# Rapport de Tests
## Projet GL

* Groupe g3 | Equipe gl16 *
Ange Romuald Ossohou
Nadir Ait Lahmouch
Hamza Benjelloun
Oussama Fennane
Younes Zaibila

30 janvier 2020

## Table des matières

**1 Descriptif des tests** **2**

**2 Types de tests pour chaque étape/passe** **2**
2.1 Tests .deca . . . . . 2
2.2 Tests unitaires . . . . . 2
2.3 Objectifs des tests, comment ces objectifs ont été atteints . . 2

**3 Les scripts de tests** **5**
3.1 Etape A (syntax) . . . . . 5
3.2 Etape B (contex) . . . . . 5
3.3 Etape C (codeGen) . . . . . 6
3.4 comment faire passer tous les tests . . . . . 6

**4 Gestion des risques et gestion des rendus** **7**

**5 Résultats de Cobertura** **7**

# 1 Descriptif des tests

Le rendu comporte

Des tests JUnit ajoutés aux répertoires :
- src/test/java/fr/ensimag/deca
- src/test/java/fr/ensimag/deca/context
- src/test/java/fr/ensimag/deca/syntax
- src/test/java/fr/ensimag/deca/tools
- src/test/java/fr/ensimag/deca/tree

Des tests deca ajoutés aux répertoires :
- src/test/deca/context/valid/added
- src/test/deca/context/invalid/added
- src/test/deca/syntax/valid/added
- src/test/deca/syntax/invalid/added
- src/test/deca/codegen/valid/added
- src/test/deca/codegen/invalid/

Des scripts :
- auto-cobetura.sh dans src/test/script
- auto-test.sh dans src/test/script/autotest
- decac.sh dans src/test/script/autotest

# 2 Types de tests pour chaque étape/passe

## 2.1 Tests .deca

Comme demandé on a bien ajouté des tests ".deca" simulant le comportement d'un utilisateur du compilateur. Ces tests sont disposés dans les répertoires en respectant la répartition demandée par les professeurs. Deux méthodes ont été utilisées dans la génération des codes. Une première, classique, et celle de les écrire manuellement pour valider ou invalider chaque règle présente dans le poly. Et une deuxième, automatique, en utilisant des scripts python. Ces codes ont permis de générer à la fois des tests valides et invalides pour les deux étape B et C. Ces derniers concernent les opérations arithmétiques et les comparaisons où les cas sont très nombreux pour les générer manuellement.

## 2.2 Tests unitaires

Des tests JUnit ont été ajoutés pour chaque classe existante voir 2.3 .

## 2.3 Objectifs des tests, comment ces objectifs ont été atteints

**Tests unitaires :** On a créé une classe de test par classe existante à tester. Dans chaque classe de test, il y a une méthode par méthode à tester.

Dans les tests JUnit on a pas besoin de savoir exactement ce que vaut les paramètres des sous-classes situés dans un autre package que celui où on est maintenant.
En effet, grace aux frameworks de Mock, on peut générer automatiquement des objets 'mockés' sans utiliser l'opérateur 'new' pour créer un objet d'une sous-classe qu'on veut pas tester, et ceci par exemple en utilisant la méthode Mockito.mock(nomDeLaClasse.class).
Par exemple pour tester la méthode figurant dans FloatType.java du package fr.ensimag.deca.context :

```
public boolean isFloat() {
        return true;
}
```

Ce qu'on doit vérifier est que la méthode renvoie true, pour ce, on crée une instance de la classe FloatType, le constructeur requiert une variable name de type SymbolTable.Symbol, pour la création d'une telle variable sans appeler l'opérateur 'new' on peut choisir d'appeler la méthode mock en écrivant à l'interieur de la fonction testIsFloat la ligne :

```
SymbolTable.Symbol name =
   Mockito.mock(SymbolTable.Symbol.class);
```

ou rajouter l'annotation @Mock au dessus de la variable globale name pour les instances de classes.

```
@Mock
SymbolTable.Symbol name;
```

Pour la méthode sameType de la meme classe FloatType

```
public boolean sameType(Type otherType) {
        return otherType.isFloat();
    }
```

on a besoin de vérifier que la méthode renvoie "true" si otherType.isFloat() et "false" sinon.
On peut donc prédire le résultat de la méthode sameType grace au « Stubbing » (ou Définition du comportement des objets mockés). On va indiquer à Mockito quelle valeur on souhaite retourner lorsque la méthode sameType() est invoquée, grace à la commande :

```
when(otherType.isFloat()).thenReturn(Boolean.TRUE);
```

On peut s'assurer enfin que pour une instance de la classe FloatType, le résultat de la méthode sameType appliquée sur cette instance est "true" grace à l'assertion :

```
boolean result = instance.sameType(otherType);
assertEquals(true, result);
```

Pour les méthodes :

- decompile
- iterChildren
- prettyPrintChildren

La couverture de ces méthodes est simple, il suffit de créer une instance de la classe qu'on veut tester et appliquer dessus une des trois méthodes ci dessus sans oublier de mocker les paramètres nécessaires pour la méthode.
Par exemple, pour la méthode decompile de la classe Selection :

```
public void decompile(IndentPrintStream s) {
        this.expr.decompile(s);
        s.print(".");
        this.ident.decompile(s);
}
```

On peut faire le test suivant :

```
public void testDecompile() {
        System.out.println("decompile");
        IndentPrintStream s =
           Mockito.mock(IndentPrintStream.class);
        Selection instance = new Selection(expr,
           ident);
        instance.decompile(s);
}
```

Mais on peut également choisir de tester ligne par ligne à l'intérieur de la méthode decompile en utilisant verify qui vérifie que la méthode "print" a été appelée sur l'objet s de type IndentPrintStream et que la méthode "decompile" a été appelée sur les attributs expr et ident de la classe Selection avec comme paramètre l'objet s de type IndentPrintStream.

```
public class SelectionTest {
    @Mock
    AbstractExpr expr;
    @Mock
    AbstractIdentifier ident;
    @Before
    public void setup() throws ContextualError {
        MockitoAnnotations.initMocks(this);
    }

    /**
     * Test of decompile method, of class Selection.
```

```
     */
    @Test
    public void testDecompile() {
        System.out.println("decompile");
        IndentPrintStream s =
           Mockito.mock(IndentPrintStream.class);
        Selection instance = new Selection(expr,
           ident);
        instance.decompile(s);
        verify(expr).decompile(s);
        verify(s).print(".");
        verify(ident).decompile(s);
    }
}
```

**Remarque :** Comme la première façon de tester la méthode decompile est plus rapide et suffisante pour couvrir la totalité de la méthode puisqu'il n'y a pas d'instructions if et donc on est sûr de tester toutes les méthodes à l'intérieur de la fonction decompile rien qu'en l'appliquant sur l'instance de la classe qu'on veut tester, alors c'est elle qu'on a choisi à la fin après avoir utilisé la deuxième façon pas mal de fois sur d'autres classes.

# 3 Les scripts de tests

## 3.1 Etape A (syntax)

Le script auto-test.sh permet d'automatiser l'exécution de **test_synt** sur l'ensemble des tests se trouvant dans src/test/deca/syntax/valid/added et src/test/deca/syntax/invalid/added, de générer un fichier **.lis** pour chaque fichier .deca considéré, contenant le résultat de l'exécution de l'exécutable et le stocker dans Projet_GL/src/test/Autotest/syntTest/valid et Projet_GL/src/test/Autotest/syntTest/invalid puis comparer ces résultats respectivement avec les résultats attendus des tests archivés à la base dans Projet_GL/src/test/correcttest/syntax/valid et Projet_GL/src/test/correcttest/syntax/invalid permettant donc en particulier de comparer systématiquement les résultats obtenus lorsque l'exécutable a été modifié. (**tests de non régression**)

## 3.2 Etape B (contex)

Le script auto-test.sh permet d'automatiser l'exécution de **test_context** sur l'ensemble des tests se trouvant dans src/test/deca/context/valid/added et src/test/deca/context/invalid/added, et de générer un fichier **.lis** interprétable par IMA pour chaque fichier .deca considéré, contenant le résultat de l'exécution de l'exécutable et le stocker dans

Projet_GL/src/test/Autotest/contextTest/valid et
Projet_GL/src/test/Autotest/contextTest/invalid puis comparer ces résultats respectivement avec les résultats attendus des tests archivés à la base dans Projet_GL/src/test/correcttest/contex/valid et
Projet_GL/src/test/correcttest/contex/invalid permettant donc en particulier de comparer systématiquement les résultats obtenus lorsque l'exécutable a été modifié.

### 3.3 Etape C (codeGen)

De même le script auto-test.sh permet d'automatiser l'exécution de **decac** sur l'ensemble des tests se trouvant dans src/test/deca/codegen/valid/added, et de générer un fichier **.ass** pour chaque fichier .deca considéré et le stocker dans Projet_GL/src/test/Autotest/codeGenTest/valid puis comparer ces résultats avec les résultats attendus des tests archivés à la base dans Projet_GL/src/test/correcttest/codeGen/valid permettant donc en particulier de comparer systématiquement les résultats obtenus lorsque l'exécutable a été modifié.

### 3.4 comment faire passer tous les tests

En plus, des tests évoqués ci dessus auto-test.sh permet également à la fin de tester les options en exécutant le fichier decac.sh se trouvant dans Projet_GL/src/test/script/autotest. L'idée vient du fait que le test unitaire DecacMainTest.java du package fr.ensimag.deca ne peut pas couvrir tout le code de DecacMain.java car on a le droit de créer qu'une seule instance de classe DecacMain, du coup ayant déjà testé le maximum d'options possible dans le test unitaire avec une seule instance DecacMain avec le code ci-dessous :

```
public class DecacMainTest {
    @Test
    public void testMainArgsp() throws Exception {
        System.out.println("main");
        String[] args = {"-p",
           "./src/test/deca/context/valid/added/and.deca",
           "-n", "-r ", "10",
           "./src/test/deca/context/valid/added/and.deca",
           "-b"};
        DecacMain instance = new DecacMain();
        instance.main(args);
    }
}
```

Il fallait également tester les cas où les options relèvent des exceptions d'où le fichier decac.sh qui teste toutes les combinaisons et permet de compléter

la couverture de la classe DecacMain.

Le fichier auto-cobetura.sh dans src/test/script permet d'executer tous les tests et inclure les tests unitaires et ouvrir le rapport de couverture à la fin en executant les commandes :

- **mvn clean** #Suppression des .class et des autres fichiers générés
- **mvn compile** #Compilation du programme
- **mvn test-compile** #Compilation des tests et du programme
- **mvn test** #Lancement des tests
- **mvn cobertura :clean** #Pour éviter de mélanger les résultats des tests avec des tests précédents
- **mvn cobertura :instrument** #Instrumenter les classes pour enregistrer la couverture.
- **mvn cobertura :cobertura** #Instrumenter les classes, puis lancer les tests automatiques, puis générer un rapport de couverture dans target/-site/cobertura/
- **./src/test/script/autotest/auto-test.sh**
- **cobertura-report.sh** #Générer un rapport de couverture
- **firefox target/site/cobertura/index.html &**

# 4 Gestion des risques et gestion des rendus

Une fois qu'on trouve une erreur dans une partie, on informe immédiatement les membres du groupe qui travaillent sur cette partie pour corriger le bug s'il s'agit d'un code très long.

# 5 Résultats de Cobertura

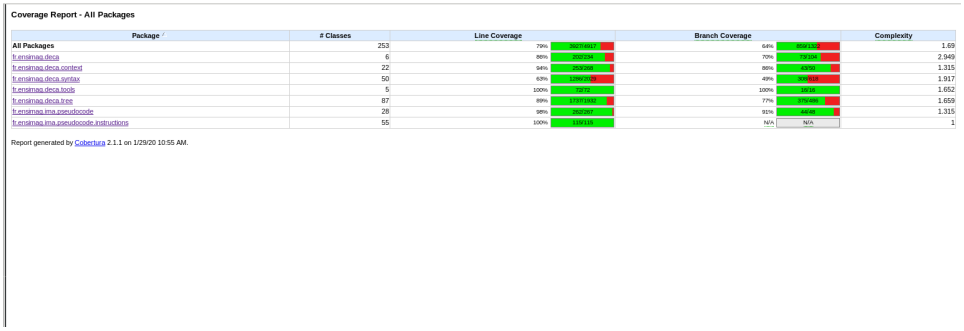


FIGURE 1 – Rapport de couverture

Grâce à la totalité des tests qu'on a présenté on a pu couvrir 80% de latotalié des codes du projet sur cobertura.