

Projet Génie Logiciel

Documentation : Validation

Gr2 - G110 - 2023 - 2024

Membres du groupe:

CHRIF M'HAMED Mohamedou

DIAB Dana

KONE Madou

MOHAMED AHMED Mohamed Lemine

Sommaire

1. Validation de l'étape A :	2
2. Validation de l'étape B :	5
3. Validation de l'étape C :	10
4. Jacoco :	11
5. Conclusion :	12

1. Validation de l'étape A :

Pendant cette phase, l'accent est mis sur la validation de l'analyseur lexical et syntaxique. Cela implique de vérifier si les mots existent déjà parmi les tokens que nous avons déjà implémentés, et d'examiner la syntaxe pour nous assurer que les phrases sont correctes, afin de fournir un arbre complet et fonctionnel à la partie B.

1.1 Tests Deca :

Dans cette étape, tout comme dans les autres sections, les tests ont été répartis en deux grandes catégories, à savoir "valide" et "invalid". Chacune de ces catégories est subdivisée en plusieurs sous-répertoires correspondant à chaque étape d'avancement du projet, tels que HelloWorld, le Sans Objet et le complet. Chacun de ces sous-répertoires contient des tests spécifiques liés à sa partie respective.

- **HelloWorld :**

Dans cette section, nous évaluons les fonctions liées aux chaînes de caractères. Ainsi, les tests invalides se concentrent sur les caractères ou les phrases non acceptés dans cette partie spécifique. Quant aux tests valides, nous avons cherché à explorer toutes les possibilités valides, comme les caractères “ protégés par \ et d'autres exemples.

- **Sans Objet :**

Nous avons testé ici pratiquement toutes les fonctions, à l'exception de celles liées aux classes et au casting. Cela s'applique aussi bien aux cas acceptés dans les tests valides qu'aux cas non acceptés dans les tests invalides.

- **Complet :**

Nous avons pratiquement essayé d'évaluer toutes les fonctions acceptées par notre langage, que ce soit celles concernant uniquement les chaînes de caractères, les classes ou autres. Nous avons adopté une approche optimiste tout en évitant la répétition autant que possible.

1.2 Exécution des tests :

Dans cette section, nous effectuons la vérification lexicale des tests en utilisant la commande **test_lex** sur chaque test. Si un mot inconnu est détecté, une erreur est générée. Pour la vérification syntaxique, nous utilisons la commande **test_synt** sur un test spécifique. En cas d'erreur, une autre erreur est signalée ; sinon, l'arbre correspondant à l'entrée de la partie B est généré.

1.3 Vérification des résultats :

Nous avons validé les résultats par plusieurs méthodes. Initialement, nous avons comparé les résultats des tests avec les attentes du sujet, mais cette approche s'est avérée insuffisante. Nous avons ensuite opté pour une vérification manuelle, anticipant le résultat correct avant de lancer le test et de le comparer. En outre, nous avons mis en œuvre la décompilation en exécutant la commande "decac -p" sur un test, comme P1, redirigeant le résultat vers un fichier appelé test.deca. Ensuite, nous avons relancé la décompilation sur ce fichier (test.deca), stockant le résultat dans test2.deca, et enfin utilisé la commande diff sur test.deca et test2.deca. En cas d'erreur à l'existence d'une différence, le message "Comparaison FAILED" est affiché, sinon, "Comparaison PASSED" est affiché.

1.4 Scripts Shell :

À mesure que nous progressions dans le projet et que le nombre de tests augmentait, ainsi que la nécessité de vérifier la validité des tests précédents, nous avons introduit plusieurs scripts shell pour simplifier le processus de lancement des tests et de vérification. Des scripts dédiés ont été créés pour chaque partie du projet. Voici la liste de ces scripts :

- test-HelloWorld-valide.sh
- test-HelloWorld-invalid.sh
- test-syntSansObjet-valide.sh
- test-syntSansObjet-valide.sh
- test-synt-objet-valide.sh
- test-synt-objet-invalid.sh
- Test-decompile.sh

Voici à titre d'exemple : celui du HelloWorld valide

```

1  #!/bin/sh
2  echo "===== valide : test HelloWorld ====="
3
4
5  test_synt_valide () {
6      # $1 = premier argument.
7      if test_synt "$1" 2>&1 | grep -q -e "$1:[0-9][0-9]*:"
8      then
9          echo "Echec inattendu pour test_synt sur $1."
10         exit 1
11     else
12         echo "Succes attendu de test_synt sur $1."
13     fi
14 }
15 }
16
17 for cas_de_test in src/test/deca/syntax/valid/HelloWorld/*.deca
18 do
19     test_synt_valide "$cas_de_test"
20 done

```

Et celui de la décompilation :

```

1  #!/bin/sh
2  echo "===== décompilation ====="
3
4
5  for cas_de_test in src/test/deca/syntax/valid/Objet/*.deca
6  do
7      decac -p "$cas_de_test" > test.deca
8      decac -p test.deca > test2.deca
9
10     if diff test.deca test2.deca >/dev/null; then
11         echo "Test case $cas_de_test: PASSED"
12     else
13         echo "Test case $cas_de_test: FAILED"
14     fi
15 done
16
17

```

Il est possible d'observer que nous parcourons l'ensemble des tests ciblés en lançant la commande correspondante. En ce qui concerne la décompilation, nous effectuons également la comparaison, affichant une confirmation de validation en cas de réussite et signalant les erreurs de manière appropriée pour chaque partie du projet.

2. Validation de l'étape B :

Pour l'étape B, nous nous concentrons particulièrement sur les tests de cette étape, car c'est une étape très importante pour la génération du code en C et pour le rejet du code non valide contextuellement. Nous effectuons de nombreux tests en deca environ 300 tests et des tests unitaires avec JUnit 5, et des tests d'intégration dont nous discuterons plus tard dans cette section, et des scripts shell pour automatiser les tests.

2.1 Tests Deca :

Les tests en Deca sont les plus importants pour valider l'étape B. Pour la conception des tests, nous utilisons la méthode d'examen systématique des erreurs présentées dans les vidéos de l'étape B afin de répertorier toutes les conditions à vérifier pendant l'étape B. Pour chaque condition, nous réalisons tous les tests possibles, valides et invalides.

Pour chaque test, nous ajoutons en commentaire une description du test, le résultat attendu, et un historique.

2.1.1 Tests invalides en deca:

Pour les tests invalide en deca on deux répertoires de tests :

- **sans-objet/ :**

Ce répertoire contient 61 tests invalides, et tous ces programmes doivent être rejetés par l'étape B en levant une exception ContextualError avec le format de message décrit dans le poly :

<nom fichier.deca>:<numero de ligne>:<numero du colonne>: description

Avec une description très précise de la cause de l'erreur [*ce qui est bien détaillé dans le Manuel-Utilisateur pour les messages d'erreurs*].

Par exemple, nous testons toutes les combinaisons possibles pour les opérations. À titre d'exemple pour l'opération +, dans le cahier des charges, nous avons que l'opération + est valide entre int et float ou float et int. Ainsi, dans les tests invalides, nous testons cette opération avec (int, boolean), (int, string), (string, string) ...etc



Donc on essaie de faire toutes les cas possibles non valide

- **complet/ :**

Ce répertoire contient 99 tests non validés pour tester la partie Objet de notre compilateur. Nous commençons tout d'abord par tester la déclaration des classes et l'héritage des classes, ce qui valide la passe 1.

Ensuite, nous commençons par des tests sur la déclaration des méthodes et des champs, ainsi que des redéfinitions invalides, comme la redéfinition d'une méthode avec un nombre de paramètres différent de celui de la méthode de la super classe. Ces tests valident bien la passe 2.

Ensuite, pour valider la passe 3, nous procédons de la même manière qu'avec le sans-objet. Pour chaque règle de la grammaire, nous essayons de faire passer tous les tests possibles invalides.

Il y a aussi de nombreux tests sur l'accès à des champs protégés, surtout pour valider que notre compilateur prend en charge les deux conditions pour accéder à un champ protégé.

2.1.2 tests valide en deca :

Pour les tests valide en deca on aussi deux répertoires :

- **sans_objet/**

Dans ce répertoire, il y a 40 tests valides pour tester toutes les fonctionnalités qui devront être valides dans l'étape B. Le plus important dans ces tests est les tests sur les opérations, que ce soit arithmétique, de comparaison, des opérations logiques, et aussi

des tests sur l'enrichissement de l'arbre avec Convfloat, et des autres tests sur if, while et print .

- **complet/ :**

Dans ce répertoire, il y a 88 tests pour valider la passe 1, passe 2 et passe 3 de notre compilateur.

- Dans la passe 1 : les tests se concentrent sur la déclaration des classes et sur l'héritage des classes valides.

- Dans la passe 2 : nous testons les déclarations des méthodes et des champs, ainsi que les redéfinitions des méthodes avec des types de retour qui sont des sous-types des types de retour des méthodes de la super classe. Nous effectuons également des tests sur la méthode equals de la classe Object.

- Dans la passe 3 : le plus important de ces tests se trouve dans cette partie. Nous avons fait beaucoup de tests pour vérifier toute la décoration de l'arbre dans cette passe. Il s'agit donc de tests sur les appels de méthodes, les accès valides aux champs protégés, et des tests plus complexes comme une hiérarchie des classes complexe.

2.2. Script Shell :

Pour automatiser les choses on a écrit des shell script pour lancer les toutes les tests et on a utiliser dans ce script **test_context** qui est fourni

Pour les tests de l'étape B on 4 script shell :

- **test-context-sans-objet-valide.sh**
- **test-context-sans-objet-invalid.sh**
- **test-context-complet-invalid.sh**
- **test-context-complet-valide.sh**

Par exemple :


```

1 echo "===== invalide : test-context objet ====="
2
3 test_context_invalide() {
4     # 2>&1
5     if test_context "$1" 2>&1 | grep -q -e "$1:[0-9][0-9]*:[0-9][0-9]*"
6     then
7         echo "Echec attendu pour test_context sur $1"
8     else
9         echo "Succes inattendu de test_context $1"
10        exit 1
11    fi
12 }
13
14
15 for cas_de_test in ./src/test/deca/context/invalid/complet/*.deca
16 do
17     test_context_invalide "$cas_de_test"
18
19 done

```

Pour chacun des tests, le script parcourt le dossier spécifique qui lui est associé et lance **test-context** sur tous les fichiers. Pour les tests invalides, si il capte un message d'erreur (avec **grep**) de format spécifié dans la section précédente, il affiche **Échec attendu** et continue à traiter les autres fichiers. Si aucun message d'erreur n'est capturé, il affiche **Succès inattendu** et arrête l'exécution. Pour les tests valides, le comportement est inverse.

Pour automatiser davantage les tâches, nous insérons ces scripts comme des plugins dans le fichier **pom.xml**, afin que **Maven** puisse traiter directement ces tests.

2.3. tests avec JUnit:

Lors de la phase de développement, nous écrivons des méthodes et des fonctions utilitaires qui nous aident à simplifier certaines fonctionnalités dans notre compilateur. Cependant, pour être certain que ces méthodes ou fonctions fonctionnent correctement, il faut les tester. C'est ici que le framework de test JUnit intervient. Durant notre phase de tests, nous avons largement utilisé JUnit pour nous assurer que ces fonctions utilitaires fonctionnent correctement.

Dans le répertoire **src/test/java/context/**, on trouve l'ensemble des tests unitaires en JUnit que nous avons réalisés.

Nous avons utilisé ces tests pour évaluer le package **util** qui contient les fonctions utilitaires de l'étape B.

Par exemple, nous pouvons mentionner le fichier **ClassTypeTest.java**, qui contient des tests sur la méthode **isSubClassOf**.

```

1  @Test
2  public void testMethodeSuClass()
3  {
4      DecacCompiler compiler = new DecacCompiler(null, null);
5      Symbol ASymb = compiler.createSymbol("A");
6      Symbol BSymb = compiler.createSymbol("B");
7
8
9
10     ClassType A = new ClassType(ASymb, Location.BUILTIN, compiler.environmentType.Object.getDefinition());
11     ClassType B = new ClassType(BSymb, Location.BUILTIN, A.getDefinition());
12
13     assertTrue(A.isSubClassOf(A));
14     assertTrue(A.isSubClassOf(compiler.environmentType.Object));
15     assertTrue(A.getDefinition().getSuperClass().getType().getName().equals(compiler.environmentType.Object.getName()));
16     assertFalse(A.isSubClassOf(B));
17     // assertFalse(A.isSubClassOf(nullType));
18
19     assertTrue(B.isSubClassOf(A));
20     assertTrue(B.isSubClassOf(compiler.environmentType.Object));
21     assertTrue(B.isSubClassOf(B));
22
23 }

```

2.4. Tests d'intégration :

Pour l'étape B, nous exécutons un test d'intégration qui vérifie si les indexes sont bien gérés dans l'étape B.

Afin de tester cette fonctionnalité, nous avons développé un fichier Java dans le répertoire : **src/test/java**, portant le nom de **TestIndex.java**. Ce fichier effectue tout d'abord l'étape A, puis l'étape B, et ensuite, pour chaque classe, il affiche les index des champs et des méthodes.

Pour lancer ce test, nous avons également développé un script shell dans le répertoire : **src/test/script**, nommé **test_index.sh**. Ce script prend en argument un fichier Deca et lance le fichier Java préalablement mentionné.

```

1  public class TestIndex {
2
3      public static void main(String[] args) throws IOException {
4          Logger.getRootLogger().setLevel(Level.DEBUG);
5          DecaLexer lex = AbstractDecaLexer.createLexerFromArgs(args);
6          CommonTokenStream tokens = new CommonTokenStream(lex);
7          DecaParser parser = new DecaParser(tokens);
8          DecacCompiler compiler = new DecacCompiler(new CompilerOptions(), null);
9          parser.setDecacCompiler(compiler);
10         AbstractProgram prog = parser.parseProgramAndManageErrors(System.err);
11         if (prog == null) {
12             System.exit(1);
13             return; // Unreachable, but silents a warning.
14         }
15         try {
16             prog.verifyProgram(compiler);
17         } catch (LocationException e) {
18             e.display(System.err);
19             System.exit(1);
20         }
21
22         prog.displayIndex(compiler);
23     }
24 }
25
26

```

3. Validation de l'étape C :

Pendant cette phase, l'accent est mis sur la validation de la génération de code. L'utilisation de la machine abstraite IMA nous a également aidé à vérifier la validité des fichiers assembleurs. Cependant, l'étape C représente dans notre projet une base de tests moins importante en nombre que les autres étapes.

Tout comme pour les deux autres étapes, les tests Deca sont répartis dans deux répertoires "valid/" et "invalid/". De plus, pour cette étape, nous avons un répertoire "interactive/" qui contient les tests (avec readFloat() et readInt()) nécessitant une entrée de l'utilisateur.

1. valid/ :

Ce sous-répertoire contient deux répertoires : "SansObjet/" et "Complet/". Le répertoire "SansObjet/" contient des tests Deca pour les sous-langages HelloWorld et SansObjet à la fois. Pour le répertoire "SansObjet/", nous avons essayé de tester le maximum d'instructions possible, pensant avoir une bonne base de tests pour le langage SansObjet, avec au moins un test par instruction. Les résultats attendus semblent corrects selon IMA.

Quant au langage Complet, nous avons créé les tests en parallèle avec le développement du compilateur. Étant donné que le développement de toutes les instructions demandées n'est pas terminé, certains tests ne compilent pas avec decac.

Nous les avons donc placés dans "Complet/nopass/". La base de tests pour le langage complet est encore incomplète et moins rigoureuse.

2. invalid/ :

Ce répertoire contient un seul sous-répertoire, "SansObjet/", qui regroupe des tests provoquant des erreurs telles que la division par zéro, le reste de la division entière par zéro, etc. Malheureusement, en raison de contraintes de temps, nous n'avons pas pu développer de tests invalides pour le langage Complet.

3. intervatice/:

Ce répertoire contient deux tests distincts, chacun testant readFloat et readInt() séparément. Il inclut également un jeu, "plusOuMoins", où l'utilisateur doit saisir des entiers et tenter de deviner le nombre mystère fixé dans le jeu.

Dans le but d'automatiser les tests le plus possible, nous avons mis en place deux scripts shell, "decac-valide-sans-objet.sh" et "decac-valid-objet.sh", qui testent les tests Deca valides. Ces deux scripts se ressemblent : ils parcourent les fichiers, les compilent avec decac en affichant le succès attendu ou l'échec inattendu. Ensuite, IMA est lancé sur les fichiers assembleur générés et affiche également s'il y a un échec inattendu ou non. Cependant, nous ne vérifions pas l'exactitude des résultats avec IMA, mais seulement si l'exécution est réussie ou si une erreur est signalée. Enfin, nous compilons une fois de plus les fichiers, cette fois-ci avec l'option -v qui sert à vérifier contextuellement les fichiers.

Ces deux scripts ont été ajoutés au pom.xml.

La base de tests pour l'étape C et leur automatisation peuvent être améliorées significativement, même pour les instructions non encore implémentées. Cela représente la seule faiblesse de la base de tests de notre projet.

4. Jacoco :

Les résultats fournis par JaCoCo nous semblent satisfaisants, avec une couverture de 80% sur l'ensemble des instructions Java de notre projet. La base de tests dans son ensemble est assez robuste, même si elle présente quelques faiblesses dans certains endroits.

Deca Compiler



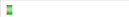















Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.ima.pseudocode.instructions		75%		n/a	17	62	28	111	17	62	13	54
fr.ensimag.ima.pseudocode		75%		75%	28	84	45	180	23	74	2	26
fr.ensimag.deca.util		98%		94%	7	64	2	52	1	8	0	1
fr.ensimag.deca.tree		92%		87%	91	690	140	1,876	50	512	3	86
fr.ensimag.deca.tools		98%		100%	1	21	1	46	1	17	0	3
fr.ensimag.deca.syntax		75%		57%	469	673	492	2,005	248	369	2	48
fr.ensimag.deca.context		80%		66%	30	156	57	304	20	132	0	21
fr.ensimag.deca.codegen		85%		90%	5	23	7	49	4	18	0	1
fr.ensimag.deca		52%		51%	43	81	124	278	8	35	2	5
Total	4,165 of 21,614	80%	352 of 1,194	70%	691	1,854	896	4,901	372	1,227	22	245

Figure 1 : Résultats de JaCoCo

5. Conclusion :

Les bases de tests sont d'une importance cruciale dans le développement logiciel. Elles constituent un outil essentiel pour évaluer la robustesse, la fiabilité et la conformité d'une application. Désormais, nous avons pris conscience de l'importance de l'automatisation des tests, qui facilite les processus de validation et de vérification, accélérant ainsi le cycle de développement et réduisant les risques d'erreurs dans le compilateur final. Une base de tests solide nous a montré comment garantir le succès et la pérennité de notre compilateur.