

# Bilan sur l'étape de vérification

Antoine Turkie, Augustin Chenevois, Aurélien Delory, Mihaja  
Razafimahefa, Sarah Ramaharobandro

## **Introduction**

Ce document vise à mettre en évidence la partie validation de notre compilateur. En effet pour pouvoir vérifier toutes les fonctionnalités de notre compilateur, nous avons effectué de nombreux tests visant à couvrir les différentes étapes du compilateur:

- **Étape A** : (lexer/parser)
- **Étape B** : (vérification contextuelles et décoration de l'arbre)
- **Étape C** : (génération de code)

## **Descriptif des tests**

Nous avons décidé de réaliser des tests qui vérifient à chaque fois la fonctionnalité que nous avons décidée de mettre en place lors de la User Story par des cas très facile visant à reproduire le résultat attendu pour tous les tests qui devaient être fonctionnels (**cas valides**), en même temps nous rajoutions des tests qui n'étaient pas censés marcher afin de vérifier qu'on respectait bien convenablement la spécification de notre compilateur (**cas invalides**).

Une fois ces tests faits, nous avons opté pour tester différents cas plus durs visant à manipuler plusieurs fois la fonctionnalité correspondante dans des cas limites et ensuite essayer de mixer cette fonctionnalités avec les autres fonctionnalités qu'on a testé et qui marchaient afin d'être sûr d'avoir correctement respecté la spécification du client.

Finalement, une fois les spécifications respectées, étant donné que nous avons choisi l'extension OPTIM, nous cherchions à vérifier la cohérence de nos optimisations en réduisant au maximum le nombre de cycles en faisant des tests visant à challenger la performance.

Sans oublier les diverses options de decac que nous avons testé sur de nombreux exemples afin de respecter le cahier des charges.

Voici le type de tests que nous avons choisi de réaliser pour chaque étape de notre projet :

Pour l'**Étape A**, qui représente l'analyseur lexicale et syntaxique:

nous devons réaliser 3 types de tests afin de vérifier la bonne implémentation de cette étape:

Tout d'abord vérifier la cohérence de notre partie lexicale (**lexer**) à chaque implémentation d'une nouvelle fonctionnalité, pour cela nous avons décidé de faire des tests fonctionnels visant à vérifier que les tokens décrits étaient bien ceux qu'on attendait en sortie. Ainsi on pouvait rapidement savoir si les tokens reconnus étaient les bons ou bien comprendre que l'ordonnancement de notre lexer devait être cohérent puisque par exemple: nous avons repéré que si le `readInt` était placé après `ident` dans le lexer, ce dernier allait être reconnu comme un identifiant et cela n'aurait donc pas eu le comportement attendu. Le script `test_lex` fournit dans le sujet est celui qui nous permettait de tester si le lexer fonctionnait, mais surtout s'il fonctionnait pas.

#### *Sortie test\_lex qui fonctionne*

```
OBRACE: [@0,158:158='{',<31>,10:0]
IDENT:  [@1,164:166='int',<24>,11:4]
WHILE:  [@2,168:172='while',<1>,11:8]
EQUALS: [@3,174:174='=',<35>,11:14]
INT:    [@4,176:176='3',<8>,11:16]
SEMI:   [@5,177:177=';',<30>,11:17]
CBRACE: [@6,179:179='}',<32>,12:0]
```

#### *Sortie du test\_lex qui ne fonctionne pas*

```
OBRACE: [@0,138:138='{',<31>,13:0]
PRINTLN: [@1,141:147='println',<6>,15:0]
OPARENT: [@2,148:148='(',<33>,15:7]
INFO fr.ensimag.deca.syntax.DecacErrorListner.syntaxError(DecacErrorListner.java:50) - no token, no LocationException => using input.getSourceName(): line=15
StringSimple.deca:15:8: token recognition error at: ''
```

Ensuite, une fois la partie du lexer testée, nous devons vérifier que l'arbre généré soit cohérent en analysant la partie syntaxique (**parser**) de notre implémentation. Pour cela nous devons vérifier 2 choses :

- la création de l'arbre est cohérente avec ce qu'on attend (que ce soit par exemple : les différentes méthodes appelées, les différents paramètres, le nombre, leur localisation etc... .

Pour cela nous utilisons le script `test_synt` qui permet de vérifier cela

```

> [1, 0] Program
+> ListDeclClass [List with 1 elements]
|   []> [1, 0] DeclClass
|       +> [1, 6] Identifier (A)
|       +> [builtin] Identifier (Object)
|       +> ListDeclField [List with 0 elements]
|       `> [2, 4] ListDeclMethod [List with 1 elements]
|           []> [2, 4] DeclMethod
|               +> [2, 4] Identifier (int)
|               +> [2, 8] Identifier (nombreDeCartePokemon)
|               +> ListDeclParam [List with 0 elements]
|               `> [2, 30] MethodBody
|                   +> [3, 8] ListDeclVar [List with 0 elements]
|                   `> [3, 8] ListInst [List with 1 elements]
|                       []> [3, 8] Return
|                           `> [3, 15] Identifier (a)
|   `> EmptyMain

```

Sinon le test\_synt nous renvoie l'erreur correspondante comme par exemple:

```
while-conditionWithoutParenthesis.deca:2:10: missing '(' at '4'
```

Ainsi pour tester la validité de notre parser, on a commencé par regarder les cas de base valides qui sont censés marcher, ensuite pour tester la robustesse de ce qu'on a implémenté, on s'est mis à la place du client et on a commencé à écrire divers programmes qu'il était susceptible d'écrire tel qu'un programme qui fait "x +=3; ", un nom de programme qui correspond à un mot réservé, la mauvaise manière de déclarer une classe etc... .

On devait réaliser chacun de ces tests afin de s'assurer que notre compilateur était robuste et pour vérifier cela, on devait réfléchir à tout ce qu'on ne pouvait pas faire d'une manière évidente et des cas plus complexes où c'est notamment grâce à la documentation que nous avons pu trouvé comment créer des tests invalides intéressants.

Finalement, nous devons vérifier les étapes de **décompilation** (decac -p), en effet si l'arbre a bien été créé, il fallait vérifier que la décompilation implémentée permettait de revenir au code deca à partir de l'arbre et notamment de vérifier le théorème suivant :

**Théorème 1 (spécification de l'analyseur syntaxique)** *Pour tout programme Deca syntaxiquement correct, il existe un arbre de la syntaxe abstraite qui se décompile dans un programme Deca syntaxiquement correct « équivalent ».*

Pour cela, nous avons décidé d'appliquer decac -p sur nos programmes de tests pour créer un fichier de la forme testp.deca puis nous avons décidé de réappliquer decac -p afin de vérifier que les 2 fichiers soient identiques en

réalisant la commande diff. Nous avons essayé de le rajouter à notre script de test mais cela vérifiait assez mal le théorème donc on a décidé de l'enlever.

Pour l'**Étape B** (Vérification contextuelles et décoration de l'arbre)

Cette étape est essentielle afin de vérifier que les programmes valides dérivent des grammaires attribuées de la spécification. De plus, on vérifiait que les opérations de typage ont bien été ajoutées lors de la décoration de l'arbre.

Pour effectuer la première tâche, une fois que l'étape B a été implémentée, nous avons décidé de commencer par regarder la règle que nous avons implémentée dans la fonctionnalité en question. Puis nous réalisons une série de tests invalides qui devaient renvoyer une erreur correspondante à la règle indiquée si l'implémentation avait bien été faite. Pour cela nous avons décidé d'utiliser decac -v afin de voir si l'erreur était bien repérée ou non.

Si l'on obtient une erreur correspondante à la règle qu'on voulait tester, alors on détecte bien l'erreur voulue...

```
/user/3/.base/turkiea/home/GL/Projet_GL/gl38/src/test/deca/context/invalid/while
-sansBooleen.deca:11:11: Le paramètre de l'instruction n'est pas de type boolean
: règle (3.29)
```

sinon la fonctionnalité est incomplète...

Une fois ceci réalisé, nous pouvons passer sur les tests valides et vérifier que le compilateur ne détecte aucune erreur.

Pour cette étape la plupart de nos tests sont des vérifications qui ont été faites règle par règle et revérifiées à la toute fin à l'aide de Jacoco.

Pour ce qui est de la vérification de la décoration, celle-ci est un peu moins développée puisque cette dernière se faisait à la main pour vérifier que tout se passait bien pour la suite mais malheureusement sans rajouter à nos scripts de tests une manière automatique de check.( Par l'intermédiaire de l'étape C et du résultat final, on pouvait voir si les décorations avaient bien été faites (pour le ConvFloat notamment).

Concernant l'**étape C**, nous avons testé notre génération de code en vérifiant les différents codes assembleurs et leur cohérence. Les scripts de tests vérifiaient le résultat avec celui attendu que l'on connaissait ou calculait de

notre côté (par calcul mental ou bien par calculatrice selon le jour). De plus à côté de ces tests là, on testait que la sortie des overflow, division par 0 et déréférencement null était bien une erreur du bon type par Ima.

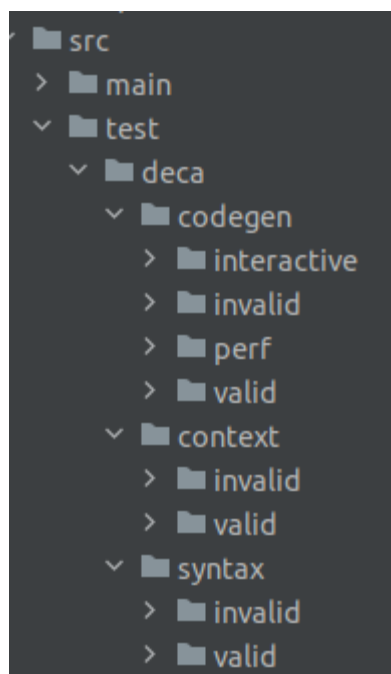
Voici un exemple sur le terminal:

```
turkiea@ensipc586:~/GL/Projet_GL/gl38/src/test/deca/codegen/invalid$ ima arithmetic-divisionPar0modulo.ass
2.30000e+00
Erreur de division par 0
```

Pour la partie de génération de code, nous pouvons automatiser la comparaison de résultat mais il faut néanmoins pouvoir vérifier la partie assembleur pour optimiser et ceci malheureusement doit se faire à l'œil.

## **Organisation et objectifs des tests**

Voici comment l'organisation des tests est effectuée :



Chacune des 3 étapes A, B et C (syntax, context et codegen) a un sous-dossier composé des tests valides et invalides (développé dans le

paragraphe précédent), pour la génération de codes, il y'a 2 dossiers supplémentaires : un composé des tests interactifs pour permettre aux personnes qui ont envie de faire des tests interactifs de pouvoir les lancer à partir de dossier sans gêner les scripts de tests.

Pour nos scripts de tests, nous avons décidé de procéder de la manière suivante pour les différents type de fichiers :

- Pour les tests syntaxiques valides, en plus de test.deca, on ajoute un test.pp dans le même dossier qui est le résultat attendu de le decac -p du fichier.
- Pour les tests syntaxiques invalides, nous avons en plus du test.deca, un fichier de type .lex ou .synt (selon le type d'erreurs que l'on doit avoir) qui contient la ligne où le problème est censé arriver sur le fichier .deca, c'est la chose que l'on va comparer afin de savoir si l'erreur a bien eu lieu au bon endroit.

Les tests lexicaux sont gérés par le script run\_lex.sh qui a chaque fichier deca va rechercher le .lex correspondant et le comparer avec la sortie standard.

Les tests syntaxiques sont gérés par le script run\_synt.sh qui a chaque fichier deca va rechercher le .synt correspondant et le comparer avec la sortie standard.

Les tests de décompilation devraient être gérés par le script run\_décompilation.sh(pour l'instant il est incomplet), en récupérant les .pp et les comparant à la sortie de decac -p

- Pour les fichiers contextuelles valides, nous ne rajoutons pas d'autres fichiers, en effet pour vérifier la décoration de l'arbre, il faut effectuer le script test\_context à la main.

- Pour les fichiers contextuelles invalides, nous rajoutons un fichier .context où l'on écrit la ligne où l'erreur est censée être détectée.

Les tests contextuels sont gérés par le script `run_context.sh` qui a chaque fichier deca va rechercher le `.context` correspondant et le comparer avec la sortie standard.

- Pour les fichiers gencode valides, nous rajoutons un fichier `.res` contenant le résultat attendu du test.
- Pour les fichiers gencode invalides, nous rajoutons un fichier `.ima` qui indique la sortie que `ima` doit ressortir.

Les tests gencode sont gérés par le script `run_gencode.sh` qui a chaque fichier deca va rechercher le `.res` et `.ima` correspondant et le comparer avec la sortie standard.

L'objectif de faire ça est de toujours comparer les sorties attendues aux résultats dans les fichiers non deca correspondant afin de pouvoir automatiser cela facilement pour pouvoir effectuer tous nos tests avant chaque push et donc être sûr que tout fonctionne et que la comparaison des résultats, ou l'endroit où se trouve l'erreur est cohérent.

## **Lancement et scripts de tests**

Pour pouvoir lancer tous les tests avec notre programme il suffit d'utiliser la commande

"`mvn test`", cette commande permet de faire plusieurs choses à la fois très pratiques même si très coûteuse en termes de ressources. On a rajouté dans le fichier `pom.xml` nos scripts de tests personnalisés pour que maven puisse l'intégrer à son cycle de vie lors du lancement de `mvn test`. Elle effectue dans l'ordre:

- `mvn compile` ( qui compile le programme)
- Une fois cela fait, la 1ère vérification est celle des tests unitaires et common tests donnés par le professeur (si cette étape ne marche pas, l'exécution s'arrête et renvoie l'erreur associé)
- Une fois les vérifications de bases établies, on va lancer les vérifications de tests indiqués dans les parties précédentes en



comparant les sorties de ima avec les résultats attendus. A la moindre erreur, tout s'arrête. On effectue les tests dans l'ordre des étapes (syntax, context and codegen)

- Si tout s'est bien passé on obtient le résultat suivant

```
[OK] decac  print-true.deca
[OK] decac  return-bigExpression.deca
[OK] decac  return-constantMethod.deca
[OK] decac  return-convfloat.deca
[OK] decac  return-new.deca
[OK] decac  return-param.deca
[OK] decac  selection-fieldWithProtected.deca
[OK] decac  this-fibo.deca
[OK] decac  this-implicitMethod.deca
[OK] decac  this-multitest.deca
[OK] decac  this-orNotThis.deca
[OK] decac  this-variable.deca
[OK] decac  thisImplicit-var.deca
[OK] decac  unary-minus.deca
[OK] decac  -r 4 unary-minus.deca
[OK] decac  whileFalse.deca
Fin tests codegen
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:01 min
```

## Gestion des risques et gestion des rendus

Pour la gestion des risques, nous avons établi une stratégie simple et précise: toujours après l'ajout de fonctionnalités que ce soit par l'une des 3 étapes, il faut vérifier que tous les tests que nous avons fait depuis le début continuent de marcher. Pour cela nous relançons mvn test, afin de vérifier le build success ainsi que l'ajout des nouveaux tests. On aurait dû peut-être utilisé les divers scripts pour simplifier la vérification surtout d'un point de vue énergétique.

















Pour la partie sans-objet, nous avons opté pour réaliser fonctionnalité par fonctionnalité afin d'être sûr de pouvoir rendre quelque chose de fonctionnel le jour J même si nous n'aurions pas fini. Pour la gestion des risques, cette méthode était très bien puisqu'elle nous permettait de ne pas stresser jusqu'à la dernière seconde pour le rendu.

En revanche, pour la partie avec objet, nous avons décidé de faire étape A puis B et puis C et nous avons réussi à finir les fonctionnalités que l'on voulait que peu de temps avant le rendu (ce que je ne recommande pas).

Il faut de plus gérer mieux les merge conflicts qui peuvent venir de nulle part même avec une bonne organisation et donc s'y prendre à l'avance. Cela n'a malheureusement pas été fait et nous a coûté plusieurs moments tendus. Pour les prochains projets, nous avons donc appris à mieux gérer ce phénomène afin de rendre les rendus paisiblement.

## Résultats de Jacoco

### Deca Compiler

Element	Missed Instructions	Cov.	
<a href="#">fr.ensimag.deca.syntax</a>		77%	
<a href="#">fr.ensimag.deca.tree</a>		82%	
<a href="#">fr.ensimag.deca</a>		77%	
<a href="#">fr.ensimag.ima.pseudocode</a>		71%	
<a href="#">fr.ensimag.deca.context</a>		84%	
<a href="#">fr.ensimag.ima.pseudocode.instructions</a>		76%	
<a href="#">fr.ensimag.deca.codegen</a>		99%	
<a href="#">fr.ensimag.deca.tools</a>		100%	
Total	4,590 of 23,070	80%	

On a un coverage à 80% des instructions ce qui est assez complet. Tout au long de notre implémentation : nous avons essayé de tout faire afin d'utiliser

le moins de code mort possible et de vérifier que nos tests vérifient tous les cas, que ce soit tous les cas dans le codeGen, certaines règles qui peuvent être oubliées etc... . Ainsi en dehors de certains catches et certains pretty print que l'on teste manuellement, on arrive à passer dans tous les cas des fonctions importantes.

On aurait pu peut-être rajouter certains tests unitaires pour avoir les 100% mais nous n'avons pas trouvé cela utile.

## **Méthodes de validation utilisées autres que le test**

En dehors de la validation pour les tests, nous avons dû vérifier certaines parties de l'implémentation à la main :

- La décoration de l'arbre après l'étape B, que ce soit par la localisation , la vérification des types, des ConvFloats etc... .
- De même pour la vérification de la partie assembleur et notamment pour l'optimisation, nous avons décidé de regarder chaque ligne afin de voir s'il y avait la possibilité d'optimisation de réduction du nombre d'instructions et de cycles.