MINISTÈRE
DE L'ENSEIGNEMENT
SUPÉRIEUR
ET DE LA RECHERCHE

Liberté
Égalité
Fraternité

GRENOBLE INP UGA Ensimag

PROJET GÉNIE LOGICIEL 2023-2024

DOCUMENTATION DE VALIDATION

Ingénieur 2ème année

Professeur encadrante:

Karine ALTISEN

Equipe projet :

- Stéphane KOUADIO
- Julian COUX
- Breno MORAIS
- Loan GATIMEL
- Hugo MERCIER

SOMMAIRE

INTRODUCTION

La validation est le pilier qui soutient la qualité et la fiabilité de tout compilateur. Sans validation rigoureuse, les compilateurs sont susceptibles de comporter des erreurs, menant à des bugs dans les programmes compilés.

Le présent document offre un descriptif détaillé des tests réalisés dans le cadre du développement du compilateur Deca, soulignant les méthodes de validation employées à chaque étape du processus, de l'analyse lexicale et syntaxique à la génération de code, en passant par l'extension TRIGO.

La structuration méthodique des tests, accompagnée de scripts spécifiques, assure une couverture exhaustive et une évaluation rigoureuse de la qualité du compilateur. En parallèle, la gestion des risques et des rendus, ainsi que l'utilisation de JaCoCo pour mesurer la couverture de code, renforcent notre engagement envers la qualité, la robustesse et la conformité aux exigences du cahier des charges du produit final.

Cette démarche de validation, cruciale pour la fiabilité d'un compilateur, permet également son amélioration continue, s'adaptant aux besoins changeants des développeurs et aux évolutions du langage.

Descriptif des tests

1. Type de tests

A. Analyse lexicale, syntaxique

Lexer

Nous avons conçu des tests exhaustifs pour le lexer afin d'assurer une couverture complète de tous les tokens pouvant être reconnus. Cette approche vise à garantir que chaque élément lexical est correctement identifié par le lexer et qu'on obtient aucun mauvais caractère.

Parser

Les tests pour le parser ont été conçus pour évaluer la capacité du compilateur à détecter et gérer les erreurs de syntaxe. Nous avons exploré différentes règles syntaxiques afin de vérifier la robustesse du parser face à divers scénarios.

B. Analyse contextuelle

Pendant la validation de la partie B de notre compilateur, un soin particulier a été apporté à la variété des tests pour garantir la robustesse et respecter les exigences du cahier des charges. Nous avons déployé des scénarios exhaustifs qui englobent toutes les décorations d'arbres envisageables. Cela inclut des tests couvrant différentes configurations de déclarations de variables, permettant ainsi de valider la capacité du compilateur à traiter diverses structures syntaxiques et s'assurer de la cohérence des résultats.

Un autre volet crucial de nos tests a été la validation des erreurs contextuelles. Nous avons conçu des scénarios spécifiques visant à provoquer délibérément des erreurs contextuelles, évaluant ainsi la réactivité du compilateur et sa capacité à signaler correctement les erreurs dans des situations variées.

Pour finir nous avons ajouté au fichier ***pom.xml*** un fichier de test JUnit qui nous a permis de pouvoir efficacement tester la classe ***EnvironmentExp*** qui nous a été très utile.

C. Génération de code

Les tests de cette étape ont visé à vérifier la concordance entre la sortie générée par le compilateur et les attentes réelles, en mettant l'accent sur les comparaisons booléennes et les structures conditionnelles.

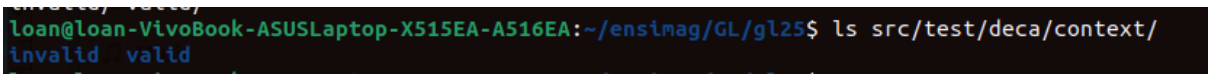
Cela a permis de confirmer que le compilateur effectue les branchements conditionnels de manière appropriée, garantissant ainsi l'affichage conforme à nos attentes.

D. Extension TRIGO

Pour tester l'extension que nous avons sélectionnée, nos tests visaient à exploiter les fonctions implémentées dans la bibliothèque Math.decah en utilisant des valeurs spécifiques. L'objectif était de comparer mathématiquement les résultats obtenus avec ceux attendus, vérifiant ainsi la justesse des calculs.

2. Organisation des tests

Les tests ont été regroupés par étapes dans des répertoires dédiés, suivant une structure organisée facilitant la gestion et la compréhension. Par exemple, les tests de la partie portant sur l'analyse contextuelle sont situés dans les répertoires *src/test/deca/**context**/valid/...* et src/test/deca/**context**/invalid/....



Cette organisation a contribué à maintenir la clarté et l'ordre, facilitant ainsi la reproduction des tests et la localisation rapide d'éventuelles erreurs.

3. Objectifs des tests

L'objectif global des tests était multiple. Tout d'abord, ils visaient à assurer que le compilateur respectait scrupuleusement les exigences du cahier des charges, garantissant ainsi sa conformité fonctionnelle. En outre, les tests ont joué un rôle crucial dans la surveillance de la progression du projet, nous permettant ainsi de vérifier que nous avancions dans la bonne direction sans commettre d'erreurs, tout en facilitant le processus de débogage.

La conception soignée des tests a également contribué à minimiser les risques liés aux commits, grâce à une organisation cohérente des répertoires de tests.

En somme, la méthodologie de validation a été conçue de manière à permettre une reproduction aisée des tests, une identification rapide des lacunes de la couverture des tests. Cette approche a efficacement renforcé le processus de développement en assurant la qualité du compilateur à chaque étape de son évolution.

Scripts de tests

Afin de garantir le bon fonctionnement de notre compilateur Deca, nous avons développé une série complète de tests, accompagnée de scripts facilitant leur exécution. Ces tests, minutieusement conçus et répartis dans le répertoire */src/test/deca/...*, couvrent de manière exhaustive chaque étape critique du processus de compilation.

Ces scénarios de validation, destinés à tester à la fois les cas valides et invalides, sont essentiels pour assurer la qualité de notre compilateur Deca. Leur exécution méthodique revêt une importance particulière, évaluant la performance du compilateur dans une variété de contextes. Ils représentent notre engagement envers la qualité du compilateur Deca.

1. Script Principal

Pour faciliter la visualisation de chaque étape de la compilation, nous mettons à disposition un script principal, ***./scriptTest.py***, un exécutable Python. Ce script prend en argument un fichier ***.deca*** et exécute individuellement les trois étapes de la compilation, affichant les résultats sur le terminal. Avant utilisation, assurez-vous de rendre le script exécutable avec la commande *chmod +x scriptTest.py* à exécuter à la racine du compilateur.

Pour l'utilisation, lancez le script avec un fichier de test comme argument, par exemple :

./scriptTest.py *src/test/deca/context/.../test_complet_ultime.deca*
ou (si la commande *chmod* n'a pas été exécutée) :
python3 ***scriptTest.py*** *src/test/deca/context/.../test_complet_ultime.deca*

Ce script offre une visualisation rapide du programme et permet d'identifier rapidement et précisément les erreurs, où qu'elles se trouvent.

Exemple sur Hello World :

```
test_lex

OBRACE: [@0,159:159='{',<38>,12:0]
PRINTLN: [@1,169:175='println',<13>,13:8]
OPARENT: [@2,176:176='(',<36>,13:15]
STRING: [@3,177:191='"Hello, world!"',<50>,13:16]
CPARENT: [@4,192:192=')',<37>,13:31]
SEMI: [@5,193:193=';',<42>,13:32]
CBRACE: [@6,195:195='}',<39>,14:0]

test_synt

`> [12, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [12, 0] Main
      +> ListDeclVar [List with 0 elements]
      `> ListInst [List with 1 elements]
         []> [13, 8] Println
             `> ListExpr [List with 1 elements]
                []> [13, 16] StringLiteral (Hello, world!)
```

```
test_context

DEBUG fr.ensimag.deca.tree.Program.verifyProgram(Program.java:
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClass(ListD
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClass(ListD
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClassMember
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClassMember
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClassBody(L
DEBUG fr.ensimag.deca.tree.ListDeclClass.verifyListClassBody(L
DEBUG fr.ensimag.deca.tree.Main.verifyMain(Main.java:39) - ver
DEBUG fr.ensimag.deca.tree.ListDeclVar.verifyListDeclVariable(
DEBUG fr.ensimag.deca.tree.ListDeclVar.verifyListDeclVariable(
DEBUG fr.ensimag.deca.tree.ListInst.verifyListInst(ListInst.ja
DEBUG fr.ensimag.deca.tree.AbstractPrint.verifyInst(AbstractPr
DEBUG fr.ensimag.deca.tree.StringLiteral.verifyExpr(StringLite
DEBUG fr.ensimag.deca.tree.StringLiteral.verifyExpr(StringLite
DEBUG fr.ensimag.deca.tree.AbstractPrint.verifyInst(AbstractPr
DEBUG fr.ensimag.deca.tree.ListInst.verifyListInst(ListInst.ja
DEBUG fr.ensimag.deca.tree.Main.verifyMain(Main.java:57) - ver
DEBUG fr.ensimag.deca.tree.Program.verifyProgram(Program.java:
`> [12, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [12, 0] Main
      +> ListDeclVar [List with 0 elements]
      `> ListInst [List with 1 elements]
         []> [13, 8] Println
             `> ListExpr [List with 1 elements]
                []> [13, 16] StringLiteral (Hello, world!)
                    type: string
```

```
test_codeGen

        TSTO #1
        BOV pile_pleine
        ADDSP #1
; ------------------------------------------------
;       Construction des tables des methodes
; ------------------------------------------------
; Construction de la table des methodes de Object
        LOAD #null, R0
        STORE R0, 1(GB)
        LOAD code.Object.equals, R0
        STORE R0, 2(GB)
; ------------------------------------------------
;            Code du programme principal
; ------------------------------------------------
; Variables declarations:
; Beginning of main instructions:
        WSTR "Hello, world!"
        WNL
        HALT
```

etc…

2. Script de validation automatique Lexer

À la racine du compilateur, un script, ***./lexer_test.py***, permet de tester spécifiquement la première étape du compilateur, le lexer. Son utilisation est simple :

./lexer_test.py *src/test/deca/context/.../test_complet_ultime.deca*
ou
python3 ***lexer_test.py*** *src/test/deca/context/.../test_complet_ultime.deca*

Ce script prend en entrée un fichier ***.deca*** avec les tokens attendus en commentaire au début du fichier. Il exécute le fichier avec notre compilateur puis compare la sortie attendue (en commentaire) avec celle obtenue, automatisant ainsi au maximum les tests.

```
Test_lex [OK]
```
: Pour un test valide ou bien :

```
Test_lex [ECHEC]
Erreur : Token inattendu trouvé : TIMES: [@15,231:231='*',<31>,6:30]
```

3. Script Génération de code

Un autre script, ***basic-gencode.sh***, permet de valider automatiquement la génération de code. Pour son exécution, il faut juste utiliser la commande à la racine du projet :

*./src/test/script/****basic-gencode.sh***
ou
mvn test (cette commande sera détaillé plus loin)

Ce script bash exécute la commande ***decac*** avec un fichier ***.deca*** en paramètre, récupère la sortie avec ***ima***, puis utilise un script Python pour extraire le résultat attendu du fichier de test, comparant ainsi les deux résultats.

Les fichiers ***.deca*** à tester doivent suivre un format spécifique, avec les résultats attendus en commentaire, facilitant ainsi l'automatisation des tests pour toutes les étapes du compilateur.

Ces scripts, combinés aux tests et à la documentation fournie, offrent une expérience complète et automatisée pour la validation du compilateur Deca.

4. Script decompile

Dans le répertoire ***src/test/launchers/***, il y a un fichier nommé ***test_decompile*** qui opère de manière similaire à "test_lex" et appel le fichier ***ManualTestDecompile***. La fonction de ce fichier est d'afficher la sortie décompilée d'un programme, que ce soit en le spécifiant en tant que paramètre ou en le saisissant manuellement.

Concrètement, en exécutant ***test_decompile*** avec un programme en tant que paramètre ou en saisissant le programme manuellement, la sortie décompilée est affichée. Cette sortie peut ensuite être visuellement comparée avec le programme source pour vérifier la précision et l'efficacité de la fonction de décompilation. Cette approche manuelle offre une méthode visuelle et rapide pour évaluer la qualité du processus de décompilation dans le cadre du développement de notre compilateur Deca.

```
Fichier deca :
// Description:
//    Programme minimaliste utilisant println.
//
// Resultats:
//    Hello
//
// Historique:
//    cree le 01/01/2024


{
        println("Hello, world!");
}


Sortie decompile :
{
        println("Hello, world!");
}
```

Gestion des risques et gestion des rendus

1. Gestion des risques

A. Tests exhaustifs

Un des risques majeurs était lié à la couverture des tests. Pour y remédier, nous avons mis en place des tests exhaustifs à chaque étape du compilateur (lexer, parser, génération de code et extension). Cette approche a permis de détecter rapidement les lacunes de la couverture des tests et d'y remédier de manière efficace.

B. Erreurs de conception

Afin de réduire les risques liés aux erreurs de conception, nous avons régulièrement effectué des revues de code en équipe. Ces sessions ont favorisé le partage des connaissances, l'identification de problèmes potentiels et l'amélioration continue de la qualité du code.

C. Scripts de tests automatisés

La mise en place de scripts de tests automatisés a été cruciale pour atténuer le risque de régression lors des évolutions du code. Ces scripts ont été conçus pour être évolutifs, facilitant ainsi l'ajout de nouveaux tests au fur et à mesure du développement.

De plus, cette automatisation a permis de vérifier systématiquement à chaque itération que les anciens tests continuaient de passer avec succès.

2. Gestion des risques

A. Structuration des Tests

La structuration rigoureuse des tests dans des répertoires dédiés à faciliter la gestion des rendus. Chaque étape du projet a été clairement identifiée dans l'arborescence des répertoires, simplifiant ainsi la localisation des tests spécifiques à une étape donnée.

B. Scripts de Tests Unifiés

La mise à disposition de scripts de tests unifiés, tels que ***./scriptTest.py*** et ***./lexer_test.py***, a simplifié le processus de validation pour les utilisateurs. Ces scripts, inclus dans le fichier ***pom.xml***, permettent l'exécution de l'ensemble des tests avec une seule commande (***mvn test***), facilitant ainsi les rendus réguliers.

La commande ***mvn test*** nous offre la simplicité de lancer tous nos tests, étant donné que nous avons ajusté tous les scripts disponibles pour qu'ils s'exécutent dans les répertoires de tests appropriés. Par ailleurs, nous avons également créé d'autres scripts tels que ***basic_decompile.sh***, que nous avons naturellement intégrés pour assurer un lancement fluide et complet de l'ensemble des tests.

Couverture des tests (résultats de Jacoco)

L'utilisation de **Jacoco** dans notre projet a profondément impacté la qualité de notre application. En utilisant Jacoco pour la mesure de la couverture de code, nous avons pu obtenir une visibilité approfondie sur les parties du code qui avaient été testées et celles qui nécessitaient une attention particulière. Les informations fournies par Jacoco ont guidé notre équipe tout au long du processus de tests, nous permettant de cibler spécifiquement les zones de code nécessitant une amélioration et assurant une couverture robuste.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		70%		53%	496	683	605	2,027	260	371	6	48
fr.ensimag.deca.tree		87%		73%	188	803	242	2,057	78	573	4	90
fr.ensimag.deca		71%		69%	26	79	60	216	6	42	2	5
fr.ensimag.deca.context		78%		71%	45	135	49	224	39	121	1	22
fr.ensimag.ima.pseudocode		77%		71%	29	93	43	199	20	77	2	26
fr.ensimag.ima.pseudocode.instructions		73%		n/a	18	62	31	111	18	62	15	54
fr.ensimag.deca.codegen		82%		62%	4	16	1	27	1	12	0	2
fr.ensimag.deca.tools		94%		83%	3	21	3	44	1	15	0	3
Total	4,825 of 22,098	78%	429 of 1,170	63%	809	1,892	1,034	4,905	423	1,273	30	250

Résultat obtenu avec Jacoco

Nous pouvons voir que la couverture de nos tests est très bonne, toujours au dessus de 70% et quasiment à 80% pour les trois étapes de notre compilateur (syntax, context, codegen). On s’approche de la barre des 90% pour ce qui concerne l’arbre. Notre couverture de tests et donc très bien ce qui garantit une bonne qualité du code pour notre compilateur.

Afin que Jacoco puisse donner une analyse précise de la couverture de nos tests, comme expliqué plus tôt, nous avons ajouté à nos scripts tous les fichiers de tests que nous avons créés au cours du projet permettant à Jacoco de fournir des

analyses précises et pertinentes pour orienter notre démarche d'amélioration continue.

Un exemple concret illustrant notre expérience avec JaCoCo concerne la découverte d'un oubli dans notre suite de tests. Nous avons constaté que nous ne testions pas les différentes options du compilateur, bien que celles-ci aient été correctement implémentées dans le code. Cette observation grâce à JaCoCo nous a permis de rectifier cet écart et de renforcer la robustesse de notre base de tests.

Ce cas spécifique souligne l'efficacité de JaCoCo en identifiant des lacunes potentielles dans notre couverture de tests, ce qui a contribué à l'amélioration continue de la qualité de notre code

Méthodes de validation autres que le test

En complément de l'utilisation de tests, notre approche de vérification de la qualité du code s'est étendue à d'autres méthodes.

1. Revues de Code

Nous avons également opté pour des revues de code systématiques au sein de notre équipe. Ces revues ont favorisé une approche collaborative et un partage de connaissances au sein de notre équipe. Les membres de l'équipe ont examiné attentivement le code les uns des autres, échangé des idées et proposé des améliorations.

Cela a permis de détecter des erreurs potentielles, mais elle a également favorisé une meilleure entente du groupe. Les revues de code se sont avérées être un complément précieux aux tests automatisés dans notre démarche de garantie de qualité.

2. Conformité du cahier des charges

Certains aspects spécifiques définis dans le cahier des charges n’ont pas été pleinement couverts par nos tests. Ainsi, nous avons mis en place des validations manuelles pour garantir que chaque fonctionnalité requise était implémentée conformément aux spécifications du cahier des charges.

Un exemple concret concerne la validation manuelle de la conformité des messages d'erreurs, effectuée en les comparant directement aux spécifications du cahier des charges.

CONCLUSION

La documentation concernant l'utilisation des scripts de tests, les structures des répertoires de tests, et les résultats de Jacoco constitue un suivi exhaustif de l'évolution du projet. Cette démarche a favorisé une transparence renforcée quant à la qualité du code, simplifiant la gestion des livrables en offrant une assise robuste basée sur des tests approfondis.

L'association d'une gestion des risques à notre base de tests et de scripts a permis d'optimiser la qualité du compilateur Deca, tout en réduisant au minimum les éventuels obstacles associés aux risques de développement. Cette approche rigoureuse souligne notre engagement envers la qualité et la conformité du cahier des charges.