École nationale de l'informatique et des mathématiques appliquées de Grenoble

Projet Génie Logiciel

Manuel Validation
Année universitaire 2020-2021

Groupe 8 Equipe 40
Étudiants 2A

Table des matières

1 Descriptif des tests

1.1 Types de tests

1.1.1 Étape A

Les tests de l'étape A ont été réalisés dans le but de guider l'implémentation. En effet, surtout mis en place lors de la création du parser pour les objets, nous avons d'abord créé le test que l'on voulait passer puis nous avons étudié et travaillé le parser pour que ce test précisément s'exécute. Nous avons donc créé de nombreux tests simples pour avoir les bases du parser puis nous avons pu mettre en place des tests un peu plus complets et complexes pour s'assurer de la complétude de notre code.

Cette étape a aussi été testée lors des tests des parties suivantes, c'est pourquoi certaines particularités du compilateur n'ont pas été explicitement testées pour la partie A.

Tous les tests ont en leur en-tête une description rapide de leur contenu et le résultat ou l'erreur attendue.

Pour la partie *hello world*, le script *lex-hello* vérifie que les scripts passent correctement *test_lex* et vérifie qu'il n'y a pas de message d'erreur ou affiche le bon message d'erreur. Le script *parser-hello* quant à lui, vérifie que *test_synt* passe sur les fichiers valides, que le résultat de ces fichiers valides est le bon (modèles disponibles pour la comparaison), et que les fichiers invalides soient invalides avec l'erreur attendue (erreur disponible en première ligne du fichier pour la comparaison).

Pour la partie sans objet, un répertoire sansobjet les regroupe. Pour les tests spécifiques au lexer, ceux-ci contiennent dans leur nom *_invalid* ou *_valid* car ils peuvent être valides pour le lexer mais pas pour le parser. Les scripts *lexer-so* et *parser-so* vont comparer la sortie avec les fichiers de résultat et voir s'il n'y a pas de différence. Si ce n'est pas le cas, la différence est affichée avec le résultat attendu (contenu dans l'entête des fichiers tests) . Pour les fichiers non valides pour le lexer et le parser, on regarde si le code erreur est contenu dans l'entête du fichier test et si ce n'est pas le cas, l'erreur et le résultat attendu sont retournés.

Pour la partie objet, les scripts *lexer-o* et *parser-o* sont disponibles et sont de la même forme que les scripts précedemment décrits.

Pour la partie extension, seul un script *parser-ext* est disponible, les tests lexer étant testés finalement en même temps que la vérification syntaxique.

Si on veut rajouter des tests :
Dans le cas où on veut rajouter un test à ceux déjà présents pour **le lexer** : si le test n'est pas valide du point de vue syntaxique, il doit être mis dans le dossier *invalide* puis *partie_concernée*. Si ce test est valide, il faut y joindre un fichier nommé *nom_du_fichier.deca.lex.ok* qui servira

de modèle lors du passage du sript.
Dans le cas où on veut rajouter un test à ceux déjà présents pour **le parser** : si le test est valide on le met dans le dossier *valide* puis *partie_concernée* et on y joint un fichier *nom_du_fichier.deca.ok* qui servira de modèle lors du passage du script. Dans le cas d'un test invalide, on le met dans le dossier *invalide* puis *partie_concernée* et on met en première ligne du fichier *\\erreur obtenu* pour pouvoir passer le script.

1.1.2 Étape B

Les tests de l'étape B sont formés de deux types de tests : les tests valides qui regroupent les programmes devant fonctionner et ne devant rien renvoyer. Les tests invalides testant les erreurs de contexte fournis par la grammaire sont disponibles dans *ContextualError.java*. Pour chaque erreur possible, un fichier de test *.deca* est créé avec en première ligne commentée l'erreur de *ContextualError* attendue. Le script d'automatisation *context-so-err.sh* va alors prendre tous les fichiers invalides, lancer *test_context* et vérifier que l'erreur en sortie est la même que celle donnée par la première ligne du fichier test. Le script va ensuite vérifier les fichiers valides en cherchant les sorties d'erreurs pour signaler toute erreur inattendue. Les tests unitaires permettent de vérifier les fonctionnalités propres à chacune des classes de *Tree*. Ils permettent également de vérifier des comportements compliqués à déceler avec un fichier *.deca*.

Les tests concernant la partie décompilation ont été faits pour se comparer à un fichier résultat déjà implémenté. Ainsi, le script va exécuter la commande *decac -p* sur chaque fichier test de décompilation et le comparer à son modèle prévu. Comme la décompilation se base seulement sur le lexer et le parser, il n'y a pas besoin de faire des tests valides et invalides, puisque c'est déjà pris en compte dans les tests de l'étape A. Il est seulement nécessaire de s'assurer que la décompilation est telle que nous l'aurions attendu (d'où les fichiers modèles dans le dossier *src/test/deca/decompile/* avec le nom *.deca.ok*).

Tous les tests ont en leur en-tête une description rapide de leur contenu et le résultat ou l'erreur attendue.

Un script *decompilation* est disponible pour tester tous les fichiers dans le dossier du même nom. Il décompile les fichiers à l'aide de *decac -p* puis les comparent à leur modèle respectif *.deca.ok*. Un script *Redecompile* est aussi disponible. Il utilise un fichier intermédiaire pour faire deux décompilations à la suite, un fichier intermédiaire qui est à la fin supprimé (pour ne pas gêner l'exécution d'autre script notamment du script *decompilation*).

Un script *context-...-err* (à adapter selon la partie, o pour objet, so pour sans objet, ext pour extension) est disponible pour tester la partie de vérification contextuelle. Dans ce script, les fichiers valides sont exécutés par *test_context_nodebug* (un dérivé de *test_context* fait pour ne pas afficher les lignes de débug) et leur résultat est comparé avec les fichiers modèles *nom_du_fichier.deca.ok*. Les fichiers invalides sont exécutés et leur erreur est comparée avec l'erreur attendue (disponible en première ligne du fichier test).

Si on veut rajouter des tests :
Dans le cas où on veut rajouter un test à ceux déjà présents pour **la décompilation** : un seul et même dossier est disponible pour les tests de décompilation. Aucune différence n'est faite entre les fichiers tests valides et invalides car la décompilation repose sur l'exécution du parser et du lexer, tous les fichiers mis dans le dossier test de décompilation sont donc valides. Chaque fichier ajouté doit avoir un fichier modèle contenant le résultat attendu et portant le nom *nom_du_fichier.deca.ok*.
Dans le cas où on veut rajouter un test à ceux déjà présents pour **la vérification contextuelle** : si le test est invalide, on met l'erreur attendue en commentaire en première ligne du fichier test (*erreur attendu* et on le met dans le dossier *context\invalid\partie_concernée*

1.1.3 Étape C

Pour l'étape C, les tests ont été réalisés un peu différemment. Étant la dernière étape du processus, nous avons utilisé les deux étapes précédentes (une fois celles-ci testées et validées) pour tester l'ensemble A-B-C. Nous avons écrit des programmes deca simples ayant dans un premier temps pour but de vérifier des points très précis (déclarations de variables, avec ou sans initialisation, opérations une par une avec les différentes possibilités.. etc) puis dans un second temps nous avons écrit quelques tests plus complets, ressemblant davantage à un programme réaliste (par exemple un programme de factorielle) pour tester l'ensemble.

Une fois les premiers tests passés, et en particulier durant la partie objet, il a été nécessaire d'adapter les tests aux changements opérés dans le code. En effet, nous avons essayé de diversifier les tests, une fois un problème trouvé il était nécessaire de le modifier puis de tester le plus d'alternatives possibles pour certifier que le changement de code était valide.

Les tests de la partie C, mais aussi des parties précédentes, sont donc parfois répétitifs car certains tests ont été conçus au début du code, pour mettre en place le programme, et d'autres au cours du débogage du code.

Tous les tests ont en leur en-tête une description rapide de leur contenu et le résultat ou l'erreur attendue.

Les scripts *gencode-...* (à adapter selon la partie, objet, so pour sans objet avec spécification de valide ou invalide, ext pour extension) sont disponibles pour les tests de la partie génération de code. Ils testent tout d'abord que les tests valides passent l'exécution de *decac* puis l'exécution de *ima*, puis ils comparent que le résultat obtenu est le bon avec un fichier modèle contenant le résultat attendu sous le nom de *nom_du_fichier.ass.ok*. Les tests invalides quant à eux sont testés d'abord à l'exécution de *decac* (qu'ils doivent passer, sinon ce sont des test invalides de contexte) puis l'erreur relevée lors de l'éxectuion de *ima* est comparée à celle attendue. L'erreur attendue est disponible dans le fichier *nom_du_fichier.ass.ok* correspondant (ou le fichier *MessageErreur.ok* dans le cas où plusieurs tests donnent la même erreur résultat).

Le script *codegen-less-register* teste le contenu de tous les scripts de génération de code mais avec l'option -r de decac pour limiter le nombre de registres.

Dans les scripts disponibles, certains tests invalides sont dans des sous-dossiers (type *debordementPile*) car leur exécution nécessite une particularité du type option à rajouter lors de l'exécution de *decac*.

Si on veut rajouter des tests :
Dans le cas où on veut rajouter un test à ceux déjà présents pour **la génération de code :** si le test est valide, on le place dans le dossier *codegen \valid \nom_partie_concernée* et on y joint un fichier modèle contenant le résultat désiré à l'exécution de *ima* sous le nom *nom_du_fichier.ass.ok*. Si le test est invalide, on le place dans le dossier *codegen \valid \nom_partie_concernée* avec un fichier contenant l'erreur attendue sous le nom *nom_du_fichier.ass.ok* après l'exécution de *ima*. Si ce test invalide a besoin d'une particularité à l'exécution, on le place dans le sous dossier correspondant ou, dans le cas où le sous dossier correspondant n'existe pas encore, on le crée et on rajoute l'exécution particulière dans le script correspondant.

1.1.4 Extension

Les tests pour l'extension sont regroupés dans les dossiers nommés *extension*.

Les tests concernant les bibliothèques sont dans le dossier nommé *bibliothèque*

Les scripts décris précédemment sont applicables à l'extension (présence des scripts *gencode-ext*, *parser-ext* et *context-ext-err*).

Un script *biblio* est disponible pour tester les fichiers tests présents dans le dossier *codegen\valid\biblio* et comparer la valeur obtenue avec celle attendue disponible dans le fichier associé finissant par *.ass.ok*.

Si on veut rajouter des tests :
Dans le cas où on veut rajouter un test à ceux déjà présents pour **l'extension** : il faut suivre les directives données dans la mise en place de nouveaux tests dans les parties A, B et C (voir parties précédentes). Dans le cas où on veut rajouter un test à ceux déjà présents pour **les bibliothèques** : si le test est valide, on l'ajoute dans le dossier correspondant et on lui joint un fichier modèle contenant la valeur attendue après exécution de *ima* portant le nom *nom_du_fichier.ass.ok*. Dans le cas d'un fichier invalide, il faudra créer le dossier invalide et ajouter une partie d'évaluation des erreurs des tests invalides dans le script (possible de s'inspirer des tests de génération de code déjà établis).

1.2 Organisation des tests

Chaque partie (A, B et C) ainsi que décompile ont été faites plus ou moins indépendamment les unes des autres (des personnes différentes s'en chargeaient en même temps). Chaque partie a donc été testéz par une batterie de tests résumée dans un script propre à la partie.

Ainsi, le projet comprend différentes batteries de tests, chacune disponible dans un sous fichier du dossier *src/test/deca* (par exemple les tests de context pour le sans objet seront disponibles dans les dossiers *src/test/deca/context/valid/sansobjet* et *src/test/deca/context/invalid/-sansobjet*).

De plus, certains tests sont mis dans des sous dossiers particuliers car leur exécution est particulière. On retrouve ainsi des sous dossiers pour des tests avec certaines options de decac par exemple.

1.3 Objectifs des tests

La création de nos tests s'est faite dans deux buts différents.

Le premier but était de faire un test puis de coder pour que ce test passe (comme expliqué pour les tests de la partie A).

Le second était de mettre en valeur certains cas spécifiques qui pouvaient difficilement être résolus par notre projet afin de s'assurer que notre travail soit valide dans ces cas spéciaux.

2 Scripts de tests

2.1 Comment faire passer tous les tests

Tous les scripts créés au cours du projet ont été ajoutés au test déjà existants dans le fichier *pom.xml* pour pouvoir s'exécuter lors de l'exécution de la commande *mvn test*.

Ces tests sont aussi tous disponibles pour être lancés de manières unitaires, par exemple dans le cas où un script ne passe pas mais que l'on veut pouvoir étudier les autres.

2.2 Organisation des scripts

Chaque script comporte au début du fichier .sh une description du script de test ainsi que des actions qui y sont implémentées. Dans la grande majorité des scripts, l'organisation est la suivante : dans un premier temps on exécute les fichiers valides en vérifiant que aucune erreur n'est relevée ; dans un second temps on compare le résultat des fichiers valides avec un modèle pré établie et vérifié ; dans une troisième et dernière partie on exécute les fichiers invalides en vérifiant que l'erreur relevée est la bonne (l'erreur attendue est soit en première ligne du fichier test, soit dans un fichier modèle).

3 Gestion des risques et gestions des rendus

3.1 Liste des dangers critiques

Voici une liste des dangers critiques que notre équipe peut rencontrer au sein de ce projet et des solutions que nous avons imaginé pour les résoudre :

- **Problème de configuration du projet sur son ordinateur personnel :** entraide entre les différents membres de l'équipe pour que chacun ait la configuration attendue dès le deuxième jour du projet, sachant que cela a mis plus ou moins de temps selon les membres.
- **Manque de communication** : Utilisation de Discord pour communiquer à distance et création de différents canaux (parser, lexer, étape B, étape C, général) pour ordonner la communication. Motivation de chaque membre à poser ses questions pour s'assurer de la bonne compréhension du sujet et des attentes.
- **Manquer une date de rendu :** utilisation d'un agenda et discussion en permanence résumant qui travaille sur quoi et où en sont chaque partie du projet (permet de savoir si le travail avance comme nous l'avions prévu sur le planning).
- **Ne pas avoir fini le travail en temps et en heure :** utilisation régulière du planning et répartition perpétuelle des tâches encore à faire (lorsqu'une personne finit son travail actuel, elle l'annonce, propose son aide aux autres membres de l'équipe, en cas de refus elle choisit une nouvelle partie à développer et avance dessus).
- **Doubler/Répéter notre code :** chaque membre de l'équipe annonce son travail régulièrement. Si ce dernier est terminé, il annonce la prochaine partie sur laquelle il va travailler et attend la réponse des autres membres de l'équipe (évite de travailler sur la même partie sans le savoir).
- **Le programme ne passe pas les tests "simples" :** développement des tests en même temps que la programmation (on code certaines parties dans le but de passer un test et non pas l'inverse). Certains de nos tests simples se croisent ce qui permet de s'assurer que chaque partie développée passe les tests "simples" (exemple : programme constitué seulement d'un print, ou seulement d'une déclaration d'une variable).
- **Le programme ne passe pas mvn test :** test du code de chaque partie avec mvn test avant de push dans une branche git majeure (sauf pour les parties qui n'ont pas encore le code nécessaire à la réalisation de certain tests, par exemple : les parties A et B qui n'ont pas le code nécessaire à la bonne réalisation de basic-context).
- **Toutes les fonctionnalités de notre projet ne fonctionnent pas ensemble :** à chaque partie développée on crée un fichier bash qui automatise cette partie de test, on ajoute ce script à mvn test. Si une nouvelle partie provoque une régression dans une autre partie, cela se remarque lors de l'exécution de mvn test avant de push sur une branche importante de notre git.
- **Les fichiers tests sont faux :** Nous essayons de mettre des personnes différentes sur l'écriture des scripts de tests. Ainsi, nous essayons de diversifier nos tests mais aussi de s'assurer que si un script n'est pas valide, les autres scripts puissent quand même être fonctionnels.

- **Les fichiers de tests ne sont pas révélateurs des problèmes possibles et de l'ensemble du code :** vérification des tests par les pairs et vérification de la couverture de ceux-ci par Jacoco. Discussion permanente entre tous les membres de l'équipe pour réfléchir à des tests qui couvrent un maximum les programmes.

3.2 Check-up rendu

- Lancer mvn clean, mvn compile, mvn test-compile, mvn test.
- S'assurer d'être sur la branche du dernier commit concernant le rendu Cloner à nouveau le dépôt git pour s'assurer que tous les fichiers soient bien sur la branche en question.
- 24h avant le rendu, on s'assure que tous les fichiers à modifier sont modifiés, on ne change aucun fichier majeur le jour du rendu pour ne pas "casser" notre compilateur (si une erreur minime est trouvée elle peut être modifiée, en s'assurant que les résultats restent tous valides).

4 Résultats de Jacoco

Voici le résultat de *mvn verify*

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		78 %		60 %	630	928	492	2 379	275	423	2	53
fr.ensimag.deca.tree		90 %		78 %	172	870	189	2 301	47	542	0	90
fr.ensimag.deca		69 %		52 %	62	140	129	411	15	74	2	5
fr.ensimag.ima.pseudocode		77 %		75 %	28	89	41	189	23	79	1	27
fr.ensimag.deca.context		89 %		82 %	36	181	33	281	21	138	0	25
fr.ensimag.ima.pseudocode.instructions		69 %		n/a	22	64	38	115	22	64	18	54
fr.ensimag.deca.tools		93 %		100 %	1	16	3	37	1	13	0	3
fr.ensimag.deca.codegen		98 %		97 %	3	50	2	139	2	33	0	2
Total	4 406 of 26 801	83 %	590 of 1 856	68 %	954	2 338	927	5 852	406	1 366	23	259

Ce que nous avons retenu de cette analyse par jacoco est que notre couverture de tests est raisonnable. En particulier, lorsqu'on s'intéresse à *deca.tree*, qui est, pour nous, la partie la plus révélatrice de nos tests car elle montre les classes utilisées par nos tests, on obtient :

fr.ensimag.deca.tree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
NewArray		70 %		54 %	16	31	31	117	0	8	0	1
Identifier		71 %		83 %	7	39	23	92	2	24	0	1
AbstractExpr		77 %		65 %	14	46	15	89	4	26	0	1
ArrayLiteral		87 %		80 %	8	31	12	95	2	11	0	1
Tree		78 %		81 %	6	30	13	64	4	22	0	1
ArraySelection		90 %		84 %	6	23	10	88	2	10	0	1
AbstractOpCmp		81 %		69 %	9	22	6	41	0	4	0	1
ReadInt		32 %		n/a	2	7	8	15	2	7	0	1
ReadFloat		32 %		n/a	2	7	8	15	2	7	0	1
DeclClass		94 %		69 %	14	34	5	116	0	11	0	1
FloatLiteral		73 %		33 %	5	15	8	27	2	12	0	1
BooleanLiteral		75 %		40 %	5	16	7	23	2	11	0	1
IdentifierArray		82 %		100 %	9	17	9	30	9	15	0	1
InstanceOf		95 %		71 %	4	14	2	62	0	7	0	1
Cast		95 %		67 %	9	20	2	58	0	6	0	1
UnaryMinus		76 %		75 %	2	7	3	13	1	5	0	1
TreeList		89 %		100 %	2	15	2	26	2	12	0	1
LocationException		85 %		60 %	4	9	2	14	0	4	0	1
ConvFloat		85 %		75 %	3	7	4	17	2	5	0	1
Modulo		84 %		50 %	2	7	1	14	0	4	0	1
AbstractOpArith		95 %		90 %	4	17	4	45	2	6	0	1
MethodCall		97 %		70 %	3	13	1	68	0	8	0	1
DeclField		97 %		83 %	2	14	1	54	0	8	0	1
AbstractOpBool		93 %		50 %	2	5	1	19	0	3	0	1
DeclMethod		99 %		84 %	9	35	2	129	2	13	0	1
Location		92 %		50 %	2	8	2	14	0	6	0	1
MethodAsmBody		92 %		n/a	1	8	1	16	1	8	0	1
DeclParam		97 %		100 %	1	9	1	32	1	8	0	1
AbstractPrint		97 %		100 %	1	12	1	29	1	9	0	1
Return		95 %		100 %	2	9	2	22	2	8	0	1
Initialization		95 %		n/a	1	9	1	22	1	9	0	1
AbstractInst		84 %		n/a	1	4	1	6	1	4	0	1
AbstractBinaryExpr		98 %		50 %	1	9	0	27	0	8	0	1
IfThenElse		100 %		100 %	0	11	0	63	0	7	0	1
Assign		100 %		83 %	3	14	0	44	0	5	0	1
While		100 %		n/a	0	9	0	47	0	9	0	1
DeclVar		100 %		100 %	0	10	0	47	0	8	0	1
Selection		100 %		92 %	1	16	0	43	0	9	0	1

Ce qu'on remarque dans *deca.tree* c'est que, mise à part ReadInt et ReadFloat qui ne peuvent pas être testés directement dans le mvn verify, toutes les classes de notre programme sont plus ou moins testées, avec un minimum a 70% de couverture de tests. Ces chiffres qui ne sont pas au maximum s'expliquent par le fait que certains cas "extrêmes" n'ont pas été testés, nous cherchions en permanence des tests ambiguës mais cela est certain qu'une partie de tests complexes nous ont échappé.

En conclusion de ce *mvn verify*, nous sommes satisfaits de notre couverture de tests mais nous sommes conscients que cette couverture peut toujours être améliorée. Nous avons essayé au mieux de créer nos tests de manières diverses par tous les membres de notre équipe.

5 Méthodes de validation utilisées

Les méthodes de validation utilisées autre que les tests ont variées d'une partie à l'autre.

5.1 Pour la partie A

La partie A a été testée au travers des tests. De plus, lors de la première étape du langage (*Hello World*) presque tous les membres de l'équipe se sont familiarisés avec le lexer et/ou le parser. Il a été donc très simple que chacun donne son avis sur la partie A. Cette partie a surtout été validée par la suite lors de la mise en place des autres parties du projet. Si on prend

l'exemple de l'utilisation de *setLocation* dans la partie 1, la mauvaise utilisation ou l'oublie de cette fonction à certains endroits a été découvert lors de la mise en place de l'étape B.

5.2 Pour la partie B

En plus des tests, l'étape B a pu être validée et modifiée lors de la mise en place de l'étape C. En effet, certaines réflexions lors de la mise en place de l'étape de génération de code ont engendré une modification ou une simple validation de la partie B.

5.3 Pour la partie C

La partie C a majoritairement été testé par tests et scripts. La complexité de cette partie empêchait souvent la libre parole au sein de l'équipe, certains membres n'ayant jamais codé directement sur cette partie. Nous n'avons pas utilisé d'autre moyens importants que les tests et nous pensons que cette partie aurait une meilleure validation si nous avions une plus grande et plus complexe batterie de tests. Nous sommes conscients que, dans le cas d'une mise en service de notre compilateur, c'est cette partie en particulier qu'il faudrait fréquemment mettre à jour selon le ressenti et les expériences des utilisateurs.

5.4 Pour les scripts et tests

Pour cette partie, *mvn test* était fréquemment lancé pour être sûre que tous les tests passent. Mais cela n'a pas été la seule méthode de validation concernant les scripts et les tests. En effet, à chaque changement, les scripts étaient d'abord lancés de manière indépendante, s'ensuivait l'exécution de *mvn verify* pour être sûre que les pourcentages de couverture des tests n'avaient pas été modifiées. Afin de s'assurer que les tests étaient tous passés et qu'aucun fichier en surplus ne sera mis sur le git, un *find -inam "*.ass"* est réalisé (les scripts sont écrits pour effacer les .ass après leur utilisation).

De plus, pour s'assurer que les tests étaient assez diversifiés, chaque membre de l'équipe proposait des tests auxquels il pensait, qu'il ait travaillé sur la partie ou non.

5.5 Pour l'extension

L'extension a permis de revoir certains points à modifier dans les parties précédentes. Cependant, par manque, la partie en elle-même est sans doute celle qui a été la moins validée. C'est un point que nous aurions aimé modifier si nous avions eu un peu plus de temps à réserver à l'extension, partie finale du projet.

Cette partie du projet a dû être commencée avant d'avoir l'entièreté du projet de base, il a donc été nécessaire de tester nos bibliothèques sur d'autres langage que le langage deca (notamment java) afin d'être sûr que nos algorithmes étaient opérationnels. Lorsque le compilateur a

été terminé pour la totalité du langage deca, les bibliothèques ont pu être traduites en deca.