Projet Génie Logiciel

Compilateur Deca

Documentation Validation

Auteurs :
M. Benjamin ARGENSON
M. Jean-Baptiste LEFOUL
M. Thomas POUGET-ABADIE
M. Robinson PRADA MEJIA
M. Alexandre PROY

Encadrants :
Pr. Catherine ORIAT
Pr. Philippe BODIGLIO

Version du 26 janvier 2017

Table des matières

1 Descriptif des tests

1.1 Type de tests

Au cours de ce projet, notre travail n'a pas consisté à implémenter le **lexer** puis le **parser** puis les méthodes de vérifications contextuelles pour ensuite terminer par les méthodes de génération de code. Nous avons décidé de départager le travail en fonctionnalités perceptibles du point de vue utilisateur.

En se plaçant du point vue de l'utilisateur, nous avons décidé de faire principalement des tests d'intégration. Ainsi, chaque fichier de test est un fichier .deca valide ou invalide.

Lorsque nous parlons de fichiers .deca valides, nous faisons référence à des fichiers pouvant être compilés et exécutés sans erreur. Ainsi, un tel fichier doit pouvoir passer à travers le **lexer**, **parser**, les vérifications contextuelles et l'exécution par **ima** sans erreur.

Lorsque nous parlons de fichiers .deca invalides, nous faisons référence à des fichiers pouvant lever une erreur soit au niveau du **lexer**, soit au niveau du **parser**, soit au niveau des vérifications contextuelles, soit lors de l'exécution par **ima**.

1.2 Organisation des tests

Nos fichiers de tests sont organisés en plusieurs dossiers au sein de **src/test/deca/** :

- **syntax/**
    - **syntax/valid/**
        - **syntax/valid/our_tests**
        - **syntax/valid/synt-results**
        - **syntax/valid/lex-results**
    - **syntax/invalid/**
        - **syntax/invalid/our_synt_tests**
        - **syntax/invalid/our_lex_tests**
        - **syntax/invalid/synt-results**
        - **syntax/invalid/lex-results**
- **context/**

- **context/valid/**
    - **context/valid/our_tests**
    - **context/valid/cont-results**
- **context/invalid/**
    - **context/invalid/our_tests**
    - **context/invalid/cont-results**

- **codegen/**
- **codegen/valid/**
    - **codegen/valid/our_tests**
    - **codegen/valid/exe-results**
- **codegen/invalid/**
    - **codegen/invalid/our_tests**
    - **codegen/invalid/exe-results**
- **codegen/interactive/**

- **bytegen/**
- **bytegen/valid/**
    - **bytegen/valid/bc-results**
- **bytegen/invalid/**
    - **bytegen/invalid/bc-results**

Il est important de noter que les dossiers valides sont tous identiques. En effet, comme nous l'avons précédemment expliqué, tout fichier .deca valide l'est pour toute étape de la compilation et il est donc inutile de distinguer les tests valides comme étant spécifiques à certaines étapes.

Le dossier **syntax** contient l'ensemble des tests spécifiques à la première étape de la compilation. Puisque cette première étape repose sur le **lexer** ainsi que le **parser**, nous avons créé des dossiers spécifiques à ces deux sous-étapes. Dans les dossiers **synt-results** ainsi que **lex-results**, nous stockons les résultats corrects du **lexer** et du **parser** respectivement appliqués à l'ensemble de nos fichiers valides contenus dans le dossier **our_tests**.

A titre d'exemple, si nous disposons d'un fichier test **hello.deca**, nous trouverons un fichier **hello-result.txt** dans le dossier **lex-results** et un autre fichier du même nom mais contenant le résultat correct du parser dans le dossier **synt-results**.

Le dossier **context** contient l'ensemble des tests spécifiques à la deuxième étape de la compilation : les vérifications contextuelles.

Le dossier **codegen** contient l'ensemble des tests spécifiques à la troisième étape de la compilation : la génération de code. Pour tester cette troisième étape, nous avons décidé de nous baser sur le résultat d'**ima** sur le fichier .ass et non pas sur le code .ass obtenu. En effet, alors que l'implémentation des méthodes de génération de code peut subir des modifications induisant des différences au sein d'un fichier .ass généré pour un fichier .deca, le résultat d'**ima** sur ce fichier .ass ne devrait pas changer.

Le dossier **bytegen** contient l'ensemble des tests spécifiques à notre extension. Il est important de noter que notre extension reprend les méthodes de la première et deuxième étape de la compilation et ne nécessite donc pas de tests spécifiques pour ces étapes. Le dossier **bytegen** est donc analogue à **codegen**.

1.3 Objectifs des tests

Nos tests sont essentiellement des tests de non régression. C'est pourquoi nous stockons les résultats corrects des fichiers .deca ayant subis les différentes étapes de la compilation. De cette manière, à la suite de l'implémentation d'une nouvelle fonctionnalité, relancer la totalité des tests nous permet de vérifier qu'aucune autre partie du code a été affecté. Lors du lancement des tests, nos scripts calculerons le résultat d'un fichier .deca avec les lançeurs **test_lex**, **test_synt** et **test_context**, avec le compilateur par la commande **decac** et par **ima**. Ces résultats seront ensuite comparés aux résultats déjà stockés. Toute différence sera considérée comme un **FAIL** et un **PASS** sinon.

2 Scripts de tests

2.1 Exécution des tests

Afin d'exécuter l'ensemble de nos tests, nous avons crée plusieurs scripts python contenus dans le dossier **src/test/deca/script**.

- **basic-lex.py**

Ce script permet d'exécuter l'ensemble de tests **lexer** valides et invalides. Le script commence par exécuter les tests valides. Pour ce faire, il utilise **test_lex** afin de calculer l'ensemble des résultats des fichiers valides .deca contenus dans le dossier **valid**. Ces résultats sont stockés dans un dossier **test-results** crée par le script. Ensuite, le script compare ces résultats aux résultats stockés dans **valid** et considérés comme corrects. Toute différence est considérée comme un test **FAIL**, le cas contraire comme un test **PASS**. Ce même processus est utilisé pour les fichiers invalides.

- **basic-synt.py**

Ce script est analogue au précédent et permet d'exécuter nos tests spécifiques au **parser**. Ce script fait appel à **test_synt**.

- **basic-context.py**

Ce script est analogue aux précédents et permet d'exécuter nos tests spécifiques à la deuxième étape de la compilation : les vérifications contextuelles. Ce script fait appel à **test_context**.

- **basic-exe.py**

Ce script est analogue aux précédents et permet d'exécuter nos tests spécifiques à la troisième étape de la compilation : la génération de code. Comme expliqué précédemment, nous utilisons les résultats de l'exécution d'**ima** sur les fichier .ass obtenus suite à la compilation des fichiers .deca correspondants. Ce script fait appel à **decac** ainsi qu'à **ima**.

Ces différents scrips facilitent la vérification manuelle des différents composants du compilateur. Afin d'exécuter un de ces scripts, il suffit d'exécuter la commande suivante :

- **python <script>.py**

Une autre manière d'exécuter la totalité de nos tests est d'utiliser la commande suivante :

- **mvn test**

Pour permettre l'exécution des nos tests par cette commande, nous avons rajouté un script bash au **pom.xml** appelé **our-tests.sh**. Ainsi, lors du lancement de **mvn test**, **basic-lex.py** est d'abord exécuté, puis **basic-synt.py**, puis **basic-context.py** pour ter-

miner avec **basic-exe.py**. le nombre de **PASS** et de **FAIL** est affiché pour chacun de ces scripts.

2.2 Création des tests

Afin de permettre une création facile de nouveaux fichiers tests, nous avons créé un script **test-manager.py**. Lors de son lancement, on demande à l'utilisateur quel type de test il souhaite créer, à savoir valide ou invalide. Si l'utilisateur demande de créer des tests invalides, il lui sera demandé pour quelle étape de la compilation le fichier est invalide. Sinon, cette question ne lui sera pas posée étant donné que nous considérons qu'un fichier valide l'est pour toute étape de la compilation.

L'utilisateur doit ensuite fournir le nom de son fichier test ainsi que le code deca qu'il peut taper directement dans le terminal. À la suite de cette étape, le **test-manager** va créer le fichier .deca correspondant et calculer les résultats avec les **launchers** et/ou **decac** et/ou **ima** avant de stocker ces résultats dans les bons dossiers.

Le **test-manager** permet donc aux utilisateurs d'éviter de calculer manuellement les résultats des .deca et de stocker les résultats dans les bons dossiers. De plus, nos script permettant d'exécuter les tests prendront automatiquement en compte les tests crées par le **test-manager**.

3 Gestion des risques

Pour limiter au maximum les risques sur notre compilateur, d'éventuelles mauvaises "manipulations", notamment avant les rendus, nous nous sommes efforcés de respecter certaines régles tout au long de notre projet.
Ainsi, nous nous sommes imposés de :

- Cloner notre dépôt git sur le serveur de l'école avant chaque rendu, afin d'être certain que notre compilateur s'exécute correctement sur les machines de l'école.
- De ne pas modifier en même temps un même fichier, afin d'éviter au maximum les problèmes de **merge** dans le dépôt git.
- De ne "pusher" des modifications dans le dépôt qu'à condition que le projet compile correctement.
- De prévoir une certaine marge avant les rendus, afin de pouvoir gérer d'éventuels imprévus.

4 Résultats de Cobertura

Pour vérifier quelles instructions de notre code sont exécutées par nos tests, nous avons utilisé Cobertura. Cet outil nous permet en effet de calculer la couverture d'un jeu de tests sur un programme.

Pour l'utiliser avec la batterie de tests que nous avons implémentées, il faut entrer la commande suivante dans un terminal : **our_tests_cobertura.sh**. Cobertura créé ainsi un rapport regroupant les résultats, auxquels on accède en exécutant les deux commandes, **cobertura-report.sh** + **firefox target/site/cobertura/index.html** . On obtient alors le rapport ci-dessous.

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	424	25%	4850/19041	13%	1161/8429	3.14
bytecode	2	87%	7/8	N/A	N/A	1
fr.ensimag.deca	6	71%	196/274	58%	50/86	2.833
fr.ensimag.deca.context	22	88%	203/229	84%	22/26	1.188
fr.ensimag.deca.syntax	50	70%	1442/2043	47%	362/758	2.12
fr.ensimag.deca.tools	4	100%	40/40	100%	6/6	1.333
fr.ensimag.deca.tree	88	97%	2086/2135	92%	370/400	1.536
fr.ensimag.ima.pseudocode	26	76%	134/175	75%	15/20	1.176
fr.ensimag.ima.pseudocode.instructions	54	69%	77/111	N/A	N/A	1
org.objectweb.asm	23	15%	665/4360	13%	336/2524	6.465
org.objectweb.asm.commons	27	0%	0/2794	0%	0/1309	3.466
org.objectweb.asm.signature	3	0%	0/157	0%	0/59	2.195
org.objectweb.asm.tree	30	0%	0/1160	0%	0/556	2.476
org.objectweb.asm.tree.analysis	13	0%	0/1039	0%	0/660	8.56
org.objectweb.asm.util	16	0%	0/2902	0%	0/1453	3.341
org.objectweb.asm.xml	60	0%	0/1614	0%	0/572	2.714

FIGURE 1 – Rapport de Cobertura pour l'ensemble des paquetages du projet

On remarque que sur l'ensemble de nos paquetages on obtient une couverture total de 25%, mais cela est à relativiser du fait que la bibliothèque ASM utilisée pour l'extension est présente dans notre répertoire. En effet le paquetage dans lequel nous avons implémenté la très grande majorité de notre compilateur est le paquetage fr.ensimag.deca (71 % de couverture) et plus particulièrement dans le dossier **fr/ensimag/deca/tree** pour lequel on obtient 97 % de couverture (cf *figure 2* ci-dessous). Ceci prouve que notre batterie de tests parcourt bien la quasi-totalité du code que nous avons implémenté pour notre compilateur.

Classes in this Package	Line Coverage		Branch Coverage		Complexity
AbstractBinaryExpr	96%	31/32	50%	1/2	1
AbstractDeclClass	100%	1/1	N/A	N/A	1
AbstractDeclField	100%	1/1	N/A	N/A	1
AbstractDeclMethod	100%	1/1	N/A	N/A	1
AbstractDeclParam	100%	1/1	N/A	N/A	1
AbstractDeclVar	100%	1/1	N/A	N/A	1
AbstractExpr	97%	46/47	92%	13/14	1.778
AbstractIdentifier	100%	1/1	N/A	N/A	1
AbstractInitialization	100%	1/1	N/A	N/A	1
AbstractInst	100%	3/3	N/A	N/A	1
AbstractLValue	100%	1/1	N/A	N/A	0
AbstractMain	100%	1/1	N/A	N/A	1
AbstractMethodBody	100%	1/1	N/A	N/A	1
AbstractOpArith	100%	19/19	100%	12/12	5
AbstractOpBool	100%	8/8	N/A	N/A	1
AbstractOpCmp	100%	7/7	100%	2/2	1.5
AbstractOpExactCmp	100%	42/42	100%	28/28	6.667
AbstractOpIneq	100%	21/21	100%	12/12	5
AbstractPrint	100%	41/41	88%	16/18	1.9
AbstractProgram	100%	1/1	N/A	N/A	1
AbstractReadExpr	100%	2/2	N/A	N/A	1
AbstractStringLiteral	100%	1/1	N/A	N/A	1
AbstractUnaryExpr	100%	18/18	N/A	N/A	1
And	100%	38/38	N/A	N/A	1
Assign	97%	41/42	100%	4/4	1.833
BooleanLiteral	100%	25/25	100%	6/6	1.333
Cast	98%	53/54	100%	16/16	2.667
ConvFloat	90%	10/11	N/A	N/A	1.2
ConvInt	90%	10/11	N/A	N/A	1.2
DeclClass	100%	79/79	75%	3/4	1.545
DeclField	100%	70/70	100%	10/10	1.889
DeclMethod	100%	60/60	87%	7/8	2
DeclParam	100%	27/27	75%	3/4	1.429
DeclVar	100%	48/48	83%	5/6	1.857
Divide	100%	18/18	87%	7/8	2
EmptyMain	100%	8/8	N/A	N/A	1
Equals	100%	20/20	100%	2/2	1.25
FloatLiteral	92%	25/27	66%	4/6	1.091
Greater	100%	18/18	N/A	N/A	1
GreaterOrEqual	100%	18/18	N/A	N/A	1
Identifier	84%	58/69	92%	13/14	2.19
IfThenElse	100%	70/70	100%	4/4	1.286

FIGURE 2 – Rapport pour le paquetage fr.ensimag.deca.tree

Le rapport du dossier **fr/ensimag/deca/tree** permet d'avoir les détails de la couverture pour chaque classe que nous avons implémentée. Ainsi, nous avons pu faire des tests de façon à maximiser notre couverture, en regardant les portions de notre code qui n'étaient pas encore parcourues par nos tests.
Cependant, il faut tout de même prendre un certain recul quant à ces résultats car Cobertura ne donne aucune information quant à la pertinence des tests. En effet, avoir une bonne couverture de tests ne signifie en aucune façon que ceux-ci sont fiables et qu'ils prennent en compte divers cas particuliers qu'il peut être pertinent de tester.

5 Conclusion

En décidant de faire essentiellement des tests d'intégration, nous nous somme placés dès le début du point de vue de l'utilisateur. Pour faciliter l'exécution de nos fichiers tests, nous avons partagé les tâches entre plusieurs scripts python, chacun responsable de l'exécution des tests spécifiques à une étape de la compilation.

Avec le **test-manager**, nous avons cherché à optimiser notre création de tests en automatisant une grande partie du travail.

L'ajout de notre script bash au fichier **pom.xml** permettant d'exécuter nos tests avec la simple commande **mvn test** nous a permis de valider au fur et à mesure notre code.

Afin d'étendre notre validation, des tests unitaires se focalisant sur des composants très spécifiques du code auraient pu être implémentés ainsi qu'un script python spécifique à l'exécution de ces tests unitaires.

ENSIMAG
681, rue de la Passerelle
38400 Saint-Martin-d'Hères