

Decac

Groupe Génie Logiciel n°29



Compilateur pour le langage Deca

Rapport de validation

Janvier 2024

Table des matières

1	Descriptif des tests	2
1.1	Étape A	2
1.1.1	Test du lexer	2
1.1.2	Test du parser	2
1.2	Étape B : Test de l'analyse contextuelle	2
1.3	Étape C : Test de génération du code assembleur	3
1.4	Décompilation	3
1.5	Vérification	4
1.6	Registres	4
1.7	Synthèse	5
2	Les scripts de tests	6
2.1	jacoco-report	6
2.2	run-all-tests	6
3	Gestion des risques et gestion des rendus	7
4	Résultats de Jacoco	9

1 Descriptif des tests

Les tests sont regroupés selon les étapes testés, et dans chaque étape, les tests sont séparés en deux groupes : ceux dits **corrects**, qui ont pour objectif de passer avec succès, et ceux dits **incorrects** qui ont pour objectif d'échouer. Dans toute la documentation, on dira qu'un test est **réussi** lorsqu'il produit le résultat attendu. Dans le cas contraire, on dira que le test a **échoué**.

1.1 Étape A

1.1.1 Test du lexer

Les tests du lexer ont pour objectif de tester les unités lexicales de Deca. Les tests corrects s'assurent qu'elles sont identifiées par le lexer, pendant que les tests incorrects vérifient que seules les unités lexicales du cahier des charges sont reconnues.

```
===== Etape A =====
==> Test du lexer
--> Tests des programmes syntaxiquement corrects
    [Total : 9/9]
--> Tests des programmes syntaxiquement incorrects
    [Total : 6/6]
```

Figure 1: Résultat du `run-all-tests` à l'étape du lexer.

Tous les tests de cette partie ont réussi, comme en atteste le `run-all-tests` sur la capture ci-dessus.

1.1.2 Test du parser

Les tests effectués sur le parser nous permettent de vérifier que les éléments conformes au cahier des charges ont une syntaxe correcte, tandis que les cas incorrects nous permettent de reconnaître uniquement les éléments définis dans le cahier des charges.

```
==> Test du parser
--> Tests des programmes syntaxiquement corrects
    [Total : 16/16]
--> Tests des programmes syntaxiquement incorrects
    [Total : 4/4]
```

Figure 2: Résultat du `run-all-tests` à l'étape du parser.

1.2 Étape B : Test de l'analyse contextuelle

Cette étape de test du processus de compilation vérifie l'arbre pour s'assurer qu'il est correctement décoré. Nous avons organisé cette étape en deux étapes distinctes, ayant pour objectif d'améliorer notre compilateur.

Notre approche s'est concentrée sur la conception de tests initiaux dans la première phase. Ces tests basés sur des usages simples visaient à vérifier la correcte décoration de l'arbre pour les éléments

constitutifs de la partie Sans objet, Essentiel et Complet du compilateur. En identifiant les fonctionnalités de base et en garantissant la stabilité des composants essentiels, nous avons pu établir une base solide. Cette première étape nous a permis de passer à la deuxième étape. Nous avons augmenté notre batterie de tests avec des tests plus complexes pour vérifier la capacité de notre compilateur. Ce processus a été d'une grande aide pour l'amélioration de notre compilateur, car il a permis de détecter et de corriger de nombreux bugs. Ce qui a conduit à une amélioration et une optimisation la qualité de notre compilateur.

```
===== Etape B =====
==> Test du contexte
--> Tests des programmes corrects
    [Total : 67/67]
--> Tests des programmes incorrects
    [Total : 72/72]
```

Figure 3: Résultat du `run-all-tests` à l'étape du contexte (Étape B).

Notre approche a été bénéfique, car tous les tests de notre batterie de tests sont passés correctement ou non s'il devait échouer.

1.3 Étape C : Test de génération du code assembleur

Au cours de cette étape, nous avons appliqué la même stratégie que celle mise en œuvre lors de l'étape B du processus. Cette stratégie s'est avérée bénéfique, contribuant à la correction d'un nombre substantiel de bugs et d'anomalies identifiés tout au long de cette étape. Cette approche systématique a renforcé notre capacité à résoudre efficacement les problèmes rencontrés, consolidant ainsi la robustesse de notre compilateur.

```
===== Etape C =====
==> Test de génération du code assembleur
    [Total : 63/63]
```

Figure 4: Résultat du `run-all-tests` à l'étape du code assembleur (Étape C).

Notre stratégie a été bénéfique, car tous nos tests ont réagi comme il le devait. Pour savoir si le test avait la bonne sortie, nous avons comparé sa sortie à un fichier portant le même nom finissant par `.exp`, si les deux sont identiques alors le test était réussi sinon il était marqué comme échoué.

1.4 Décompilation

Nous vérifions ensuite la décompilation. Sur nos tests, notre compilateur lance la commande `decac -p`. La décompilation est réussie si le résultat d'une décompilation d'un test et de sa décompilation reste inchangé. On peut voir ici que tous nos tests réussissent.

```

===== Décompilation =====
==> Test de décompilation des programmes deca
[Total : 66/66]

```

Figure 5: Résultat du `run-all-tests` à l'étape de décompilation.

1.5 Vérification

Au moment de la vérification, la commande `decac -v` est lancée sur les fichiers tests Deca. Les tests réussissent s'il n'y a aucun retour suite à l'exécution de la commande. Nous pouvons remarquer qu'ici, tous nos tests réussissent.

```

===== Vérification =====
==> Test que la sortie avec -v des programmes deca est vide
[Total : 66/66]

```

Figure 6: Résultat du `run-all-tests` à l'étape de vérification.

1.6 Registres

Enfin, nous vérifions que les tests Deca valides fonctionnent toujours avec une limitation de registres¹. Donc la commande `decac -r 4` est lancée sur plusieurs fichiers Deca. Nous avons d'ailleurs ici notre seule et unique erreur.

```

===== Registres =====
==> Test de la limitation des registres
[FAILED] : run_exmathdoc.deca
[Total : 62/63]

```

Figure 7: Résultat du `run-all-tests` à l'étape de limitation des registres.

¹cf Manuel d'utilisation pour plus de précisions à ce sujet.

1.7 Synthèse

En conclusion, notre stratégie pour l'élaboration des tests a été bénéfique. Cela nous a permis de détecter et de corriger de nombreuses anomalies. Ainsi, nous avons pu améliorer notre compilateur tout au long du processus.

```
===== Synthese =====  
Temps total d'exécution : 97.085 s  
Il y a eu 1 erreur(s)
```

Figure 8: Résultat du `run-all-tests` à l'étape de synthèse.

Nous pouvons voir cela, car tous nos tests sont passés correctement sauf pour un test. Il nous reste encore un bug pour la limitation de registre pour l'extension.

2 Les scripts de tests

Pour faire passer tous les tests, il faut lancer le script `run-all-tests.sh` qui se trouve dans le répertoire `tests` du projet, dans le dossier `script`. Vous aurez le rapport du nombre de tests réussis sur le nombre total de tests par catégorie, ces dernières étant spécifiées dans la section 1. Nous vous fournissons quelques précisions sur deux scripts en particulier : `jacoco-report.sh` et `run-all-tests.sh`.

2.1 jacoco-report

Le script `jacoco-report.sh` nous a servi au cours du projet afin de vérifier le niveau de couverture de nos tests sur l'ensemble de notre compilateur. Il a été modifié afin d'ignorer les potentielles erreurs lors du lancement du `mvn verify`, et le résultat que nous avons obtenu au moment du rendu est une couverture de tests de 95%, puis 100% peu après le rendu.

2.2 run-all-tests

Ce script, dont vous avez eu quelques aperçus dans la partie précédente, lance tous les tests Deca présents dans le répertoire, et vérifie que les tests réussissent. Seuls les tests qui échouent sont affichés. Ce script est sans aucun doute celui dont nous nous sommes le plus servi lors des phases de vérification de notre compilateur, puisqu'il permet d'avoir une indication précise des faiblesses de notre programme, et qu'il autorise l'ajout de nouveaux tests.

3 Gestion des risques et gestion des rendus

Dans le cadre du projet, nous avons configuré un Bot sur notre serveur Discord qui, à chaque push sur le Git, peut avoir deux réactions différentes (on ne comptera pas le cas où tout se passe bien). Si un push est effectué alors que la commande `mvn compile` renvoie **BUILD FAILURE**, le bot nous prévient que la version présente sur le dépôt Git ne compile pas. Cela nous a notamment permis de veiller à toujours effectuer un `mvn compile` avant chaque push afin de vérifier que l'on ne "cassait" pas notre compilateur avec les modifications que nous apportions. Puis, l'étape de vérification suivante était liée au `mvn verify`. Chacun de nos scripts de tests peut renvoyer deux sortie : 0 si tout s'est bien passé, 1 sinon. Si au moins un de nos scripts de tests renvoie 1, alors le bot nous prévient que la version du dépôt Git compile, mais ne passe pas tous les tests. Cela nous apporte beaucoup en terme de gestion des rendus, puisque nous étions assurés de rendre au moins un projet qui compile, à condition bien sûr de tenir compte des messages du bot Discord.

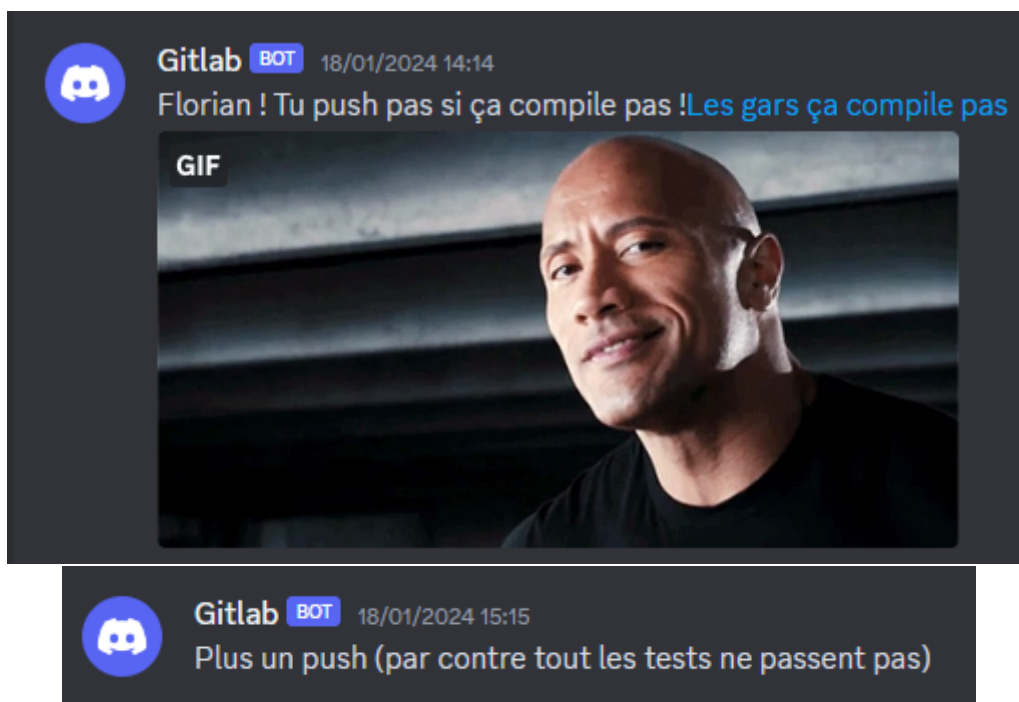


Figure 9: Illustration des réactions possibles du bot : le premier si ça ne compile pas (le bouc émissaire est consentant), le deuxième si ça compile mais que les tests ne réussissent pas tous.

En terme de gestion des risques, la vérification était moins automatisée cette fois, puisqu'il s'agissait surtout de vérifier le bon fonctionnement de notre base de tests. En effet, nous procédions à une double vérification. La première consistait à lancer le script `run-all-tests.sh`, et de comparer le résultat au précédent pour s'assurer que les modifications que chacun apporte génère autant ou moins d'erreurs. Puis, en cas d'erreur jugée inhabituelle, comme par exemple un test dont on est sûr du résultat mais qui persiste à faire une erreur qui ne semble pas en rapport avec un bug du compilateur, chacun scrutait le test concerné en détail afin de vérifier les potentielles erreurs de rédaction du code. Cela nous a été utile puisque nous avons pu non seulement corriger des erreurs dans notre base de tests, mais aussi ajouter des tests sur des points auxquels nous n'avions pas pensé initialement, améliorant ainsi notre couverture de tests. Finalement on peut dire que nous avons tous été des testeurs durant le projet, ce qui est un plus par rapport à laisser une seule personne en charge

de tous les tests. Cela a certainement joué sur la qualité de notre couverture de tests puisque nous étions bien plus exhaustifs.

Enfin, même si nous connaissons les résultats attendus de nos tests, il est malgré tout facile de les oublier, et les recalculer aurait été une perte de temps énorme, en plus de multiplier les chances de faire une erreur de calcul, et donc de chercher un bug qui n'existe pas. C'est pourquoi dans les tests valides du répertoire codegen notamment, nous avons ajouté des fichiers `.exp` qui contiennent les résultats attendus des tests, et qui sont comparés aux résultats obtenus lors du lancement de `run-all-tests.sh`.

4 Résultats de Jacoco

Lors de l'élaboration de notre batterie de tests, nous avons utilisé l'outil jacoco pour nous permettre de pouvoir tester chacune des fonctionnalités que nous avons codées pour notre compilateur. Grâce à celui nous avons pu atteindre le pourcentage de 95% pour le rendu puis après le rendu nous avons atteint 100%.

Deca Compiler												Sessions
Deca Compiler												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		74%		56%	479	679	530	2,055	261	380	3	50
fr.ensimag.trigo		0%		0%	50	50	130	130	19	19	2	2
fr.ensimag.deca		68%		72%	26	72	58	212	7	36	2	6
fr.ensimag.ima.pseudocode		77%		75%	30	90	44	191	24	78	1	26
fr.ensimag.deca.context		84%		75%	27	142	36	241	18	116	0	21
fr.ensimag.deca.codegen		84%		82%	12	70	22	163	4	39	0	1
fr.ensimag.deca.tools		75%		60%	10	38	15	79	2	23	0	5
fr.ensimag.ima.pseudocode.instructions		76%		n/a	16	62	28	111	16	62	12	54
fr.ensimag.deca.tree		100%		95%	21	688	0	1,732	0	461	0	85
Total	4,400 of 22,882	80%	378 of 1,288	70%	671	1,891	863	4,914	351	1,214	20	250

fr.ensimag.deca.tree												
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
DeclClass		100%		83%	2	20	0	87	0	14	0	1
DeclMethod		100%		93%	1	19	0	78	0	11	0	1
CallMethod		100%		92%	2	20	0	74	0	7	0	1
Identifier		100%		95%	2	45	0	71	0	22	0	1
DeclField		100%		93%	2	26	0	75	0	10	0	1
GetAttribut		100%		94%	1	29	0	70	0	20	0	1
Program		100%		100%	0	10	0	60	0	9	0	1
Tree		100%		100%	0	27	0	60	0	20	0	1
Cast		100%		83%	3	15	0	44	0	6	0	1
IFThenElse		100%		100%	0	11	0	55	0	8	0	1
Return		100%		95%	1	17	0	44	0	7	0	1
AbstractExpr		100%		91%	1	21	0	49	0	15	0	1
InstanceOf		100%		100%	0	7	0	28	0	5	0	1
ListDeclClass		100%		100%	0	14	0	36	0	8	0	1
BinShift		100%		90%	1	10	0	35	0	5	0	1
ListDeclField		100%		100%	0	10	0	28	0	5	0	1
AbstractPrint		100%		100%	0	17	0	29	0	8	0	1
DeclVar		100%		100%	0	9	0	40	0	6	0	1
AbstractBinaryExpr		100%		90%	1	14	0	39	0	9	0	1
New		100%		100%	0	8	0	27	0	6	0	1
StringLiteral		100%		100%	0	13	0	26	0	9	0	1
MethodBody		100%		100%	0	12	0	41	0	7	0	1
While		100%		100%	0	9	0	40	0	8	0	1
Assign		100%		100%	0	10	0	24	0	5	0	1
AbstractOpArith		100%		92%	1	10	0	23	0	3	0	1
AbstractOpCmp		100%		100%	0	11	0	20	0	3	0	1
DeclParam		100%		100%	0	8	0	24	0	6	0	1
Main		100%		100%	0	8	0	33	0	7	0	1
TreeList		100%		100%	0	14	0	23	0	11	0	1
AbstractUnaryExpr		100%		87%	1	11	0	24	0	7	0	1
Initialization		100%		100%	0	9	0	22	0	6	0	1
FloatLiteral		100%		100%	0	11	0	17	0	9	0	1
ListDeclParam		100%		100%	0	8	0	21	0	4	0	1
ListDeclMethod		100%		100%	0	9	0	17	0	5	0	1

Figure 10: Résultat Jacoco































































 ListExpr		100%		100%	0	4	0	7	0	2	0	1
 And		100%		n/a	0	3	0	7	0	3	0	1
 Or		100%		n/a	0	3	0	7	0	3	0	1
 AbstractDeclParam		100%		n/a	0	4	0	9	0	4	0	1
 ConvInt		100%		n/a	0	4	0	7	0	4	0	1
 Lower		100%		n/a	0	3	0	6	0	3	0	1
 Equals		100%		n/a	0	3	0	6	0	3	0	1
 GreaterOrEqual		100%		n/a	0	3	0	6	0	3	0	1
 Greater		100%		n/a	0	3	0	6	0	3	0	1
 NotEquals		100%		n/a	0	3	0	6	0	3	0	1
 LowerOrEqual		100%		n/a	0	3	0	6	0	3	0	1
 ConvFloat		100%		n/a	0	4	0	7	0	4	0	1
 AbstractOpIneq		100%		n/a	0	2	0	5	0	2	0	1
 Plus		100%		n/a	0	3	0	5	0	3	0	1
 Minus		100%		n/a	0	3	0	5	0	3	0	1
 PrintIn		100%		n/a	0	3	0	6	0	3	0	1
 Multiply		100%		n/a	0	3	0	5	0	3	0	1
 Visibility		100%		n/a	0	1	0	3	0	1	0	1
 NoInitialization		100%		100%	0	8	0	9	0	7	0	1
 NoOperation		100%		n/a	0	6	0	7	0	6	0	1
 EmptyMain		100%		n/a	0	7	0	7	0	7	0	1
 Print		100%		n/a	0	2	0	3	0	2	0	1
 AbstractInst		100%		n/a	0	2	0	3	0	2	0	1
 AbstractOpExactCmp		100%		n/a	0	1	0	2	0	1	0	1
 AbstractDeclMethod		100%		n/a	0	1	0	1	0	1	0	1
 AbstractProgram		100%		n/a	0	1	0	1	0	1	0	1
 AbstractDeclField		100%		n/a	0	1	0	1	0	1	0	1
 AbstractReadExpr		100%		n/a	0	1	0	2	0	1	0	1
 AbstractLValue		100%		n/a	0	1	0	1	0	1	0	1
 AbstractDeclClass		100%		n/a	0	1	0	1	0	1	0	1
 AbstractMethodBody		100%		n/a	0	1	0	1	0	1	0	1
 AbstractStringLiteral		100%		n/a	0	1	0	1	0	1	0	1
 AbstractDeclVar		100%		n/a	0	1	0	1	0	1	0	1
 AbstractIdentifier		100%		n/a	0	1	0	1	0	1	0	1
 AbstractInitialization		100%		n/a	0	1	0	1	0	1	0	1
 AbstractMain		100%		n/a	0	1	0	1	0	1	0	1
Total	0 of 7,643	100%	21 of 454	95%	21	688	0	1,732	0	461	0	85

Figure 11: Résultat Jacoco