# Documentation de validation

## 1 Descriptifs des tests

Le compilateur decac étant un programme informatique composé de plusieurs étapes, chacune composée de parties indépendantes, il est nécessaire, pour assurer le bon fonctionnement du compilateur, que des tests soient effectués. Ces tests ont pour objectif de vérifier la validité de chacune des étapes du compilateur, pour cela il faut réaliser des tests validant ou invalidant unitairement chacune des parties que compose le compilateur. De plus, il n'est pas suffisant de vérifier la validité de chaque partie indépendamment, il est aussi nécessaire de réaliser des tests systèmes, qui vont vérifier la validité du programme dans son intégralité. L'ensemble des tests ne peuvent pas couvrir la totalité des cas possibles pour le programme decac, il est donc impossible d'avoir une couverture complète du programme. Cependant, il reste nécessaire de couvrir, avec des tests, une grande partie des cas possibles, pour assurer une stabilité pour le compilateur dans la majorité des possibilités.

L'ensemble des scripts de tests, ainsi que les fichiers sont regroupés dans un répertoire commun *"src/test/"*. Le répertoire contenant la totalité des scripts de tests automatisé est situé dans *"src/test/scripts/"*, on y retrouve plusieurs fichiers *'.sh'*, qui sont les scripts et dont chacun d'eux a un nom évocateur de la fonctionnalité qu'il test, par exemple *"test-synt.sh"* qui vérifie la validité et l'invalidité syntaxique de plusieurs fichiers sources. En ce qui concerne les fichiers sources qui sont testés, il faut suivre le chemin *"src/test/deca/"*, où sont regroupés l'ensemble des fichiers *.'deca'*. Pour chacune des étapes qui constituent le compilateur, on associe une répertoire différent (*"src/test/deca/syntax"*, *"src/test/deca/context"*, *'src/test/deca/codegen"*). Dans chacun de ces répertoires sont situés deux autres répertoires *"/valid"* et *"/invalid"* permettant de dissocier les fichiers sources devant être validé et ceux devant être invalidé par le test associé à l'étape. Finalement, on retrouve dans les deux répertoire précédent un répertoire *"/provided"* contenant le ou les fichiers sources fournis, et un répertoire *"/non_provided"* contenant les multiples fichiers sources devant couvrir le plus grand nombre de cas possible dans chaque étape du compilateur.

Pour valider la première étape du compilateur, l'étape A, concernant la vérification lexicale et syntaxique des fichiers sources, des tests unitaires ont été mis en place. Cette étape étant construite en deux parties, chacune a été testé indépendamment. Cependant, étant donné que la vérification syntaxique suit la vérification lexicale, il est nécessaire de les tester dans un ordre cohérent. Pour chacune des deux vérifications (lexicale et syntaxique), des scripts de test ont été implémenté, ces scripts permettent de tester la validité et l'invalidité d'une multitude de fichiers sources. Respectivement "*basic-lex.sh"* et "*basic-synt.sh"* effectuent la vérification lexicale et syntaxique sur plusieurs fichiers, chacun de ces scripts s'arrêtent en cas de résultat inattendu, tout en affichant un message approprié pour détecter le fichier ayant posé problème.

En ce qui concerne la validation par des tests de la seconde étape du compilateur, l'étape B, concerne la vérification contextuelle ainsi que la décoration de l'arbre abstrait construit par la première étape du compilateur. Pour valider les différentes vérifications de contextuelles, on applique un script appliquant l'étape B sur un grand nombre de fichiers, pouvant être valide ou invalide. Ces fichiers ont pour but de couvrir, au maximum, la multitude de cas possibles. Le script réalisant cet ensemble de vérification de manière automatique est "*basic-context.sh"*, le script s'arrête en cas d'échec, pour une validation ou une invalidation, en affichant le nom du fichier sur lequel le test a échoué. Pour ce qui est de la vérification de la décoration, elle est faite en utilisant le compilateur decac directement sur les différents fichiers à l'aide d'une méthode nommée "*checkAllDecorations"*.

Pour valider la dernière étape du programme decac, l'étape C, qui concerne la génération du code machine, on utilise directement le compilateur sur un ensemble de fichiers. Pour effectuer une vérification de la génération de code machine, on réalise deux étapes pour chaque fichier devant être vérifié. Tout d'abord, on vérifie la création du fichier '*.ass'* associé, composé du code machine, puis dans un second temps, on vérifie le résultat de l'exécution du fichier compilé à l'aide d'un fichier texte reprenant le nom du fichier auquel il est associé suivi du suffixe "*_result.txt"*. Par exemple, pour une fichier "*hello.deca"* correct, un fichier "*hello_result.txt"* est associé, et lors de la compilation un fichier "*hello.ass"* est créé puis appliqué à la machine abstraite "*ima"*.

L'ensemble de ces tests ont pour but de couvrir le plus grand nombre de cas possible tout en automatisant la validité des différentes étapes du compilateur decac. Pour couvrir au mieux les multiples cas possibles de programmes sources, des fichiers de test ont été produit à chaque implémentation d'une nouvelle fonctionnalité dans chaque étape

du compilateur. Cependant, tous les scripts sont tous indépendants et la vérification totale de la validité du programme doit pouvoir être réalisé en une unique commande. Pour effectuer l'ensemble des scripts de test, il suffit d'utiliser la commande *"mvn test"* à la racine du compilateur, cette commande exécute les tests successivements et de manière automatique. Chaque test est précédé d'une bannière permettant de savoir quel lequel de ces tests est en cours d'exécution.

# 2 Les scripts de tests

Pour réaliser l'exécution de l'ensemble des jeux de tests, il est possible d'utiliser la commande de maven: *"mvn test"*, mais il est aussi possible de lancer ces tests sans utiliser maven, pour cela la commande: *"src/test/script/global-test.sh"* exécutera successivement tout les tests dans un ordre cohérent.

Lors de l'exécution des différents scripts, pour s'assurer qu'aucune erreur n'a été détectée, un indicateur visuel est présent. Pour chacune des étapes, une bannière précède les tests et donc indique quelle partie est en cours de validation. De plus, pour chaque fichier testé, une préfixe *"[OK]"* apparaît si le test a eu le résultat attendu sur le fichier, dans le cas contraire, le test de la partie en cours s'arrête sur le fichier qui pose problème et passe à la partie suivante.

Par exemple, pour un fichier *"hello-world.deca"* valide au point de vue contextuelle (et donc lexicale et syntaxique aussi), l'affichage sera de la forme:

*[OK] Succes attendu de test_context sur  provided/hello-world.deca.*

Pour un fichier *"comp_inf.deca"* non valide contextuellement, l'affichage sera de la forme:

*[OK] Echec attendu pour test_context sur  non_provided/comp_inf.deca.*

Pour la dernière étape de validation, la génération de code, la validation se réalise en deux étapes, une première vérification de la compilation, suiv par une vérification de l'exécution avec des fichiers textes associés pour vérifier le résultat attendu.

# 3 Gestions des risques et gestions des rendus

Dans l'objectif de rendre le compilateur le plus proche possible du zéro défaut, un ensemble de tests unitaires, accompagnés d'une multitude de fichier .deca valide et invalide, ayant pour but de couvrir au maximum les cas d'erreur du langage Deca pour chacune des parties, l'analyse lexicale, l'analyse syntaxique, l'analyse contextuelle ainsi que la décoration de l'arbre. Cet ensemble de tests utilisant les scripts de tests fournis pour le projet, il est assez simple de se convaincre que les risques de bugs dans notre infrastructure de tests unitaires sont réduits. De plus, pour compléter cette base de tests, des tests systèmes visant à tester la partie de génération de code sont aussi effectués. L'utilisation du compilateur decac, ainsi que le programme ima directement sur l'ensemble des fichiers .deca de notre base de tests permet la vérification des fichiers .ass ainsi que la vérification des résultats sur la sortie standard. Avec cet ensemble de tests unitaires et systèmes, on couvre une grande partie des possibilités du langage Deca et du compilateur decac.

Au cours du projet génie logiciel, des dangers potentiels ont été identifié, dans le but de pouvoir les éviter ou de les résoudre plus facilement. Les dangers identifiées sont :

- Ne pas rendre des documents attendu lors des différents suivis
- Se focaliser uniquement sur l'aspect code et ne pas produire de la documentation régulièrement
- Oublier les dates de rendus et se retrouver pris de cours pour finir
- Tests simples qui ne passent pas (perte de crédibilité)
- Base de tests insuffisante (ne couvre pas assez les différent cas)
- Erreur dans l'implémentation des différentes parties (on a fait des tests)
- Ne pas respecter la classification, ce qui rendrait le rendu non testable par les enseignants
- Les membres oublient de add, commit et de pull assez souvent (problème de cohérence dans les fichiers, conflits)

En ce qui concerne les rendus de documents lors des suivis, les principaux dangers étaient d'oublier les dates de ces rendus et donc de ne pas les préparer ou bien les préparer trop tard. Pour empêcher que cela arrive, un agenda sur smartphone a été utilisé, ce qui permet d'avoir des rappels réguliers pour l'ensembles des rendus et donc de

pouvoir les anticiper. De plus le chef d'équipe, en utilisant les moyens de communications de l'équipe, tel que Trello, qui permet de savoir pour chacunes des fonctionnalitées, l'état dans laquelle elle se trouve (à faire, en cours ou fini). a rappelé régulièrement l'arrivée du prochain rendu et il a demandé aux membres de l'équipe de rédiger des parties de la documentation prioritaire.

Un des dangers principal était de se focaliser uniquement sur l'aspect code et donc de négliger le reste des rendus, tel que la documentation. Pour réduire ce problème, les différentes documentations ont été commencé au cours de la deuxième semaine du projet génie logiciel. Cela avait pour objectif de compléter la documentation étape par étape pour réduire la masse de travail à la fin du projet. Cependant les problèmes d'implémentations rencontrés au cours du projet ont empêché l'avancement constant de la documentation, et donc le danger en question n'a pas pu être complètement évité.

Un danger était que des tests simples ne fonctionnent pas sur not compilateur et donc créer une perte de crédibilité auprès de l'utilisateur. Pour conter ce danger, plusieurs scripts de tests ont été implémenté, pour chaque étape du compilateur, au moins un script de test a été associé pour vérifier unitairement chacune des étapes.

En ce qui concerne les risques liés au programme decac, un des principaux dangers était de ne pas suffisamment couvrir les multitudes de cas possibles avec l'ensemble de tests. Pour pallier à ce danger, la mise en place d'un script de test unitaire facilement compréhensible, et validant un ensemble de fichiers .deca, enrichi tout au long du projet génie logiciel, pour couvrir chaque cas qui a été implémenté. Ce script de test accompagné de cet ensemble de fichiers permet de limiter le manque de couverture des cas possibles. Cette solution impliquant un nouveau danger, ne pas avoir une base de fichier à tester suffisante, et donc de faire trop confiance au test et ne pas voir ce qu'il manque, pour pallier à ce problème, des fichiers de tests valides et invalides ont été produits pour chaque étape du compilateur et pour chaque nouvelle fonctionnalité implémentée. De plus, l'utilisation de Cobertura permet de voir la couverture de notre programme par les tests, et ainsi d'ajuster la quantité de tests.

Pour les potentiels problèmes de classification, ce qui empêcherait les enseignants de tester le rendu, l'unique solution mise en place par le groupe est la rigueur de chacun, ainsi qu'une vérification régulière par les membres de l'équipe pour s'assurer que les fichiers de test était bien situé au bon endroit avec un bon nommage pour éviter toute erreur.

Pour pallier au problème de git, étant donné que l'équipe travaillait sur une unique branche, il était important d'éviter les conflits entre les versions de chacun. Pour éviter cela, tout comme précédemment, on compte sur la rigueur de l'équipe, mais aussi le chef d'équipe qui rappelait régulièrement aux membres de *pull* et *commit* tout au long de l'avancement de leur partie. Etant donné que les membres de l'équipe n'était pas expérimenté avec l'outil Git, il était important de garder un fonctionnement simple mais efficace pour ne pas créer de gros problèmes avec le projet. De plus, il était nécessaire de lancer la commande de test avant chaque *commit* pour s'assurer que ce qu'on ajoutait au programme decac était correct et n'empêchait pas la compilation ni l'exécution du programme.

Gestion des mises en production

- Chacun des membres fait un git add afin de s'assurer que toutes les modifications ont été ajoutées sur le git
- Des membres de l'équipe désignés vérifient que la classification des tests, des commentaires
- Si une modification est faite au git, on recommence toute la checklist
- Faire repasser les tests unitaires ainsi que les tests système pour s'assurer de la validité du projet
- Vérifier que les commentaires sont pertinents et assez détaillés

# 4 Résultats de Cobertura

Cobertura est un outil permettant de calculer la couverture d'un jeu de tests sur un programme. On peut savoir quelle instruction a été exécutée (ou non), et quelles branches d'une instruction conditionnelle (if,while) ont été prises.

Pour notre compilateur, on souhaite couvrir au maximum chacune des branches de l'arbre pour s'assurer que l'ensemble des cas possibles a été couvert. Etant donné que l'on ne peut pas couvrir totalement le programme et s'assurer que tous les cas sont traités, on cherche à maximiser le nombre de branches parcouru.

Ci dessous, le rapport du Cobertura sur notre jeu de test.

| Package | # Classes | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|---|
| (default) | 1 | 0% 0/73 | 0% 0/56 | 0 |
| All Packages | 252 | 69% 3504/5043 | 49% 740/1493 | 1.7 |
| fr.ensimag.deca | 6 | 50% 114/224 | 29% 30/101 | 3.125 |
| fr.ensimag.deca.codegen | 1 | 85% 497/584 | 67% 113/168 | 2.458 |
| fr.ensimag.deca.context | 23 | 72% 178/246 | 56% 36/64 | 1.445 |
| fr.ensimag.deca.syntax | 50 | 66% 1362/2056 | 41% 310/744 | 1.969 |
| fr.ensimag.deca.tools | 4 | 38% 18/47 | 40% 4/10 | 1.643 |
| fr.ensimag.deca.tree | 87 | 74% 1135/1527 | 70% 232/330 | 1.513 |
| fr.ensimag.ima.pseudocode | 26 | 77% 136/175 | 75% 15/20 | 1.176 |
| fr.ensimag.ima.pseudocode.instructions | 54 | 57% 64/111 | N/A N/A | 1 |

| Classes in this Package | Line Coverage | Branch Coverage | Complexity |
|---|---|---|---|
| Math2 | 0% 0/73 | 0% 0/56 | 0 |

Certaines parties de ce rapport sont moins pertinentes que d'autres, en l'occurrence, "*fr.ensimag.ima.pseudocode.instructions*" regroupant les différentes instructions possibles pour le langage machine *ima*, ne peut être couvert totalement car seulement une partie des instructions ont été utilisé lors de la génération de code.

On constate que les packages concernant les étapes B et C, "*tree*", "*pseudocode*", "*context*" et "*codegen*", ont une couverture par le jeu de tests plutôt satisfaisante avec un pourcentage supérieur à 70% pour chacun de ces packages.

Une couverture aussi conséquente implique que les tests automatisés vérifient et utilisent une grande partie du code concernant les étapes de vérifications contextuelles et de génération de code. Concernant le package "*tree*", il contient l'ensemble des éléments de l'arbre syntaxique, et il est important d'avoir une bonne couverture de ses classes car elles représentent la base du code source. L'objectif de l'ensemble des fichiers à tester est de d'utiliser un maximum de code pour couvrir tous les cas traité par les différentes vérifications, cela permet de s'assurer de la validité des vérifications.

Le package "*codegen*", contenant l'ensemble de la gestion de la génération de code, obtient une couverture de l'ordre de 80%, on peut extrapoler le fait qu'on couvre 80% des cas de la génération de code avec nos tests.

Il est toujours possible d'améliorer la couverture, et l'outil Cobertura permet de savoir à quel pourcentage chacune des classes a été couverte et donc de savoir quel élément n'ont pas été traité pour améliorer la couverture du programme par les tests. Au cours du projet génie logiciel, Cobertura a permis de constater ces manques de

couverture au niveau du code et donc d'enrichir la base des fichiers test efficacement, en ciblant les éléments de l'arbre les moins couvert.

Finalement, pour l'extension, la couverture est de 0 %, car il ne nous a pas été possible de la compiler avec notre propre compilateur decac et donc de pouvoir tester chacune de fonctionnalités offertes par l'extension trigonométrique, qui permet d'avoir accès à des fonctions mathématiques complexes.

# 5 Autres méthodes de validation

Les tests automatiques ont été une grande part de la validation du programme decac, cependant, d'autres méthodes ont été employé tout au long du projet génie logiciel par chacun des membres de l'équipe en fonction de leur besoin.

Concernant l'extension trigonométrique, une grande partie des tests concernant les algorithmes et leur précision ont été testé avec la machine virtuelle java, pour s'assurer de leur fiabilité. De plus, pour le développement du compilateur, les exemples du polycopié fournis ont permis de construire des bases pour chacune des étapes, tout en pouvant vérifier les résultats obtenu.