèdent. Les codes de test écrits sont censés compiler et donc on a toujours un retour du compilateur après l'exécution soit pour les tests valides et les tests invalides et donc les **.des** sont présents dans les deux répertoires. Pour le répertoire des codes invalides, c'est pareil aux tests contextuels et syntaxiques, ils contiennent les messages d'erreurs attendus. Or pour le répertoire des codes valides, on met la sortie désirée après exécution du code assembleur généré .

## D. OPTIONS-DECAC.SH

Pour valider toutes les spécifications de notre compilateur, il était nécessaire que nous écrivions un script de test pour valider le fonctionnement des options du compilateur. Ce script lance plusieurs commandes **decac -options** et s'assure que la sortie ne contient pas des résultats inattendus. La logique dans ce script ne consiste pas sur le parcours de tous les codes de test mais on s'est contenté de tester un fichier test pour économiser de l'énergie puisque ce parcours est jugé inutile et n'apporte aucun plus à la validation des options du compilateur.

## E. AUTOMATISATION DES TESTS
















L'automatisation de ces tests est une étape très importante dans ce projet puisque ça rend le procédé de validation plus facile a mené. Cette automatisation se fait à partir du fichier **pom.xml** qui est déjà fourni et il nous suffit de mettre les scripts qu'on veut lancer. Ainsi, on lance la commande **mvn test** qui déclenche le processus de validation et affiche les tests passés de toutes les étapes du testing. Cette automatisation est aussi importante pour rendre l'étape de la validation plus extensible vu qu'il suffit au client de mettre le code du test dans le répertoire associé

avec le fichier **.des** (si besoin) pour qu'il soit aussi inclus dans la batterie des tests évalués.

Les scripts écrits jouent aussi un rôle non négligeable pour rendre les tests réutilisables dans le sens où le client à la liberté de choisir n'importe quel répertoire et le parcourir pour vérifier le fonctionnement du compilateur juste en changeant le chemin vers le fichier désiré.

## IV. COUVERTURE DES TESTS

Afin de quantifier la couverture de nos tests et pour s'assurer que notre procédé de validation a une certaine fiabilité on a utilisé l'outil Jacoco à l'aide de la commande **mvn -Djacoco.skip=false verify** qui génère un fichier **html** dans **target/site/**. Le résultat est représenté dans la figure ci-dessous :

Deca Compiler													
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes	
fr.ensimag.deca.tree		61%		57%	372	825	910	2,328	205	513	3	86	
fr.ensimag.deca.syntax		76%		54%	562	767	490	2,051	244	370	1	48	
fr.ensimag.ima.pseudocode		57%		70%	52	112	104	248	45	100	7	31	
fr.ensimag.deca		71%		61%	44	102	89	283	16	54	1	6	
fr.ensimag.deca.codegen		56%		53%	19	46	67	161	7	26	0	2	
fr.ensimag.ima.pseudocode.instructions		44%	n/a		43	83	80	153	43	83	28	65	
fr.ensimag.deca.context		83%		89%	28	145	43	266	24	126	0	22	
fr.ensimag.deca.tools		43%		37%	11	19	28	43	8	15	1	3	
Total	8,038 of 25,450	68%	665 of 1,551	57%	1,131	2,099	1,811	5,533	592	1,287	41	263	

Created with JaCoCo 0.8.7.

Fig. Résultat de la couverture des tests par Jacoco

Ces résultats sont moyennement rassurants, on a une bonne couverture de 76% pour les tests syntaxiques et 83% pour les tests du vérificateur contextuel. Par contre, la couverture des tests de la génération du code dépasse la moitié de 6%. Donc, il faudrait faire plus de tests de la partie C mais dans la globalité on peut dire que le testing a une légitimité considérable et il fait son travail de tester le fonctionnement du compilateur.

## V. GESTION DES RISQUES

Être proactif et prévoir les cas de risque et d'urgence dans un projet est une compétence très essentielle pour des jeunes ingénieurs. Dans ce projet, la gestion des risques est une étape nécessaire pour gagner la confiance du client et faire preuve d'un haut professionnalisme. Ci-dessous, on a marqué les dangers potentielles qui pourraient nous rencontrer durant le projet et comment on pourrait y faire face pour assurer l'avancement normal du projet:

Danger	Action
Manquer la date d'un rendu	Merge vers la branche master au moins 1 heure avant l'heure de rendu
Oublier d'implémenter une des options de la commande decac.	Créer un test qui lance decac avec des fichiers vides mais des options différentes
Oublier un token dans DecaLexer qui pourra par la suite interrompre tout le processus de compilation	Lancer test_lex sur un fichier contenant tous les caractères de la partie Lexicographie du poly.
Oublier d'implémenter une règle de grammaire	Mettre en place un test pour chaque règle de grammaire / Relecture croisée du code et tests
Oublier un générateur de code	Relire la grammaire abstraite et faire le point sur les fichiers associés.
Version du projet sur git ne compile pas à cause d'un manque de fichiers par exemple	Checker régulièrement les versions et garder toujours une archive du clonage git dans sa machine perso.
Corruption du git du projet	////
bug dans l'infrastructure de test	
Manquer la documentation d'une partie du projet	Relecture attentionné des documentations et de ce qui est attendu



Oublier de tester le projet sur la machine ciblée, c'est-à-dire la machine client (un pc ensimag).	Rencontres régulières à l'ensimag pour tester sur machine.
Manquer de temps pour l'implémentation de l'extension	Codage au possible de l'extension en parallèle de l'étape C
Oublier la date d'un suivi et ne pas pouvoir préparer les documents demandés à temps.	Check régulier de l'emploi du temps du projet pour préparer les suivis à l'avance.
Mélanger les générateurs de code IMA et ARM	Toujours tester le fichier assembleur généré.
Ne pas pouvoir cerner les dimensions énergétiques du projet	Se fixer des deadlines pour réfléchir aux méthodes de tests d'énergie.

## VI. GESTION DES RENDUS

Durant ce projet, il y avait des dates de suivi et des rendus qui nécessitent la préparation de quelques démos et des documentations. Dans ce sens, on devrait faire des démarches et des actions pour éviter qu'on se plonge dans le projet et qu'on laisse les rendus à la dernière minute, chose qui va impacter la qualité de nos livrables. Ci-dessous, on trouve les démarches qu'on a décidé de faire pour éviter tout oubli d'un rendu.

- Tester avec les scripts de tests donnés.
- Lancer les tests unitaires.
- Lancer la batterie de tests précédemment élaborée sur le projet sur toutes nos machines personnelles avant de push.
- Relancer tous ces tests sur une "machine témoin" à l'école, afin de s'assurer du bon fonctionnement de notre compilateur.
- Push la version fonctionnelle vers la branche de travail.
- Clonage du dépôt de cette branche puis on relance tous nos tests sur les machines personnelles et la machine de l'école.
- Merge la branche de travail sur le master puis on clone et on relance les tests.

- Vérifier les documents à rendre et se mettre d'accord sur leur contenu final.
- Ecrire les documentations au fur et à mesure de l'avancement du projet
- Faire des réunions quotidiennes pour discuter et faire le bilan
- Revoir le code par plusieurs personnes pour avoir différents avis

## VII. CONCLUSION

Pour conclure, après toutes ces procédés de validation qu'on a effectué, on peut dire avec confiance que nous avons réussi à faire un compilateur fonctionnel qui respecte les spécifications exigées. Or, on est loin du compilateur parfait et il y a plusieurs pistes d'améliorations possibles qui peuvent être effectuées pour raffiner ce compilateur.