# Document de validation

Grenoble-INP ENSIMAG

Groupe 19:

Salaheddine ASLOUNE
Wen LUO
Anass BOUBKRI
El Mehdi KHADFY
Amine BELFKIRA

# Sommaire:

# INTRODUCTION

Une partie importante du projet consiste à s'assurer que le code remplit son objectif. Pour cela, plusieurs techniques sont utilisées.

Dans notre projet, la validation du compilateur repose sur les tests. C'est pourquoi, une architecture de tests particulière a été suivie. Puisque cela évaluera la qualité du système et s'assurera que ce dernier fait ce qu'on attend de lui.

# I. Descriptif des tests:

Des tests ont été conçus pour différentes étapes du projet. Concernant la première étape (étape A), les tests portent sur la construction de l'arbre. Dans l'étape B, les tests portent sur le parcours de l'arbre abstrait, la gestion des environnements, les enrichissements de l'arbre et la décoration. En ce qui concerne l'étape C, les tests portent sur les différentes constructions Deca, en particulier, sur les séquences d'instructions assembleur.

Les tests avaient pour but de tester le bon fonctionnement de notre compilateur. Ainsi, la stratégie était de classifier les tests selon les étapes du projet. C'est-pourquoi, on retrouve plusieurs dossiers de tests chacun contenant 2 autres dossiers nommés respectivement valide et invalide. Le but est certes de vérifier le bon fonctionnement du programme, mais aussi de vérifier que les messages d'erreurs ou même l'identification de cette dernière sont ceux recherchés à afficher.

On a cherché à tester les parties lexicographiques, syntaxe, context textuelle et la partie génération de code.

## I.1 Tests Lexicographiques:

Les tests lexicographiques se situent dans le dossier *./src/test/deca/syntax/valid/lex* et *./src/test/deca/syntax/invalid/lex*. Ces tests peuvent être exécutés soit manuellement par la commande *test_lex*

<*nom fichier test.deca*>, ou bien grâce à un script automatisé *all_test_lex.sh*. Ce dernier effectue des tests lexicographiques de tous les fichiers présents dans ses deux répertoires et envoie le résultat de compilation dans les fichiers <*nom fichier test.lis*> présents dans le répertoire *./src/test/deca/TestsLexroResultat*. Il permet aussi de vérifier si le test renvoie le résultat voulu, c'est-à-dire un fichier valide doit s'exécuter sans erreur, alors qu'un test invalide doit envoyer un message d'erreur lexical.

I.2 Tests Syntaxiques:

Les tests syntaxiques se situent dans le dossier *./src/test/deca/syntax/valid/synt* et *./src/test/deca/syntax/invalid/synt*. Ces tests peuvent être exécutés soit manuellement par la commande *test_synt <nom fichier test.deca>*, ou bien grâce à un script automatisé *all_test_synt.sh*. Ce dernier effectue des tests syntaxiques de tous les fichiers présents dans ses deux répertoires et envoie le résultat de compilation dans les fichiers <*nom fichier test.lis*> présents dans le répertoire *./src/test/deca/TestsSynchroResultat*. Il permet aussi de vérifier si le test renvoie le résultat voulu, c'est-à-dire un fichier valide doit s'exécuter sans erreur, et affiche un arbre abstrait décorée, alors qu'un test invalide doit envoyer un message d'erreur.

I.3 Tests Contextes:

Les tests contextuels se situent dans le dossier *./src/test/deca/context/valid* et *./src/test/deca/syntax/context/invalid*. Ces tests peuvent être exécutés soit manuellement par la commande *test_context<nom fichier test.deca>*, ou bien grâce à un script automatisé *all_test_context.sh*. Ce dernier effectue des tests contextuels de tous les fichiers présents dans ses deux répertoires et envoie le résultat de compilation dans les fichiers <*nom fichier test.lis*> présents dans le répertoire *./src/test/deca/TestsContextResultat*. Il permet aussi de vérifier si le test renvoie le résultat voulu, c'est-à-dire un fichier valide doit s'exécuter sans erreur, alors qu'un test invalide doit envoyer un message d'erreur. Ensuite, chaque message d'erreur est examiné individuellement à dessein de vérifier si ce dernier renvoie le message

souhaité. On essaye de dégager tous les messages d'erreurs contextuelles possibles afin de vérifier si le compilateur se comportait comme voulu

I.3 Tests Codegen:

Les tests syntaxiques se situent dans le dossier *./src/test/deca/codegen/objet, ./src/test/deca/codegen/sansObjet, ./src/test/deca/codegen/perf/provided, ./src/test/deca/syntax/valid/provided*. Ces tests peuvent être exécutés soit manuellement par la commande *deca <nom fichier test.deca> , suivi de la commande ima <nom fichier test.ass>* ou bien grâce aux scripts automatisés *TestGenCode.sh et decompile-Gen.sh*. Ce dernier effectue des tests de tous les fichiers présents dans ses répertoires et envoie le résultat de compilation dans les fichiers *<nom fichier test.res>* présents dans le répertoire *./src/test/deca/codegen/TestsRes* .

# II. Les scripts de tests:

Afin d'automatiser l'exécution de *test_ synt, test_ lex*, *test_ context*, *decac et ima* sur chacun des répertoires, différents scripts shell ont été créés. Les scripts shells se trouvent dans le dossier *./src/test/script:*
-Les scripts shell all_test_context.sh, all_test_lex.sh, all_test_synt.sh ont été redigé de la même manière, à savoir:

```
echo [INFO] "Tests invalides:"
for cas_de_test in ./<REPERTOIRE>/*.deca
do
  <type de test>_invalide  "$cas_de_test"
done

echo

echo [INFO] “Tests valides:"
for cas_de_test in  ./<REPERTOIRE>/*.deca
do
  <type de test>_valide "$cas_de_test"
```

```
done
```

Avec <type de test>_invalide et <type de test>_valide deux fonctions  redigées aussi sous la même forme, à savoir:

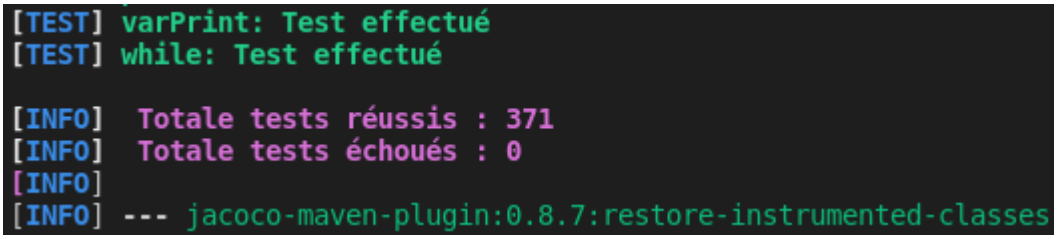
```
<type de test>_valide ou invalid ( ) {
   if <type de test>  "$1" 2<&1 | grep -q -e "$1:[0-9][0-9]*:"
   then
      filename=$(echo ${cas_de_test} | xargs basename)
      resultat=$(<type de test>"$1")
      echo [TEST ] "Echec inattendu de ./<REPERTOIRE>  sur $filename"
   else
      filename=$(echo ${cas_de_test} | xargs basename)
      resultat=$(test_lex "$1")
      echo [TEST ] "Succes attendu pour <type de test> sur $filename"
   fi
}
```

La particularité des tests valide est que le résultat est enregistré dans le fichier <nom du test>.lis. Deux variables de types statiques *reussi* et *echec* sont utilisées afin de compter le respectivement le nombre des tests réussis et échoués.

- Les scripts shell decompile_Gen.sh et TestGenCode.sh sont deux scripts pour tester le bon fonctionnement de la partie C. Ces tests automatisés permettent dans un premier lieu d'exécuter la commande decac sur tous les fichiers présents dans le répertoire ./src/test/deca/codegen. On lance ensuite la commande ima dessus et on compare le résultat de sortie. La particularité de script decompile-Gen.sh est le lancement de la commande decac -p, aussi, il

  conserve le résultat de la commande ima dans le fichier <Nom du fichier test>.res.  À la fin de l'exécution, on obtient le nombre des tests exécutés.
  -Le script ext.sh permet  de lancer la commande decac -j. Ce script exécute les tests liés à la partie extension BYTE.

-Le script  shell all_tests.sh c'est une sorte de combinaison de tous ses derniers scripts, ce qui permet d'exécuter tous les tests dans tout le répertoires tests. Ainsi, à la fin on obtient le nombre total des tests réussi et échoué grâce au variables statique reussi et echoue.




# III. Gestion des risques

- Il est essentiel au sein d'une équipe de prévoir les risques auxquels on peut être confronté afin d'éviter d'éventuels problèmes qui pourraient entraîner un dysfonctionnement du groupe. Le but est de trouver des solutions.

- On distingue deux catégories de risques:

- Risques liés à l'organisation
- Risques techniques

| RISQUES | Solution et dispositifs de prévention | Importance /3 |
|---|---|---|
| - Oubli d'un document dans un rendu de suivi | -Check-list des éléments à rendre pour chaque suivi | 3 |
| - Oubli de la date d'un suivi | -Mise en place d'un planning, avec des suivis | 3 |

| | | |
|---|---|---|
| | intermédiaires permettant de garantir la complétion des tâches requises pour un suivi. | |
| - Branching très lourd et des problèmes de branches et version divergents. | - Mise en place d'un git-Master qui veillera à la vérification du bon état de la branche Master de notre GIT. Par ailleurs, chaque membre travaillant sur une branche qui n'est pas la sienne veille à faire un pull avant toute modification et est censé faire une relecture du code qu'il a modifié ou ajouté et une vérification du bon fonctionnement des tests avant de faire un push. Une relecture du code qu'on a push durant la journée est aussi faite par le Git-Master. | 3 |
| -Inaptitude d'un membre à travailler suite à un problème de santé. | - Faire une réunion entre les autres membres de l'équipe pour trouver une | 3 |

| | | |
|---|---|---|
| | méthode de répartition du travail permettant de compenser son absence temporaire. | |
| - Mésentente entre des membres du groupe. | - Mise en place d'une petite réunion afin d'essayer de trouver la source du problème et essayer de la résoudre rapidement pour ne pas nuire au bon déroulement du projet. | 2 |
| -Erreur de compilation | -Vérification de la bonne compilation du code avant tout push dans Git et éviter de faire des pushs de parties conséquentes de code près des dates échéances. | 3 |
| -Avoir une couverture de tests trop faible. | -Nous utiliserons l'outil Jacoco pour vérifier si certaines parties sont non couvertes par le tests. | 2 |
| - Problèmes de régression. | - Exécution de tous nos tests après l'ajout de nouvelles | 3 |

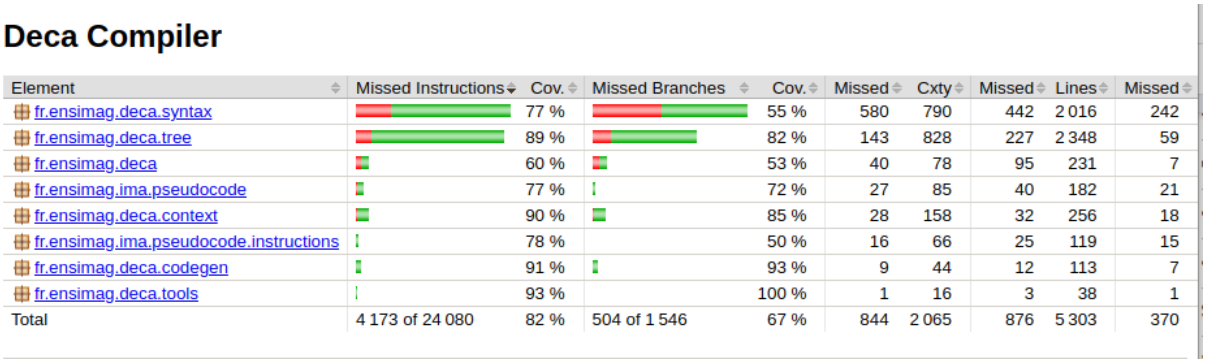
| | | |
|---|---|---|
| | fonctionnalités afin d'éviter ces problèmes de régression. | |

# IV. Gestion des rendus:

Après la date de récupération des projets au Lundi 24 janvier 2022, tous les membres de l'équipe se sont rassemblés afin de répartir les tâches de rédaction. Chaque membre s'est chargé d'une d'une partie du compilateur, et il s'est impliqué à la finire avant les dates imparties des rendus. Le git-Master veillait à la vérification du bon état de la branche principale, ainsi des branches ont été créées. Aussi, chaque membre veillait à faire un pull avant toute modification, relire le code et le tester.

# V. Couverture des tests:

La mesure de la couverture du code par la base des tests s'est avérée très pratique car elle fournit une idée de la qualité des tests et de leur efficacité. Elle nous à permis de bien identifier les parties non couvertes. Les scripts all_tests.sh et ext_sh permettent d'avoir une couverture de 82% de notre code, ce qui est un bon résultat . Certes la couverture dépasse les 75%, toutefois il aurait fallu une couverture de 100%. En effet, certaines parties du code sont difficiles à couvrir et il s'est avéré difficile de les couvrir.



**Deca Compiler**

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed |
|---|---|---|---|---|---|---|---|---|---|
| fr.ensimag.deca.syntax | | 77 % | | 55 % | 580 | 790 | 442 | 2 016 | 242 |
| fr.ensimag.deca.tree | | 89 % | | 82 % | 143 | 828 | 227 | 2 348 | 59 |
| fr.ensimag.deca | | 60 % | | 53 % | 40 | 78 | 95 | 231 | 7 |
| fr.ensimag.ima.pseudocode | | 77 % | | 72 % | 27 | 85 | 40 | 182 | 21 |
| fr.ensimag.deca.context | | 90 % | | 85 % | 28 | 158 | 32 | 256 | 18 |
| fr.ensimag.ima.pseudocode.instructions | | 78 % | | 50 % | 16 | 66 | 25 | 119 | 15 |
| fr.ensimag.deca.codegen | | 91 % | | 93 % | 9 | 44 | 12 | 113 | 7 |
| fr.ensimag.deca.tools | | 93 % | | 100 % | 1 | 16 | 3 | 38 | 1 |
| Total | 4 173 of 24 080 | 82 % | 504 of 1 546 | 67 % | 844 | 2 065 | 876 | 5 303 | 370 |

# VI. Méthodes de validation autres que le test:

Pour valider notre code, les membres de l'équipe à chaque pull se sont chargés de relier les nouvelles parties du code écrit. Aussi la hiérarchie du projet nous à permis de déboguer plusieurs problèmes, en effet la partie C permet de déboguer plusieurs failles dans la partie B il faut que le code en amont soit fonctionnel.