Projet Génie Logiciel **Documentation de validation**

Fait par les étudiants

Mohamed Hadi Chamsi
Li Rong
Oussama Lamzaouri
Ziad Razani
Safia Echarif

Janvier 2022

Table des matières

1 Tests

1.1 Descriptif des tests

Comme notre compilateur est composée de plusieurs étapes, il était indispesable, pour assurer son bon fonctionnement, de tester chacune de ses étapes. Pour cela, il fallait préparer une batterie de tests importantes contenant des tests valides et invalides pour chaque étape. Et puisque l'ensemble des tests ne peuvent pas couvrir la totalité des cas possibles, il reste nécessaire de couvrir une grande partie.

1.1.1 types de tests

Au sein du projet, nous pouvons trouver des tests unitaires par etape :

- Tests pour l'analyse syntaxico-lexicale(étape A).
- Tests pour l'analyse contextuelle (étape B).
- Tests pour la generation du code assembleur et la semantique (étape C).

En plus nous trouverons des scripts permettant de vérifier tester le système, autrement dit, ils nous permettent de s'assurer que toutes les tests passent, après la mise en place d'une nouvelle fonctionnalité.

1.1.2 organisation de tests

les tests sont regroupés par étape, ils sont donc 3 et tous dans le réportoire src/test/deca :

- l'analyse syntaxico-lexicale (étape A) :
  - src/test/deca/syntax/invalid : contient les tests syntaxiquement invalides (même si c'est lexiquement correctes).
  - src/test/deca/syntax/valid : contient les tests syntaxiquement et lexicalement correctes.
- l'analyse contextuelle (étape B) :
  - src/test/deca/context/invalid : contient les tests contextuellement invalides .
  - src/test/deca/context/valid : contient les tests contextuellement correctes.
- la generation du code assembleur et la semantique (étape C) :
  - src/test/deca/codegen/invalid : les tests invalides pour la géneration du code.
  - src/test/deca/codegen/valid : les tests valides pour la géneration du code.

1.2 Les scripts de tests

Pour éxecuter l'ensemble de jeu des tests on lace la commande : **mvn test**. Cependant il est aussi possible de lancer les tests par étape :

- pour lancer les tests syntaxiquement correctes, on exécute le script synt_valid.sh à l'aide de la commande : **synt_valid.sh**.

— pour lancer les tests syntaxiquement incorrectes, on exécute le script synt_invalid.sh à l'aide de la commande :**synt_invalid.sh**.

— pour lancer les tests contextuellement correctes, on exécute le script cont_valid.sh à l'aide de la commande : **cont_valid.sh**.

— pour lancer les tests contextuellement incorrectes, on exécute le script cont_invalid.sh à laide de la commande :**cont_invalid.sh**.

— pour lancer les tests de generation de code correctes, on exécute le script cont_valid.sh à l'aide de la commande : **cont_valid.sh**.

— pour lancer les tests de generation de code incorrectes, on exécute le script cont_invalid.sh à laide de la commande :**cont_invalid.sh**.

Lors de l'exécution du code des différents scripts, un indicateur visuel est présent pour s'assurer que les tests passent bien, de plus chaque nom fichier est préfixé par un **OK** si le test a le résultat attendu sur le fichier, sinon le script s'arrête sur le fichier qui pose problème.

```
ensimag@linux:~/Projet_GL$ synt_valid.sh
OK ./src/test/deca/syntax/valid/2_class.deca
OK ./src/test/deca/syntax/valid/Instanceof.deca
OK ./src/test/deca/syntax/valid/asm.deca
OK ./src/test/deca/syntax/valid/assign.deca
OK ./src/test/deca/syntax/valid/bad_order.deca
OK ./src/test/deca/syntax/valid/basic_hello.deca
OK ./src/test/deca/syntax/valid/cast.deca
OK ./src/test/deca/syntax/valid/cast_class_2.deca
OK ./src/test/deca/syntax/valid/castbetweenclass.deca
OK ./src/test/deca/syntax/valid/champ.deca
OK ./src/test/deca/syntax/valid/class_basic.deca
OK ./src/test/deca/syntax/valid/class_basic_1.deca
OK ./src/test/deca/syntax/valid/class_vide.deca
OK ./src/test/deca/syntax/valid/compatible_types.deca
OK ./src/test/deca/syntax/valid/complex_class.deca
OK ./src/test/deca/syntax/valid/complex_class_1.deca
OK ./src/test/deca/syntax/valid/convFloat.deca
OK ./src/test/deca/syntax/valid/decl_var.deca
OK ./src/test/deca/syntax/valid/decla_var_2.deca
OK ./src/test/deca/syntax/valid/empty_class.deca
OK ./src/test/deca/syntax/valid/empty_file.deca
OK ./src/test/deca/syntax/valid/empty_file_1.deca
OK ./src/test/deca/syntax/valid/empty_main.deca
OK ./src/test/deca/syntax/valid/equals.deca
```

2 Gestion des risques

Dans un projet aussi vaste que le génie logiciel, il est nécessaire d'établir des règles ainsi que des méthodologies précises, car le compilateur à mettre en œuvre est un programme impossible à tester de manière exhaustive. Du point de vue de la programmation, les erreurs de programmation sont variées et parfois difficiles à détecter. En plus des erreurs de programmation, nous avons également des risques humains ou liés à la gestion du projet à gérer. C'est pourquoi nous voyons qu'il est indispensable d'avoir une bonne gestion des risques dans un projet, car la meilleure manière d'éviter les erreurs, c'est de ne pas les faire. Par exemple pour **synt_valid.sh** :

2.1 Risques potentiels

— Perte de la synchronisation du dépôt sur GitLab

— Incapacité de détecter toutes les erreurs.

— Le non-respect de la Date limite du rendu final.

— Conflits au sein de l'équipe donc perte de temps.

— Incapacité de l'équipe ou d'un membre à réaliser une tâche / Mauvaise affectation des tâches.

— Erreur dans la classification des tests.

— Ambiguïté au niveau des spécifications par rapport à l'extension.

2.2 Face aux risques

Ainsi, à chaque étape de l'analyse d'un projet, nous rassemblons les risques éventuels et les solutions que nous pouvons proposer.

2.2.1 Au niveau de la gestion de projet

On prédéfinit le planning, l'organisation et toutes les tâches pour que chaque membre du groupe travaille de manière plus efficace. Aussi, nous organise quotidiennement de réunions afin d'avoir une bonne ambiance de travail et d'assurer une bonne connaissance des avancées du projet pour chaque membre.

2.2.2 En ce qui concerne la programmation

L'élément essentiel est une vérification adéquate du code. Nous adoptons deux approches, la première étant le codage par paire pour identifier rapidement les erreurs et accélérer le processus de codage.

2.2.3 Validité du code

Une autre personne du groupe révise le code régulièrement, cela nous permet d'identifier des problèmes le plus tôt possible, l'ensemble de l'équipe peut aussi avoir une vue d'ensemble de la progression de code qui a été mis en œuvre. De plus, en en découvrant de nouvelles visions et de nouvelles façons de résoudre les problèmes, elle permet d'apporter des améliorations de code.

2.2.4 Pour GitLab

Avant de faire un dépôt d'un nouveau fichier, nous vérifions tout d'abord que le projet compile et s'exécute correctement dans la version actuelle. Ensuite, le programme doit réussir tous les tests valides. Enfin, le code doit être bien commenté et facile à comprendre afin de s'assurer des révisions ou corrections du code par les autres. De plus, nous enregistrons constamment notre dépôt durant avant les push sur GitLab pour éviter toute sorte de risque.

2.3 Outils

— Automatisation des tests

— Utilisation de Trello, qui permet de créer un tableau par User Story, donc on peut gérer des tâches sous la forme d'action à réaliser/ à tester/ terminé, s'organiser de manière très facile et lisible.

— Un Poker Planning pour la classification des taches par leur niveau de difficulté.

— Visualisation des Burndown Chart pour la gestion d'effort ainsi que l'efficacité durant les heures de travail.

3 Gestion des rendus

Avant chaque rendu intermédiaire, il est nécessaire de s'assurer que le code rendu fonctionne parfaitement. Ensuite, le code doit être testé et validé pour s'assurer qu'il répond au besoin.

☐ Chacun est dans la branche la plus récente (the latest branch), a bien récupéré la dernière version du projet

☐ Vérifier mvn compile

☐ Vérifier passer tous les tests

☐ Si un test loupe, on l'exécute seul pour trouver l'erreur

☐ Corriger cette erreur et relance le test jusqu'à tous les tests sont passés

☐ Assurer le code est bien commenté et facile à comprendre

☐ Faire git commit et push dans le dépôt GitLab.

☐ Mettre à jour Trello et Realisation.planner afin d'éviter la duplication du travail

☐ Examen des documentations

3.1 Résultats de Jacoco :

Après génération, les rapports de couverture obtenus sont les suivants :

Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.tree		68%		59%	494	1,043	733	2,593	102	426	0	86
fr.ensimag.deca.syntax		75%		54%	577	782	475	2,032	247	372	2	50
fr.ensimag.deca		67%		50%	45	94	96	260	11	50	2	5
fr.ensimag.ima.pseudocode		74%		75%	29	88	48	185	24	78	2	26
fr.ensimag.deca.context		83%		67%	42	151	39	224	25	117	1	21
fr.ensimag.ima.pseudocode.instructions		56%		n/a	26	62	48	111	26	62	19	54
fr.ensimag.deca.tools		36%		33%	10	16	27	39	8	13	1	3
fr.ensimag.deca.codegen		88%		92%	4	35	8	99	2	21	0	2
Total	7,428 of 26,117	71%	920 of 2,200	58%	1,227	2,271	1,474	5,543	445	1,139	27	247

D'ailleurs, il est à signaler qu'il y a des parties qui sont moins pertinentes que d'autres, notamment fr.ensimag.ima.pseudocode.instructions qui n'a pas été couvert totalement car on implémenté juste la partie dont on a besoin pour la génération du code. D'autre part, les packages qui concerne les étapes : A, B et C : fr.ensimag.deca.tree,fr.ensimag.deca.syntax, fr.ensimag.deca.context, et fr.ensimag.deca.codegen , ont une couverture par jeu de tests satisfaisante

3.2 Autres Méthodes de validation utilisées

Bien plus que les tests, d'autres méthodes de validation ont été employé tout au long du projet :

- La relecture du code, par d'autres membres (c'était très pratique pour vérifier la conformité de notre code surtout pour la partie B).
- l'extension nous a permis également de valider et vérifier pas mal de cas.
- les tests de poly nous ont a permis également de détecter pas mal de défaillance et les corriger.