Documentation de validation

GL37
Thomas Avare, Quentin Candaele, Helena Cazals, Germain Vu, Andry Rakotonjatovo

22 janvier 2022

Ce document a pour but d'expliquer notre démarche de validation, comment nos tests ont été développé et dans quel but.

Table des matières

1 Organisation des fichiers de test

Les tests permettent de vérifier les différentes étapes permettant d'aboutir à la construction du compilateur `Deca` tout au long du projet Génie Logiciel. Tous nos tests sont des petits programmes au format `.deca`. Ces derniers permettent de valider une règle syntaxique ou contextuelle de notre langage, de donner un contre-exemple des règles ou de vérifier que la génération de code fonctionne. C'est pourquoi, le répertoire test est séparé en trois fichiers correspondant au trois étapes des différents phases du processus de compilation :

(i) Syntax

(ii) Context

(iii) Gencode

Ces trois étapes seront évoquées tout au long de ce document de validation. Ces derniers permettent de vérifier le bon fonctionnement des fonctionnalités du compilateur. Il se doit que ces tests soient le plus exhaustif possible afin d'assurer un taux de couverture de code le plus élevé possible. Nous utilisons l'outil Jacoco afin d'effectuer un diagnostic automatique de ce taux de couverture. La plupart de nos tests sont des tests en boîte noire et sont considérés comme dissociables de l'implémentation des fonctionnalités.

Pour la partie syntax, cela fait référence à la vérification lexico-syntaxique. Les tests présents dans ce répertoire détectent les types d'unité lexicale et construisent l'arbre abstrait défini par la syntaxe du code.

Ensuite, pour la partie context, cela fait référence à la vérification contextuelle. De nombreux tests ont été créés afin de détecter les erreurs contextuelles des trois passes de l'arbre. Les erreurs contextuelles sont levées de manière systématique à l'aide de l'exécution des scripts. Ces erreurs sont répertoriés dans le manuel Utilisateur. Une partie des tests permettent également de vérifier la génération de l'arbre abstrait.

Enfin, la partie Gencode de tests permet de valider le fonctionnement de la partie génération de code du compilateur. Ces tests soulèvent des erreurs arithmétiques par exemple telles que la division par 0 et vérifient des fonctions mathématiques de la bibliothèque `Math.decah`. Le sous-répertoire Gencode/Interactive contient des tests liés à `ReadInt()` et `ReadFloat()`. Ces derniers requièrent donc une interaction avec l'utilisateur de notre compilateur.

2 Lancement des tests

2.1 Exécution manuelle des tests

Afin d'exécuter un test, il est possible d'exécuter manuellement nos tests .deca à l'aide de la commande :

```
decac /chemin/vers/le/test/<test>.deca
```

3 Exécution automatique des tests

Afin d'exécuter un test, il est possible d'exécuter manuellement nos tests .deca à l'aide de la commande :

```
decac /chemin/vers/le/test/<test>.deca
```

3.1 Exécution automatique des tests

Mais, dû au grand nombre de tests présents dans nos répertoires, nous avons une possibilité d'exécuter les tests de manière automatique à l'aide des scripts que nous avons créés. Chacun vérifie une étape de fonctionnement pour la compilation.

3.1.1 Étape A : Vérification du bon fonctionnement du parseur et du lexeur

Type de tests présents :

- Tests d'analyse syntaxique
- Tests d'analyse lexical

L'analyse lexicale est testée grâce à nos tests d'analyse lexicale. Il est possible d'exécuter tous ces tests à l'aide du script /src/test/script/basic-lex-2.sh. De plus, pour l'analyse syntaxique, il est possible d'exécuter tous ces tests à l'aide du script `/src/test/script/basic-synt-2.sh`.

La décompilation, elle, permet de tester manuellement à l'aide de l'appel de l'exécutable decac avec l'option -p. Un exemple de cette exécution donnerait : `decac -p gl37/src/deca/syntax/valid/test`

Le chemin du fichier peut être évidemment adapté par le test qui intéresse l'utilisateur à vérifier

```
decac -p gl37/src/deca/syntax/valid/<test.deca>
```

Cette option est un moyen de vérifier la syntaxe de notre langage.

Nous avons remarqué que la partie A demandait une attention automatique vis-à-vis de l'anti-régression de notre compilateur lors de l'ajout de la programmation avec objet. Ce

suivi se fait heureusement grâce à l'exécution de la base de scripts deca. La décompilation des tests invalides visibles sur gl38/src/test/syntax/invalid est également testée de la même manière que ceux de /valid.

Finalement, on remarquera que l'analyse lexical et syntaxique est indirectement testée par tous les autres tests liés à l'évaluation contextuelle. En effet, ces derniers nécessitent à ce que l'arbre syntaxique primitif soit opérationnel.

3.1.2 Etape B : Vérification contextuelle

Type de tests présents :

sans classe :

- Tests liés à l'analyse contextuelle
- Tests conditionnelles : if, else, while

avec classe :

- Test lié aux includes
- Tests liés aux classes, aux champs et méthodes

Pour l'analyse contextuelle liée à l'étape B, l'exécution de `src/test/script/basic-context-2.sh` permet d'exécuter tous les tests de format `.deca` liés à l'analyse contextuelle liée à l'étape B et donc présents dans `src/test/deca/context/valid/*.deca` et

`src/test/deca/context/invalid/*.deca`.

La vérification contextuelle d'un fichier de test .deca peut également être exécutée à l'aide de l'option `-v` de l'exécutable `decac`. Pour information, la commande `decac -v <test.deca>` ne renvoie rien s'il n'y a pas d'erreur contextuelle et retourne une erreur si une erreur est relevée.

3.1.3 Etape C : Vérification de la génération de code

Type de tests présents :

- Tests sur le calcul arithmétique.
- Test de comparaisons numériques et booléennes.
- Tests liés au calcul booléen, d'entiers et de flottants.
- Tests de simulation.

L'étape de génération de code n'a pu être valable que lorsqu'on s'assure que le codage des conditions, des affectations et des affichages fonctionnait bien.

C'est au rôle du script `src/test/script/basic-gencode-2.sh` de vérifier que la génération de code fonctionne. Il exécute les tests présents dans `src/test/deca/codegen/*.deca`. Les tests liés à la bibliothèque `Math.decah` sont également testés dans ce répertoire.

3.1.4 Exécution de tout les test

Il est également possible d'exécuter l'ensemble de ces scripts simplement à l'aide de la commande :

```
mvn test
```

Cette commande va exécuter l'ensemble des scripts mentionnés précédemment :

- `src/test/script/common-test.sh`
- `src/test/script/basic-context-2.sh`
- `src/test/script/basic-gencode-2.sh`
- `src/test/script/basic-lex-2.sh`
- `src/test/script/basic-lex-2.sh`

L'ensemble de ces scripts permettent la vérification de toute la batterie de tests. Ces scripts seront alors exécutés (cela peut prendre quatre à cinq minutes) et affiche le résultat de chacun des tests.

3.2 Résultats des tests effectués

On distingue deux types de message sur la sortie standard.

Pour un test qui passe correctement, donc sans erreur pour un test valide et avec une erreur pour un test invalide :

```
test : <fichier-deca> - PASSED
```

Pour un test qui ne passe pas correctement, donc avec une erreur pour un test valide et pas d'erreur pour un test invalide :

```
test : <fichier-deca> - NOT PASSED
```

Remarque : Quelques tests ne passent pas car la fonctionnalité testée n'a pas encore été implémentée dans notre compilateur Deca.

Suite à la création des tests, l'outil Jacoco permet d'optimiser le taux de recouvrement en examinant les nœuds d'arbre non exploités.

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		75 %		54 %	589	783	487	2 011	254	367	4	48
fr.ensimag.deca.tree		79 %		74 %	123	641	361	2 134	55	477	2	87
fr.ensimag.deca		65 %		69 %	32	85	84	262	7	36	2	5
fr.ensimag.deca.context		79 %		70 %	50	138	65	281	42	114	0	22
fr.ensimag.ima.pseudocode		73 %		70 %	29	86	46	182	22	74	2	26
fr.ensimag.ima.pseudocode.instructions		55 %		n/a	26	62	45	111	26	62	19	54
fr.ensimag.deca.codegen		93 %		88 %	6	41	9	111	3	28	0	3
fr.ensimag.deca.tools		100 %		100 %	0	18	0	42	0	14	0	3
Total	5 337 of 23 140	76 %	487 of 1 300	62 %	855	1 854	1 097	5 134	409	1 172	29	248

FIGURE 1 – Aperçu du taux de recouvrement sur Jacoco

Nous avons essayé d'optimiser le taux de recouvrement et sommes montés à un total de 76%. Un exemple ci-dessous illustre un cas de résolution afin de pouvoir parcourir toutes les branches de l'arbre.

En l'occurrence dans cet exemple, nous remarquons en rouge que nous ne traitons pas la branche de l'arbre où un type null serait à droite ou à gauche d'une opération binaire.

```
32.   @Override
33.   public Type verifyExpr(DecacCompiler compiler, EnvironmentExp localEnv,
34.           ClassDefinition currentClass) throws ContextualError {
35.       LOG.debug("verify Expr AbstractOpArith: begin");
36.       if (getLeftOperand() == null){
37.           throw new ContextualError("operand gauche invalide à " + compiler.getSource().getAbsolutePath() + ":" + this.getLocation().getLine() + ": ", this.getLocation());
38.       }
39.       if (getRightOperand() == null) {
40.           throw new ContextualError("operand droite invalide à " + compiler.getSource().getAbsolutePath() + ":" + this.getLocation().getLine() + ": ", this.getLocation());
41.       }
42.       Type type1 = this.getLeftOperand().verifyExpr(compiler, localEnv, currentClass);
43.       Type type2 = this.getRightOperand().verifyExpr(compiler, localEnv, currentClass);
44.       String op_name = this.getOperatorName();
45.       if((type1.isInt() && type2.isInt()) || (type1.isFloat() && type2.isFloat())){
46.           this.setType(type1);
47.           LOG.debug("verify Expr AbstractOpArith: end");
48.           return type1;
49.       }
50.       //conversion implicite
51.       else if(type1.isInt() && type2.isFloat()){
52.           AbstractExpr new_left_op = new ConvFloat(this.getLeftOperand());
53.           Type new_type1 = new_left_op.verifyExpr(compiler, localEnv, currentClass);
54.           this.setLeftOperand(new_left_op);
55.           this.setType(type2);
56.           LOG.debug("verify Expr AbstractOpArith: end");
57.           return type2;
58.       }
59.       else if(type1.isFloat() && type2.isInt()){
60.           AbstractExpr new_right_op = new ConvFloat(this.getRightOperand());
61.           Type new_type2 = new_right_op.verifyExpr(compiler, localEnv, currentClass);
62.           this.setRightOperand(new_right_op);
63.           this.setType(type1);
64.           LOG.debug("verify Expr AbstractOpArith: end");
65.           return type1;
66.       }
67.       else{
68.           String error_msg = "Erreur contextuelle : opération impossible entre "+type1.toString()+" et "+type2.toString();
69.           throw new ContextualError(error_msg, this.getLocation());
70.       }
71.   }
```

FIGURE 2 – Aperçu d'une partie de code de l'arbre de recouvrement

3.3 Comment ajouter un test ?

Afin de pouvoir exécuter un test, ce dernier doit suivre un certain format. En effet, chaque test suit un en-tête faisant office de convention :

```
//  Description:
//   Test de l'addition entre float et int
//
//  Résultat:
//       tout marche bien
```

Pour les tests invalides, il est nécessaire de renseigner, dans la partie Résultat, le numéro de la ligne où se situe l'erreur ainsi que le type d'erreur . Cela donnerait :

```
//  Description:
//  Test de l'addition entre float et int
//
// Résultat:
//    Ligne 10 : Les types "int" et "boolean" sont incompatibles pour
//    l'affectation (règle 3.8)
```

3.4 Méthodes de validation utilisées autres que le test

Nos techniques de validation reposent essentiellement sur lancement de decac et des scripts de tests mentionnés précédemment. Toutefois, les résultats peuvent également être vérifiés manuellement. Par exemple, la commande `decac -n` permet de vérifier l'option no check.

3.5 Gestion des risques

Danger	Risque	Comment l'éviter
Oubli d'une date de rendu	Modéré	Diagramme de Gantt et mise au point quotidienne sur les avancées.
Mal se répartir les tâches et se retrouver avec du travail fait en doublon	Modéré	Mise au point quotidienne sur nos tâches de la journée.
Un confinement	Moyen	Chacun doit pouvoir travailler sur sa machine personnelle
Le travail de quelqu'un produit des erreurs sur le code des autres	Elevé	S'assurer après un "push" que la version du git compile correctement + communiquer des problèmes lors de la mise au point quotidienne

Règle de grammaire non respectées dans notre implémentation	Moyen	Relecture du code pas une personne exterieur à cette partie du programme
L'absence long terme d'un ou plusieurs membres de l'équipe	Moyen	Redistribuer les tâches qui lui étaient attribués
La présence de bug dans notre infrastructure de test	Élevé	Vérification à la main des tests et/ou demander à un autre groupe de faire tourner nos tests. Correction immédiate
Le git de l'ensimag ne fonctionne plus	Faible	Création d'un nouveau git (sur github par exemple)

3.6 Gestion des risques

Rendus	Réalisation
S'assurer que le projet rendu compile bien comme il faut	Utiliser un clone du dépôt
S'assurer que les fonctionnalités implémentées marchent	Lancer les scripts de tests afin de vérifier ce qu'il devrait marcher
Respecter le coding-style	Vérifier tous les codes sources
Vérifier la pertinence et la cohérence des documents à rendre	Relire les documents
Vérifier que les documents à mettre sur le git sont bien mis	Vérifier le nom des documents à rendre par rapport à la consigne