Documentation de validation

Samy Ahjaou Hamza Bouihi Omar Guessous
Grégoire Sename Mamadou Thiongane

Groupe 9 – Équipe 42

26 janvier 2024

Table des matières

Dans le cadre du développement de notre projet, l'aspect des tests revêt une importance cruciale, non seulement pour garantir la qualité et la fiabilité de notre travail, mais aussi parce qu'ils constituent une part significative de notre note globale. Les tests, souvent perçus comme une étape technique et rigoureuse, sont en réalité une composante essentielle de tout processus de développement logiciel. Ils permettent de vérifier la conformité du produit aux exigences initiales, d'assurer sa stabilité et sa performance, et de détecter les erreurs et les points faibles avant la mise en production.

Ce document, élaboré collectivement par toute l'equipe vise à présenter de manière détaillée et structurée notre approche en matière de tests. Nous y abordons les différents types de tests mis en œuvre.Chaque section, confiée à des membres spécifiques de l'équipe, traite d'un aspect distinct du processus de test, reflétant ainsi notre organisation et notre collaboration. Les scripts de tests illustrent concrètement comment nous avons procédé pour tester les différents composants de notre projet. La gestion des risques et des rendus, montre notre capacité à anticiper et à gérer les problèmes potentiels. Les résultats obtenus grâce à l'outil Jacoco fournissent une mesure objective de la couverture de nos tests, tandis que les méthodes de validation alternatives démontrent notre volonté d'adopter une approche holistique en matière de qualité logicielle.

Lors de la réalisation de l'étape C, moment où nous avons commencé à avoir une vue globale du compilateur, nous avons intégré des tests interactifs. Ces tests étaient particulièrement pertinents car ils nous ont permis de simuler l'utilisation finale du compilateur par les utilisateurs. Cette approche interactive a été cruciale pour évaluer l'expérience utilisateur et pour détecter des problèmes qui ne sont pas toujours évidents lors de tests plus automatisés.

1 Descriptif des tests

1.1 Type de tests effectués

Dans le cadre du développement de notre compilateur, nous avons mis en œuvre une stratégie de test exhaustive, adaptée à chaque étape du projet pour assurer une qualité et une fiabilité optimales du produit final. Cette stratégie comprenait différents types de tests, chacun ciblant des aspects spécifiques du compilateur.

Au début, nous avons effectué des tests unitaires surtout pour la partie B d'analyse contextuelle où on teste chaque erreur contextuelle que nous avons implémenté, ce qui nous a permis de vérifier le bon fonctionnement de chaque module indépendamment. Ces tests étaient essentiels pour détecter les erreurs précocement dans le processus de développement et assurer que chaque composant individuel fonctionnait comme prévu.

À mesure que les modules étaient développés et que leur interaction devenait plus complexe, nous avons procédé à des tests d'intégration. Ces tests étaient cruciaux pour s'assurer que les différents modules interagissent correctement entre eux, formant un ensemble cohérent et fonctionnel. Lors de la réalisation de l'étape C, moment où nous avons commencé à avoir une vue globale du compilateur, nous avons intégré des tests interactifs. Ces tests étaient particulièrement pertinents car ils nous ont permis de simuler l'utilisation finale du compilateur par les utilisateurs. Cette approche interactive a été cruciale pour évaluer l'expérience utilisateur et pour détecter des problèmes qui ne sont pas toujours évidents lors de tests plus automatisés.

1.2 Organisation des tests

En suivant l'architecture initialement définie par les enseignants pour notre projet, nous avons adopté une approche progressive et structurée pour l'intégration des tests dans notre environnement de développement. Les tests ont été organisés dans un répertoire spécifique, nommé src/test/deca, qui était subdivisé en cinq dossiers distincts, correspondant à différentes composantes du compilateur : lex, syntax, context, codegen, et decac. Chacun de ces répertoires comprenait à son tour deux sous-dossiers, valid et invalid, destinés à accueillir des cas de test spécifiques. Cette structure reflétait étroitement les différentes étapes de notre projet, permettant une validation ciblée et efficace à chaque phase du développement. Par exemple, durant l'étape A, nous concentrions nos efforts de test sur les répertoires lex et syntax. Dans le dossier lex, nous ajoutions des tests pour vérifier le bon fonctionnement du lexer, c'est-à-dire la capacité du compilateur à lire et à analyser correctement les tokens du langage. Parallèlement, le dossier syntax était utilisé pour tester le parser, assurant ainsi que la structure syntaxique des programmes était correctement interprétée.

Lors de l'étape B, notre attention se portait sur le répertoire context. Ici, les tests étaient axés sur la vérification des aspects liés au contexte, comme l'analyse des portées, la résolution des noms et la vérification des types. Cette étape était cruciale pour s'assurer que le compilateur decorait correctement l'arbre produit par l'etape A.

Enfin, pour l'étape C, nous utilisions principalement le répertoire codegen. Les tests de cette phase étaient centrés sur la génération de code, évaluant la capacité du compilateur à produire un code exécutable correct et efficace à partir du code source. Cette étape était essentielle pour garantir que le compilateur pouvait transformer le code source en un programme fonctionnel. Pour la partie Orientee objet, nous avons ajoute parfois des repertoirees a l interieur des repertoires valid invalid juste pour que ca soit plus specifique, par exemple pour context, on a ajoute obj wobj pour differencier en tre sans objet et objet puis plusieurs fichiers comme DeclField pour tester la declaration des fields a l interieur d une classe, declParam pour verifier les parametres d une methode a l interieur d une classe, declMethode pour la declaration d une methode...

1.3 Descriptif des tests

Nos scripts de test suivent une architecture cohérente et méticuleuse, facilitant l'évaluation précise de divers aspects du compilateur. Le script test_init_subtype.deca illustre parfaitement cette méthodologie comme test valid de la partie orientee objet . Voici sa structure détaillée :

En-Tête de Documentation :

- Description : Une brève explication du but du test. Pour ce script, il est mentionné que le test vérifie une initialisation compatible, étant donné que A est un sous-type de B et pas l'inverse.
- Résultats Attendus : La documentation précise que le test doit réussir l'initialisation et la sous-typage.
- Historique : Cette section indique la date de création du test, renforçant la traçabilité.

test_init_subtype.deca 337 Bytes

```
// Description:
//    initialisation compatible car A soustype de B et pas l'inverse
//
// Resultats:
//  soustypage et initialisation reussis
//
// Historique:
//    cree le 01/01/2024



class A {
    int x;
}
class B extends A{
    boolean x;
    void test(A a){

    //A a ;
    a.x = 1;
    }
}
class C {
    B b;
    A a = b ;


}
```

La structure rigoureuse de nos tests est constante, même lorsqu'il s'agit de vérifier des cas d'erreur. Prenons pour exemple un test spécialement conçu pour identifier les non-conformités dans la partie B du projet, où l'objectif était de tester le comportement du compilateur en l'absence d'objets. Bien que nous conservions la même en-tête standard pour maintenir une uniformité documentaire, la section des résultats se distingue en cela qu'elle fournit des informations détaillées sur la nature et l'emplacement de l'erreur attendue.

Dans ce test invalide, l'en-tête de documentation reste inchangée pour préserver la clarté et la cohérence, permettant à quiconque examine le test de comprendre rapidement son but et son contexte. La section Résultats est particulièrement cruciale ici, car elle ne se contente pas seulement d'indiquer l'échec anticipé du test, mais elle spécifie également la ligne précise du code source où l'erreur est censée se manifester. Cela est essentiel pour le débogage et l'optimisation du compilateur, car cela aide les développeurs à localiser immédiatement la source de l'erreur.

test_affect_incompatible_float.deca 211 Bytes

```
1  // Description:
2  //    Affectation et typage
3  //
4  // Resultats:
5  //    Ligne 15: Affectation de types incompatibles.
6  //
7  // Historique:
8  //    cree le 01/01/2024
9
10 {
11    float b = 1.0;
12    boolean a = true;
13    b = a;
14 }
```

2 Scripts de tests

Pour faire passer notre base de tests, nous avons implémenté différents scripts shell, dans le répertoire `src/test/script`, pour chaque étape de développement du compilateur. L'ensemble de ces scripts de tests ayant été intégrés au fichier `pom.xml`, la commande `mvn test` suffit pour lancer l'ensemble des tests.

Pour l'analyse lexicographique, le script `test-lex.sh` teste d'abord l'ensemble des fichiers Deca lexicalement corrects, situés dans le répertoire `src/test/deca/lex/valid`, en lançant le programme `test_lex` sur chaque fichier. Lorsque l'exécution de `test_lex` sur un fichier lexicalement correct renvoie un message d'erreur, le script affiche

`[FAILED] Erreur inattendue: text_lex <fichier deca>`

et renvoie 1, sinon, le script affiche simplement

`[PASSED] Analyse reussie: test_lex <fichier deca>`

Ces messages permettent ainsi de trouver facilement le fichier mal répertorié ou la fonctionnalité à corriger. Le script teste ensuite l'ensemble des fichiers Deca lexicalement incorrects, situés dans le répertoire `src/test/deca/lex/invalid`. Le script affiche

`[PASSED] Erreur bien detectee: text_lex <fichier deca>`

pour une exécution de `test_lex` renvoyant bien un message d'erreur et

`[FAILED] Erreur non detectee: test_lex <fichier deca>`

ainsi que le code de retour 1 dans le cas contraire.

De la même manière, pour l'analyse syntaxique, le script `test-synt.sh` teste l'ensemble des fichiers Deca syntaxiquement corrects et incorrects, respectivement situés dans les répertoires `src/test/deca/syntax/valid` et `src/test/deca/syntax/invalid`, en lançant le programme `test_synt`.

Pour l'analyse contextuelle, le script `test-context.sh` teste l'ensemble des fichiers Deca corrects et incorrects, respectivement situés dans les répertoires `src/test/deca/context/valid` et `src/test/deca/context/invalid`, en lançant le programme `test_context`.

Pour la génération de code, le script `test-gencode.sh` teste l'ensemble des fichiers Deca des répertoires `src/test/deca/gencode/valid` et `src/test/deca/gencode/invalid`. Le script

lance la commande `decac` sur chaque fichier et renvoie une erreur lorsque la compilation a échoué ou lorsque le fichier .ass n'a pas été généré. Sinon, le script exécute le fichier .ass généré avec la commande `ima` et affiche un message d'erreur lorsqu'un fichier valide renvoie une erreur ou lorsque l'exécution d'un fichier invalide ne renvoie pas d'erreur.

Le script `test-decac.sh` teste les différentes fonctionnalités et options du compilateur et affiche un message d'erreur ainsi que le code de retour 1 dans les cas suivants :

- Le non-respect de la syntaxe de la commande `decac` ne renvoie pas d'erreur
- Le passage en paramètre d'un fichier dont l'extension est différente de .deca ne renvoie pas d'erreur
- L'échec de la compilation ne renvoie pas d'erreur
- Aucun fichier .ass n'est généré
- `decac -b` n'affiche pas de bannière
- `decac` n'affiche pas les options disponibles
- `decac -p` renvoie une erreur pour les fichiers Deca syntaxiquement corrects
- La sortie de `decac -p` est un programme syntaxiquement incorrect ou la decompilation n'est pas idempotente pour les fichiers Deca syntaxiquement corrects
- `decac -v` affiche une sortie pour les fichiers Deca valides
- `decac -v` ne produit pas de message d'erreur pour les fichiers Deca invalides
- `decac -n` ne supprime pas les tests à l'exécution
- `decac -r` utilise un registre indisponible
- `decac -d` n'affiche pas les traces de debug
- Pour un ensemble de fichiers Deca donné, `decac -P` produit des fichiers .ass différents que ceux générés par une compilation séquentielle

3 Gestion des risques et gestion des rendus

3.1 Gestion des risques

Le risque principal est d'avoir une branche `master` non fonctionnelle, nous avons donc mis en place des actions pour éviter cela.

La première est de ne pas travailler directement sur la branche `master`, et de ne pas *push* directement dessus sans vérification. Nous avons donc réfléchi la mise en place d'un *pipeline* pour bloquer un *commit* qui ne passerait pas les tests. Cette manière automatique ne remplace pas notre besoin de lancer les tests manuellement lors du développement, mais permet d'éviter un risque humain (un membre qui *push* son travail sur la branche `master` en ayant oublié de lancer les tests avant).

Nous avons donc commencé à écrire une base de fichiers Deca valides et invalides couvrant un maximum de possibilités. Nous avons également réfléchi à l'écriture de scripts qui exécutent les programmes `test_lex`, `test_synt`, `test_context` et `decac` sur l'ensemble des fichiers et vérifient la validité des fichiers. Le pipeline décrit ci-dessus exécutera ces scripts et bloquera le *commit* en cas d'erreur ou de succès inattendu.

D'autres risques ont également été remarqués. En effet, un problème technique pour déposer le rendu final en raison de la congestion des serveurs. Nous mettons ainsi régulièrement à jour la branche `master` pour éviter de tout envoyer au dernier moment.

Un membre du groupe peut tomber malade et ne peux pas travailler. C'est pourquoi personne ne travaille en totale autonomie afin que quelqu'un puisse poursuivre la suite du travail et on communique très régulièrement sur l'avancement des tâches.

3.2 Gestion des rendus

Une action mise en place pour la gestion des rendus est la création d'un Trello. Il s'agit d'un outil permettant la gestion de tâches et de nos sprints. Nous avons donc créé des tâches pour tous les rendus, avec leurs *deadlines*, et cette application envoie des notifications à l'approche de celles-ci.

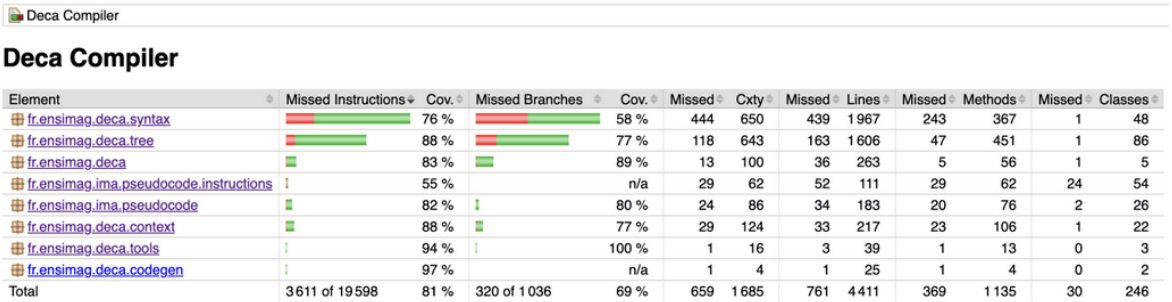
Nous avons également organisé les différentes tâches dans le diagramme de Gantt afin d'avoir une vision plus temporelle de ces tâches.

4 Couverture des tests

Jacoco est un outil de mesure de couverture de code intégré à l'écosystème Java, qui joue un rôle essentiel dans l'assurance qualité du logiciel. Il fournit une analyse quantitative de la couverture de tests, c'est-à-dire qu'il indique quel pourcentage du code source est réellement testé par les tests unitaires et d'intégration. Cela permet aux développeurs de visualiser les parties du code qui n'ont pas été exercées pendant les tests, soulignant ainsi les zones potentiellement vulnérables aux défauts.

L'outil fonctionne en instrumentant le bytecode de l'application au moment de son exécution de test, enregistrant ainsi quelles lignes de code ont été exécutées et lesquelles ne l'ont pas été. Les résultats sont ensuite compilés dans des rapports lisibles et détaillés, fournissant des métriques telles que le pourcentage de branches, de lignes et d'instructions couvertes. Ces rapports peuvent être générés en différents formats, y compris HTML pour une visualisation facile et XML pour l'intégration avec d'autres outils ou services.

Voici les resultats que nous avons eu pour notre compilateur :


Deca Compiler

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		76 %		58 %	444	650	439	1 967	243	367	1	48
fr.ensimag.deca.tree		88 %		77 %	118	643	163	1 606	47	451	1	86
fr.ensimag.deca		83 %		89 %	13	100	36	263	5	56	1	5
fr.ensimag.ima.pseudocode.instructions		55 %		n/a	29	62	52	111	29	62	24	54
fr.ensimag.ima.pseudocode		82 %		80 %	24	86	34	183	20	76	2	26
fr.ensimag.deca.context		88 %		77 %	29	124	33	217	23	106	1	22
fr.ensimag.deca.tools		94 %		100 %	1	16	3	39	1	13	0	3
fr.ensimag.deca.codegen		97 %		n/a	1	4	1	25	1	4	0	2
Total	3 611 of 19 598	81 %	320 of 1 036	69 %	659	1 685	761	4 411	369	1 135	30	246

Le rapport de couverture généré par Jacoco pour le projet "Deca Compiler" illustre une surveillance approfondie de l'exhaustivité des tests sur l'ensemble du projet. Il permet d'évaluer la qualité des tests réalisés et identifie les domaines qui nécessitent une attention supplémentaire. Voici une analyse détaillée des résultats :

- Package fr.ensimag.deca.syntax : Avec une couverture d'instructions de 76 %, ce package, qui gère vraisemblablement l'analyse syntaxique, montre que la majorité du code a été exécutée lors des tests. Cependant, la couverture des branches à 58 % suggère que certains

cas conditionnels n'ont pas été entièrement explorés. Cela pourrait indiquer le besoin d'ajouter des tests pour couvrir les scénarios de décision qui sont actuellement négligés.

- Package fr.ensimag.deca.tree : Ce package affiche une excellente couverture avec 88% des instructions testées et 77% des branches, impliquant une attention particulière aux structures de données de l'arbre, qui sont essentielles pour la représentation des constructions du langage de programmation.
- Package fr.ensimag.deca : Il présente une couverture solide, avec 83% pour les instructions et 89% pour les branches. Ces chiffres élevés témoignent d'une suite de tests bien conçue qui aborde efficacement la majorité des cas de figure.
- Package fr.ensimag.ima.pseudocode.instructions : Avec seulement 55% de couverture d'instructions, ce package spécifique aux instructions de pseudocode pourrait bénéficier d'une augmentation des tests, surtout qu'il n'y a pas de données disponibles pour la couverture des branches.
- Package fr.ensimag.deca.context : Ce package a une couverture d'instructions de 88% et de branches de 77%, ce qui est assez robuste, mais laisse encore une marge pour améliorer les tests de certains chemins de code critiques liés au contexte des programmes.
- Package fr.ensimag.deca.tools : Avec une couverture de 94%, cet ensemble d'outils semble être parmi les mieux testés du projet, avec presque tous les cas d'utilisation probablement couverts.
- Package fr.ensimag.deca.codegen : Ce package a une couverture impressionnante de 97% des instructions, indiquant que la génération de code est soigneusement testée. Cependant, l'absence de données sur la couverture des branches nécessite une attention particulière pour s'assurer que tous les chemins conditionnels sont vérifiés.

5 Méthodes de validation utilisées autres que le test

Les tests ne constituaient pas notre unique méthode pour vérifier l'exactitude de notre implémentation. Nous avons fréquemment utilisé un polycopié contenant de nombreux exemples illustrant différentes étapes du processus, accompagnés de leurs corrections. Ces exemples se sont révélés extrêmement utiles pour confirmer le fonctionnement adéquat de notre compilateur. Par exemple, l'exemple détaillé aux pages 206, 207 et 208, présentant un arbre syntaxique entièrement décoré, a été crucial pour valider à la fois la justesse de notre analyse syntaxique et l'adéquation de la décoration de l'arbre.

```
class A {
   protected int x ;
   int getX() {
      return x ;
   }
   void setX(int x) {
      this.x = x ;
   }
}

{
   A a = new A() ;
   a.setX(1) ;
   println("a.getX() = ", a.getX()) ;
}
```

Arbre décoré

```
`> [1, 0] Program
   +> ListDeclClass [List with 1 elements]
   |  []> [1, 0] DeclClass
   |      +> [1, 6] Identifier (A)
   |      |  definition: type defined at [1, 0], type=A
   |      +> [builtin] Identifier (Object)
   |      |  definition: type (builtin), type=Object
   |      +> ListDeclField [List with 1 elements]
   |      |  []> [2, 17] [visibility=PROTECTED] DeclField
   |      |      +> [2, 13] Identifier (int)
   |      |      |  definition: type (builtin), type=int
   |      |      +> [2, 17] Identifier (x)
   |      |      |  definition: field defined at [2, 17], type=int
   |      |      `> NoInitialization
   |      `> ListDeclMethod [List with 2 elements]
   |         []> [3, 3] DeclMethod
   |         ||  +> [3, 3] Identifier (int)
   |         ||  |  definition: type (builtin), type=int
   |         ||  +> [3, 7] Identifier (getX)
   |         ||  |  definition: method defined at [3, 3], type=int
   |         ||  +> ListDeclParam [List with 0 elements]
   |         ||  `> [3, 14] MethodBody
   |         ||     +> ListDeclVar [List with 0 elements]
```

En outre, un autre exemple, situé dans les pages 212 et 213 du polycopié, a été spécifiquement utile pour vérifier que la partie B du compilateur attribue correctement l'indice de la méthode et du champ aux emplacements appropriés. Ce genre d'exemples pratiques, intégrés dans notre matériel de référence, a joué un rôle essentiel dans le processus de validation et d'assurance de la qualité de notre compilateur.

```
class A {
   void m() { }
   void p() { }
}

class B extends A {
   void p() { }
   void q() { }
}
```