Documentation sur la validation

Projet génie logiciel

Equipe : gl07

Naima AMALOU Yidi ZHU Jiayun Po Zaineb Tiour An Xian

30 Janvier 2020

Table des matières

Table des figures

1 Descriptif des tests

Pendant le développent de notre compilateur nous avons construit au fur et à mesure des tests qui permettent de tester le bon fonctionnent de notre compilateur. Pour chaque étape des tests ont été construit avant le développent de l'étape correspondantes , d'autre tests ont était ajouté au fur et à mesure pour exhiber d'autre erreurs possible. Le nombre total de tests construit est 523 tests.

1.1 Types de tests pour l'étape A

Les tests construit pour l'étape A ont pour but, de vérifier la construction de l'arbre abstrait, et de vérifier la location de chaque non-terminal.

Les tests valide de l'étape A se trouve dans le répertoire ***src/test/deca/syntax/valid***, et les tests invalides sont dans ***src/test/deca/syntax/invalid*** les tests de langage sans objet sont dans le sous répertoire ***sansObjet***, et les tests de langage objet dans le sous répertoire ***class***. Pour lancer un test il suffit de lancer ***test_synt*** qui se trouve dans ***src/test/script/launchers*** suivie de chemin de test.

1.1.1 Les tests Valides

Le but des tests valides étaient de vérifier la bonne constructions de l'arbre abstrait, donc nous avions essayer de créer des tests en utilisant le plus grand nombre possibles des non-terminal de la grammaire syntaxique.

- **Les listes** : Des tests ont étaient créer pour vérifier la bonne construction des listes de déclarations des classes, champs, paramètres et variables,
- **If Then Else** : Une attention particulier a était attribué à ce noeud, pour sa difficulté, nous avons donc tester ce noeud avec les cas possibles : absence de la branche else dans le if principal, une branche if à l'intérieur dans la branche else, tests des if imbriqué.

1.1.2 Les tests invalides

Les tests invalid vérifie la sémantique de langage Deca.

- **Les tests de la syntaxe** : Des tests ont étaient créer qui ne respecte pas la syntaxe de langage Deca , par exemple une chaîne de caractère sont les guillemets, les déclarations des variables après une liste d'instructions...
- **les inclusion** deux erreurs ont étaient testé pour le token **include** l'unclusion d'un fichier qui n'existe pas dans le dossier **src/main/resources/include** , ou les inclusion circulaire de même fichier.
- **les débordements syntaxique** : Deux tests ont étaient créer pour tester le débordements des entiers et celui des flottants.

1.2 Types de tests pour l'étape B

L'étape B est l'une des étape les plus importantes pendant le développement de notre compilateur, elle permet de vérifier si un langage Deca est « bien typés » Cette étape permet aussi de gérer un très grand nombre des erreurs contextuelles. Au total nous avons construit 219 tests dans cette étape.

```
`> ListInst [List with 1 elements]
   []> [5, 6] Assign
       type: A
       +> [5, 4] Identifier (a)
       |  definition: variable defined at [4, 6], type=A
       `> [5, 8] New
          type: C
          +> [5, 12] Identifier (C)
          |  definition: type defined at [2, 6], type=C
```

FIGURE 1 – Le type de lvalue est A et le type de rvalue est C , le type de assign doit être de type A.

Les tests valide de l'étape B se trouve dans le répertoire ***src/test/deca/context/valid***, et les tests invalides sont dans ***src/test/deca/context/invalid*** les tests de langage sans objet sont dans le sous répertoire ***sansObjet***, et les tests de langage objet dans le sous répertoire ***class***. Pour lancer un test il suffit de lancer ***test_context*** qui se trouve dans ***src/test/script/launchers*** suivie de chemin de test.

1.2.1 Les tests valides

Les tests valide de cette étape ont pour but de tester les bons décorations de l'arbre abstraite, et les enrichissements de l'arbre, et la bonne vérification des conditions contextuelles, les tests de l'étape B s'appuie sur la syntaxe contextuelle du langage Deca.

— Chaque identificateur utilisé dans le programme **Deca** doit avoir une définition et un type pendant la vérification contextuelles, cette définition dépends de la nature de identificateur et son type. La majorité des tests construits permet de vérifier la bonne décoration de identificateur. la définition d'un type prédéfinie est sous la forme suivante : definition : type (builtin) , type= <type> Pour un type non prédéfinie (les classes) , la définition est sous la forme : definition : type defined at [<ligne>,<colonne>] , type=<type> avec [<ligne>,<colonne>] représente la position de la définition de la classe. Ce type de décoration a été tester pour tous les types prédéfinies (int, void , boolean, Object, equals). et le classes définie par l'utilisateur.

— Les identificateurs sont aussi décorés par leurs types pendant leurs déclarations. Les tests construits pour l'étape B permet la vérification de bon type pour chaque identficateur.

— Chaque expression à un type , Nous avons vérifier les bons types attribué à une expression, par exemple dans un assign , le type qui doit être attribué à assign et le type de lvlaue et non pas de rvalue. et pour un appel de méthode on vérifie que le type à la déclaration de la méthode est bien son type de retour et aussi au moment d'appel de la méthode le type de **MethodCall** est son type de retour.

```
class A {}
class C extends A {}
{
    A a;
    a = new C();
}
```

— **Les décorations des champs d'une classe :** Un champ d'une classe a en plus de son type, une visibilité, l'arbre syntaxique est aussi décorée avec la visibilité des champs, on vérifie cette visibilité avec des déclarations des champs publique et d'autres protégé . la visibilité à la forme suivante : [visibility=PUBLIC] pour les champs publiques et [visibility=PROTECTED] pour les champs protégés. On vérifie que lacée est possible pour les champs public en dehors de la classe, et que lacée est possible pour les champs protégés à l'intérieur de la classe et dans les classes filles.
— Les tests valide permet aussi de vérifier le bon respect de toutes les conditions sur les attributs fournie par la syntaxe contextuelles.

- **Comptabilité pour l'affectation :** Dans le langage sans Objet, nous avons testé cette condition en créant des tests qui affecte à une expression de type **float** une expression de type **int**. et aussi pendant les initialisations. Dans le langage objet, cette comptabilité pour l'affectation à était vérifier au moment de l'appel d'une méthode avec des paramètres qui sont différents de ceux qui ont était déclaré au moment de la création de la méthode. mais qui sont des sous type pour les classes.

- **Compatibilité pour la conversion** : Pour le langage sans objet le seul cas de conversion possible c'était la conversion entre les deux type float et int , un test était fait pour faire cette vérification, pour le langage objet les cas de conversion possible sont entre les classes plusieurs, des tests ont étaient créer pour ces vérifications.

- **Relation de sous-tapage :** ces relations ont étaient vérifier au moment de la redéfinition d'une méthode dans une classe fille pour le type de retour de la méthode. et au moment de la sélection d'un champ d'une classe.

- **Les opérations partielles :** ces opérations ont étaient majoritairement testé dans le langage sans objet, en vérifiant la possibilités de faire des additions entre les flottants et les entiers.

— **Arbre enrichi :** Nous avons testé le noeud ConvFloat avec des affectation simple entre un identificateur de type float et un autre de type int, nous l'avons également tester au moment de la lecteur en utilisant **readInt()**, et au moment des opération arithmétique entre les deux types int et float, ce noeud est testé aussi au moment de l'appel d'une méthode pendant la conversion des paramètres.

— **Compilation concurrente :** quelques tests qui porte le nom **compilConcurN** avec 1 <= N <= 5, contient des milliers de déclaration de variables, ont étaient générer avec des scripts en **Python** pour tester l'option **-P** de compilateur.

1.2.2 Les tests invalid :

La totalité des tests invalid construits permet de tester tous les cas des erreurs possibles en se basant sur les conditions imposés par la syntaxe contextuelle du langage Deca.

— **Les double définitions** : Cette erreur à était vérifier par des tests qui contient des doubles définitions (les déclarations des variables dans le programme principal ou dans le corps d'une méthode , les déclarations des paramètres d'une méthode, les déclarations des champs d'une classe.

— **Les affectations incompatibles :** Nous avons tester des affectation incompatible pendant la déclaration des variables avec une initialisation incompatible, avec tous les types possible (booléen, int, float, classe), les affectations incompatibles sont aussi testé dans le noeud **Return**, en créant des test avec des méthodes qui retourne des types qui ne sont pas compatible pour l'affectation avec le type déclaré pendant la déclaration de la méthode. Cette affectation a était aussi tester lors de l'appel d'une méthode en passant en paramètre des types incompatibles.

— **Les opérations binaires et unaire entre des types non autorisé** : Dans le langage sans objet nous avons créer quelque tests, qui effectue des opérations arithmétiques entre des booléen et des chaînes de caractères, d'autres tests avec des opérations de comparaison entre de booléen et des entiers ou flottants.

— **Les conversions incompatible** : Cette erreur a était vérifier en créant des tests avec des conversions entre les flottants et les booléens.

— **Impression des types non autorisé** Cette erreur a était testé en impriment des booléens, des méthodes et des classes.

— **Utilisation des types non prédéfinies** : Cette erreur a testé en criant des tests qui déclare des variables avec des types non prédéfinie de type **string**, **null**, et tous les autres types qui ne sont pas définie dans l'environnement types.

— **Accé non autorisé pour les champs** : Cette erreur a était testé, en créant des classes avec de champs protégés, que le l'utilisateur appel dans le programme principale.

— **Types non autorisé pendants les déclarations** : Nous avions créer des tests qui déclare qui variables, des champs d'une classe, des paramètres d'une méthode de type **void**.

— **Appel des méthodes** : Les erreurs pendant l'appel d'une méthode ont étaient testé avec des appels de méthode inexistante dans la classe courante, ou dans le programme principale sans désigner un identifiant de classe.

— **Le retours non autorisé** : Nous avons testé les retour non autorisé avec des tests qui contient le noeud **return** dans des méthodes de type **void** et dans le programme principale. Nous avons aussi testé plusieurs retour dans une liste d'instructions avec plusieurs **return** dans la même méthode.

1.3 Types de tests pour l'étape C

Les tests de l'étape C, tests des programmes Deca qui sont correcte syntaxiquement et contextuellement, s'exécutant correctement ou provoquant une erreur à l'exécution.

Les tests valide de l'étape B se trouve dans le répertoire ***src/test/deca/codegen/valid***, et les tests invalides sont dans ***src/test/deca/codegen/invalid***, les tests interactifs sont dans le dossier ***src/test/deca/codegen/iteractive*** , les tests de langage sans objet sont dans le sous répertoire ***sansObjet***, et les tests de langage objet dans le sous répertoire ***objet***. Pour lancer un test il suffit de lancer ***decac*** qui se trouve dans ***src/main/bin*** suivie de chemin de test. decac génère un fichier assembleur dans le même dossier que le fichier d'origine, ensuite il faut

lancer **ima** dans la racine de projet suive de chemin de fichier assembleur.

1.3.1 Les Tests valides

— **Les opération arithmétiques** : Nous avions créer des tests pour vérifier le bon fonctionnements des opérations arithmétiques.

— **Les expressions imprimables** : Des tests ont étaient créer pour tester l'ensemble des expressions imprimable : des entiers, flottants et chaîne de caractères.

— **Les instructions de contrôle** : Pour tester les instructions de contrôles nous avons créer des tests qui tests l'ensemble des cas possible pour l'instruction **if** et **while**.

— **Les conversions compatibles** : Nous avons tester les conversions compatibles dans le langage sans objet entre les flottants et les entiers , et dans le langage objet entre les classes.

— **Les redéfinitions des méthodes et des champs** : Pendant une redéfinition d'une méthode ou un champ, c'est la dernière définition qui doit être utilisée dans le programme pour la classe correspondant. Nous avons donc tester ceci par des tests qui redéfinit les méthodes d'une classe supérieur dans la classe fille.

— **Test de limitation de nombre de registres** : Pour tester le bon fonctionnement de l'option **-r X** de compilateur, tels que X est le nombre de registres à utiliser dans la génération de code. Nous avons relancer les tests précédant toute en fixant le nombre de registre à 4. Nous avons aussi tester ceci avec un grand nombre d'instructions imbriqué.

1.3.2 Les tests invalides

— **L'utilisation des variables non initialisée** : Les variables définit dans le programme principale ne sont pas initialisé par défaut, leur utilisation sans initialisations est donc interdite. Nous avions créer des tests qui utilise des variables non initialisés dans les opérations arithmétiques ou dans à l'intérieur de **print**.

— **Débordement arithmétique sur les flottants** : Nous avions tester cette erreur en créant des programmes avec un débordement arithmétique sur les flottants et aussi pendant la division par zéro.

— **Dépassement de la pile** : Nous avions tester le dépassement de la pile, en utilisant un très grand nombre de variable global.

— **Absence de return lors de l'exécution d'une méthode** : L'absence de return dans une méthode qui doit retourner un type diffèrent de void est interdit lors de la génération de code, nous avions tester cela en créant des méthodes qui sont censé de retourner un résultat sans avoir le non-terminal **return** dans leur corps.

— **Conversion de type impossible** : Nous avons testé cette erreur avec des conversion impossible entre les classes.

2 Les scripts de tests

Les scripts de tests sont en disposition pour lancer tous les tests de chaque étape avec un seul commande. Nous avons crée deux types de script pour faciliter les tâches. Le scripts dans le répertoire **Projet_GL/src/test/script/script_deca** sont faits de sorte que les résultats et commentaires de tests sont affiché sur l'écran. Il y en a 8 scripts en disposition pour lancer les tests valids et invalids pour la génération de code, vérification de l'analyse contextuelle, verification de l'analyse syntaxique et lexicographique.
Un autre type de script que nous avons crée affiche que le résultat globale de chaque test sans afficher les commentaires. Si un test est validé, un message de "Succes attendu" est affiché sur l'écran, et le compilateur continue à exécuter le test suivant. Si on attend un résultat positive sur un test, mais une erreur est survenu, un message d'erreur de "Échec inattendu" est affiché, et l'exécution de test s'arrête tout de suite. Il y en a 4 scripts en disposition pour les 4 tests différents dans le répertoire **/Projet_GL/src/test/script/auto_test_resultat**.

3 Gestion des risques et gestion des rendus

3.1 Facteurs de risque

- Valoriser une partie du projet plus qu'une autre. (ex : se concentrer sur coder et oublier la documentation ..)
- Conflits entre membre de l'équipe.
- Manque de places dans les salles machines de l'ensimag.
- Incompréhension des subtilités de la grammaire.
- Engagement des membres dans d'autres activités, taches, ou projets.
- Oublie des dates des rendus.
- Soucis relatif au git.

3.2 Facteurs de sécurité

- Division du projets en axes principaux et désignation d'une responsable pour chaque axe.
- -Désignation d'une chef de projet qui saura agir selon la charte de travail en groupe signé par tous les membre de l'équipe.
- Configuration de tous les ordinateurs personnels des membres de l'équipe et réservation en amont des salles de travail à la bibliothèque universitaire.
- Se partager les moindres soucis de compréhesion et les questions qu'on juge souvent basiques mais qui constituent une base non négligeable de la grammaire.
- Parler des nouvelles "découvertes" sur le language deca dans la réunion du jour.
- Chaque membre se doit de déclarer les engagements qui peuvent provoquer un soucis d'organisation en amont.
- Respect total des autres membres de ces engagements.
- Mise en place d'un Google Calendar et d'un board sur Trello.
- Activation des notifications pour ces deux applications.
- Mise en place d'un "guide git" sur le drive par la personne qui s'y connait mieux.
- Ne pas faire un merge de branches que si tout le monde qui travaille dessus l'approuve.
- Ne pas pusher un code qui compile pas.
- Demander de l'aide en cas de doute.

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	256	75 % 4390/5812	57 % 1022/1786	1,791
fr.ensimag.deca	5	57 % 167/292	48 % 40/83	2,13
fr.ensimag.deca.codegen	5	90 % 78/86	72 % 16/22	1,542
fr.ensimag.deca.context	22	71 % 263/369	60 % 66/110	1,581
fr.ensimag.deca.syntax	52	71 % 1471/2059	45 % 333/737	1,923
fr.ensimag.deca.tools	4	87 % 50/57	70 % 14/20	2,214
fr.ensimag.deca.tree	88	80 % 2148/2654	67 % 536/792	1,877
fr.ensimag.ima.pseudocode	26	73 % 136/184	77 % 17/22	1,181
fr.ensimag.ima.pseudocode.instructions	54	69 % 77/111	N/A N/A	1

FIGURE 2 – Le résultat de Cobertura.

4 Résultats de Cobertura

Cobertura permet de tester chaque ligne de code en créant les tests unitaire.

4.0.1 L'utilisation basique :

— @Mock (Mockito) : qui crée les objets factices.

— @Before(Junit) : qui permet de mettre en place les conditions ou les variables prédéfinies.

— @Test(Junit) : où nous pouvons créer les tests unitaires.

4.0.2 Les tests en Junit :

— opération arithmétique : comme plus, soustraction, diviser, multiplier et moduler.

— affection : assign.

— opération binaire pour la comparaison : comme "equal", "greater", "lower".

— opération unitaire : comme "not" et "minus".

4.0.3 Le résultat :

Nous avons créé les tests en Junit qui vérifient principalement les résultats de l'opération binaire en comparant le résultat attendu et le résultat exécuté. En même temps, nous avons aussi construit pleins de tests écrit en Deca. A la fin, nous avons obtenu un résultat de Cobertura dont le ratio de la couverture de test est affiché dans la figure 2.