ELEFTHERIOU Theophanis - ELIEZER Gwennan - GINDRIER Clément
PETITEAU Matthieu - POLICET Rémi

DOCUMENTATION DE VALIDATION

Vous trouverez dans ce document les méthodes et tests utilisés pour valider notre produit

Equipe GL60

GRENOBLE INP UGA Ensimag

SOMMAIRE

Descriptifs des tests **2**
Types de tests 2
Étape A 2
Étape B : 2
Étape C : 3
Tests d'intégration : 3
Organisation des tests 3
Scripts de tests **4**
Gestion des risques et gestion des rendus **5**
Plan de gestion des risques 5
Gestion des mises en production 5
Couverture des tests (Jacoco) **6**
Méthodes de validation alternatives **6**

Descriptifs des tests

Types de tests

Nous avons mis en place des tests à faire à la main, ainsi que des tests automatiques. Certains tests sont spécifiques à une étape, d'autres sont plus généraux et testent decac en entier. Les tests lancés à partir de scripts qui sont automatiques se trouvent dans la section **scripts de tests**, et nous avons listé ici les autres tests.

Étape A

test_lex : Ce test permet de tester le lexer et vérifie que les tokens appelés sont bien reconnus

test_synt : Ce test permet de tester le parser. Il construit l'arbre abstrait non décoré d'un fichier deca en vérifiant la grammaire et affiche l'arbre. Si le test ne passe pas, il est alors possible de débugger le parser et s'il passe, il faut observer que l'arbre construit correspond parfaitement à l'arbre attendu.

decac -p : Décompile l'arbre abstrait obtenu. On peut donc ensuite comparer le programme obtenu en sortie avec le programme saisi en entrée

Étape B :

L'objectif des tests de l'étape B est de déterminer si d'une part, les bonnes erreurs sont renvoyées lorsque l'arbre n'a pas une forme correcte, et d'autre part de s'assurer que l'arbre est décoré comme il faut.

Nous disposons donc pour cela de 3 types de tests

Les tests manuel : à l'aide la de la commande *" test_context file_name"* nous sommes capable de visualiser l'arbre décoré et de voir si tout les nœuds ont bien été ajouté comme il faut, et qu'ils ont le bon type.

Les tests automatique invalide : Cette batterie de tests sert à vérifier que les bonnes erreurs sont renvoyées aux bon endroits. On utilise donc un script de test automatique qui exécute la commande *"decac file_name"* sur tous les tests de la banque de test et qui compare la sortie erreur avec l'erreur attendue pour le fichier.

Ainsi, on possède deux répertoire, l'un contenant tous les fichiers .deca et l'autre contenant toutes les solutions attendues.

On utilise la commande *"test_context_auto"* pour lancer la comparaison entre résultat obtenu et attendu sur toute la batterie de fichier.

Les tests automatiques valides : Cette batterie de tests garantit que l'étape B compile sans erreur et qu'elle contient bien le résultat nécessaire pour l'étape C. Les tests d'intégrations qui garantissent que la structure de l'arbre décoré est bonne et possèdent les bonnes décorations.

Étape C :

Pour l'étape C spécifiquement, nous avons fait des classes java qui construisent des arbres décorés pouvant sortir de l'étape B, et qui sont inclus dans "mvn test". Ces tests ont été utilisés pour valider le langage Sans-Objet, mais demandaient trop de travail pour le langage essentiel. Ces 40 tests ont été très utiles pour avancer la partie C sans attendre les parties A et B. Les tests les plus basiques ne sont pas inclus dans les tests automatiques.
Pour lancer ces tests facilement à la main, nous avons fait le script test_codegen.
Le reste des tests ont pour but de vérifier la bonne exécution du code déca généré. Ce sont donc des tests automatiques, lancés via des scripts.

Tests d'intégration :

Pour tester decac en entier, nous avons fait plusieurs scripts, qui sont tous automatisables :
Nous avons également créé un script nommé "tester", qui est capable de lancer tous les tests automatisables sur les dossiers qui les correspondent. "tester" est aussi capable de ne lancer qu'une sélection de tests, marqués comme tests de non-régression, via la commande "tester reg". Grâce à ce système, nous pouvons faire des tests plus grands et plus rapides que plein de tests individuels, ce qui va bien pour des tests exécutés fréquemment. Malheureusement, cette liste n'est actuellement pas à jour.

Organisation des tests

Nous avons organisé les tests par partie. Les tests de l'étape A sont dans les dossiers src/test/deca/lexer et src/test/deca/syntax selon qu'ils servent à tester le lexer ou le parser. Les tests lexer ne comportent souvent qu'un seul token et ne servent qu'à tester le lexer. Les tests parser n'ont pas non plus vocation à être passés par les étapes suivantes et ne servent qu'à tester le parser et le lexer.
Les tests vis-à-vis de l'étape B se trouvent quant à eux dans le dossier src/test/deca/context.
Et finalement, les tests pour l'étape C se trouvent dans le dossier src/test/deca/codegen. Ils contiennent des tests passant par decac tout entier, mais sont conçus pour tester des fonctionnalités de l'étape C.
Nous avons aussi src/test/deca/exec dont la visée est de tester l'exécution d'ima sur le résultat de decac, dans le but de tester decac tout entier, et src/test/deca/linker, qui est comme src/test/deca/exec, mais en utilisant linker.

Scripts de tests

Comment faire passer tous les tests

Nous avons fait de nombreux scripts de test automatiques spécialisés:

- test_lexer : Ce script de test a pour but d'automatiser test_lex. Ayant un programme deca et une suite de lexèmes, test_lexer est capable de comparer le résultat de test_lex et la suite de lexèmes.
- test_decompile : Ce script a pour but d'automatiser "decac -p". Il lance "decac -p" deux fois à la suite (la deuxième compilation étant faite sur la sortie de la première), et compare le fichier produit par la première compilation et celui de la deuxième compilation.
- test_exec : ce test lance decac, puis lance ima sur le fichier résultant. Il compare ensuite la sortie d'ima à un fichier donné.
- test_context : permet de visualiser l'arbre abstrait décoré obtenu en fin de parti B, à partir d'un fichier deca
- test_context_auto : permet de tester toutes les sorties erreurs obtenues en Etape B lorsque l'arbre n'a pas la bonne forme, c'est-à-dire que le code ne respecte pas la structure imposée.
- test_run_decac : comme test_exec, mais permet de donner une entrée à l'exécution d'ima
- test_run_cli : Lance decac avec les arguments donnés, et compare la sortie standard et la sortie d'erreur à une solution donnée (sans lancer ima). Utile pour les tests d'erreurs de compilation
- test_run_linker : Comme test_run_decac, lance decac puis ima, mais ajoute également linker au milieu pour faire les éditions de liens requises

Tous ces scripts automatiques peuvent être lancés indépendamment, mais ils peuvent aussi être lancés en même temps grâce au script "tester". Ce script recense tous les scripts de test automatique, et les dossiers sur lesquels les lancer. Lorsque l'on exécute "tester", il va lancer chacun d'entre eux, et une fois terminé il va indiquer tous les scripts qui ont terminé avec une erreur (c'est-à-dire tous les scripts dont au moins un test n'est pas passé).
Dans chacun des dossiers, il est également possible de mettre un fichier "regression.txt", contenant une liste de tests. Ce fichier sera lu par "tester" lorsqu'il est lancé avec l'argument "reg" ou "régression", et son contenu sera ajouté en argument à la commande de test lancée sur ce dossier. Tous les scripts de tests sont capables de prendre une liste de tests en plus du dossier sur lequel travailler en argument, et peut ajuster son exécution pour se limiter aux tests donnés.

Gestion des risques et gestion des rendus

Plan de gestion des risques

RISQUE	ACTIONS
Dépasser les deadlines	Vérifier le planning Évoquer les dates importantes lors des réunions quotidiennes
Avoir un code non fonctionnel sur la branche master	Mise en place de différentes branches avec une hiérarchie Test de la branche master avant les rendus
Avoir un code non fonctionnel sur l'environnement de travail demandé	Test sur les machines de l'Ensimag couramment
Duplication de certaines tâches due à une mauvaise communication	Récapitulatif du travail de chacun lors des réunions quotidiennes Planification
Oublier de faire une tâche/de rendre un document	Les chefs de projet doivent avoir en tête les différents rendus, les attentes pour les suivis, les spécificités nécessaires, etc En parler lors des réunions quotidiennes à l'approche des deadlines
Respecter le format imposé (pour les messages d'erreurs par exemple)	Bien lire le poly en particulier pour sa partie

Gestion des mises en production

- S'assurer que la version présente sur la branche master est la bonne :
  Permet de se rendre compte d'un oubli de merge
- Tester l'ensemble des programmes sur une machine de l'Ensimag :
  Permet de s'assurer de la compatibilité du compilateur sur l'environnement de travail proposé
- Rendre le dépôt final une heure avant la deadline :
  Permet de pouvoir gérer quelques problèmes de dernières minutes

Couverture des tests (Jacoco)

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.
fr.ensimag.deca.syntax		70 %		50 %
fr.ensimag.deca.tree		88 %		79 %
fr.ensimag.deca.codegen		78 %		54 %
fr.ensimag.deca		77 %		70 %
fr.ensimag.deca.context		70 %		70 %
fr.ensimag.ima.pseudocode		81 %		75 %
fr.ensimag.ima.pseudocode.instructions		70 %		n/a
fr.ensimag.deca.tools		93 %		87 %
Total	8 024 of 32 627	75 %	965 of 2 235	56 %

19 945/32 627 instructions correspondent au deca.syntax

fr.ensimag.deca.syntax

Element	Missed Instructions	Cov.	Missed Branches	Cov.
DecaImportParser		75 %		48 %
DecaParser		79 %		51 %

Méthodes de validation alternatives

En plus de tester tout notre code, nous avons validé notre travail par la relecture d'un 'naïf'. Cette méthode consiste à présenter notre code à quelqu'un ayant déjà travaillé sur la même partie (et donc ayant une idée précise du code à produire) mais pas sur le code en question. Cette méthode nous a permis de réaliser de nombreux débogages rapidement. En effet le 'naïf' possède une vision libre de toute idée préconçue et permet souvent de déboguer encore plus rapidement qu'avec les tests.