

# Documentation de validation

## I/Descriptif des tests

### I.1/Types de tests pour chaque étape/passe (tests unitaires, tests système, ...)

Pour chaque étape, nous avons écrit des fichiers `*.deca` de tests valides et invalides. Les résultats valides sont conservés dans un fichier `.txt` dans un répertoire 'expected result' s'il nous convient.

Ces tests sont des tests système : ils permettent de comparer une sortie obtenue par rapport à une entrée donnée. Les données traversent le système et grâce au message de sortie on peut savoir ce qu'il s'est passé. Pour avoir plus d'information, on peut utiliser des outils de débogage mais notre objectif est d'avoir des messages d'erreur suffisamment limpides pour éviter cela.

Pour l'étape A, on vérifie que tous les tokens sont reconnus (ou que les charabias sont bien rejetés) et que quelques exemples sont correctement analysés. À ce niveau-ci, il serait contreproductif de faire des tests plus poussés car seule la validité des mots pris individuellement est surveillée.

Les fichiers invalides sont les plus exhaustifs possibles pour l'étape B (afin de vérifier que toutes les erreurs contextuelles sont bien détectées). Les arbres décorés ou non sont vérifiés manuellement : il est vrai que cette vérification peut-être fastidieuse, dans les faits, seuls quelques fichiers bien choisis sont entièrement vérifiés. Le but de garder tous les résultats des tests en mémoire est de détecter des changements inattendus (ou voulus) et de valider ces modifications ou bien lancer une alerte à la partie mise en cause.

L'implémentation de la fonction `decompile()` et de la commande `decac -p` permet une vérification supplémentaire de l'interprétation de l'entrée.

Les tests de l'étape C offrent l'avantage de pouvoir se vérifier eux même (notamment une fois que les fonctionnalités `if` et `print` sont implémentées).

Nous avons donc créé des tests qui permettent de visualiser le contenu des variables et donc de valider le comportement du compilateur.

Nous avons aussi ajouté quelques tests interactifs qui ne sont pas lancés lors de la vérification générale mais permettent de tester les fonctionnalités de lecture d'entrée (vous pouvez notamment vous essayer au jeu du juste prix).

## I.2/Organisation des tests

Les fichiers de tests sont stockés dans le répertoire `src/test/deca` et organisés selon leur étape (lexique, syntaxe, contexte, génération de code). Chaque dossier contient deux types de base de tests : les valides (censé réussir) et les invalides (censé renvoyer une erreur). Ces deux répertoires contiennent des tests réalisés par nos soins (`created`) ou que l'on nous a fournis (`provided`). Au sein même de cette classification, nous avons parfois jugé bon de rajouter des répertoire pour classer les tests en fonction de leur thème.

L'entête des tests respecte toujours le même format (sauf dans le cas du test du `fichier_vide.deca` de la partie syntaxe, où il n'y a rien d'écrit dans le fichier).

### Fichiers valides :

```
1 // Description :  
2 //     <Description>  
3 //  
4 // Resultats :  
5 //     OK
```

### Fichiers invalides :

```
1 // Description :  
2 //     <Description>  
3 //  
4 // Resultats :  
5 //     <type erreur>  
6 //     Ligne <numero> : <message d'erreur attendu>
```

## I.3/ Objectifs des tests, comment ces objectifs ont été atteints.

Nous avons plusieurs objectifs :

- vérifier que les bonnes erreurs sont détectées aux bons endroits
- vérifier que la syntaxe est respectée
- couvrir un maximum notre programme
- vérifier que les fonctionnalités donnent le résultat attendu
- détecter des erreurs de conception et donc aider l'implémentation

Ses objectifs ont été réalisés par la création de très nombreux fichiers de tests que nous avons voulu les plus exhaustifs possible. Cependant, avoir une infinité de tests très efficaces est inutile si nous n'avons pas d'outils efficaces pour les analyser. C'est ici que sont intervenus les scripts d'automatisation des tests que nous détaillerons dans la partie suivante.

La validation est toujours double : les tests invalides doivent être invalidés pour les bonnes raisons, les tests valides doivent avoir une sortie conforme à nos attentes. Nous avons réussi à automatiser une bonne partie des vérifications pour les fichiers invalides (notamment contextuelles), mais des analyses manuelles les ont complétées.

Les erreurs détectées par les scripts sont isolées et analysées afin de trouver la source du problème. Puis, le fichier responsable de l'erreur ainsi que le détail de la sortie sont envoyés à la personne la plus à même de corriger le bug.

Pour la partie C, nous avons fonctionné un peu différemment. En effet, les fichiers valides étant souvent denses, nous avons procédé par fichiers auxiliaires : dans `src/test/codegen/valid`, diverses `test_problem.deca` ont été ajoutés afin de recréer la configuration minimale pour produire le bug.

L'outil trello nous a permis de communiquer efficacement : les bugs et leur détails étaient classés en trois listes : "Bug Tests qui devraient passer", "Bug Tests qui ne devraient pas passer" et "Bug Tests qui ont une mauvaise sortie".

Enfin, une quatrième liste "Bugs divers" a servi pour les bugs liés aux scripts, ou ceux relevés dès la conception par exemple.

En procédant ainsi, les bugs ont pu être corrigés efficacement (en moyenne dans les 20 minutes qui suivent).

Pour vérifier la couverture, nous avons utilisé l'outil Jacoco (cf section IV/) ainsi qu'une liste des messages d'erreurs contextuelles (cf documentation utilisateur).

## II/ Les scripts de tests

### Explication rapide sur le fonctionnement:

Le script `master-all-test.sh` appelle le script `script_test_general.sh` qui lui-même appelle les lanceurs des tests.

Tous les scripts se trouvent dans le répertoire `src/test/script`.

## II.1/ Comment faire passer tous les tests

Il y a plusieurs manières de faire passer les tests:

On indique ici les instructions depuis le répertoire racine du projet.

### II.1.1 faire passer tous les tests de manières automatique (avec différents message de retour)

Il est recommandé de procéder de cette manière

```
./src/test/script/master-all-test.sh
```

### II.1.2 Faire un test individuel avec les lanceurs fournis et créé

```
./src/test/script/launchers/test_lex <fichier.deca>
```

```
./src/test/script/launchers/test_synt <fichier.deca>
```

```
./src/test/script/launchers/test_context  
<fichier.deca>
```

### II.1.3 Faire passer une étape à tout un répertoire

Syntaxe:

```
./src/test/script/script_test_general.sh <lanceur>  
<répertoire>
```

Exemple:

```
./src/test/script/script_test_general.sh test_lex  
./src/test/deca/lexer
```

Le script `test_decompile.sh` vérifie l'idempotence de la décompilation sur les fichiers de tests valides de la partie syntaxe. Il est aussi lancé automatiquement par `master_all_test`.

Il n'y a pas de script pour tester les options de `decac` (à part l'option `-r` dans `master_all_test`) ou pour lancer les tests spécifiques à l'extension (dans `src/test/codegen/extension`).

## II.2/Quels informations pouvez-vous obtenir

On se base sur l'utilisation du script `master-all-test.sh`

### II.2.1/Cas d'un test réussi

```
[cathelib@ensipc276 gl01]$ ./src/test/script/master-all-test.sh
./src/test/script/script_test_general.sh test_lex ./src/test/deca/lexer
Debut des test sur test_lex
Test du répertoire : ./src/test/deca/lexer/invalid/created/
Echec attendu pour test_lex sur ./src/test/deca/lexer/invalid/created/invalid_characters.deca.
Test du répertoire : ./src/test/deca/lexer/invalid/provided/
Echec attendu pour test_lex sur ./src/test/deca/lexer/invalid/provided/chaine_incomplete.deca.
Test du répertoire : ./src/test/deca/lexer/valid/created/
Succes attendu de test_lex sur ./src/test/deca/lexer/valid/created/class_vide.deca.
Succes attendu de test_lex sur ./src/test/deca/lexer/valid/created/comment_1.deca.
```

D'abord le script affiche la ligne de commande utilisée pour appeler `script_test_general.sh` avec le lanceur spécifique au répertoire.

Ensuite, en bleu, s'affiche le répertoire testé.

Puis les tests sont exécutés un par un. Le résultat est vert, le test a réussi

### II.2.2/Cas d'un test qui ne crash pas mais a une sortie incorrecte

```
Le test ./src/test/deca/lexer/valid/created/if.deca s'est déroulé sans erreur mais résultat n'est pas celui attendu
commande utilisée ::: test_lex ./src/test/deca/lexer/valid/created/if.deca
```

```
./src/test/deca/lexer/valid/created/if.deca sortie actuelle:::
'{' : [0,64:64='{' ,<34>,8:0]
'if' : [01,66:67='if' ,<6>,9:0]
'(' : [02,68:69='(' ,<32>,9:3]
```

```
sortie attendue:::
'{' : [0,64:64='{' ,<34>,8:0]
'if' : [01,66:67='if' ,<6>,9:0]
'(' : [02,68:69='(' ,<32>,9:3]
```

Ce test est considéré comme raté

### II.2.3/Cas d'un test qui échoue alors qu'il devrait réussir

```
Echec inattendu pour test_synt sur ./src/test/deca/syntax/valid/provided/programme_exemple.deca.
commande utilisée ::: test_synt ./src/test/deca/syntax/valid/provided/programme_exemple.deca
des tests ont échoués... Voici la liste
```

Ce test est considéré comme raté.

### II.2.4/Cas d'un test réussi alors qu'il devrait échouer

```
Echec attendu pour test_synt sur ./src/test/deca/syntax/invalid/created//operande_manquante/it_a.deca.
Succes inattendu de test_synt sur ./src/test/deca/syntax/invalid/created//operande_manquante/minus_a.deca.
commande utilisée ::: test_synt ./src/test/deca/syntax/invalid/created//operande_manquante/minus_a.deca
```

Ce test est considéré comme raté.

### II.2.5/Cas d'un test qui doit échouer mais qui crash

```
Le test qui devait échouer test_synt sur ./src/test/deca/syntax/invalid/provided/programme_exemple.deca a crashé et non échoué proprement  
commande utilisée ::: test_synt ./src/test/deca/syntax/invalid/provided/programme_exemple.deca  
Échec attendu pour test_synt sur ./src/test/deca/syntax/invalid/provided/simple_lex.deca.
```

Ce test est considéré comme raté.

### II.2.6/Cas d'un test qui n'a pas de numéro d'erreur ou pas d'oracle

```
Test du répertoire : ./src/test/deca/context/invalid/provided/  
Pour le test ./src/test/deca/context/invalid/provided/affect-incompatible.deca,  
Il n'y a pas de numéro d'erreur dans le fichier .deca on considère le message  
d'erreur comme valide, mais veuillez vérifier  
Test du répertoire : ./src/test/deca/context/valid/created/  
./valid/created/double_checked_instruction.deca.  
Il n'y a pas de fichier de comparaison (pas d'oracle de test) pour le test src/  
test/deca/syntax/valid/created/fichier_vide2.deca. Ce test est considéré comme  
valide mais veuillez créer un oracle de test
```

Ces tests sont considérés comme réussis.

## II.3/ Comment ajouter des tests ?

Vous pouvez ajouter directement des fichiers \*.deca dans les répertoires des étapes correspondantes. Les scripts prendront alors vos fichiers en compte. Vous pouvez aussi ajouter des répertoire à condition de le faire dans à l'intérieur des dossier valid/ ou invalid/.

### Test invalide

Un test invalide échoue avec succès uniquement si le code de retour est différent de 0.

De plus, Les tests de context doivent avoir leur numéro d'erreur indiqué dans leur entête ligne 6, toujours sous le même format:

```
1 // Description :  
2 //      <Description>  
3 //  
4 // Resultats :  
5 //      Erreur contextuelle  
6 //      Ligne <numero> : (<numero erreur>) <message d'erreur  
attendu>
```

Le message d'erreur est optionnel, seul le numéro est évalué par le script et comparé à la sortie.

### Test valide

Pour ajouter l'oracle correspondant, vous pouvez créer le fichier à la main ou utiliser

```
creation_automatique_des_resultats_pour_non_reg_test.sh  
<lanceur> <répertoire>
```

(creation\_automatique\_des\_resultats\_pour\_non\_reg\_test.sh se trouve dans le repertoire src/test/script)

Par exemple

```
creation_automatique_des_resultats_pour_non_reg_test.sh  
test_context src/test/deca/context/valid/created/
```

Lancer tel quel, vous devrez valider à la main chaque résultat en appuyant sur y/n. Pour que les résultats soient validés automatiquement, vous pouvez ajouter l'option `no_valid` à la fin de votre commande.

```
creation_automatique_des_resultats_pour_non_reg_test.sh  
test_context src/test/deca/context/valid/created/ no_valid
```

### Script

Pour lancer plus de script dans `master_all_tests`, il faut les ajouter dans le tableau de lancement :

```
tests+=(("./src/test/script/launchers/decac_r.sh ./src/test/deca/codegen/valid/created/while.deca")
```

ici par exemple on ajoute le test de l'option `-r` (avec comme argument le chemin complet du fichier)

Nous avons aussi utilisé les scripts pour générer les fichiers de tests : en effet, certains d'entre eux sont très similaires (entête et code commun)

## III/ Gestion des risques et gestion des rendus

### Gestion des risques

RISQUE	Niveau de "danger"	probabilité de se	Importance totale	PRÉVENTION
--------	--------------------	-------------------	-------------------	------------

		produire		
Oubli des dates	Très élevé (5)	très faible (1)	5	→ calendrier → rappels systématiques des échéances lors des réunions quotidiennes
Oubli d'un fichier exigé pour un rendu	Très élevé (5)	faible (3)	15	→ mise à jour personnelle sur les documents requis impérativement par relecture régulière du poly → checklist sur Trello → rappels et concertations en réunion
Travail redondant (deux personnes font la même chose)	Faible (2)	moyen (5)	10	→ communication limpide → utilisation de Trello
Mésentente dans l'équipe	élevé (4)	nulle (0)	0	→ Dialogue, médiation, séparation des parties en conflit
Accident matériel: un membre de l'équipe ne peut plus travailler pour des raisons matérielles et/ou perd son travail en cours	élevé (4)	moyen (5)	20	→ gut push fréquents → réunion en urgence de tous les membres pour trouver une alternative de travail au membre concerné ou, dans l'impossibilité, de répartir sa charge de travail parmi les autres membres provisoirement
Mauvais messages d'erreur	faible (2)	moyen (5)	10	→ Dissociation du développement des parties A et B et de la partie test → Retour des testeurs vers les développeurs si le mauvais formatage est passé sous leur vigilance malgré des tests rudimentaires
Mauvaise gestion des paramètres de la ligne de commande	très élevée (5)	faible (3)	15	→ Vérification par plusieurs membres et relecture de la documentation
Les cas élémentaires sur lesquels le compilateur ne fonctionne pas	très élevée (5)	faible (3)	15	→ comparaison avec les exemples (cf exemple sans objet) → tests



comme attendu				
Script de test erronés	élevé (4)	moyen (5)	20	→ vérification manuelle régulière

### Gestion des mises en production

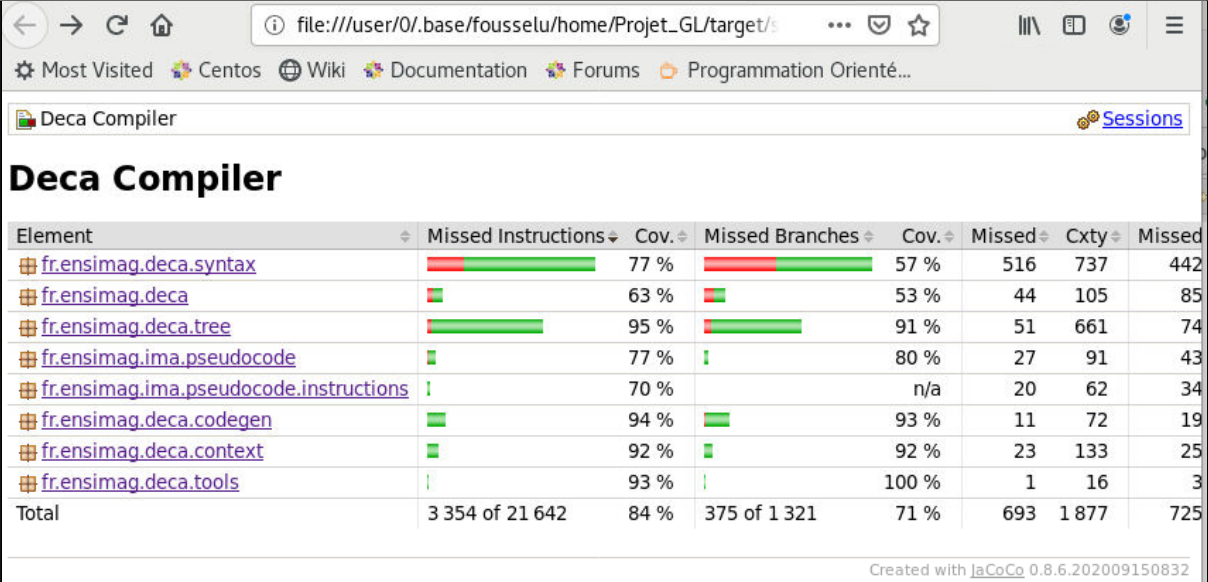
PROBLÈME	Niveau de "danger"	Probabilité d'occurrence	importance totale	PRÉVENTION
Dépôt git local pas à jour avec le dépôt commun ⇒ l'utilisateur ne récupère pas la bonne version	élevé (4)	faible (3)	12	→ git status récurrent → utilisation pour le rendu d'un dépôt vierge dans lequel on clone
Fonctionnalités non implémentées	élevé (4)	moyen (5)	20	→ la méthode agile permet de ne pas être pris de court et de concentrer les efforts sur les points critiques et urgents → Rapport présentant les limitations
Compilateur non fonctionnel	très élevé (5)	faible (3)	15	→ test d'utilisation par tous les membres
Oublie d'un // A FAIRE ou // TODO dans un rendu  Fichier .nfsxxxxxxx	très faible (1)	élevé (7)	7	→ Utilisation de grep (shell script) pour chercher les TODO dans tous les fichiers → Pour les fichier .nfsxxxxxxx, utilisation de la commande find puis rm
panne de réseau avant un rendu	élevé (4)	faible (3)	12	se renseigner sur les zones wifi gratuit les plus proches
Correction de dernière minute avant un rendu	élevé (4)	élevé (7)	28	-> avoir une version "qui marche presque" étiquetée ->se mettre d'accord sur le moment "on ne touche plus à rien" ->avoir une procédure de vérification efficace et rapide (mvn verify = 4min50)

## IV/ Résultats de JACOCO
















Nous avons introduit l'outil JACOCO au milieu de la deuxième semaine, en faisant `mvn verify`, on lance JACOCO sur `master_all_tests` (donc tous les tests non-interactifs de notre base). Les résultats sont disponibles par la commande `firefox target/site/jacoco/index.html`

Nous avons donc pu suivre et augmenter graduellement notre couverture . En effet, dans les fichiers que nous avons conçus, chaque branche non couverte était analysée pour nous assurer de la validité du compilateur. C'est par cette méthode que nous avons pu corriger de nombreux bugs silencieux.

A ce jour, la couverture est de : 84% sur les instructions et 71% sur les branches.



The screenshot shows a web browser displaying the JUnit report for the Deca Compiler. The report is titled 'Deca Compiler' and shows a table of coverage data for various components. The table has columns for 'Element', 'Missed Instructions', 'Cov.', 'Missed Branches', 'Cov.', 'Missed', 'Cxt', and 'Missed'. The data is as follows:

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxt	Missed
fr.ensimag.deca.syntax		77 %		57 %	516	737	442
fr.ensimag.deca		63 %		53 %	44	105	85
fr.ensimag.deca.tree		95 %		91 %	51	661	74
fr.ensimag.ima.pseudocode		77 %		80 %	27	91	43
fr.ensimag.ima.pseudocode.instructions		70 %	n/a	n/a	20	62	34
fr.ensimag.deca.codegen		94 %		93 %	11	72	19
fr.ensimag.deca.context		92 %		92 %	23	133	25
fr.ensimag.deca.tools		93 %		100 %	1	16	3
Total	3 354 of 21 642	84 %	375 of 1 321	71 %	693	1 877	725

Created with JaCoCo 0.8.6.202009150832

## V/ Méthodes de validation utilisées autres que le test

Nous n'avons pas utilisé d'autres méthodes de validation.