



PROJET GÉNIE LOGICIEL

2023/2024

Documentation de Validation du Compilateur Deca

Antoine Borget
Pierre Lin
Anass Haydar
Yassine Saфраoui
Jihane Zemmoura

Groupe GL-47

Table des matières

1	Introduction	2
2	Méthode de validation	3
2.1	Descriptif des tests	3
2.2	Scripts de tests	3
2.2.1	Script de test de l'étape A	3
2.2.2	Script de test de l'étape B	4
2.2.3	Script de test de l'étape C	5
2.2.4	Script de test pour toutes les étapes	6
3	Couverture des tests	7
4	Gestion des risques et gestion des rendus	9
5	Méthodes de validation utilisées autres que les tests	10
5.1	Validation de performance	10
5.2	Validation de la documentation	10

1 Introduction

Le document explore en détail les méthodes de validation mises en œuvre pour assurer la qualité du compilateur développé.

Les tests sont élaborés à chaque étape du processus de compilation, catégorisés en tests valides et invalides selon divers sous-langages.

Des scripts dédiés sont créés pour chacune des étapes, tels que les scripts :

- **tests_lexico.sh** et **tests_syntaxe.sh** pour l'étape A.
- **tests_context_invalides.sh** pour l'étape B.
- **tests_codegen.sh** pour l'étape C.

La couverture des tests est évaluée avec l'outil **Jacoco**, révélant des taux de couverture très satisfaisants.

La gestion des risques et des rendus est abordée, mettant l'accent sur la nécessité de vérifier les erreurs simples, de revoir chaque étape avant un push sur Git, et de planifier régulièrement des réunions de projet. En outre, des méthodes de validation complémentaires sont mentionnées, notamment la validation de la performance par des tests réguliers et la validation de la documentation en parallèle avec le développement de l'extension. Ces approches garantissent un processus de validation robuste et une qualité constante du code avant chaque soumission.

2 Méthode de validation

2.1 Descriptif des tests

Des tests ont été développés pour chaque étape du compilateur dans le répertoire **src/test/-deca/**. Pour les étapes A et B, les tests sont classifiés en des tests valides et des tests invalides de programmes Deca. Dans chacun des deux répertoires nous avons séparé les tests selon le sous-langage correspondant (hello-world, sans objet, essentiel, et complet).

Pour l'étape A, le répertoire *valid* contient les tests corrects lexicalement et syntaxiquement, alors que le répertoire *invalid* contient des tests incorrect syntaxiquement mais aussi incorrect lexicalement mais dont la syntaxe est correct.

Pour l'étape B, les tests sont répertoriés selon les passes de l'étape et traitent toutes les règles contextuelles du langage Deca. Les tests du répertoire *invalid* traitent toutes les erreurs contextuelles qu'on a pu identifié à partir des règles.

Pour l'étape C, les tests *interactives* contiennent tous les fichiers .deca contenant des programmes valides faisant appel aux fonctions `readInt` et `readFloat`, les tests de performance évaluent la performance énergétique de quelques programmes Deca présents dans le dossier *perf*, le dossier *invalid* répertorient des programmes ayant des erreurs à l'exécution, et les tests de programmes valides dans le dossier *valid* sont classés selon leur natures (déclarations, opérations, conditions, objets, registres...).

2.2 Scripts de tests

Nos scripts de tests pour chaque étape du compilateur se trouvent dans le répertoire **src/test/script/**.

2.2.1 Script de test de l'étape A

Nous avons créé deux scripts pour l'étape A, un script **tests_lexico.sh** pour les tests de la lexicographie et un autre script **tests_syntaxe.sh** pour la syntaxe. Le test de lexicographie vérifie le succès ou l'échec des tests du répertoire **/deca/syntax**. Lors de l'exécution du test, si l'échec ou le succès de la compilation d'un programme est correct le résultat s'affiche en vert, sinon en il s'affiche en rouge. Voici un exemple :

```

[ TESTS VALIDES POUR LE LEXEUR]
Succes attendu de test_lex sur src/test/deca/syntax/valid/helloWorld/hello.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/helloWorld/hello.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/sansObjet/noOperation.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/sansObjet/sansObjet.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/sansObjet/sansObjetComplexe.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/avecObjet/classAvecNull.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/avecObjet/classAvecThis.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/avecObjet/classe.deca
Succes attendu de test_lex sur src/test/deca/syntax/valid/avecObjet/instanceOfCast.deca
[ TESTS INVALIDES POUR LE LEXEUR]
Echec attendu pour test_lex sur chaine_incomplete.deca.
Echec attendu pour test_lex sur include_incorrect.deca.
Echec attendu pour test_lex sur crochets.deca.
Echec attendu pour test_lex sur deuxpoints.deca.
Echec attendu pour test_lex sur interrogation.deca.

```

FIGURE 1 – Test Lexeur

Le script de syntaxe vérifie l'échec des tests incorrects et compare les tests valides qui ne donnent pas d'erreur avec l'arbre abstrait attendu grâce à la fonction *diff*. Lors de l'exécution du test, si les échecs et les succès sont attendus le résultat s'affiche en vert, sinon en rouge. Voici un exemple :

```

-----Langage Hello World-----
Echec attendu pour test_synt sur src/test/deca/syntax/invalid/helloWorld/SansPointVirgule.deca.
Echec attendu pour test_synt sur src/test/deca/syntax/invalid/helloWorld/chaine_incomplete.deca.
Echec attendu pour test_synt sur src/test/deca/syntax/invalid/helloWorld/printIncomplete.deca.
Echec attendu pour test_synt sur src/test/deca/syntax/invalid/helloWorld/sansMain.deca.
Echec attendu pour test_synt sur src/test/deca/syntax/invalid/helloWorld/simple_lex.deca.
Succes attendu et arbres compatibles de test_synt sur src/test/deca/syntax/valid/helloWorld/hello.deca
-----Langage Hello World avec include-----
Succes attendu et arbres compatibles de test_synt sur src/test/deca/syntax/valid/include/include.deca

```

FIGURE 2 – Test Syntaxe

2.2.2 Script de test de l'étape B

Le script de test de l'étape B est **tests_context_invalides.sh**. Ce script teste les programmes invalides contextuellement en comparant les messages d'erreurs affichés avec les erreurs contextuelles attendus. Nous ne testons que les cas invalides puisque les cas valides seront testés grâce aux tests de l'étape C. Lors de l'exécution du test, si les échecs sont attendus le résultat s'affiche en vert, sinon en rouge. Voici un exemple :

```

=====
                        LANGUAGE SANS OBJET
=====
regle 0.1
Ligne 7: Undeclared identifier : 'a'.
===== REPONSE ATTENDUE =====
Ligne 7: Undeclared identifier : 'a'.
PASSED

regle 0.2
Ligne 7: Undeclared type identifier : 'A'.
===== REPONSE ATTENDUE =====
Ligne 7: Undeclared type identifier : 'A'.
PASSED

```

FIGURE 3 – Test Contexte

2.2.3 Script de test de l'étape C

Le script de test de l'étape C est **tests_codegen.sh**. Ce script dérive d'un script Python qui compare la sortie de l'exécution programme Deca avec la sortie attendue.

```

=====
TEST DE L'ÉTAPE C (VALIDE)
=====

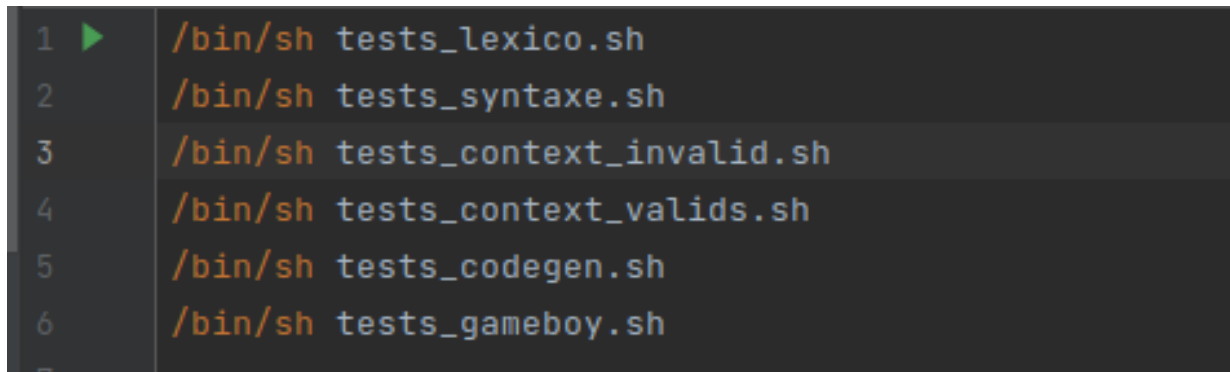
----- iostream/printString.deca ----- Passed
----- iostream/printIntFloat.deca ----- Passed
----- iostream/printFloatHexa.deca ----- Passed
----- iostream/includeSimple.deca ----- Passed
----- declarations/declVarSimple.deca ----- Passed
----- declarations/declVarMany.deca ----- Passed
----- operations/opArith.deca ----- Passed
----- operations/opArithConv.deca ----- Passed
----- operations/opShift.deca ----- Passed
----- conditions/boolLazyEval.deca ----- Passed
----- conditions/ifThenElseSimple.deca ----- Passed
----- conditions/ifThenElseComplex.deca ----- Passed
----- conditions/whileSimple.deca ----- Passed
----- conditions/whileComplex.deca ----- Passed
----- conditions/whileIfThenElse.deca ----- Passed
----- objects/fields/newSimple.deca ----- Passed
----- objects/fields/fieldSimple.deca ----- Passed

```

FIGURE 4 – Test de la Génération de Code

2.2.4 Script de test pour toutes les étapes

Pour exécuter la totalité des tests il suffit d'exécuter le script **src/test/script/test_all.sh**. Ce script exécute tous les scripts précédents. Nous avons également inclus tous nos scripts de tests dans le fichier **pom.xml**, donc nous avons également pu tester tous nos scripts avec la commande *mvn test*.



```
1 ► /bin/sh tests_lexico.sh
2   /bin/sh tests_syntaxe.sh
3   /bin/sh tests_context_invalid.sh
4   /bin/sh tests_context_valids.sh
5   /bin/sh tests_codegen.sh
6   /bin/sh tests_gameboy.sh
```

FIGURE 5 – Script qui exécute tous nos tests

3 Couverture des tests

La vérification de la couverture de nos tests s'est faite avec l'outil Jacoco.

Jacoco est un outil qui permet de calculer la couverture d'un jeu de tests sur un programme. On peut savoir quelle instruction a été exécutée (ou non), quelles branches d'une instruction conditionnelle (if, while) ont été prises, estimer la complexité du code etc...

Dans notre cas le resultat de Jacoco affirme que nous avons une couverture de **96%** pour les classes de **tree** et de **86%** en totalité.

Voici le résultat de Jacoco si on clique sur la partie **tree** :

fr.ensimag.deca.tree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Identifier		83%		94%	6	49	19	131	3	24	0	1
Tree		78%		81%	6	30	13	64	4	22	0	1
AbstractExpr		81%		87%	8	29	9	66	5	17	0	1
Assign		89%		82%	6	22	12	91	2	8	0	1
DeclMethodAsm		78%		n/a	1	13	5	35	1	13	0	1
Cast		92%		96%	3	20	9	69	2	7	0	1
DeclMethod		93%		100%	1	16	7	96	1	14	0	1
ListExpr		51%		33%	3	6	6	11	1	3	0	1
InstanceOf		94%		95%	3	17	7	65	2	7	0	1
AbstractUnaryExpr		84%		100%	1	10	6	36	1	9	0	1
Print		90%		50%	5	9	3	35	0	3	0	1
DeclClass		98%		92%	4	21	4	145	3	14	0	1
AbstractOpCmp		96%		87%	5	30	3	68	0	6	0	1
Module		96%		83%	4	13	4	53	2	7	0	1
Multiply		96%		70%	2	11	2	55	0	6	0	1
LocationException		86%		50%	3	7	2	14	0	4	0	1
AbstractBinaryExpr		98%		98%	2	39	3	142	1	11	0	1
BooleanLiteral		94%		82%	6	27	1	27	1	10	0	1
DeclField		99%		97%	1	33	2	127	0	15	0	1
DeclVar		97%		100%	0	11	2	64	0	7	0	1
AbstractDeclMethod		97%		92%	1	11	2	66	0	4	0	1
DeclParam		93%		100%	0	8	2	27	0	7	0	1
MethodCall		99%		93%	3	32	1	149	0	8	0	1
FieldSelection		98%		83%	3	18	1	83	0	9	0	1
Equals		98%		91%	2	14	1	44	1	8	0	1
NotEquals		98%		91%	2	14	1	42	1	8	0	1
AbstractOpArith		99%		95%	3	41	2	116	0	5	0	1
AbstractOpExactCmp		99%		95%	5	64	0	72	0	3	0	1
IntLiteral		96%		100%	1	18	2	29	1	14	0	1
StringLiteral		93%		n/a	1	10	1	18	1	10	0	1
Location		92%		50%	2	8	2	14	0	6	0	1
EmptyMain		60%		n/a	4	8	4	8	4	8	0	1
FloatLiteral		96%		50%	3	13	0	21	0	10	0	1
ConvFloat		94%		100%	2	7	2	16	2	6	0	1
NoOperation		75%		n/a	3	7	3	8	3	7	0	1
AbstractOpBool		98%		100%	2	11	2	48	2	5	0	1
Not		95%		100%	2	8	2	13	2	6	0	1

FIGURE 6 – Résultats de Jacoco pour tree

Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax	<div><div></div></div>	73%	<div><div></div></div>	54%	484	683	557	2,059	264	380	6	50
fr.ensimag.deca.tree	<div><div></div></div>	96%	<div><div></div></div>	92%	117	1,043	152	3,232	51	576	0	88
fr.ensimag.deca	<div><div></div></div>	60%	<div><div></div></div>	65%	38	99	103	293	10	46	2	5
fr.ensimag.deca.context	<div><div></div></div>	87%	<div><div></div></div>	77%	55	196	52	355	36	146	1	26
fr.ensimag.ima.pseudocode	<div><div></div></div>	79%	<div><div></div></div>	76%	37	127	57	266	26	101	2	28
fr.ensimag.ima.pseudocode.instructions	<div><div></div></div>	93%	<div><div></div></div>	66%	10	116	12	206	8	113	8	66
fr.ensimag.deca.codegen	<div><div></div></div>	99%	<div><div></div></div>	90%	12	202	5	461	0	138	0	12
fr.ensimag.deca.tools	<div><div></div></div>	93%	<div><div></div></div>	87%	2	19	3	43	1	15	0	3
Total	4,337 of 31,339	86%	404 of 1,870	78%	755	2,485	941	6,915	396	1,515	19	278

FIGURE 7 – Résultats Jacoco Complets

On remarque que la partie la moins couverte est celle de *identifier* car nous avons ajouté des méthodes que nous n'avons pas utilisé mais nous nous sommes rendu compte de cela qu'après le rendu final.

Cependant la couverture affichée quand nous lançons Jacoco n'est pas juste celle pour IMA mais avec tous ce que nous avons fait pour l'extension (ACONIT, sur la GameBoy) . Comme le *Multiply* où *codeGenOpArith* a une couverture de 100% mais *codeGenOpArithGb* (pour la Game-Boy) a une couverture de 96% :

Multiply

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
codeGenOpArithGb(DecacCompiler, DVal, GPRRegister)		96%		70%	2	6	2	48	0	1
codeGenOpArith(DecacCompiler, DVal, GPRRegister)		100%	n/a	n/a	0	1	0	2	0	1
Multiply(AbstractExpr, AbstractExpr)		100%	n/a	n/a	0	1	0	2	0	1
doOpInt(int, int)		100%	n/a	n/a	0	1	0	1	0	1
doOpFloat(float, float)		100%	n/a	n/a	0	1	0	1	0	1
getOperatorName()		100%	n/a	n/a	0	1	0	1	0	1
Total	9 of 296	96%	3 of 10	70%	2	11	2	55	0	6

FIGURE 8 – Jacoco-Multiply

Nous pouvons donc en conclure qu'en considérant seulement le code utilisé pour IMA, nous aurions pu avoir une encore meilleure couverture.

4 Gestion des risques et gestion des rendus

Nous avons décidé de suivre certaines étapes à effectuer avant chaque push afin d'éviter de faire un push d'un code qui ne fonctionne plus :

- Vérification des erreurs simples : Avant tout, nous recommandons l'exécution du fichier **common-tests.sh** pour détecter et corriger d'éventuelles erreurs simples qui pourraient compromettre la fonctionnalité globale du code.
- Étape A - Tests Lexicaux et Syntaxiques : Une attention particulière doit être accordée à cette étape. Nous suggérons l'utilisation des fichiers **tests_lexico.sh** et **tests_syntaxe.sh** pour garantir la conformité aux exigences lexicales et syntaxiques du projet.
- Étape B - Tests de Contexte Invalide : La validation de l'étape B nécessite l'exécution du fichier **tests_context_invalid.sh**. Cette étape critique vise à identifier toute défaillance potentielle dans la gestion des contextes invalides.
- Étape C - Tests de Génération de Code : Assurez-vous de vérifier l'intégrité de la génération de code en exécutant le fichier **tests_codegen.sh**.

Pour les erreurs difficiles à automatiser (utilisation des registres, lectures...) : Vérifier de temps en temps que tout marche bien en vérifiant dans les fichiers le code généré.

- Planificateur Temporel : Consacrer du temps régulièrement à l'examen du planificateur pour s'assurer que les délais sont respectés. La création de rappels dans l'agenda est vivement conseillée pour faciliter le suivi des échéances.
- Réunions de Projet : Organiser des réunions périodiques pour discuter de l'état d'avancement global du projet. Ces rencontres offrent l'opportunité de partager des idées, résoudre des problèmes potentiels et assurer une communication transparente entre les membres de l'équipe.
- Tests de Dépôt Git : Avant la remise finale, il est impératif de tester le dépôt Git plutôt que le dépôt local pour garantir l'intégrité du code soumis.
- Diversité dans la Création de Tests : Il est fortement recommandé de ne pas attribuer la tâche de création de tests à une seule personne. Cette approche favorise la diversité des perspectives et permet de traiter tous les cas particuliers auxquels une personne seule pourrait ne pas penser.

En suivant ces directives approfondies, nous nous assurons d'un processus de validation robuste et d'une qualité constante du code avant chaque push.

5 Méthodes de validation utilisées autres que les tests

5.1 Validation de performance

Puisque nous devons respecter certaines normes de performance, pour respecter le côté développement durable du projet, nous avons passé de temps en temps des tests de performance et nous avons comparé nos résultats avec ceux des années précédentes depuis ce [site](#). Ce qui nous a poussé à faire des optimisations lors de la conception, comme faire un décalage à droite à la place de multiplier par 2 dans la partie assembleur. Les détails de ces optimisations se trouvent dans la documentation énergétique du projet.

5.2 Validation de la documentation

Nous avons décidé de travailler sur les documentations en parallèle avec l'extension durant la dernière semaine du projet. Elles étaient écrites sur *Overleaf* qui effectue la vérification orthographique du contenu ainsi que la cohérence de la structure du document. Avant de soumettre nos documents sur Git, tous les membres du groupe les relisent et donnent leur accord pour la soumission finale.