Grenoble INP – ENSIMAG, UGA
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Projet Génie Logiciel

Documentation de validation

Yanis BOUHJOURA, Hugo DABADIE, Garance DUPONT-CIABRINI, Marek ELMAYAN, Julien LALANNE

GL09 Groupe 2

27 janvier 2022

Table des matières

1 Descriptif des tests

1.1 Types de tests

Notre base de test contient à la fois tests unitaires écrits en java et tests systèmes écrits en deca. Néanmoins, la quantité de tests unitaires dont elle dispose est bien plus faible que celle des tests systèmes. En effet, nous pouvons compter un test unitaire pour chaque opération arithmétique et booléenne pour l'étape contextuelle mais bien plus de tests systèmes (fichiers .deca) pour chacune des trois étapes du compilateur. Ainsi nous pouvons dénombrer environ une dizaine de tests unitaires contre plus de 500 tests systèmes répartis entre les trois étapes du compilateur : lexicalle/syntaxique, contextuelle et génération de code/décompilation.

1.2 Organisation des tests

En ce qui concerne l'organisation des tests, ceux-ci sont répartis dans différents répertoires et sous-répertoires. Comme vous le savez, les tests systèmes sont présents dans src/test/deca et les tests unitaires dans src/test/java/fr/ensimag. Comme précédemment stipulé, nous n'avons ajouté des tests unitaires que pour la partie contextuelle, ils se trouvent donc tous dans le répertoire src/test/java/fr/ensimag/context. Concernant les tests systèmes, bien plus de répertoires sont concernés. En effet, les tests sont répartis entre les répertoires syntax, context, codegen et decompile. Voici les graphiques de l'architecture de chacun de ces répertoire.

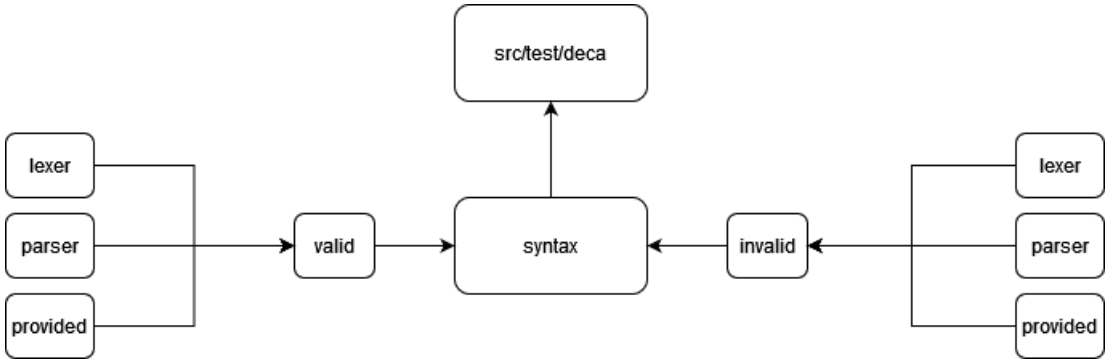


FIGURE 1 – src/test/deca/syntax

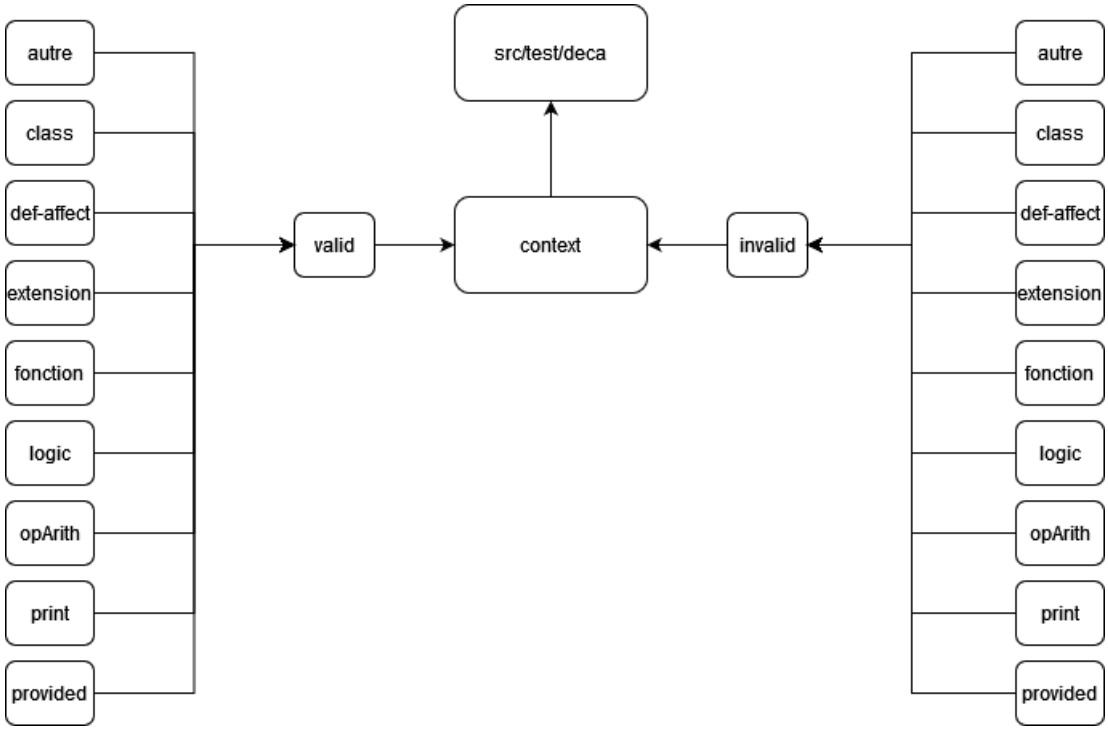


FIGURE 2 – src/test/deca/context

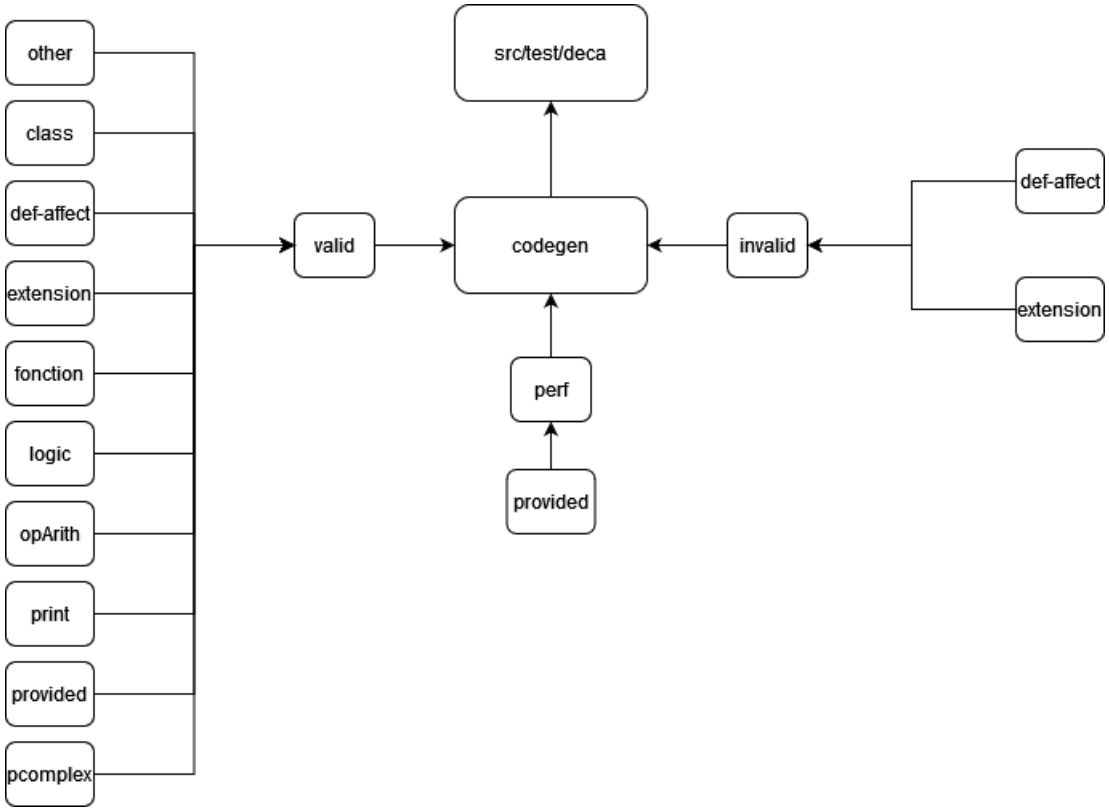


FIGURE 3 – src/test/deca/codegen

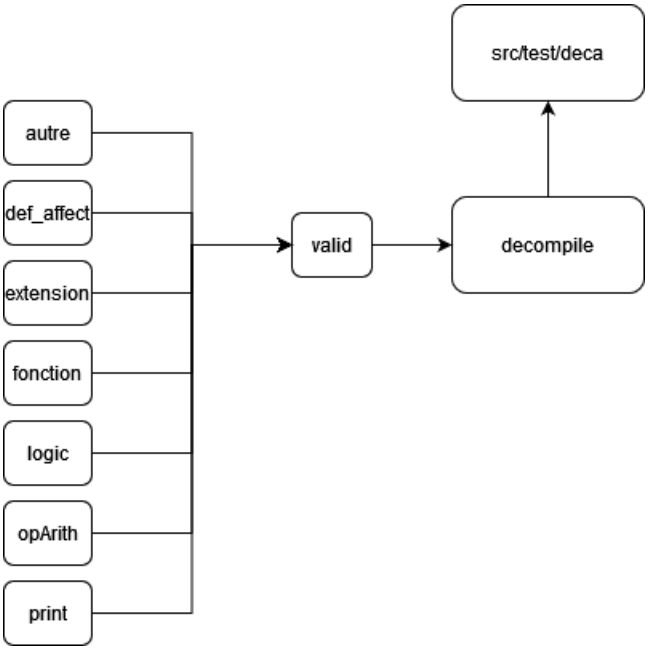


FIGURE 4 – src/test/deca/decompile

1.3 Objectifs des tests

L'objectif de ces tests était de nous permettre de vérifier efficacement les différentes fonctionnalités de notre compilateur tout en sachant, si quelque chose venait à ne pas fonctionner, à quelle étape le problème survient, qu'est ce qui ne fonctionne pas et pourquoi. En effet, les tests étant répartis en répertoires aux noms explicites et étant très concis, nous savions tout de suite ce qu'il fallait modifier et dans quel but.

De plus, la personne réalisant les tests n'étant pas celle travaillant sur la fonctionnalité pour laquelle ceux-là sont destinés, cela permettait aux tests de ne pas être fait dans le but qu'ils passent. Cela a également permis de faire du TDD (test driven development) pour un nombre relativement conséquent de fonctionnalités.

En fin de projet, les tests qui étaient rajoutés avaient pour but de trouver le plus de failles possibles afin de pouvoir les corriger. Bien évidemment si ceux-ci passaient c'était une bonne chose mais nous voulions vraiment savoir si notre compilateur comportait des failles auxquelles nous n'avions pas pensées. Nous nous sommes aidés de la mesure de couverture fournie par l'outil Jacoco afin, justement, de couvrir ces failles.

2 Scripts de tests

2.1 Explication du fonctionnement

Les scripts de test sont conçus pour pouvoir lancer la totalité des tests .deca écrits, ils affichent en vert les succès et en rouge les échecs suivi du nom du test en question et sont présents dans le répertoire src/test/script. Nous avons écrit cinq scripts permettant de lancer les tests : all-test-lex.sh, all-test-syntax.sh, all-test-context.sh, all-test-codegen.sh et all-test-decompile.sh. Chacun d'eux test une partie spécifique de notre compilateur, respectivement : les tests lexicaux, syntaxiques, contextuels, de génération de code et de décompilation.

Les scripts all-test-lex.sh et all-test-syntax.sh lancent respectivement le test test_lex et test_synt (src/test/script/launchers) sur les fichiers .deca dans src/test/deca/syntax, plus précisement le premier lance tous les tests dans les répertoires src/deca/syntax/valid/lexer et src/deca/syntax/invalid/parser et le second dans les répertoires src/deca/-syntax/valid/parser et src/deca/syntax/invalid/parser. Il est possible de lancer ces deux scripts avec l'option –no-exit afin qu'ils ne s'arrêtent pas lorsqu'un test échoue. A noter que cette option empêchera l'affichage des erreurs, celles-ci n'étant affichées que lorsque les scripts sont lancés sans option car cela serait trop encombrant à l'écran si plusieurs tests venaient à échouer (la lecture des erreurs serait très laborieuse). Dans les deux cas les tests échouent si test_lex ou test_synt renvoie une erreur.

Le script all-test-context.sh lance test_context sur les fichiers .deca présents dans le répertoire src/test/deca/context. De la même manière que les scripts précédemment évoqués, il est possible de le lancer avec l'option –no-exit ce qui aura les mêmes effets que ci-dessus listés. Néanmoins, ce test peut également être lancé, en plus de l'option –no-exit, avec l'option –nom-de-répertoire. Le nom-de-répertoire doit être un sous-répertoire de src/test/deca/context/valid ou src/test/deca/context/invalid et ne pas être provided. Cela aura pour effet de lancer les tests seulement sur le répertoire cité (à la fois dans valid et invalid) afin de travailler plus efficacement sur une fonctionnalité sans avoir à attendre que tous les répertoires soient passés pour arriver à ce sur quoi on travaille. Ainsi, si l'on cherche à vérifier que les prints fonctionnent correctement et que tous les tests passent, on peu faire ./all-test-context.sh –no-exit –print pour lancer tous les tests concernant les fonctions d'affichage. Attention, il est important de mettre cette option en second argument sinon cela ne fonctionnera pas. A noter qu'il est également possible de faire ./all-test-context.sh –exit –print afin de s'arrêter sur le premier test qui échoue dans lessous-répertoires src/test/deca/context/valid/print et src/test/deca/context/invalid/print en affichant l'erreur responsable de l'échec. Les tests échouent si test_context renvoie une erreur.

Le script all-test-gencode.sh lance la commande decac sur tous les fichiers .deca présents dans src/test/deca/codegen. Il est encore possible avec celui-ci de le lancer avec l'option –no-exit pour les mêmes effets. Il est important de noter que les répertoires de test dans codegen ne contiennent pas que des fichiers .deca. En effet, ils contiennent également un fichier .res pour chaque fichier .deca. Ces fichiers .res contiennent les sorties attendues du fichier .deca dont ils portent le nom. Cela nous permet de comparer la sortie attendue des fichiers .res à la sortie réelle obtenue après compilation et exécution des fichiers .deca. Le test vérifiera, pour le répertoire valid, si le fichier .ass correspondant au fichier .deca compilé est généré et lancera ima sur le fichier .ass si celui-ci est généré avant de comparer le fichier .res correspondant à la sortie obtenue. Le test échoue si le fichier .ass n'est pas généré ou si la sortie diffère du fichier .res. En ce qui concerne le répertoire invalid, si le fichier .deca est censé compiler mais échouer à l'exécution (sous-répertoire compile), le test échoue si le fichier .ass n'est pas généré ou si aucune erreur n'est renvoyée. Enfin, si le fichier .deca ne doit même pas compiler, le test échoue si le fichier .ass est généré.

Pour finir, le script all-test-decompile.sh lance la commande decac sur tous les fichiers .deca présents dans src/test/deca/decompile. Le test vérifiera si après avoir généré le fichier .ass puis généré un second fichier .deca à partir du fichier .ass, les deux .deca sont identiques. Ici, tous les tests sont valides, c'est-à-dire ne doivent pas générer d'erreur. En effet, si les fichiers ne compilent pas, il n'est pas possible de savoir si les tests fonctionnent. Le test échoue si le fichier .ass n'est pas généré ou si les deux .deca sont différents.

Pour ajouter des tests, il suffit donc d'ajouter des fichiers .deca dans les répertoires correspondants aux types de tests que vous voulez lancer ainsi qu'aux fonctionnalités à tester. L'ajout de test se trouve donc être très simple à réaliser.

Nous avons également réalisé un script pour connaître le nombre exact de fichiers .deca présents dans notre base de test. Il s'agît du script count.sh, il affichera le nombre de fichiers .deca ainsi que le nombre de lignes total de test.

2.2 Exemple

Pour illustrer un peu toutes ces explications concernant les tests, voici un exemple d'affichage terminal à la suite du lancement de la commande ./all-test-context.sh –no-exit –other :

Comme on peut le voir, tous les tests concernant la catégorie "autre" passent et "SUCCES" est affiché en vert suivi du nom du fichier en question. Si un test échouait, il y aurait affiché "ECHEC" en rouge suivi du nom de fichier sans pour arrêter et afficher l'erreur car cela a été lancé avec l'option –no-exit.

On remarque également qu'il y a le nombre de tests qui viennent d'être fait le pourcentage de succès pour ceux la.

3 Gestion des risques et des rendus

Lors de chaque période de rendu, lors des livraisons intermédiaires, il est important de vérifier que toutes les fonctionnalités implémentées qui doivent fonctionner passent toujours les tests les concernant mais aussi veiller à ce que le code soit propre et optimisé. Il est également important de vérifier si tous les documents sont présents et que le logiciel fonctionne sur un environnement de travail nouveau (pas les machines personnels des développeurs). Pour cela, différentes actions sont à réaliser :

— Vérifier que tous les documents sont présents
— Lancer une analyse du code
— Optimiser le code et résoudre les points de l'analyse
— Merge les branches de développement (develop par exemple) sur master
— Vérifier que la branche master soit à jour
— Récupérer les sources depuis master
— Vérifier que le projet compile
— Vérifier que tous les tests qui doivent passer passent
— Vérifier que common-test.sh passe
— Faire un mvn clean

Vérifier que tous les documents sont présents

Il faut reprendre la liste des livrables demandés pour le rendu et vérifier la présence de chacun d'eux.

Lancer une analyse du code

On lancera une analyse de code grâce à un IDE, par exemple IntelliJ.

Optimiser le code et résoudre les points de l'analyse

On se servira du rapport de l'analyse de code pour optimiser celui-ci et résoudre les possibles warnings.

Merge les branches de développement (develop par exemple) sur master

On vérifiera avoir merge toutes les branches de développement.

Vérifier que la branche master soit à jour

On regardera si la branche master a bien reçu toutes les modifications nécessaires au rendu.

Récupérer les sources depuis master

On récupérera les sources dans un nouveau répertoire afin de s'assurer d'avoir exactement ce qui va être rendu. Si possible sur une machine tiers.

Vérifier que le projet compile

On s'assurera que ce qu'on rend compile bien.

Vérifier que tous les tests qui doivent passer passent

On lancera tous les tests avec la commande mvn verify et on vérifiera que tous les tests qui sont censés passer passent.

Vérifier que common-test.sh passe

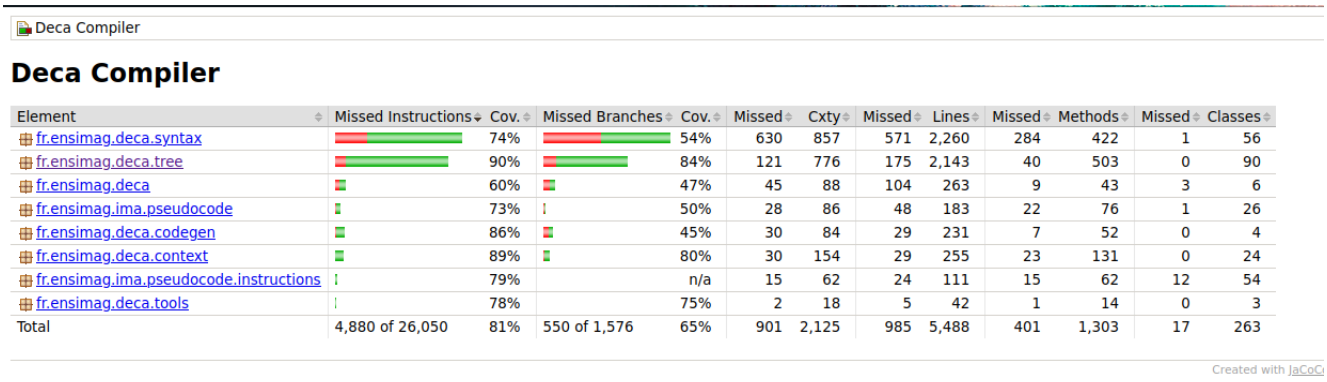
On s'assurera que common-test.sh passe bien.

Faire un mvn clean

On lancera cette commande pour nettoyer le projet.

4 Résultats Jacoco

Nous exposerons ici les résultats donnés par l'outil Jacoco, c'est-à-dire la couverture de nos tests. Voici la couverture globale de notre compilateur selon jacoco :



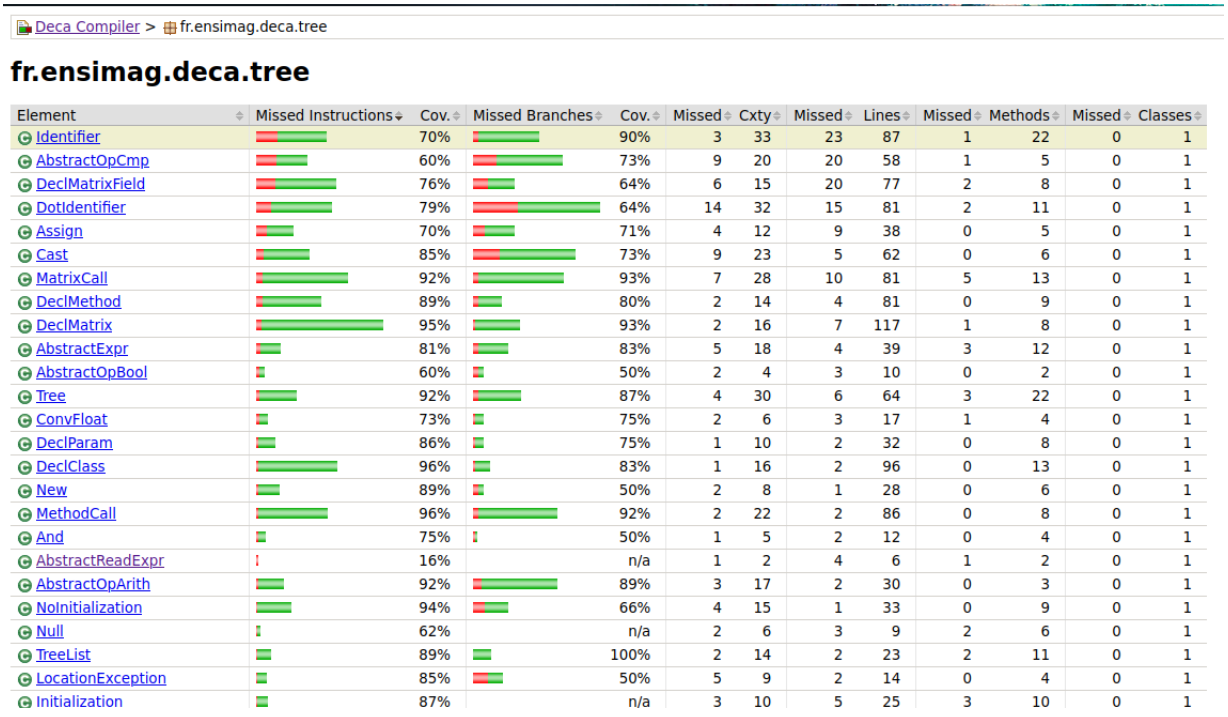
Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		74%		54%	630	857	571	2,260	284	422	1	56
fr.ensimag.deca.tree		90%		84%	121	776	175	2,143	40	503	0	90
fr.ensimag.deca		60%		47%	45	88	104	263	9	43	3	6
fr.ensimag.ima.pseudocode		73%		50%	28	86	48	183	22	76	1	26
fr.ensimag.deca.codegen		86%		45%	30	84	29	231	7	52	0	4
fr.ensimag.deca.context		89%		80%	30	154	29	255	23	131	0	24
fr.ensimag.ima.pseudocode.instructions		79%		n/a	15	62	24	111	15	62	12	54
fr.ensimag.deca.tools		78%		75%	2	18	5	42	1	14	0	3
Total	4,880 of 26,050	81%	550 of 1,576	65%	901	2,125	985	5,488	401	1,303	17	263

Created with JaCoCo

Et voici comment nous avons ajouté des tests à l'aide de jacoco. Prenons par exemple deca.tree :

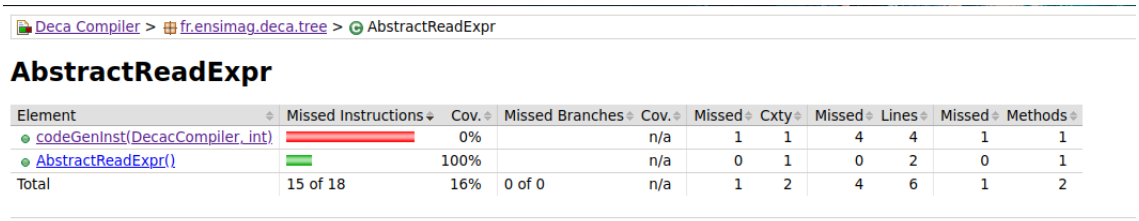


Deca Compiler > fr.ensimag.deca.tree

fr.ensimag.deca.tree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Identifier		70%		90%	3	33	23	87	1	22	0	1
AbstractOpCmp		60%		73%	9	20	20	58	1	5	0	1
DeclMatrixField		76%		64%	6	15	20	77	2	8	0	1
DotIdentifier		79%		64%	14	32	15	81	2	11	0	1
Assign		70%		71%	4	12	9	38	0	5	0	1
Cast		85%		73%	9	23	5	62	0	6	0	1
MatrixCall		92%		93%	7	28	10	81	5	13	0	1
DeclMethod		89%		80%	2	14	4	81	0	9	0	1
DeclMatrix		95%		93%	2	16	7	117	1	8	0	1
AbstractExpr		81%		83%	5	18	4	39	3	12	0	1
AbstractOpBool		60%		50%	2	4	3	10	0	2	0	1
Tree		92%		87%	4	30	6	64	3	22	0	1
ConvFloat		73%		75%	2	6	3	17	1	4	0	1
DeclParam		86%		75%	1	10	2	32	0	8	0	1
DeclClass		96%		83%	1	16	2	96	0	13	0	1
New		89%		50%	2	8	1	28	0	6	0	1
MethodCall		96%		92%	2	22	2	86	0	8	0	1
And		75%		50%	1	5	2	12	0	4	0	1
AbstractReadExpr		16%		n/a	1	2	4	6	1	2	0	1
AbstractOpArith		92%		89%	3	17	2	30	0	3	0	1
NoInitialization		94%		66%	4	15	1	33	0	9	0	1
Null		62%		n/a	2	6	3	9	2	6	0	1
TreeList		89%		100%	2	14	2	23	2	11	0	1
LocationException		85%		50%	5	9	2	14	0	4	0	1
Initialization		87%		n/a	3	10	5	25	3	10	0	1

On peut alors voir la couverture précise de chacune des classes d'arbre de notre compilateur. Prenons par exmple AbstractReadExpr :



Deca Compiler > fr.ensimag.deca.tree > AbstractReadExpr

AbstractReadExpr

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
codeGenInst(DecacCompiler, int)		0%		n/a	1	1	4	4	1	1
AbstractReadExpr()		100%		n/a	0	1	0	2	0	1
Total	15 of 18	16%	0 of 0	n/a	1	2	4	6	1	2

On remarque que dans codeGenInst, aucune des méthode n'est couverte, plus précisément, la seule méthode de la classe n'est pas appelée dans nos tests. Allons voir ce que c'est :

AbstractReadExpr.java

```
1.  package fr.ensimag.deca.tree;
2.
3.  import fr.ensimag.deca.DecacCompiler;
4.  import fr.ensimag.deca.codegen.AssError;
5.  import fr.ensimag.ima.pseudocode.Instruction;
6.  import fr.ensimag.ima.pseudocode.Register;
7.  import fr.ensimag.ima.pseudocode.instructions.LOAD;
8.
9.  /**
10.  * read...() statement.
11.  *
12.  * @author gl09
13.  * @date 01/01/2022
14.  */
15. public abstract class AbstractReadExpr extends AbstractExpr {
16.
17.     public AbstractReadExpr() {
18.         super();
19.     }
20.
21.     protected abstract Instruction mnemo();
22.
23.     @Override
24.     public void codeGenInst(DecacCompiler compiler, int n) {
25.         compiler.addInstruction(this.mnemo());
26.
27.         AssError.addErrorInput(compiler);
28.
29.         compiler.addInstruction(new LOAD(Register.R1, Register.getR(n)));
30.     }
31.
32.
33. }
```

On peut ainsi voir la totalité du fichier en question avec les parties appelées (et donc testées) et vert et les parties non appelées en rouge. Ici on comprend alors que pour résoudre le problème et tester cette méthode il faudrait tester readInt() et readFloat() dans la partie génération de code, c'est-à-dire utiliser decac sur des programmes contenant ces instructions. Néanmoins c'est ici volontaire, nous savons que ces tests manquent mais il n'est pas envisageable de les laisser dans notre compilateur. En effet, des tests en génération de code pour ces instructions interrompraient nos scripts de tests avec une demande de saisie.

Notre couverture s'élève donc à 81% au total. Il est tout de même important de noter que nous n'avons pas laissé certains tests dans notre compilateur, comme précisé plus haut pour les tests concernant readInt() et readFloat() et génération de code. Nous pouvons également citer les tests concernant l'option -P de notre compilateur. Cela demanderait une trop grande quantité de ressources pour stocker tous les programmes utilisés pour la réalisation de ces tests. De plus, ces programmes étaient conséquents afin de pouvoir observer une réelle différence entre la compilation en séquentiel (decac) ou en parallèle avec l'option -P. C'est pourquoi la mesure de couverture est en réalité légèrement plus haute que ce qui est annoncé par Jacoco.

Les résultats des tests de l'option -P sont dans docs/testOption-P, vous y trouverez les temps ainsi que l'activité des cœurs de la machine sur laquelle nous avons lancé les tests.

5 Autres méthodes de validation

Outre les tests unitaires et tests systèmes que nous avons réalisé pour s'assurer de la validation de notre compilateur, nous avons également eu recours à d'autres méthodes.

Premièrement, la personne réalisant les tests concernant une fonctionnalité n'était pas la personne ayant codé celle-ci. Cela permet de faire des tests plus objectifs, sans rapport avec l'implémentation réalisée et ainsi ne pas viser les aspects pour lesquels la personne ayant codé est sûre que cela fonctionne.

De plus, chaque membre de l'équipe travaillant globalement sur des choses différentes, il a été possible d'avoir un "candide" pour un grand nombre de parties. Cette personne étant extérieure au code à vérifier/relire à pu apporter un avis et une conception des choses différentes ce qui a pu nous faire remarquer des erreurs ou encore améliorer notre code.