# Document de validation

---

OUAZZANI CHAHDI Mohammed
BENTOUIMOU Yassine
ABDELWAHID Amine
AIT DRISS Salma
LEMDAGHRI ALAOUI Ibrahim

24 janvier 2023

# Table des matières

# 1 Introduction

Les tests ont une grande importance dans un projet aussi grand que le projet GL, pour cela on a commencé à rédiger les tests depuis le début et tout au long du projet afin de détecter les possibles bugs dans nos codes. On aimera aussi souligner que l'implémention de la base de test a été faite d'une manière progressive vu que chaque étape du projet(Etape A, Etape B, Etape C) a relevé quelques erreurs et a remonté aussi d'autres idées qu'on ajouté à notre base de tests.
Dans ce manuel, nous allons vous décrire l'ensemble des tests qu'on a pu écrire pendant ce projet et leurs objectifs, les scripts qui permettent de faire passer l'ensemble de ces tests, notre gestion des risques et du rendu, ainsi que les résultats finaux du Jacoco.

## 2 Convention d'écriture des tests

En ce qui concerne les tests de la partie C du projet (codeGen), on a décidé de suivre une convention simple, tout en respectant les exigences demandées, pour écrire un script simple qui permettra de récupérer le résultat attendu et le comparer par le résultat généré par le compilateur. Cette convention consiste à afficher un seul mot par ligne et le résultat du test doit être écrit entre les deux sections Résultat et Historique tout en laissant une ligne vide(comme le montre l'exemple ci-dessous).

```
1 // Description:
2 //    différents cas de print valides
3 //
4 // Resultats:
5 //    ok
6 //    1.00000e+00
7 //    HiGL24
8 //
9 // Historique:
10 //    cree le 08/01/2023
11
12 {
13     println("ok");
14     println(1.0);
15     print("HiGL24");
16 }
```

## 3 Tests de l'étape A

### 3.1 Tests deca

Pour les tests cette partie, on les a organisés selon des tests valides et invalides, et on a essayé de voir la grammaire du langage Deca afin de produire un maximum de tests qui vont soit concevoir des arbres incorrectes (qui ne respectent pas la grammaire), soit des arbres correctes dans plusieurs cas de figures qu'on a pu penser au cours du projet.

### 3.2 Script de la partie A

Pour cette partie, on a conçu le script *self_synt.sh* en se basant sur les scripts fournis, et qui exécutent en premier les tests invalides, et puis les tests valides conçues syntaxiquement.
Ce script exécute l'exécutable *test_synt* sur ses fichiers deca et puis fait un tuyau (pipe ) vers la commande *grep* qui détecte si une exception est lancée par le compilateur (ces exceptions suivent tous la règle suivante : *nomfichier :[ligne] :[colonne]*). Donc selon le cas de test valide ou invalide on déduit la validité du test.

## 4 Tests de l'étape B

### 4.1 Tests deca

Pour cette partie, on a gardé la même organisation des tests, et on a essayé de concevoir des tests qui vont passer par les différentes méthodes *verify* du paquetage *tree* pour vérifier le bon fonctionnement des méthodes implémentées qui devront respecter la **Syntaxe Contextuelle**. Donc en premier lieu, on a essayé d'implémenter les cas simples puis les cas plus ou moins complexe qui devront être accepter par le compilateur, ensuite on a regroupé les cas critique qui devront être nécessairement refusés et on ajoute de plus en plus les cas moins rare qu'on a déduit après plusieurs relectures de la grammaire. On a aussi conçu quelques tests unitaires pour cette partie mais on a gardé qu'un seul qui manipule les Environnements de notre compilateur.

### 4.2 Script de la partie B

On a conçu le script *self_context* en se basant sur les scripts fournis, et qui exécute en premier nos tests invalides et valides, puis les tests fournis valides et invalides du côté contextuelle.
Ce script exécute l'exécutable *test_context* fournis sur tous les tests deca dans le dossier **context**, et puis passe le résultat à la commande *grep* qui suivra le même principe du script ci-dessus.

## 5 Tests de l'étape C

### 5.1 Tests deca

Dans cette partie, l'organisation des tests est un peu différente, on a rajouté un nouveau dossier **interactive** qui contient des tests qui utilisent les différentes fonctions *readInt* et *readFloat*, ainsi que le dossier **perf** qui contient des tests relatives à la performance de notre compilateur pour des calculs profonds.
On a essayé d'écrire des tests qui vont parcourir les différents méthodes *codeGen* implémentés dans les différents classes du package *tree*, et bien vérifier son bon

fonctionnement dans les différents cas complexes qu'on a pu penser valides et invalides.

### 5.2 Script de la partie C

On a conçu le script **tes.sh** qui va commencer par parcourir les tests valides du dossier **codegen** puis les tests invalides de ce dossier. Et dans chaque cas, on commence par extraire le résultat attendu du fichier sélectionné en suivant les Convention d'écriture des tests et qui va être sauvegardé dans un fichier temporaire **res6.txt**, on va ensuite compiler le fichier sélectionné par la commande **decac** et puis on exécutera le fichier assembleur par la commande **ima** et qui va sauvegarder le résultat de son exécution dans un fichier temporaire **.res**. Enfin, on comparera les deux fichiers et on jugera un test réussit si la comparaison est exacte, sinon le test a échoué.

## 6 Tests de Décompilation

### 6.1 Tests deca

Pour cette partie, on a essayé d'écrire un **long test** qui regroupe la majorité des instructions,expressions et opérations du langage **deca** . Donc après chaque nouvelle expression conçue, la première chose à faire et de l'ajouter dans ce fichier avec les différentes formes avec laquelle elle peut exister.

### 6.2 Script de Décompilation

Pour la décompilation, on a conçu le script **test_decompile.sh** qui exécutera la compilation avec l'option **-p** sur notre fichier test cité ci-dessus et puis sauvegardera le résultat dans un fichier temporaire **res.deca**. Ensuite, on procède par recompiler ce dernier par la même méthode et on récupère son résultat dans un autre fichier temporaire **res2.deca**. Un code est syntaxiquement correct si les deux fichiers sont identiques, donc on juge à la fois une décompilation correcte et le syntaxe si les deux fichiers **res.deca** et **res2.deca** sont identiques, sinon la décompilation a échoué.

## 7 Scripts des tests

Pour cette partie des tests, on a réalisé plusieurs scripts **shell** qui permettent d'exécuter la totalité de nos tests écrits. Afin d'intégrer ces scripts à l'exécution automatique des tests via la commande **mvn verify**, on a rajouté au fichier **pom.xml** les lignes de code suivante :

$< execution >$
$< id > self\_synt < /id >$
$< configuration >$
$< executable > ./src/test/script/self\_synt.sh < /executable >$
$< /configuration >$
$< phase > test < /phase >$
$< goals >$
$< goal > exec < /goal >$
$< /goals >$
$<\backslash execution >$
$< execution >$
$< id > self\_contex < /id >$
$< configuration >$
$< executable > ./src/test/script/self\_contex.sh < /executable >$
$< /configuration >$
$< phase > test < /phase >$
$< goals >$
$< goal > exec < /goal >$
$< /goals >$
$<\backslash execution >$
$< execution >$
$< id > tes < /id >$
$< configuration >$
$< executable > ./src/test/script/tes.sh < /executable >$
$< /configuration >$
$< phase > test < /phase >$
$< goals >$
$< goal > exec < /goal >$
$< /goals >$
$<\backslash execution >$
$< execution >$
$< id > test\_decompile < /id >$
$< configuration >$
$< executable > ./src/test/script/test_decompile.sh < /executable >$
$< /configuration >$
$< phase > test < /phase >$
$< goals >$
$< goal > exec < /goal >$
$< /goals >$
$<\backslash execution >$

Ainsi, l'ensemble de nos tests pourrait être automatiquement lancés lors de l'execution de la commande **mvn verify**.

### 7.1 self_synt

Ce script a été conçue pour exécuter la vérification syntaxique à travers le launcher **test_synt** sur tous les tests deca du dossier **syntax**.
Il peut être exécuté par la commande : ./src/test/script/self_synt.sh

### 7.2 self_context

Ce script a été conçue pour exécuter la vérification contextuelle à travers le launcher **test_context** sur tous les tests deca du dossier **context**.
Il peut être exécuté par la commande : ./src/test/script/self_context.sh

### 7.3 tes

Ce script a été conçue pour exécuter la vérification du codeGen sur tous les tests deca du dossier **codegen**, ainsi que la bonne exécution dans la machine **ima**.
Il peut être exécuté par la commande : ./src/test/script/tes.sh

### 7.4 test_décompile

Ce script a été conçue pour exécuter la vérification du décompile sur un deca **Test.deca** dossier **syntax**.
Il peut être exécuté par la commande : ./src/test/script/test_decompile.sh

## 8 Test des options du compilateur

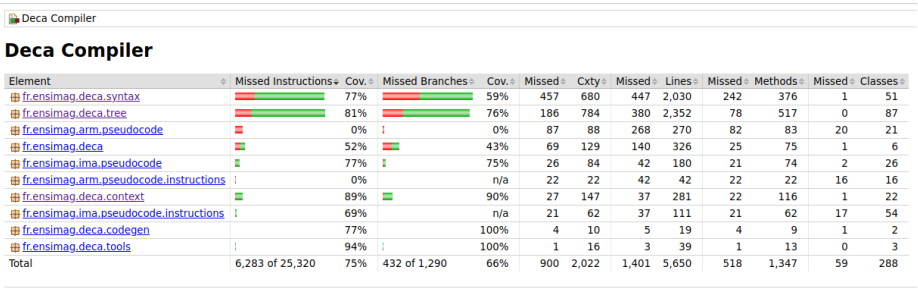
Notre compilateur est conçue pour compiler les fichiers deca avec plusieurs options de compilations qui permettent spécifier plus de contraintes au compilateur.
Pour utiliser une option de compilation précise, il suffit d'exécuter la commande suivante : **./src/main/bin/decac** [[**-p** | **-v**] [**-n**] [**-r X**] [**-d**]* [**-P**] [**-w**] < *fichierdeca* >**...**] | [**-b**]
L'option de compilation **-d** n'a pas été implémentée dans la version courante du projet, et tous les autres options de compilations ont été testés et fonctionnent comme prévu.

# 9 Gestion des risques et Gestion des rendus

| Risque : | Degré d'impact (de 0 à 4) : | Actions pour les éviter : | Actions pour résoudre et dépasser cette situation : |
|---|---|---|---|
| Retard sur les tâches et dépassement du timing consacré à une tâche | 3 | -Toute personne bloquée dans une tâche doit demander de l'aide avant le deadline du timing pour avoir de l'aide (notamment de la part l'expert sur cette étape) | -l'ajustement des délais<br>-choisir des durées des tâches réalistes |
| Oublie des détails du cahier de charges | 4 | -Chaque membre de l'équipe est un expert sur une étape du projet GL :<br>Par expert, on désigne un membre qui s'est documentée sur une des étapes du projet, de sorte à assimiler tous les détails concernant cette partie. | -Relecture récurrente du cahier de charges<br>-On a créé un document ou chaque détail retrouvé est écrit. |
| Risque des erreurs | 4 | -Une fois une sous-tâche terminée, elle passe à la phase test/vérifications. Un ensemble de tests concernant chaque instruction est testé pour savoir si tout a été bien implémenté et sinon avoir un retour sur la nature du problème afin qu'il soit rectifié au plus vite. | -Lors de l'implémentation de nos instructions, il est très important de bien commenter son code. Puisqu'il est possible qu'un membre de l'équipe se trouve dans une situation où il doit modifier une partie du code déjà modifié, il peut lire les commentaires et consulter la personne qui a modifié le code. |
| Risque des conflits sur GIT | 3 | -On travaille par branche d'instruction afin de limiter la casse (éviter les conflits) et une fois la tâche achevée, on fait un merge avec le master puis on vérifie la nouvelle version grâce au lanceur de tests automatiques. | -On a un backup pour lorsqu'il a un conflit on retrouve la dernière version qui fonctionne |
| Risque des imprévus | 4 | -Si par malheur, un ou plusieurs membres n'est pas apte à travailler à cause d'un examen, d'une maladie, un décès dans la famille, il doit informer les membres de l'équipe de son avancement afin qu'il puisse le couvrir. Il doit rester à disposition pour qu'on puisse le consulter si nécessaire. Les commentaires dans les codes sont dans ce cas très utiles. | |
| Un ordinateur défectueux. | 3 | | -On peut demander à l'école un ordinateur. Il faut juste faire en sorte d'agir le plus rapidement pour minimiser le temps perdu |

# 10 Résultats de Jacoco

L'ensemble des tests qu'on a conçues durant le projet GL nous ont permis d'avoir le résultat **Jacoco** suivant :



Deca Compiler

**Deca Compiler**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| fr.ensimag.deca.syntax | | 77% | | 59% | 457 | 680 | 447 | 2,030 | 242 | 376 | 1 | 51 |
| fr.ensimag.deca.tree | | 81% | | 76% | 186 | 784 | 380 | 2,352 | 78 | 517 | 0 | 87 |
| fr.ensimag.arm.pseudocode | | 0% | | 0% | 87 | 88 | 268 | 270 | 82 | 83 | 20 | 21 |
| fr.ensimag.deca | | 52% | | 43% | 69 | 129 | 140 | 326 | 25 | 75 | 1 | 6 |
| fr.ensimag.ima.pseudocode | | 77% | | 75% | 26 | 84 | 42 | 180 | 21 | 74 | 2 | 26 |
| fr.ensimag.arm.pseudocode.instructions | | 0% | | n/a | 22 | 22 | 42 | 42 | 22 | 22 | 16 | 16 |
| fr.ensimag.deca.context | | 89% | | 90% | 27 | 147 | 37 | 281 | 22 | 116 | 1 | 22 |
| fr.ensimag.ima.pseudocode.instructions | | 69% | | n/a | 21 | 62 | 37 | 111 | 21 | 62 | 17 | 54 |
| fr.ensimag.deca.codegen | | 77% | | 100% | 4 | 10 | 5 | 19 | 4 | 9 | 1 | 2 |
| fr.ensimag.deca.tools | | 94% | | 100% | 1 | 16 | 3 | 39 | 1 | 13 | 0 | 3 |
| Total | 6,283 of 25,320 | 75% | 432 of 1,290 | 66% | 900 | 2,022 | 1,401 | 5,650 | 518 | 1,347 | 59 | 288 |

On a réussi à couvrir 89% du package **context** qui contient les différentes classes nécessaires pour la vérification contextuelle ,81% du package **tree** qui contient les classes des différents éléments de l'arbre, 77% du package **syntax** qui contient les différents classes de l'analyse lexicographique et syntaxique. Ce qui nous a donné une couverture moyenne de 75% sur la totalité des classes du projet. Il faut souligner que vue qu'on a implémenter l'extension ARM, la base de test située dans les dossiers correspondants ne testent que la compilation pour le processeur IMA donc la couverture devra largement augmentée si on a ajouté les tests pour l'ARM.

# 11 Autres méthodes :

Pour comparer un petit peu la performance de notre compilateur, on a comparé le temps de compilation et exécution de notre compilateur avec gcc tout en écrivant des programmes plus ou moins simulaire dans le langage Deca et le langage c. Ceci nous a donné une idée sur le temps d'exécution et procéder aux actions correspondants. (A noter que cette méthode n'est pas assez exploité lors de cette projet vu qu'on n'a pas assez de temps pour la double implémentions, et il se peut qu'il soit très intéressant de faire une étude de ce genre pour sortir avec une conclusion pour valider une partie de notre compilateur)