Validation

Documentation des méthodes de validation du compilateur

Sommaire

Sommaire 1

Descriptif des tests 1

Types de tests 1

Organisation des tests 2

Objectifs des tests 4

Les scripts de tests 5

Gestion des risques et gestion des rendus 6

I. Gestion des risques 6

A. Risques liés à la gestion de projet 6

B. Risques liés à la qualité du produit rendu 7

II. Gestion des rendus 9

Résultats de Jacoco 10

Autres méthodes de validation 10

Descriptif des tests

Types de tests

Afin de tester la validité du compilateur, des tests systèmes ont été mis en place en implémentant en bash l'exécution des différentes étapes du compilateur. Ainsi, pour chaque étape, des scripts bash permettent de s'assurer de la validité de chaque fonctionnalité du compilateur et permettent de détecter d'éventuelles erreurs à chaque étape de la compilation. Cela revient à tester le lexer, le parser, l'analyse contextuelle et la génération de code sur un certain nombre de fichiers `.deca` qui sont organisés comme suit.

Organisation des tests

Les fichiers de test `.deca` sont séparés selon s'ils testent la partie A (syntax), B (context) ou C (codegen).

Ainsi :

- `src/test/deca/syntax/` contient tous les fichiers de tests `.deca` qui testent la syntaxe (partie A)
  - Dans `syntax/valid/`, on retrouve les tests qui correspondent à un code Deca syntaxiquement (a fortiori lexicalement) correct (le dossier `provided/` contient les tests fournis par les enseignants).
  - Dans `syntax/invalid/`, on retrouve les tests qui correspondent à un code Deca syntaxiquement incorrect mais lexicalement correct (le dossier `provided/` contient les tests fournis par les enseignants). Le sous-dossier `syntax/invalid/lexer/` contient les tests lexicalement incorrects.
  - Dans chacun de ces sous-répertoires, on a deux dossiers (`without_class/` et `class/`) qui contiennent respectivement les tests du compilateurs sans objet et avec objet.
- `src/test/deca/context/` contient tous les fichiers de tests `.deca` syntaxiquement corrects qui testent le contexte (partie B)
  - Dans `context/valid/`, on retrouve les tests qui correspondent à un code Deca contextuellement correct (le dossier `provided/` contient les tests fournis par les enseignants).
  - Dans `context/invalid/`, on retrouve les tests qui correspondent à un

code Deca contextuellement incorrect (le dossier **provided/** contient les tests fournis par les enseignants).
    - Dans chacun de ces sous-répertoires, on a deux dossiers (**without_class/** et **class/**) qui contiennent respectivement les tests du compilateurs sans objet et avec objet.
- **src/test/deca/codegen/** contient tous les fichiers de tests **.deca** syntaxiquement corrects qui testent le contexte (partie B)
    - Dans **codegen/valid/**, on retrouve les tests qui correspondent à un code Deca qui compile et s'exécute correctement (le dossier **provided/** contient les tests fournis par les enseignants).
    - Dans **codegen/invalid/**, on retrouve les tests qui correspondent à un code Deca contextuellement correct, qui compile mais renvoie une erreur à l'exécution (exemple : division par zéro).
    - Dans **codegen/interactive/**, on retrouve les tests qui demandent une entrée (parce que le code contient un **readInt** ou un **readFloat**). Les tests se lançant automatiquement, des entrées ont été prédéfinies dans le répertoire **src/test/script/input/**. Ces tests compilent et s'exécutent correctement.
    - Dans **codegen/perf/**, on retrouve les tests qui compilent et s'exécutent correctement utilisés pour les tests de performance.
    - Dans les sous-répertoires **invalid/** et **valid/**, on a deux dossiers (**without_class/** et **class/**) qui contiennent respectivement les tests du compilateurs sans objet et avec objet.

En ce qui concerne les tests de l'extension, ils ont été séparés des autres tests en dans le répertoire **src/test/deca/extension/**. À partir de ce répertoire, on retrouve la même organisation des fichiers de tests que dans **src/test/deca/** avec puisqu'on a à nouveau les dossiers **syntax/**, **context/**, **codegen/**, et chacun de ces dossiers a les mêmes sous-dossiers que précédemment.

Il existe également une organisation des fichiers de sorties des tests générés dans le répertoire **src/test/script/output/** :

- Dans **output/token/**, on retrouve les fichiers de sorties **.lis** suite à l'exécution de **token_test** (cf II. Les scripts de test) avec la sortie des tests valides dans le sous-répertoire **valid/** et la sortie des tests invalides dans **invalid/**.
- Dans **output/syntax/**, on retrouve les fichiers de sorties **.lis** suite à l'exécution de **syntax_test** (cf II. Les scripts de test) avec la sortie des tests valides dans le sous-répertoire **valid/** et la sortie des tests invalides dans **invalid/**.
- Dans **output/context/**, on retrouve les fichiers de sorties **.lis** suite à l'exécution de **context_test** (cf II. Les scripts de test) avec la sortie des tests valides dans le sous-répertoire **valid/** et la sortie des tests invalides dans **invalid/**.
- Dans **output/codegen/**, on retrouve les fichiers de sorties **.ass** suite à l'exécution de **codegen_test** (cf II. Les scripts de test) avec la sortie des tests valides dans le sous-répertoire **valid/** et la sortie des tests invalides dans **invalid/**. Ces sorties correspondent au code assembleur généré par le fichier de tests.
- Dans **output/ima/**, on retrouve les fichiers de sorties **.res** suite à l'exécution de **codegen_test** (cf II. Les scripts de test) avec la sortie des tests valides dans le sous-répertoire **valid/** et la sortie des tests invalides dans **invalid/**. Ces sorties correspondent au résultat de l'exécution du code assembleur généré dans **output/gencode/**.

Remarque : La sortie des tests de l'extension se trouve également dans **output/** et n'est pas dissociée de la sortie des autres tests.

Ensuite dans le répertoire **src/test/script/input/**, on retrouve les fichiers que l'on passe en entrée des tests qui en ont besoin (ceux qui sont dans **deca/codegen/interactive/**).

Enfin, dans **src/test/script/expected/**, on retrouve les résultats attendus pour certains tests interactifs et invalides de **deca/codegen/** dont la sortie est sur plusieurs lignes.

Objectifs des tests

L'objectif des tests est de pouvoir vérifier la validité de la sortie du programme à chaque étape de la compilation. En effet, en ajoutant des tests, on s'assure qu'ils testent la bonne étape de la compilation. Par exemple, on ne va pas mettre dans `test/deca/context/` un test syntaxiquement incorrect.

Ensuite, les tests permettent de s'assurer que chaque fonctionnalité du compilateur est opérationnelle dans tous les cas possibles pour chacune des étapes de la compilation. Ainsi, avant de commencer à implémenter une fonctionnalité du compilateur, un test qui assure le bon fonctionnement de la nouvelle feature est créé et donne un objectif à atteindre.

Par exemple, pour le test du `AND`, on va effectuer des tests avec une table de vérité avant de l'implémenter. On effectue les tests de chaque étape sur ce fichier de test et on s'assure que le test passe.

Les scripts de tests

Tout d'abord, les tests fournis par les enseignants permettent de tester la validité de la sortie du lexer, du parser et du contexte en automatisation la compilation des classes java correspondant à ces différentes parties dans un fichier bash. On peut ainsi effectuer des tests sur un fichier de test `.deca` particulier `<mon_fichier_test>` représente le fichier `.deca` que l'on veut tester en indiquant le chemin en partant de la racine du projet :

- `test_lex <mon_fichier_test>` → pour tester le lexer (partie A)
- `test_synt <mon_fichier_test>` → pour tester le parser (partie A)
- `test_context <mon_fichier_test>` → pour tester le contexte (partie B)

Afin de pouvoir exécuter les tests, il est nécessaire de compiler le projet au préalable avec les commandes `mvn compile` et `mvn test-compile`.

Ensuite, afin de lancer plusieurs tests simultanément, une automatisation de tests a été mise en place afin de lancer les tests d'une étape sur un certains nombres de fichiers tests en indiquant la réussite du test ou non. La sortie de l'exécution de

chaque test est sauvegardée dans le répertoire `src/test/script/output/` afin d'avoir accès aux erreurs en cas d'échec du test. Ainsi, on peut lancer l'automatisation des tests avec les commandes suivantes :

- `token_test` → pour tester le lexer (partie A)
- `syntax_test` → pour tester le parser (partie A)
- `context_test` → pour tester le contexte (partie B)
- `codegen_test` → pour tester la génération du code assembleur (partie C) et l'exécution avec ima

Ces quatre commandes sans paramètre vont lancer uniquement les tests du compilateur sans l'extension. Avec l'ajout du paramètre `-t`, les tests lancés seront uniquement ceux de l'extension. Et enfin, avec le paramètre `-a`, tous les tests seront lancés.

La commande `compile_test` permet de tester les différents paramètres de `decac` (la compilation de fichiers en parallèle, la décompilation, ...).

Ensuite, il est également possible de lancer tous les tests fournis par les enseignants avec la commande `teachers_test`.

Pour nettoyer le répertoire `src/test/script/output/` contenant tous les fichiers générés lors de l'exécution des tests, utiliser la commande `clean_test`.

Enfin, afin de lancer tous les tests qui assurent la validité du compilateur, l'outil Maven permet d'automatiser cela. On a également ajouter l'outil jacoco qui va mesurer la couverture de tests. Ainsi, en exécutant le script `jacoco` avec la commande `jacoco`, on nettoie les fichiers générés dans `target/`, on lance tous les tests en activant jacoco avec `mvn -Djacoco.skip=false verify`, on supprime toutes les sorties des tests générés dans `output/` et on affiche le rapport généré par jacoco dans Firefox.

En lançant les commandes `mvn compile` et `mvn test-compile`, on compile le projet. Et, afin d'ignorer les erreurs des tests unitaires, on peut lancer la commande `mvn test -Dmaven.test.failure.ignore` pour lancer tous les tests automatisés.

Remarque : Toutes les commandes précédentes de cette section sont des fichiers bash situés dans le répertoire `src/test/script/launchers/`. Afin de pouvoir lancer les tests sans écrire à chaque fois le chemin vers les fichiers d'automatisation des tests, il est nécessaire de rajouter ce chemin dans le PATH.

Autre remarque : Étant donné que les tests invalides de syntaxe, de contexte et de génération d'assembleur doivent être lexicalement corrects, en lançant `token_test`, on va effectuer le test du lexer sur tous ces fichiers. De même, les tests invalides de contexte et de génération de code doivent être syntaxiquement corrects. Ainsi, `syntax_test` va tester la syntaxe de tous ces fichiers.

Gestion des risques et gestion des rendus

I. Gestion des risques

A. Risques liés à la gestion de projet

1. Oublier la date d'un rendu et son contenu

Un tableau des tâches avec un agenda a été mis en place par l'équipe afin de toujours avoir en tête les différentes échéances pour les rendus et suivis. Chaque échéance s'accompagne d'une description qui permet aux membres de l'équipe de toujours avoir sous la main les différentes modalités de rendu. Cette organisation a été choisie car elle permet de synthétiser le sujet du projet en retenant les informations essentielles pour chaque échéance.

2. Empiéter sur les tâches des autres

Chaque jour, une personne de l'équipe est chargée de faire respecter l'assignation des tâches. Nous avons fait en sorte de répartir les tâches de sorte qu'elles soient indépendantes les unes des autres. C'est pourquoi, la plupart du temps, les tâches sont effectuées à plusieurs.

B. Risques liés à la qualité du produit rendu

1. Avoir un compilateur qui ne détecte pas certaines erreurs de syntaxe (manque de robustesse du code)

[RISQUE D'APPARITION : 8/10]

[NIVEAU DE GÊNE OCCASIONNÉE : 4/10]

Avoir certains cas de figures du langage qui ne sont pas respectés dans notre code est sans doute très probable. De plus, ces cas particuliers qui 'cassent' notre programme sont souvent difficiles à détecter, c'est pourquoi la gêne occasionnée par ces derniers n'est pas très élevée.

Afin de garder notre analyseur syntaxique (syntaxes concrète, abstraite et contextuelle) le plus robuste possible, nous avons mis en place une base de tests conséquente pour chaque étape de programmation du compilateur (A, B et C). Chaque test généré pour une étape de compilation doit être validé par les étapes précédentes (par exemple, il est important de vérifier qu'un test concernant une erreur testée par la syntaxe contextuelle génère une erreur à cette étape-là et non à l'étape A, auquel cas le test est soit mal défini, soit il y a un problème dans l'étape A).

De plus, chaque test correspond à une fonctionnalité implémentée et une fonctionnalité implémentée contient plusieurs tests dans la plupart des cas car cette dernière implique souvent plusieurs cas particuliers. Par exemple, pour l'étape B avec la vérification de la syntaxe contextuelle, nous avons effectué plusieurs tests pour chaque erreur contextuelle existante dans la syntaxe (une erreur de type **INVALID_CAST** contient donc plusieurs cas de figures chacun présents dans un test dans `src/test/deca/context/invalid/`).

2. Avoir un compilateur qui ne passe pas les tests fournis par les professeurs

[RISQUE D'APPARITION : 3/10]

[NIVEAU DE GÊNE OCCASIONNÉE : 10/10]

Les tests fournis par les professeurs sont testés au moins à chaque demi-journée par

un des membres de l'équipe afin de s'assurer que le code écrit ne casse pas les fonctionnalités de base du compilateur. Ainsi, la probabilité que les tests des professeurs ne passent pas lors d'un rendu est assez faible.

Toutefois, si cela devait arriver, cela serait très embêtant pour l'équipe, c'est pourquoi nous faisons attention de lancer les tests régulièrement et sur plusieurs machines.

3. **Éviter les erreurs triviales qui empêchent la compilation du code**

[RISQUE D'APPARITION : 1/10]

[NIVEAU DE GÊNE OCCASIONNÉE : 10/10]

Afin d'éviter les erreurs triviales (de type : "oublier un ;") qui empêchent la compilation du code, tous les membres de l'équipe sont obligés de vérifier que leur code compile en testant ce dernier sur des tests de base déjà créés avant d'envoyer leur code sur gitlab. Cela permet d'éviter ce type d'erreurs qui rendent le compilateur tout simplement inutilisable.

Cette procédure permet donc d'éviter quasiment tout le temps ce type d'erreurs, c'est pourquoi le risque d'apparition est très faible.

4. **Partager du code sur git qui casse celui des autres**

[RISQUE D'APPARITION : 7/10]

[NIVEAU DE GÊNE OCCASIONNÉE : 7/10]

Partager du code qui casse celui des autres est assez embêtant. C'est pourquoi, il est recommandé aux membres de l'équipe de créer leur propre branche pour effectuer leurs tests jusqu'à que leur implémentation soit valide et **surtout** valide par rapport au contenu présent sur la branche principale, **main** (ou master, pour gitlab).

En pratique, chaque membre du groupe se doit de vérifier que leur code ne casse pas en lançant les tests déjà présents sur la branche principale, en plus des tests propres à la fonctionnalité implémentée. Si un des tests ne passe pas, le membre doit modifier et corriger son code jusqu'à ce que tous les tests présents soient valides. Ceci permet de

garder une certaine intégrité dans la base de test et dans le code : chaque ajout de fonctionnalité ne doit pas empêcher la validation des fonctionnalités déjà présentes.

II. Gestion des rendus

Pour chaque échéance (rendu intermédiaire ou suivis de projet), l'équipe effectue une réunion générale (souvent la veille ou 2 jours avant l'échéance) et met en place une routine (checklist) d'actions à effectuer afin de ne rien oublier :

1) Parcourir la liste des tâches à effectuer pour la partie **programmation** pour le rendu et vérifier que ces dernières ont toutes été réalisées correctement et complètement. La vérification d'une tâche assignée est réalisée par un autre membre que celui qui a fait la tâche. Ceci permet de régler les quelques soucis de dernière minute liés à l'interprétation du sujet, par exemple.
2) En plus de la vérification des tâches liées à la programmation, l'équipe doit s'assurer que la base de tests fournie à la date du rendu est valide et suffisamment complète.
3) Une fois que les tests et le code ont été vérifiés, l'équipe passe en revue toutes les fonctionnalités du compilateur qu'elle avait définies au préalable pour le sprint courant. Ceci permet de préparer le contenu du rendu lorsque ce dernier consiste en une séance de suivi et permet également de constater notre avancement et de revoir notre gestion des tâches au besoin.
4) Une fois la partie programmation vérifiée, l'équipe va s'intéresser à la partie 'management' afin de se pencher sur les livrables à fournir pour ladite échéance. Si un livrable nécessite du travail en amont et a été assigné à un membre avant la réunion (par exemple : une documentation), plusieurs membres du groupe vérifieront si ce travail respecte les consignes du sujet avec la personne qui a réalisé la tâche. S'il s'agit d'un livrable pour exposer l'avancement de l'équipe (par exemple : des diapositives, le contenu de notre présentation de suivi…), l'équipe se concerte et s'assigne les tâches à effectuer lors de la réunion pour que la présentation soit prête lors de l'échéance.

Résultats de Jacoco

Avec la commande **`jacoco`**, on lance tous les tests et on affiche le rapport de jacoco dans Firefox qui permet d'indiquer le couverture de test mesurée. On obtient les résultats suivants :

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		68%		49%	500	679	667	1,939	254	367	6	48
fr.ensimag.deca.tree		89%		86%	63	423	92	1,016	40	326	3	67
fr.ensimag.deca.context		60%		22%	76	138	107	261	57	116	2	21
fr.ensimag.deca		80%		90%	13	74	36	203	6	36	1	6
fr.ensimag.deca.memory		94%		85%	15	110	10	227	2	61	0	5
fr.ensimag.deca.codegen		96%		100%	1	18	3	64	1	14	0	3
fr.ensimag.deca.tools		94%		100%	1	16	3	38	1	13	0	3
Total	4,212 of 17,719	76%	367 of 990	62%	669	1,458	918	3,748	361	933	12	153

Pour le moment, le code qui n'est pas encore testé correspond aux classes qui ont été implémentées mais les tests ne sont pas encore présents dans la batterie de tests.

Autres méthodes de validation

On n'en a pas encore.