Documenation de validation
Compilateur Deca

Projet Génie Logiciel, groupe 15

January 27, 2023

Contents

1 Descriptif des tests **2**
- 1.1 Les différents types de tests employés 2
- 1.2 Format des tests 2
- 1.3 Méthodologie de test 4

2 Objectifs des tests **4**

3 Les scripts de tests **5**

4 Gestion des risques et gestion des rendus **6**

5 Résultats de Jacoco **8**

6 Méthodes de validation utilisées autres que le test **9**

1 Descriptif des tests

1.1 Les différents types de tests employés

- Le type de test le plus utilisé était le test unitaire.
  Chaque partie du langage (“Hello World”, “Sans Objets” et “Essentiel”) possède une batterie de tests unitaires pour valider un à un chaque élément de langage.
  Toutefois, dans les faits il est impossible de tester chaque élément indépendamment car nombreux d'entre eux ne seraient pas corrects du point de vue du langage sans la présence d'autres éléments.
  Exemple:

  ```
  {
  A a=new A;
  }
  ```

  Dans ce test on test à la fois l'instanciation de classe et l'instruction ”new”, tester ”new” tout seul aurait levé une erreur.
  Les tests sont séparés en deux dossiers pour chaque étape (A: syntax, B: context et C: codegen) :

  - Le dossier “valid”, regroupant tous les tests censés être corrects du point de vue du langage, dont on valide le bon fonctionnement en comparant ce qu'ils retournent à un résultat de référence.
  - le dossier “invalid” regroupe tous les tests censés déclencher une erreur pendant la compilation ou l'exécution. On vérifie donc qu'ils retournent bien un message d'erreur et que ce message soit celui attendu.

- Ces tests-là font partie des tests de validation du produit, desquels font aussi partie des tests plus longs, censés éprouver la robustesse du compilateur grâce à des situations plus complexes qui vont chercher d'éventuels bugs qui seraient mieux cachés.
  Ces tests complexes se retrouvent dans les dossier valid.

- Finalement, dans le cadre de l'intégration continue, chaque développeur à été amené à réaliser des tests de son côté pour s'assurer d'écrire un code fonctionnel.
  Ces tests n'ont souvent pas été gardés dans le rendu final car faisant souvent office de brouillon qui concernait des petites parties du programme qui seraient abstraites d'un point de vue extérieur.

1.2 Format des tests

Pour décrire le format des tests, nous allons nous appuyer sur l'exemple des tests de la partie A.
On commence par décrire le format des tests des dossiers “invalid”. Les fichiers de test

sont nommés en fonction de ce que le test à pour but de valider.
On prendra l'exemple du fichier "6_decl_var_manque_ident.deca". gl15\src\test\deca\syntax\invalid\Do

```
{
// Description:
//  Teste de Double Declaration d'une variable dans le corps d'une methode.
//
// Resultats:
//   Ligne 13: Double declaration
//
// Historique:
//      cree le 21/01/2023

class A{
   void f(){
      int x;
      int x;
   }
}
}
```

Le nom du fichier nous renseigne que celui-ci est censé tester si le compilateur détecte l'absence d'identificateur dans l'égalité du main.
Le fichier est préfacé d'un paragraphe commenté fournissant une courte description du fichier et de son rôle ainsi que le résultat attendu et l'historique (même si dans les faits, cette dernière section est rarement utilisée). Dans le cas des dossiers invalid, les résultats attendus seront des messages d'erreur spécifiques apparaissant dans le terminal. Le test n'est pas concluant si:

- Le message d'erreur obtenu n'est pas le bon, sous-entendant que l'erreur à été détectée d'une autre manière que celle attendue ou qu'une autre erreur à été détectée avant.
- Il n'y a pas de message d'erreur et, dans le cas de la partie A, le programme affiche l'arbre syntaxique, ce qui veut dire que l'erreur n'a pas été détectée.

Le format des tests des dossiers "valid" est très similaire à celui des test des dossiers "invalid", à ceci prêt que les résultats attendus pour les tests des dossiers "valid" sont des arbres syntaxiques conformes à ceux que l'on s'attendait à obtenir. On ne peut donc pas utiliser ces résultat pour faire une simple comparaison avec un fichier contennat les résultats attendus. Le résultat inscrit dans la section "résultat" est alors rien, où un message signalant que quelqu'un a validé le bon fonctionnement du test (toujours dans le cas de la partie A).

Pour les parties A et B, le format reste à peu près le même, les résultats et les erreurs retournés sont simplement différentes.
Les tests de la partie B retournent les mêmes arbres syntaxiques que la ceux de la partie A, à la différence qu'ils sont décorés.
Le contenu des sections "résultat" sera donc sensiblement le même.
C'est cependant différent pour la partie C, car ses tests retournent un code assembleur exécutable, on peut donc valider ces tests en exécutant le résultat du compilateur grâce

à ima, en printant les résultats et en les comparant alors avec les résultats qu'on voulait obtenir.
C'est ce résultat voulu qui occupe alors la section résultat du fichier test.

Cette particularité des tests de la partie C nous à permis d'automatiser la procédure de test.
Nous avons trouvé judicieux de faire cette automatisation à ce niveau là car, la partie C dépendant de la partie B, dépendant elle-même de la partie A, tout élément de langage validé par C l'était automatiquement par A et B, et toute erreur levée dans les parties A et B pendant la compilation où l'exécution d'un test de la partie C serait détectée.

Nous avons donc utilisé la base de test de la partie C comme indicateur de qualité du code, nous permettant de valider la bonne implémentation des dernières parties de langages implémentées à l'instant t, mais aussi d'opérer une sorte de maintenance sur le code implémenté antérieurement pour vérifier qu'il était toujours fonctionnel en ré-exécutant les anciens tests.

Pour effectuer ses tests, nous avons stocké les résultats attendus pour chacuns d'entre eux au format nom_du_fichier_expected_output.res dans un dossier "result" des dossier "valid" et "invalid", puis nous avons écrit des scripts permettant de compiler les tests, d'exécuter les fichiers assembleurs correspondant et de comparer leurs retours avec les retours attendus du fichier résultat.

1.3 Méthodologie de test

Notre philosophie de test était le test continue, c'est à dire le fait de tester régulièrement le code au long du développement plutôt que de tout tester à la fin. Cela rend les bugs plus simple à localiser et plus simple à corriger car on les corrige "à chaud".
L'utilisation de branche personnelle à chaque programmeur dans le git présente aussi un grand avantage, en effet, quand on découvre un problème, on sait que ce problème est de notre fait car la version que l'on à récupérer dans cette branche était censé être débbugué et fonctionnelle pour la partie précédente.
Cela aide encore une fois à localiser l'erreur, mais ça empêche aussi de gêner les autres avec ses propres erreurs tout en augmentant la liberté d'expérimentation de chacun sur le code.

2 Objectifs des tests

En général, le but premier de chaque test unitaire est d' examiner un petit morceau de code pour s'assurer que chaque composant du programme de chaque partie du sujet fonctionne correctement, ce qui simplifie les méthodologies de test et réduit au minimum le nombre de tests inutiles.
Ces tests nous permettent également d'améliorer notre compréhension du langage deca

et du projet en général en voyant le compilateur en action, on a régulièrement utilisés ceux si lors des phases de développement pour mieux internaliser ce qu'il se passait lors du fonctionnement du compilateur.

Le but principal des tests unitaires de l'étape A était de tester la capacité du compilateur à générer l'arbre abstrait primitif. Il était important de terminer cette étape de test pour pouvoir enclencher l'étape de test de la partie B. Finalement, la plupart des règles de syntaxe ont été explorées.

Notre objectif en ce qui concerne les tests de la partie B était de tester toutes les règles contextuelles mentionnées dans le sujet, cet objectif a été atteint. Dans cette partie, les tests invalides sont censés être corrects syntaxiquement mais pas contextuellement (ex: usage d'une variable non déclarée). Les tests valides sont censés être corrects syntaxiquement et contextuellement.

3 Les scripts de tests

Pour pouvoir automatiser la compilation et l'exécution des tests unitaires des différentes parties, on a utilisé des scripts shell.
Afin de pouvoir utiliser ces scripts avec maven, on a mis chaque scripts dans la section "execution" du fichier pom.
Ainsi, on peut les exécuter en utilisant la commande "mvn verify". Voici un exemple de script shell utilisé:

```
{
    <execution>
     <id>Test-Syntaxe-Valide</id>
     <configuration>
     <executable>./customLaunchers/TestSyntaxique/autoSynValide.sh</executable>
     </configuration>
     <phase>test</phase>
     <goals><goal>exec</goal></goals>
</execution>

}
```

Les raison pour laquelle on a choisi d'utiliser le langage "bash shell script" sont:

1. Bash permet l'utilisation de structures de contrôle (typiquement if, for, while...) qui permet d'écrire des scripts plus compacts et capables d'utiliser un grand nombre de fichiers.
2. On peut aisément colorer les résultats des test ce qui les rends plus lisibles et donc plus faciles à parcourir et à interpréter.

Les scripts automatique sont tous rassemblés dans un même répertoire "gl15\customLaunchers".Ch
partie du code possède son propre dossier de script. On a donc "TestSyntaxique", "TestContexuelle", "TestCodegen" et "TestExtension".

Dans chaque dossier de script, on retrouve un script pour déclencher les tests invalides et les tests valides.

Tout les scripts ont été conçus en partant de la même base: Par exemple, dans un script ”valid”, on utilise deux fonctions principales “test_context_valide ()” et “redirect_result()”. La première sert à exécuter le test unitaire et détecter si ce test est passé en utilisant une condition if. Le cas échéant, la fonction affiche ”TEST PASS!” et ”TEST NON PASSE” sinon. La deuxième est exactement la même pour valid et invalid, elle sert à rediriger le résultat du test de la sortie courante (la console) dans un fichier ”.lis” du fichier ”result”. En dehorsdes fonctions, dans la partie principale du script, on utilise une boucle ”for” pour parcourir tout les fichiers du chemin indiqué et appeler les 2 fonctions présentées ci-dessus. La première fonction dans le cas d'un fichier de test invalid suit la logique inverse de celle du cas valid. Voici un exemple de la squelette d'une script automatique:

```
{
    test_context_invalide (){
    if test_context "$1" 2>&1 | grep -q -e "$1:[0-9]*"
    then
        echo -e "${GREEN}TEST PASS! ${ENDCOLOR} ${GREEN_BOLD} $(basename "$i")
${THUMBS_UP}"
    else
        echo -e "${RED}TEST NOT PASS!!  Issue file :${ENDCOLOR}${RED_BOLD} $(basename
    fi
}

redirect_result(){
    mkdir -p src/test/deca/context/invalid/result
    test_context "$1" > src/test/deca/context/invalid/result/$(basename "${1%.deca}").
}

for i in src/test/deca/context/invalid/*.deca
do
    test_context_invalide "$i"
    redirect_result "$i"
done

}
```

4 Gestion des risques et gestion des rendus

Risque	Solution
Prise de retard sur le développement. Risque de ne pas rendre tous les livrables, de ne rendre que des fonctionnalités incomplètes.	- Augmenter le rythme de travail, s'organiser plus efficacement. - Ne pas attendre la dernière minute pour commencer les livrables autres que le code. - Redéfinir les objectifs: il vaut mieux rendre quelques fonctionnalités terminées que beaucoup de débuts de fonctionnalités.
Conflit, divergence de vision au sein de l'équipe. Risque d'accentuer le retard et de donner un code qui marche par morceau mais qui ne marche plus une fois assemblé, qui n'est pas cohérent.	- Suivre la charte pour les règles de communications (parler des problèmes lors des réunions notamment).
Risque d'absence (maladie ou autre) d'une personne s'occupant d'une des tâches du chemin critique. Risque de bloquer l'équipe.	- Travailler à plusieurs sur une même tâche, le travail peut continuer même si une personne est malade.
Risque de non cohérence du travail de l'équipe: Si une partie A de l'équipe travaille sur un morceau de code se basant sur le travail d'une autre partie, on risque que la partie A travaille sur un code daté ce qui peut induire d'importantes pertes de temps autrement évitables.	- Régulièrement mettre à jour le travail de sa branche à l'aide d'un push. - Communiquer constamment pour se tenir mutuellement au courant des avancées de nos parties respectives.

Figure 1: Caption

Risque	Solution
Risque de non exhaustivité de la base de test: Le nombre de règles à tester est très grand, et il peut devenir compliqué de détecter tous les points faibles de notre code en testant. Le risque est de rendre un code que l'on pense bon mais qui cache des défauts qui ne se révéleront qu'à l'usage par le client.	- L'outils Jacoco permet d'avoir une bonne idée de la couverture de la base de test, c'est-à-dire de la proportion du code réellement sollicitée lors de leur exécution. Ce n'est pas full proof mais tout de même pratique. - L'utilisation de tests unitaires couplés avec une méthode de test continue pendant le développement garantit un bon ratissage des erreurs et problèmes potentiels.

Figure 2: Caption

5 Résultats de Jacoco

Afin de mesurer la couverture du code, c'est à dire la proportion de notre code ayant été solicité pendant l'exécution de notre base de tests, on a utilisé jacoco.
Cet outils nous permet de visualiser cette proportion à l'aide de nombreuses données et de jauges. Il nous permet également d'aller dans le détail et de savoir quelles parties du code n'ont pas été utilisées. Jacoco utilise un rapport qu'on peut générer avec la commande "mvn verify", on peut y accéder par le chemin "gl15/target/site/index.html".
Voici une capture d'écran du rapport Jacoco:

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		76%		56%	532	738	490	2,181	259	377	3	48
fr.ensimag.deca.tree		93%		82%	158	936	138	2,492	49	601	0	88
fr.ensimag.deca		63%		45%	60	121	115	321	17	67	3	6
fr.ensimag.deca.codegen		76%		73%	24	75	39	177	12	41	1	5
fr.ensimag.deca.context		90%		72%	47	171	41	327	27	131	0	23
fr.ensimag.ima.pseudocode		86%		85%	21	90	25	190	18	80	1	27
fr.ensimag.ima.pseudocode.instructions		80%		n/a	14	62	24	112	14	62	12	54
fr.ensimag.deca.tools		93%		100%	1	16	3	37	1	13	0	3
Total	4,553 of 29,085	84%	503 of 1,603	68%	857	2,209	875	5,837	397	1,372	20	254

D'après ce rapport, nos tests ont réussis à tester:

- 84% des instructions de tous les fichiers au global.
- 76% des instrcutions de la partie syntaxique (Partie A).
- 90% des instructions de la partie contextuelle (Partie B) avec dedans, 93% du package tree.
- et 76% de la partie de génération de code (Partie C).

En particulier, à cause de problèmes de limitation de temps, la plupart des instructions manquantes viennent des traitements d'exception, surtout dans la partie syntaxe.
De même, il manque des tests des branches qui testaient les instructions conditionnelles, par exemple if, else if, switch. Par conséquent on obtient un taux de couverture pour les tests des branches 68%.

6 Méthodes de validation utilisées autres que le test

On utilise également d'autres méthodes de validation en plus des tests en continue:

- On utilise la commande "mvn verify" et les scripts de tests automatiques pour effectuer une révision des anciens tests et s'assurer qu'ils sont toujours valide. Cela nous permet de nous prémunire de l'éventualité qu'un des membres de l'équipe casse le code déjà fonctionnel en implémentant de nouvelles fonctionnalités.
- La relecture du code hors tests permet d'éviter de se retrouver avec des erreurs génantes plus tard.
  En particulier, la révision par les pairs (en l'occurence, les autres membres du groupe travaillant sur la même partie), permet de bénéficier d'un regard différent pour confirmer la validité du code et ça cohérence avec le travail des autres.
- Jacoco permet de valider partiellement la qualité de la base de tests et indirectement celle du code.
- Travailler à deux sur un même écran (ce qui c'est régulièrement fait au travers du projet) est aussi un moyen de valider le code écrit en le soumettant à l'analyse critique de deux personnes simultanément, ce qui réduit la possibilité d'erreurs et de fautes d'inattention.