

Projet GL 2021: Documentation de validation

*Équipe n°31 : Antoine Briançon, Axel Glorvigen,
Thibault Launay, Maxime Martin et Pierre Pocreau*

Sommaire:

I.	Introduction	p.1
II.	Descriptif des tests	p.1
III.	Scripts de tests	p.7
IV.	Résultats de Jacoco	p.11
V.	Gestion des risques et rendus	p.12

I. Introduction

Ce document de validation, illustrant les méthodes utilisées pour les tests du projet, leur structure et leur fonctionnement, est une base pour l'utilisateur souhaitant développer de nouveaux tests et continuer l'évaluation du compilateur. Les tests représentent une partie très importante du Projet GL car il permettent de vérifier le bon fonctionnement du compilateur, d'accompagner le débogage et d'illustrer les limites de certains choix techniques, poussant ainsi à améliorer le compilateur pour le rendre plus rapide et plus robuste. Nous allons ainsi revenir en détail sur l'utilisation des tests dans ce projet, afin qu'ils soient accessibles et compréhensibles pour les utilisateurs souhaitant les compléter.

II. Descriptif des tests

La partie dédiée aux tests est essentiellement composée de scripts voués à l'automatisation et de fichiers .deca contenant des programmes valides et invalides selon ce qu'ils sont censés tester. Par exemple, un programme contenu dans le dossier lex/valid sera considéré comme valide uniquement dans le cadre de la lexicographie mais ne le sera pas forcément dans celui de la syntaxe et encore dans celui de l'analyse contextuelle. De plus, ces programmes sont triés en fonction de la partie du compilateur qu'ils testent (langage hello-world, sans-objet ou complet).



Exemple d'organisation des programmes de test (syntaxiques pour langage hello-world)

Ces programmes sont tous écrits dans le même format:

Dans la suite nous allons nous intéresser à la nature des différents tests en fonction de la partie du compilateur qui les concerne.

A. Lexicographie

Localisation: src/test/deca/lex/

Ces tests permettent de vérifier que les règles lexicales écrites dans le Lexer identifient correctement ce qui est lexicalement valide ou invalide dans les programmes.

Par exemple, pour le langage hello-world, on doit savoir si le Lexer est capable de détecter si une chaîne de caractère (string) est correctement écrite. Pour cela, on écrit une batterie de programmes contenant uniquement des chaînes de caractère, respectant les règles rattachées au Lexer dans le cas des tests valides, ne les respectant pas dans la partie invalide.

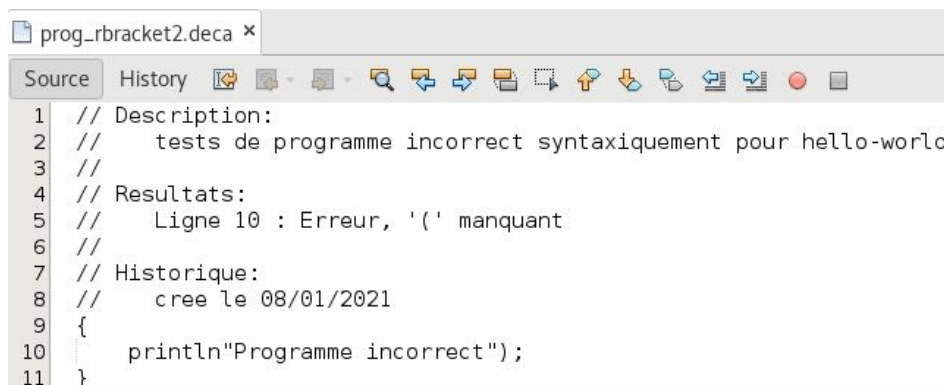
Les règles d'écriture d'un chaîne de caractère est résumée comme ceci:

STRING = ' ' ' (STRING_CAR + '\"' + '\\')* ' ' '

Avec STRING_CAR l'ensemble des caractères excepté ' ' , ' \ ' et EOL, la marque de fin de ligne.

Ainsi, dans le cas des tests valides, on écrit un ensemble de programmes testant chacun une combinaison possible:

- "chaîne de caracteres"
- "chaîne de \\ caracteres"
- "chaîne avec
saut de ligne"
- etc.



```
1 // Description:
2 //   tests de programme incorrect syntaxiquement pour hello-world
3 //
4 // Resultats:
5 //   Ligne 10 : Erreur, '(' manquant
6 //
7 // Historique:
8 //   cree le 08/01/2021
9 {
10     println"Programme incorrect";
11 }
```

Dans le cas des tests invalides, dans l'idéal, on doit avoir écrit autant de programmes qu'il y a de manières de briser les règles du Lexer. Dans le cadre des chaînes de caractères:

- "chaîne pas finie
 - chaîne pas commencee"
 - "chaîne"guillemet en trop"
- etc.

Et ainsi de suite pour toutes les règles lexicales décrites dans le cahier des charges. A noter qu'à ce stade des tests, le Lexer seul va surinterpréter les éléments du programme. Ainsi il est délicat de tester, par exemple, qu'un IDENT ou un FLOAT est mal écrit, puisque sans règles syntaxiques ni contexte il est impossible pour le Lexer de comprendre exactement ce que vous souhaitez tester (c'est plus facile dans le cadre des chaînes de caractères puisqu'elles possèdent des marqueurs distincts représentés par les guillemets ' " ').

B. Syntaxe

Localisation: src/test/deca/syntax/

Ils permettent de vérifier que les règles syntaxiques écrites dans le Parser identifient correctement ce qui est syntaxiquement valide ou invalide dans les programmes.

Ainsi, comme pour le Lexer, on reprend chaque règle et on écrit, dans le cas des tests valides, toutes les combinaisons possibles de programmes qui doivent être considérés comme justes syntaxiquement. Pour les tests invalides, on doit écrire un programme par manière possible de briser chacune des règles.

Si on reprend le cahier des charges et qu'on s'arrête sur l'ensemble des règles concernant l'écriture des instructions:

inst

```
→ expr ';'
|
| ';'
|
| 'print' '(' list_expr ')' ';'
|
| 'println' '(' list_expr ')' ';'
|
| 'printx' '(' list_expr ')' ';'
|
| 'printlnx' '(' list_expr ')' ';'
|
| if_then_else
|
| 'while' '(' expr ')' '{' list_inst '}'
|
| 'return' expr ';'
|
```

On peut très bien imaginer un programme valide contenant toutes ces instructions écrites correctement, même si tout écrire dans un même programme peut amener à des complications si ce test considéré comme valide n'est pas accepté par le Parser à cause d'une erreur dans ce dernier.

En effet, il ne faut pas oublier que ces tests sont avant tout mis en place pour aider les développeurs à détecter les erreurs dans leur code. Un seul programme valide testant toutes les règles ne sera pas pertinent puisque si le Parser le rejette, il n'y a aucun moyen efficace de savoir quelle ligne du programme n'est pas passée ni combien d'autres sont également considérées comme fausses.

Ainsi il est recommandé de fractionner au plus possible les programmes valides et d'éventuellement d'en ajouter des plus complexes une fois que les premiers sont considérés comme valides par le Parser (d'ailleurs, cette règle s'applique aussi pour les tests contextuels).

Pour les tests invalides, il est, comme pour les tests lexicaux, impératif de dédier un programme pour chaque manière possible de briser les règles syntaxiques.

Par exemple, en analysant les règles rattachées au 'while', on peut:

- Oublier d'écrire une parenthèse, ' (' ou ') '
 - Oublier d'écrire un bracket, ' { ' ou ' } '
 - Mal écrire le mot réservé ' while '
 - Ne pas mettre une *expr* là où c'est demandé
- etc.

C'est un travail qui demande une méthode rigoureuse, d'où la nécessité de réfléchir l'ensemble de ses tests et de vérifier à la main si l'ensemble des règles ont été brisées avant d'écrire concrètement le moindre programme de test.

Enfin il faut aussi noter que ces tests, qu'ils soient valides syntaxiquement ou non, doivent être valides lexicalement pour éviter tout problème.

C. Contexte

Localisation: src/test/deca/context/

Ces tests permettent de vérifier que l'analyse contextuelle se fait correctement. Comme pour les deux parties précédentes, on va tester des programmes valides et invalides.

Cette partie est plus complexe car elle demande à prévoir un nombre de cas beaucoup plus grand et complexe. Comme pour les autres parties, l'ensemble des règles contextuelles sont décrites dans le cahier des charges.

Si on prend à nouveau un exemple parmi ces règles:

$$\text{type_arith_op} : \text{Type} \times \text{Type} \rightarrow \text{Type}$$

$\text{type_arith_op}(\underline{\text{int}}, \underline{\text{int}})$	\triangleq	$\underline{\text{int}}$
$\text{type_arith_op}(\underline{\text{int}}, \underline{\text{float}})$	\triangleq	$\underline{\text{float}}$
$\text{type_arith_op}(\underline{\text{float}}, \underline{\text{int}})$	\triangleq	$\underline{\text{float}}$
$\text{type_arith_op}(\underline{\text{float}}, \underline{\text{float}})$	\triangleq	$\underline{\text{float}}$

Ainsi on sait que, à part si les deux variables de l'opération sont de type int, le résultat sera un float. A partir de là on crée un ensemble de programmes valides contenant les opérations et affectations possibles en fonction des types concernés.

De l'autre côté, on prépare également une batterie de tests invalides essayant d'attribuer, par exemple, un int comme résultat d'une opération entre un int et un float.

Comme on peut le voir, la démarche n'est pas différente des autres parties, il faut simplement parvenir à prévoir tous les cas possibles pour chaque règles et rester organisé, en procédant règle par règle et en notant les règles dont les tests ont déjà été écrits.

A l'issue de ce travail, il est également recommandé de créer des programmes valides plus complexes qui font appel à plusieurs de ces règles à la fois.

Enfin, notons que, comme pour la partie syntaxique, les tests valides et invalides contextuellement doivent impérativement être valides syntaxiquement.

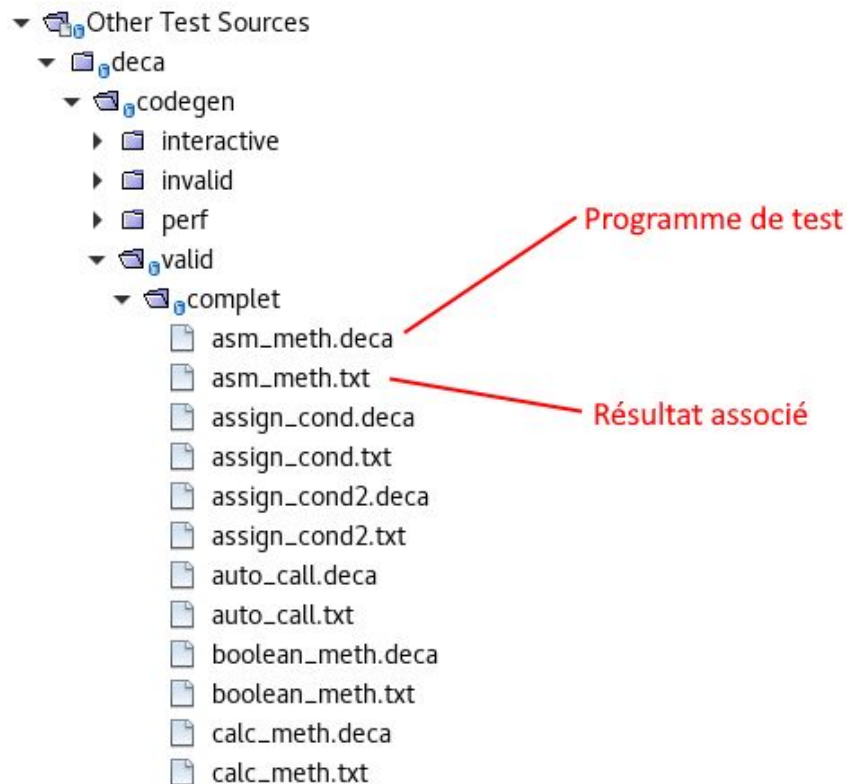
D. Génération de code:

Localisation: src/test/deca/codegen/

Ces tests permettent de vérifier que la génération de code se fait correctement, notamment en vérifiant que la sortie concrète du programme est celle attendue théoriquement.

Pour cela on procède d'une manière différente par rapport aux autres parties précédemment décrites. On écrit un ensemble de programmes test valides possédant un ou plusieurs appels à la fonction print (ou à l'une de ses dérivées comme printx ou println) afin de pouvoir réellement vérifier le bon fonctionnement du programme.

Pour pouvoir comparer les sorties concrètes et celles attendues, on écrit pour chaque programme un fichier .txt contenant la sortie attendue portant le même nom.



Comme pour les parties précédentes, il faut essayer de prévoir chaque cas pour vérifier que le programme sorte exactement la réponse attendue à l'exécution. Par exemple, concernant l'affichage des floats, il faut vérifier que le nombre de chiffres significatifs est suffisant et qu'on ne perd pas d'information lors des calculs. Pour les booleans on vérifie que les opérateurs respectent la logique binaire, etc.

Il ne faut pas hésiter à créer des programmes complexes qui feront appel à plusieurs composantes du langage étudié. Par exemple, pour le sans-objet on peut imaginer plusieurs if et while imbriqués les uns dans les autres pour vérifier que les conditions sont bien interprétées par le compilateur.

Une petite partie des tests est également consacrée à quelques tests invalides. Si on prend l'exemple du langage sans-objet, on vérifie par exemple que le compilateur génère une erreur lorsqu'on déclare un int trop grand ou un float trop petit.

A noter que les différents programmes de test doivent être corrects contextuellement.

III. Scripts de tests

Localisation: src/test/script/

Les scripts permettent d'automatiser les tests en lançant la vérification d'un grand nombre de programmes à l'aide d'une simple ligne de commande. Il existe un script pour chaque partie du compilateur (lexical, syntaxique, etc.) et chaque langage (hello-world, sans objet et complet).

A. Lexicographie

Ces scripts fonctionnent en appelant la classe Java ManualTestLex (localisation: src/test/java/fr/ensimag/deca/syntax) avec pour argument le programme testé. Cette classe renvoie la liste des tokens qu'elle a identifiés si le programme est correct lexicalement. Elle indique aussi via sa sortie si une erreur se trouve dans le programme.

Ainsi on passe par une boucle for afin de lancer l'ensemble des programmes contenus dans un dossier spécifique et on arrête le script prématurément en renvoyant une erreur si on a un succès inattendu de la part d'un programme invalide (sous-entendu le Lexer n'a pas vu de problème dans le programme analysé alors qu'il y a une erreur) ou un échec inattendu (dans le cas où le Lexer a détecté une erreur alors que le programme n'en a pas).

```
test_lex_valide () {  
    if test_lex "$1" 2>&1 | grep -q -e "$1:[0-9][0-9]*:"  
    then  
        echo "Echec inattendu pour test_lex sur $1."  
        exit 1  
    else  
        echo "Succes attendu de test_lex sur $1."  
    fi  
}  
  
echo "SANSOBJET LEX"  
  
for cas_de_test in $WAY/valid/sansobjet/*.deca  
do  
    test_lex_valide "$cas_de_test"  
done
```

Extrait du script sansobjet-lex.sh

B. Syntaxe

Ces scripts fonctionnent en appelant la classe Java ManualTestSynt (localisation: src/test/java/fr/ensimag/deca/syntax) avec pour argument le programme testé. Cette

classe renvoie l'arbre abstrait du programme si ce dernier est correct syntaxiquement. Il renvoie une erreur sinon.

Pour les tests invalides, ces scripts fonctionnent comme ceux du Lexer. Si on a un succès inattendu, on arrête le script et on renvoie une erreur.

En ce qui concerne les tests valides, on vérifie d'abord que la syntaxe du programme est juste. Si c'est le cas, on appelle la commande `decac -p` qui va s'arrêter à la création de l'arbre abstrait avant de le décompiler et sortir le programme né de la décompilation qu'on va stocker dans un fichier `.deca`. On appelle à nouveau la commande `decac -p` sur ce nouveau programme et on stocke le résultat dans un second fichier `.deca`. On compare ensuite les deux fichiers `.deca`, si ils sont identiques, cela signifie que l'arbre abstrait généré est correct. S'ils sont différents, on arrête le script et on renvoie une erreur.

Ci-dessous un extrait du script `sansobjet-synt.sh` qui montre concrètement cette méthode de vérification.

```
test_synt_valide () {
    if test_synt "$1" 2>&1 | grep -q -e "$1:[0-9][0-9]*:"
    then
        rm $WAY/program_from_decompile1.deca $WAY/program_from_decompile2.deca 2>/dev/null
        echo "Echec inattendu pour test_synt sur $1."
        exit 1
    else
        echo "Succes attendu de test_synt sur $1."
        decac -p $1 2>&1 > $WAY/program_from_decompile1.deca
        decac -p $WAY/program_from_decompile1.deca 2>&1 > $WAY/program_from_decompile2.deca
        if diff $WAY/program_from_decompile1.deca $WAY/program_from_decompile2.deca
        then
            echo "Arbre de $1 correct"
        else
            rm $WAY/program_from_decompile1.deca $WAY/program_from_decompile2.deca
            echo "Arbre de $1 incorrect"
            exit 1
        fi
    fi
}
```

A noter qu'il faut faire attention, en cas d'erreur du script, à ne pas déduire trop vite que le problème vient du Parser. Les fonctions de décompilation écrites dans le compilateur peuvent être la source du problème, et donc mener à des erreurs (dans notre projet, en l'état il n'y a plus d'erreurs dans les fonctions de décompilation, mais dans le cas où on voudrait ajouter de nouvelles fonctionnalités au langage, il faut bien prêter attention à ce genre de problèmes).

C. Contexte

Ces scripts fonctionnent en appelant la classe Java ManualTestContext (localisation: src/test/java/fr/ensimag/deca/context) avec pour argument le programme testé. Cette classe renvoie l'arbre abstrait avec indications contextuelles du programme si ce dernier est correct contextuellement. Il renvoie une erreur sinon.

De plus, un test validé ne signifiait pas forcément qu'il n'y avait pas d'erreurs à l'analyse contextuelle. Pour certaines fonctions comme le ConvFloat, il fallait bien regarder l'arbre en sortie afin de s'assurer que la fonctionnalité est bien implémentée.

Le fonctionnement de ces scripts est parfaitement identique à ceux s'occupant des tests syntaxiques. La seule différence est la classe utilisée pour conclure sur la validité ou non du programme de test.

D. Génération de code:

Ces scripts diffèrent beaucoup de ceux décrits précédemment. On appelle directement la commande decac sur chaque programme.

S'il est valide, on vérifie qu'un fichier .ass du même nom a été créé dans le même dossier. On exécute ensuite ce fichier .ass et on vérifie que la sortie est bien celle attendue (en comparant avec le fichier .txt dont on a parlé dans la partie Programmes de test). Si il y a un problème à la moindre de ces étapes (si, par exemple, le fichier .ass n'a pas été créé), on arrête le script et on renvoie une erreur.

Ci-dessous un extrait du script sansobjet-synt.sh qui montre concrètement cette méthode de vérification.

```
for cas_de_test in $WAY/valid/sansobjet/*.deca
do
    echo "$cas_de_test :"
    decac $cas_de_test
    if(! test -f "${cas_de_test%$DE}$AS")
    then
        echo "Le fichier .ass n'a pas été créé"
        remove_ass
        exit 1
    fi
    ima ${cas_de_test%$DE}$AS > $WAY/resultat.txt
    if diff $WAY/resultat.txt ${cas_de_test%$DE}$TE > /dev/null
    then
        echo "Le résultat est celui attendu"
        rm $WAY/resultat.txt
    else
        echo "Erreur: le résultat n'est pas celui attendu"
        echo "Attendu:"
        cat ${cas_de_test%$DE}$TE
        echo "Ce qui est sorti:"
        cat $WAY/resultat.txt
        rm $WAY/resultat.txt
        remove_ass
        exit 1
    fi
done
```

S'il est invalide, on vérifie bien qu'aucun fichier .ass du même nom n'a été créé.

E. Tests de la commande decac:

Nous avons deux scripts de test pour vérifier que la commande decac fonctionne correctement.

Le script test_deca.sh vérifie basiquement que la commande decac compile un programme correct possédant l'extension .deca. Il vérifie également qu'elle ne compile pas les fichiers qui n'ont pas cette extension. Les programmes testés se trouvent dans src/test/deca/codegen/valid/hellotest et ../invalid/hellotest.

Le script OptionParsing.sh vérifie que chaque option de la commande decac rempli correctement son rôle et que le parsing des arguments fonctionne comme décrit dans le cahier des charges:

- decac -b doit renvoyer le numéro d'équipe
- les options -p et -v sont incompatibles
- l'option -P compile plusieurs programmes en parallèle
- etc.

En cas de problème, ces scripts s'arrêtent et renvoient une erreur.

F. Résultats des tests:

Pour lancer un script de test, on utilise la commande ./script en se plaçant dans le dossier des scripts (rappel: src/test/script/).

Ci-dessous un exemple d'exécution du script hello-lex.sh:

```
[ensimag@ensimagvm script]$ ./hello-lex.sh
HELLO-LEX
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/chaine_complete_backslash2.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/chaine_complete_backslash.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/chaine_complete_complexe.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/chaine_complete.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/chaine_complete_EOL.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/commentaire_simple.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/include_espaces.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/include_fichier_complexe.deca.
Succes attendu de test_lex sur src/test/deca/lex/valid/hellotest/include_simple.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_backslash2.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_backslash.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_EOL_backslash.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_EOL_fail.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_guillemet.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/chaine_incomplete2.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/class_bad_multistring.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/include_fichier_incorrect.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/include_sans_guillement.deca.
Echec attendu pour test_lex sur src/test/deca/lex/invalid/hellotest/include_vide.deca.
HELLO-LEX OK
```

Le script `tests_etu.sh` permet de lancer tous les scripts décrits précédemment les uns après les autres. A la moindre erreur le script s'arrête et renvoie une erreur. C'est un moyen efficace mais relativement long (compter plusieurs minutes, dépendant de la puissance de votre machine) de vérifier que le compilateur passe tous les tests écrits depuis le début. Ce script passe actuellement entièrement pour notre version du compilateur.

IV. Résultats de Jacoco

Pour vérifier la couverture des tests, c'est-à-dire vérifier les parties du code Java par lesquelles le script `tests_etu.sh` va passer, on utilise l'outil Jacoco. Cette analyse de couverture est très utile pour savoir si nos tests prennent en compte un maximum de possibilités et de règles du langage. Si il s'avère que la couverture est mauvaise par rapport aux standards qu'on s'impose, on peut savoir précisément quelles parties du code ont été sollicitées et lesquelles ne l'ont pas été. De là, on peut déduire quels tests ajouter pour optimiser la couverture.

Pour lancer l'outil Jacoco, il faut se placer à la racine du projet et écrire les commandes suivantes l'une après l'autre:















```
mvn compile test-compile
mvn clean
mvn -Djacoco.skip=false verify
```

Compter quelques minutes d'attente après le lancement de la dernière commande. Une fois que l'analyse de couverture est terminée, il est possible d'analyser la couverture en lançant la commande suivante:

```
firefox target/site/jacoco/index.html
```

Ci-dessous le site qui s'ouvre après exécution de la commande:

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		76 %		55 %	601	803	463	2 017	252	370	3	49
fr.ensimag.deca.tree		87 %		78 %	153	697	210	1 918	65	457	3	86
fr.ensimag.deca		81 %		79 %	22	84	45	248	6	43	0	6
fr.ensimag.deca.codegen		75 %		64 %	28	83	40	187	9	51	0	3
fr.ensimag.deca.context		86 %		82 %	31	145	32	239	24	116	1	22
fr.ensimag.ima.pseudocode		83 %		82 %	21	93	31	198	15	76	1	26
fr.ensimag.ima.pseudocode.instructions		65 %		n/a	22	62	38	111	22	62	17	54
fr.ensimag.deca.tools		94 %		100 %	1	17	3	40	1	14	0	3
Total	4 310 of 23 271	81 %	523 of 1 531	65 %	879	1 984	862	4 958	394	1 189	25	249

En cliquant sur les différents éléments, il est possible d'avoir plus de précisions sur la couverture de chaque composant du code, jusqu'à voir le code lui-même. La couverture actuelle est plutôt satisfaisante mais il est possible d'encore l'améliorer.

V. Gestion des risques et des rendus

Nous présentons ici une liste des risques les plus importants liés à la réalisation du projet. En effet, de nombreux points critiques sont identifiables à toutes les étapes du projet, et chacun de ces points peut mettre en danger la pérennité du travail effectué. C'est la raison pour laquelle nous proposons des actions à mettre en place afin de maintenir une organisation efficace et cohérente de l'équipe, ainsi qu'un bon fonctionnement technique du projet, même en cas de difficulté ou de complication. Voici donc la liste des risques potentiels identifiés, avec leur solution associée sous forme d'action concrète.

- Le dépôt sur la branche master, après un nouveau commit, ne fonctionne pas. Ainsi, le corps du projet devient non fonctionnel. On met en place des tests automatiques depuis Gitlab en amont afin de valider chaque commit ou fusion vers la branche master, et on valide ainsi que tous les tests prévus passent.
- L'oubli d'une date de rendu. On affiche toutes les dates de rendus et de suivis sur le planning du projet, on rappelle ces dates à l'oral lors des réunions quotidiennes de l'équipe et on crée des "issues" sur Gitlab afin de ne pas oublier ces dates critiques.
- Les tests ne sont pas complets, ils laissent passer des bugs/des erreurs. On relit la base de tests et on vérifie que tous les cas de figure sont traités. On utilise ensuite Jacoco pour évaluer la couverture des tests et créer des test pour les fonctions non testées.
- Les outils d'aide au développement ne sont pas entièrement maîtrisés (Git, IDE, Maven, ...) par un des membres de l'équipe. On prend le temps nécessaire pour voir quels sont les problèmes rencontrés et on trouve des solutions collectives, avec l'aide d'un professeur si besoin.
- La méthode de gestion du projet est inefficace, l'équipe n'est pas coordonnée et n'arrive pas à se synchroniser dans le cadre du travail à distance. On organise une réunion afin de restructurer la gestion du projet et repartir sur de nouvelles techniques de gestion.
- Le projet est en retard, les fonctionnalités basiques ne sont pas présentes. On se concentre alors sur le test common-tests.sh et on reprend étape par étape les fonctionnalités nécessaires à la réussite de ce test, afin de les implémenter correctement.
- Les documentations à rendre ne sont pas complètes lors de la date de rendu. On se concentre sur la qualité de ce qui est déjà fait plutôt que sur la taille du

document, et on ajoute si possible les sujets clés attendus dans les documentations.

- Les spécifications précises du projet (format des messages d'erreur, arborescence du projet, etc ...) ne sont pas respectées. On relit entièrement les passages liés aux spécifications imposées et on ajoute ces fonctionnalités au projet existant.

La gestion des rendus correspond aux actions et vérifications effectuées avant chaque séance du suivi ou date de rendu. Ces actions doivent être mises en place pour assurer la coordination entre les membres de l'équipe et pour certifier du bon fonctionnement du projet. On propose ainsi la liste des actions suivante:

- Tous les membres de l'équipe passent les tests en local.
- Toutes les branches de développement sont fusionnées dans master, et on teste la branche master en deux étapes : en local et sur Gitlab.
- On s'assure que tous les tests sont présents dans le dépôt Gitlab.
- On relit toutes les documentations.
- On prépare les entretiens en avance, avec des slides et/ou les documents attendus.
- On met à jour les documents de gestion de projet : diagramme de Gantt, "issues" de Gitlab, etc ...