Documentation de validation

CHENGUEL Sofièn
FEGUIRI Wael
HASSOUNE Hamza
OLLIER Valentin
SAUNIER Victor

January 2020

Table des matières

Chapitre 1

Descriptif des tests et des scripts

1.1 Partie A

Chacun des tests que nous avons implémenté est un programme court au format .deca. Ils sont situés dans les sous-dossiers /src/test/deca/syntax/invalid s'ils sont syntaxiquement incorrect et /src/test/deca/syntax/valid sinon. Pour la partie sans-objet, chaque test a une mise en page particulière afin de pouvoir automatiser leur utilisation avec un script. Ainsi, en début de programme, un commentaire contient la description du test et le résultat attendu. Pour un test invalide, le résultat sera une partie du message d'erreur renvoyé par notre compilateur à l'exécution du programme. Pour un test valide, le résultat sera sobrement "Succès". Afin de tester les fonctionnalités du lexer, certains tests sont lexicalement incorrects, et d'autres lexicalement corrects mais syntaxiquement incorrects. Cependant, tous ces tests se trouvent dans le répertoire invalid, car le sujet demande à ce que les tests valides le soient pour l'analyse syntaxique. Les tests lexicographique concernent uniquement la génération de tokens valides. Les tests syntaxiques garantissent que le programme ait du sens, que chaque opérateur ait le nombre d'opérandes requis et qu'ils soient bien placés, que les structures conditionnelles soient correctement écrites, que les parenthèses ou accolades soient toujours par deux, qu'il ne manque pas de points virgules à la fin des instructions et autres erreurs qui ne respecteraient pas la syntaxe abstraite du langage deca. L'ensemble des tests valides permet de vérifier que les programmes syntaxiquement corrects ne renvoient pas d'erreur.
Les tests sur la partie objet se trouvent dans le sous-répertoire /src/test/deca/syntax/valid/test_objets
Pour ajouter un programme de test à la base, il suffit de créer un nouveau fichier .deca dans le répertoire adéquat, d'écrire en en-tête un commentaire sur les six premières lignes, suivant le format suivant :

```
// Description :
// Description du test
//
// Résultat :
//Partie du message d'erreur attendu
//
```

L'exécutable src/test/scripts/syntax.sh est un script qui permet de lancer la batterie de tests et pour chacun d'entre eux vérifier si le résultat est bien celui attendu. Cela permet de trouver les bugs potentiels beaucoup plus rapidement qu'en lançant chaque test un par un dans son terminal, même si cela peut s'avérer nécessaire. Pour savoir si un test invalide est réussi ou pas, le script compare ce que le programme lancé renvoie et la chaîne de caractère écrite à la 5ème ligne du fichier .deca. Si le script trouve que cette chaîne de caractère est exactement une partie du message d'erreur renvoyé par l'analyseur lexical ou syntaxique, alors le test est réussi. Sinon le test a un comportement inattendu. Dans le cas d'un test valide, le script vérifie qu'aucune

exception ni erreur n'est renvoyé par les les analyseurs, on peut donc penser que ces derniers fonctionnent correctement.

1.2 Partie B

Chacun des tests que nous avons implémenté est un programme court au format .deca. Ils sont situés dans les sous-dossiers /src/test/deca/context/invalid s'ils sont contextuellement incorrect et /src/test/deca/context/valid sinon. Les tests de la partie B et le script d'automatisation sont très similaires à ceux de la partie A. Il faut respecter le même format d'en-tête, avec une description et le résultat attendu. Les tests invalides permettent de vérifier que l'implémentation de la partie B est complète et capable de détecter n'importe quelle erreur contextuelle d'un programme deca. Ainsi pour la partie sans objet, on vérifie entre autre la cohérence entre opérandes et opérateurs, la déclaration de chaque variable, la justesse des assignations et le typage. La partie objet elle permet de vérifier le bon traitement de la déclaration de classes, de méthodes et de champs, les sélections de champs et de méthodes, les mécanismes d'héritages, la redéfinition de méthode ou encore la protection des champs. Les test valides de la partie B permettent de vérifier qu'en cas de syntaxe correcte, des codes simples contextuellement corrects en deca sont bien transformés en arbres décorés afin de pouvoir être traités dans la partie C. De la même manière que pour la partie A, on peut ajouter un test en créant un fichier .deca dans le répertoire adapté, avec l'en-tête nécessaire. Lancer le script /src/test/scripts/context.sh permet de vérifier que chaque test renvoie le résultat attendu. En cas de succès d'un test valide, l'arbre n'est pas tracé par le script afin de ne pas surcharger le terminal. On peut s'assurer de la correction de l'arbre en lançant le test soupçonné individuellement.

1.3 Partie C

Chacun des tests que nous avons implémenté est un programme court au format .deca. Ils sont situés dans le sous-dossier /src/test/deca/codegen/valid. Une grande majorité des tests qui sont invalides pour la partie C le sont pour la partie A ou B, à part les erreurs dynamiques qui ne peuvent être reconnus qu'à l'exécution du code à proprement parler, comme la division par 0 par exemple. On s'est donc concentré sur les tests qui génèrent du code valide. Ainsi, on compile des programmes syntaxiquement et contextuellement corrects qui ne doivent normalement pas lever d'erreurs, et créer un fichier .ass qu'on pourra ensuite exécuter avec ima afin de vérifier leur correction. Nous avons un script qui permet de générer automatiquement l'ensemble des fichier .ass à partir des .deca, mais contrairement aux premières parties, il ne vérifie pas si le résultat est bien celui attendu ou pas. On a donc nécessairement besoin de lancer une commande ima pour qu'un test soit utile.

Chapitre 2

Gestion des risques et des rendus

Le projet GL est d'envergure suffisamment importante pour que de nombreux imprévus de diverses natures entravent son déroulement. Il semble donc nécessaire de prendre des mesures à l'avance afin de ne pas être perturbés voir complètement paralysés lors de la rencontre avec un tel obstacle. On peut donc énumérer les différents risques potentiels et la manière avec laquelle nous devions y faire face.

- Oublier une deadline.
  -Consulter régulièrement et suivre le planning mis en place.
- Ne pas passer un test fourni.
  -Toujours tester les premiers tests afin de vérifier la non-régression.
- Oublier de tester une fonctionnalité importante.
  -Faire faire les tests par un membre de l'équipe qui n'a pas forcément travaillé sur la partie testée, utiliser cobertura
- Push une version qui ne fonctionne pas.
  -Toujours vérifier que le projet compile.
- Assigner une tâche importante à une personne qui n'est pas à l'aise avec ou qui n'a pas les compétences requises.
  -Être plusieurs à travailler sur la même partie, afin de pouvoir s'entraider.
- Plusieurs personnes travaillant sur la même fonctionnalité sans le savoir.
  -Tenir au courant les autres de ce sur quoi on travaille.
- Travailler sur des versions obsolètes ou très différentes de celles des autres.
  -Toujours pull dès que possible après que quelqu'un ait push une version.
- Problème organisationnel au sein du groupe.
  -Se réunir et communiquer régulièrement
- Problème d'un membre de l'équipe mettant en péril le bon déroulement du projet.
  -Adhérence à la charte de travail en équipe.
- La dernière version push avant le rendu final ne fonctionne pas.
  -Ne pas coder de fonctionnalités majeures juste avant la deadline, se réunir et vérifier que le travail rendu est le meilleur que l'on ait.
- Obstacle qui bloque complètement l'avancée d'une partie importante du projet.
  -Réorganiser les ressources humaines de l'équipe en conséquence, travailler à plusieurs.

Chapitre 3

Résultats

Pour évaluer l'efficacité de notre base de tests, on utilise Cobertura, qui nous renvoie une couverture de 51% des lignes et 43% des branches. L'utilisation de cet outil nous permet de nous rendre compte que la couverture globale est encore insuffisante et qu'il manque encore de nombreux tests pour couvrir la totalité des erreurs possibles. En regardant plus en détail, on comprend qu'il semble normal que les packages pseudocode soient peu couverts ou encore des lignes comme celles de DecaParser dans la syntax. Les package context et syntax sont les mieux couverts, mais il y a encore de nombreuses méthodes inutilisées. Ainsi, même si Cobertura est un outil très puissant et pratique, les chiffres renvoyés ne sont pas absolus, et il serait déraisonnable de tenter d'avoir une couverture totale, puisque certaines lignes et branches ne sont pas testables.

En conclusion, notre base de tests est facile d'utilisation, en partie automatisée, relativement fournie pour certaines parties comme l'analyse contextuelle, mais sa couverture est encore trop faible en règle générale.

****