Documentation de validation

gl58

January 25, 2017

1 Etape A

L'analyse lexicale est automatiquement testée via les tests sur l'analyse syntaxique. Une erreur lexicale ferait en effet échouer un test atomique syntaxique. Il n'y a donc pas de tests purement lexicaux dédiés, cependant, il est possible de tester l'analyse lexicale à l'aide du script test_lex dans "/src/test/script/launchers/".

Ce dernier peut être utilisé en tapant du code manuellement puis en arrêtant la saisie avec Ctrl+D, il est également possible de fournir directement une entrée à ce script en utilisant les fonctions d'I/O fournie par Linux, par exemple :
./src/test/script/launchers/test_lex src/test/deca/syntax/valid/testInt1.deca
ce qui est très utile pour les tests manuels. Nous reviendrons sur la base de scripts deca de test disponibles plus bas.

1.1 Les tests syntaxiques

1.1.1 Les tests manuels

Il est possible de tester la construction de l'arbre d'analyse syntaxique manuellement, similairement à l'utilisation de test_lex. Les deux scripts sont dans le même dossier et s'utilisent de façon identique.

La décompilation peut aussi se tester manuellement à l'aide de l'appel de l'exécutable decac avec l'option -p.

En étant à la racine du projet, il faudra alors appeler:
decac -p src/test/deca/syntax/valid/testInt1.deca
Le chemin du fichier pouvant bien sûr être remplacé par celui de votre choix.

1.1.2 Les tests automatisés

La partie A bénéficie d'un fort suivi automatique anti-régression. Ce dernier procède en testant la décompilation d'une base de scripts deca.

Le fonctionnement est le suivant: decompile-still-equals-to.sh est un script shell dans le dossier src/test/script, qui est lancé automatiquement à la compilation avec tests, comme spécifié dans le fichier pom.xml à la racine.
Ce script va appeler la décompilation sur chaque script deca du dossier src/test/deca/syntax/valid/, et le compare, si la décompilation réussit, à son résultat attendu, script du même nom, dans le sous-dossier idempotent.

Il y a en effet plusieurs sous-dossiers:

- Les scripts de /provided sont ceux fournis initialement.

- Les scripts de /homebrew sont les scripts rajoutés par l'équipe, dans le cadre d'une compilation de

- script sans-objet.

- Les scripts de /classes sont répartis en deux sous-dossiers, /homebrew et /idempotent, ayant une fonction similaire à ceux un stade supérieur.

- Les scripts de /idempotent sont les scripts générés par la décompilation des fichiers dans ses 3 dossiers, vérifiés comme valides à leurs créations.

La comparaison de la décompilation des scripts de /provided, /homebrew et /classes à leurs équivalents /idempotent permet donc une protection contre la régression, et la vérification d'idempotence des scripts dans /idempotent est une vérification supplémentaire du fonctionnement de la décompilation. La décompilation ne retournant un code pertinent que si l'arbre syntaxique a été correctement construit.

La décompilation des tests invalides (src/test/deca/syntax/invalid) est également testée. L'organisation est similaire à celle du dossier /valid, avec des sous-dossiers /provided, /homebrew et /classes, dont la décompilation des fichiers contenus doit échouer, mais évidemment pas de dossier /idempotent.

Enfin, l'analyse lexicale et syntaxique est passivement testée par tous les autres tests, ces derniers ayant besoin de la construction de l'arbre syntaxique.

1.1.3 Description des scripts deca

Les rôles des scripts deca contenus dans les 2 dossiers /homebrew sont décrits par leur nom. Ainsi, pour donner des exemples, emptyClass.deca teste l'analyse d'une classe vide:

```
class EmptyClass {
}
```

emptyClassExtendsSomething teste la création d'une classe qui hérite d'une autre:

```
class emptyClass extends Something {
}
```

classWithAMethodThatReturnsAnAttribute.deca teste une méthode qui renvoie son attribut.

```
class MyClass {
    int x=1;
    int returnOne() {
        return x;
    }
}
```

Et ainsi de suite. Il est important de noter que ces tests ne testent que l'analyse syntaxique, c'est-à-dire la construction de l'arbre, et la décompilation. Ils ne sont donc pas corrects contextuellement, comme montré dans l'exemple emptyClassExtendsSomething, où la classe Something n'est pas définie.

Nous utilisons également des tests non-unitaires, qui testent le programme sur des scripts inspirés de conditions réelles (syntaxiques, pas contextuelles!) plutôt que de tester des concepts précis. Ces derniers scripts, dans le cas de la partie A, sont:

Pour la partie sans-objet:

- exprCompliquee.deca, qui vise à tester l'évaluation d'une expression complexe.
- grandChelem.deca, qui teste la majorité des possibilités sans-objet.

Pour la partie avec-objet:

- z-anActualProgram1, qui simule une classe de vecteur relativement basique avec getters et setters.
- z-anActualProgram2, qui porte sur une structure d'arbre et un parcours de cette dernière.

1.1.4 Exemple de décompilation

Voici un exemple de décompilation du programme scriptThatUsesNull.deca, situé dans le fichier /classes/homebrew, à partir du répertoire de tests syntaxiques.

```
{int x=1;
if (x!=null) {
    print("Hey, listen !");
}
}
```

donne après décompilation:

```
int x = 1;
if ((x != null)) {
    print("Hey, listen !");
} else {
}
```

1.1.5 Procédure en cas de changement de la décompilation

On peut voir dans l'exemple précédent que le code généré diffère assez nettement de l'original, et c'est normal pour du code généré.

Entre autres, x!=null est devenu (x != null), un else vide est apparu. Or chaque espace ou tabulation suffit à faire échouer les tests de non-régression.

Il est donc important, si on modifie la procédure de décompilation pour par exemple, changer le coding style, de pouvoir mettre à jour la base de tests.

Pour ce faire, un script shell est optimal. Après avoir manuellement testé sur les tests concernés par vos changements que la décompilation donne effectivement ce que vous vouliez, voici comment mettre à jour la base de tests anti-régression:

- Ajouter `<votre chemin vers le projet>/src/main/bin/` à votre `PATH`. (`PATH=<votre chemin vers le projet>/src/main/bin/:$PATH`) si ce n'est pas déjà fait.

- Dans un terminal, se rendre dans src/test/deca/syntax/valid.

- Aller dans /homebrew pour mettre à jour les tests sans-objet, ou /classes/homebrew pour mettre à jour les tests avec-objet.

- Lancer la commande suivante:
  `for $cas_de_test in *; do; decac -p $cas_de_test > ../idempotent/$cas_de_test; done;`

Il est possible de customiser * pour n'affecter que certains tests. Par exemple, testInt*.deca n'affectera que 5 fichiers si lancé dans le répertoire /homebrew sans-objet.

1.2 Gestion des risques et des rendus

La partie A, si on exclut des problèmes ponctuels de setLocation, était à jour pour le prochain rendu plus de 3 jours avant ce dernier, qu'il s'agisse du rendu intermédiaire ou du rendu final. Elle a donc été largement testée, sans précipitation.

Le fait qu'elle ne fasse pas planter les tests des parties suivantes en ayant besoin a également démontré sa fiabilité. J'ai donc conclu qu'elle ne présentait pas de danger critique pour les rendus, ce qu'elle n'a pas fait.

2 Etape B

2.1 Les tests contextuels

1. Les tests manuels

   La vérification contextuelle d'un fichier deca peut être testée manuellement a l'aide de l'option -v de l'exécutable decac. La commande `decac -v <fichier deca>` ne retourne rien si aucune erreur contextuelle est relevée et retourne l'erreur si elle a lieu.

   L'analyse contextuelle peut aussi être testée à l'aide du script shell `src/test/script/launcher/test_context`. Ce script s'utilise de la même manière que le script `test_lex` décrit précédemment, c'est-a-dire qu'il peut s'exécuter avec ou sans arguments, nécessitant une saisie manuel du code deca dans ce dernier cas.

   Le script `test_context` a aussi l'avantage d'afficher l'arbre décoré du programme après la vérification contextuelle. On peut donc aussi sans servir pour vérifier que la décoration et l'enrichissement de l'arbre sont bien gérés .

2. Les tests automatiques

   L'analyse contextuelle dispose, dans un premier temps, de tests Java, trouvés dans `src/test/java/fr/ensimag/`
   La majorité de ces tests étaient fournis et servaient a tester les opérations binaires. De nouveaux tests ont été ajoutés pour tester les déclarations de variables et certaines fonctions comme les `if` et les `while`.
   Ces tests sont lancés par le main de la classe `ManuelTestContext`. Celui-ci est executé automatiquement a la compilation du projet (sur Netbeans).
   Si ces tests sont peu développés, c'est parce qu'ils étaient utiles uniquement avant la réalisation de l'étape d'analyse syntaxique, car les classes Java construisent eux-même les arbres qui doivent être construit par le parser.

En effet, pour tester l'analyse contextuelle de manière plus complète, on dispose de plusieurs fichiers deca, situés dans `src/test/deca/context/`. Ces fichiers sont testes par deux script shell différents (dans `src/test/script/`):

`basic-context.sh`
Ce script effectue le test des fichiers sources qui se trouvent dans `invalid/pierre/`,`invalid/provided`, `valid/pierre/` et `valid/provided/`, des sous dossiers de `src/test/deca/context/`. Seul les sous-dossiers `objects/` des dossiers `pierre/` sont exclus de ce script.
Ces fichiers permettent de tester la vérification de la grammaire "sans-objet" de Deca.

`full-context.sh`
Ce script effectue le tests des fichiers sources qui se trouvent dans les dossiers `pierre/objects/`, dans `invalid/` et `valid/`.
Ces fichiers concerne la grammaire complète de Deca.

Le but de ces tests est, notamment, de tester chaque erreurs contextuelles possibles et de vérifier que des erreurs inattendues ne surviennent pas dans des cas valides un peu délicats.

2.2 Gestion des risques et des rendus

Comme la réalisation de cette étape a été plus longue que celle de l'étape A, les tests de vérifications contextuelles sont un peu moins étoffés que les tests syntaxiques.

Néanmoins, je pense que le risque que présente l'analyse contextuelle au bon fonctionnement du projet n'est pas trop conséquent. Les cas invalides sont suffisamment bien testés pour assurer que des erreurs contextuelles ne passeront pas inaperçues.

Les cas valides sont un peu plus délicats a tester car les situations imprévues qui peuvent lever des erreurs ne sont pas toujours intuitives.
C'est pour cette raison que le test `src/test/deca/context/valid/pierre/allHellBreaksLoose.deca` a été créé. Dans ce test, on a essaye d'inclure tout les cas délicats imaginable, que ce soit l'initialisation de super-classes par leur sous-classe, l'utilisation de champs et de paramètres de même nom, l'appel de champs `protected` dans des sous-classes et bien d'autres cas particuliers. La validation de ce test assure donc, a notre avis, une certaine fiabilité à l'analyse contextuelle.

3 Etape C

Pour valider la partie de la génération du code, il a fallut dans un premier temps vérifier que le code des conditions et des affichage fonctionnait bien. Une fois que l'on en été assuré nous avons pus commencé à automatiser les tests à l'aide scripts.

3.1 Vérification des fonctions de bases

Comme précisé plus tôt nous avons du vérifier manuellement que chaque fonction nécessaire aux tests automatique fonctionnait correctement. Nous avons donc pour cela crée quelques tests que l'on peut exécuter manuellement lorsque l'on a un doute sur le fonctionnement du programme. Nous avons aussi ajouté un test Junit qui permet de vérifier que ma gestion de la pile correspond bien aux attente que l'on en fait.

3.2 Vérification automatique

Afin d'accéléré la suite des vérification nous avons crée deux script cherchant les `0` ou les `FAIL` dans l'exécution des test. Ainsi la batterie de test pouvais s'exécuter automatiquement sans intervention. Le script quand à lui ne s'arrête que si il trouve un test qui rate. Nous possédons plusieurs dossier de tests pour la génération de code. Dans le dossier `ours` nous avons nos propre

tests sur la partie sans objet. Il y a donc dedans tout les tests sur les opération entières, flottantes ou booléennes. Dans un second dossier appelé `julien` nous avons entreposé les tests relatifs aux classes. Cependant nous n'avons pas eu le temps de créer plus de test pour valider la partie de la génération du code.
Finalement la stabilité du code de la génération n'a pas toujours été très bonne. Mais il est difficile de corriger un programme tout en le développant. Je pense ne pas avoir été assez bon communicant, et ne pas avoir assez bien expliqué ce que je développé pour aider mes camarades à fournir des tests.