Documentation de validation

Document de référence des tests - Version 1.0 (release)

Table des matières

Introduction

Pourquoi tester, et quels types de tests ?

L'écriture de tests a pour objectif de s'assurer que le code développé est fonctionnel et répond aux exigences. Il existe différents types de tests, chacun ayant un objectif spécifique.

- Les tests unitaires ont pour but de tester des parties isolées du code, généralement des fonctions ou des méthodes, pour s'assurer qu'elles fonctionnent comme prévu, avant qu'elles ne soient intégrées au reste du système.
- Les tests d'intégration ont pour but de tester comment des parties distinctes du code se comportent lorsqu'elles travaillent ensemble. Ces tests permettent de s'assurer que différents modules ou composants du système fonctionnent correctement ensemble.
- Les tests système ont pour but de tester le système dans son ensemble pour s'assurer qu'il réponde aux exigences fonctionnelles.
- Il existe d’autres types de tests (acceptance, black-box), mais que nous n’avons pas utilisés dans le cadre de ce projet.

En écrivant des tests pour chaque niveau, on est sûr de couvrir les différents aspects du système en développement, et de s'assurer que tout est fonctionnel. Les tests permettent également de détecter les bugs de manière plus rapide et facile, et de vérifier si les corrections apportées ont un impact sur d'autres parties du système.

Objectifs

Lors de la réalisation de ce projet, deux objectifs majeurs ont été identifiés qui peuvent parfois entrer en conflit :

1. Assurer une qualité maximale du compilateur en effectuant un grand nombre de tests. Il est primordial que le compilateur soit fiable et stable pour garantir un fonctionnement optimal de celui-ci.
2. Minimiser l'impact énergétique du processus de développement et du produit final en cherchant à optimiser l'efficacité de celui-ci. Il est important de maintenir un bon équilibre entre ces deux objectifs pour garantir la meilleure expérience utilisateur possible.

Procédure de tests

1. Identification de la fonctionnalité à tester : Quelle méthode, classe, bibliothèque ou autre fonctionnalité doit être testée.
2. Écriture des tests : Création des cas de test pour vérifier le bon fonctionnement de la fonctionnalité en question.
3. Exécution des tests : Les tests sur le système ou sur la partie du système en cours de test sont lancés.
4. Analyse des résultats : Observation, évaluation et enregistrement des résultats des tests.
5. Résolution des problèmes : Résoudre les bugs ou les erreurs détectés lors des tests.
6. Évaluation de la couverture des tests : La couverture des tests est-elle suffisante pour garantir la qualité du logiciel ? Si nécessaire, on ajoute des cas de test pour couvrir les cas d'utilisation manquants.

Organisation des tests

Premièrement, nous avons des tests unitaires JUnit qui ont pour objectif d'assurer le bon fonctionnement des fonctions Java intervenant pendant la phase d'analyse contextuelle.

Deuxièmement, nous avons des tests d'intégration, sous forme de fichiers source Deca. Ils ont une place importante dans la validation du compilateur car ils permettent de vérifier que les modifications apportées à une partie du compilateur n'affectent pas les parties déjà fonctionnelles et testées. Ces tests sont organisés en dossiers en fonction de la phase de développement sur laquelle ils interviennent (analyse lexico-syntaxique, vérification contextuelle, génération de code) pour faciliter la compréhension de l'organisation des tests et faciliter la maintenance des tests.

D'autre part, nous avons des tests systèmes plus généraux, qui couvrent des cas d'utilisations réelles du compilateur et assurent son bon fonctionnement.

Fichiers Deca

Les tests Deca sont situés dans des sous-dossiers de `src/test/deca/syntax`, `src/test/deca/context` et `src/test/deca/codegen`.
Chacun de ces dossiers doit avoir au moins les sous-dossiers `valid/` et `invalid/`.
De plus, le dossier `src/test/deca/codegen` doit avoir les sous-dossiers : `interactive/`, `valid/`, `invalid/` et `perf/`.

Voici la hiérarchie des fichiers tests Deca :

```
📂 src/test/deca/
├── 📂 codegen/
├── 📂 context/
├── 📂 extension/
└── 📂 syntax/
```

Les fichiers Deca sont organisés de telle manière afin de distinguer les tests qui portent sur les différentes phases de la compilation. Ainsi, le dossier 📂 `syntax` contient des tests de la phase d'analyse lexico-syntaxique (cf Partie I - Analyse lexico-syntaxique). Cela nous permet aussi de savoir que sur les tests de ce dossier, nous devons utiliser le launcher `` `test_synt` ``. De la même manière, pour le dossier 📂 context (respectivement 📂 `codegen` et 📂 `extension`) nous devons utiliser le launcher `` `test_context` `` (respectivement `` `decac` ``).

Fichiers Java

```
📂 src/test/java/fr/ensimag/deca
├── 📂 context/
│   └── 📄 *.java
├── 📂 syntax/
├── 📂 tools/
└── 📂 tree/
```

Ce dossier contient des tests unitaires qui ont pour objectif de vérifier que la partie B de vérification contextuelle du compilateur (les fonctions `verify`) fonctionne correctement et conformément aux spécifications. Ces tests unitaires sont écrits en utilisant la bibliothèque JUnit pour exécuter des tests automatisés sur les différentes classes et méthodes de cette partie spécifique. Ils permettent de détecter les erreurs et les bugs dans le code de manière précoce et efficace, en s'assurant que cette partie fonctionne comme prévu.

Partie I - Analyse lexico-syntaxique

Le dossier 📂 `syntax` est organisé de la manière suivante :

```
📂 src/test/deca/syntax/
├── 📂 lexer/
│   ├── 📂 valid/
│   │   ├── 📂 invalid/
│   │   └── 📂 provided/
├── 📂 parser/
│   ├── 📂 valid/
│   ├── 📂 invalid/
└── └── 📂 provided/
```

Test analyse lexicale:

Pour la partie analyse lexicale, nous avons mené des tests visant à vérifier si tous les caractères reconnus par le lexer étaient correctement associés aux tokens appropriés, tout en s'assurant que les caractères non reconnus étaient correctement détectés et gérés. Ces tests ont permis de s'assurer que le lexer fonctionnait correctement et de détecter tout éventuel bug.

Test analyse syntaxique:

L'analyse syntaxique est la partie avec le plus de tests, il s'agit de parcourir l'arbre de grammaire abstraite, pour couvrir le maximum de cas et de combinaisons possibles. Ces tests ont permis de s'assurer que l'analyse syntaxique fonctionnait correctement, et de détecter une éventuelle syntaxe incorrecte. Nous avons également mené des tests pour des entrées invalides, pour vérifier que les combinaisons incorrectes étaient correctement détectées, gérées et affichées à l'utilisateur de manière claire.

Partie II - Vérification contextuelle

Le dossier 📂 `context` est organisé de la manière suivante :

```
📂 src/test/deca/context/
├── 📂 valid/
└── 📂 invalid/
```

Une grande partie des tests de la phase de vérification contextuelle est constituée de tests unitaires JUnit.

Analyse des règles contextuelles

L'analyse systémique des règles contextuelles nous permet de localiser les sources d'erreurs ou de comportements inattendus. Cela nous donne un cadre sur les tests à créer.

Exemple d'analyse systémique

- La règle (3.17) nous donne la condition que l'on ne peut déclarer une variable de type void**.** On appelle **decl_var** dans **list_decl_var** uniquement, on appelle **list_decl_var** uniquement dans **bloc** et on appelle **bloc** dans **method_body** et dans **main**.

Cela nous donne la séquence de dérivation suivante :

> **program** -> **list_decl_class** -> **decl_class** -> **list_decl_method** -> **decl_method** -> **method_body** OU **main** -> **bloc** -> **list_decl_var** -> **decl_var**

Ainsi, il faut vérifier que la déclaration d'une variable de type `void` dans une méthode ou dans main renvoie bien une erreur. On écrit donc les tests suivant, dans 📂 `invalid` :

> decl_var_void_in_method.deca
> decl_var_void_in_main.deca

La règle (3.17) nous donne aussi la condition que la définition de la variable n'existe pas déjà dans l'environnement passé en paramètre lors de la déclaration d'une variable dans une méthode ou dans main. Il faut vérifier que c'est bien le cas. On écrit les tests nécessaires.

Partie III - Génération de code

Le dossier 📂 `codegen` est organisé de la manière suivante :

```
📂 src/test/deca/codegen/
├── 📂 interactif/
├── 📂 invalid/
├── 📂 perf/
└── 📂 valid/
```

Pour la partie génération de code, nous avons principalement mené des tests en ciblant des fonctionnalités spécifiques et en testant toutes leurs variantes possibles (par exemple, en utilisant différents types d'arguments, ou plusieurs valeurs). Nous avons également effectué des tests d'intégration plus complexes pour simuler l'utilisation réelle de l'utilisateur avec des programmes simples. Les erreurs de génération de code qui ne sont pas détectées avant l'exécution sont plus rares que dans les autres parties, nous avons donc également effectué des tests pour détecter des cas tels que les débordements de pile ou les divisions par zéro.

Scripts d’automatisation

Le dossier 📂 `script` est organisé de la manière suivante :

```
📂 src/test/script.
├── 📂 create-modele/
├── 📂 launchers/
├── 📂 modele/
├── 📂 modele-extension/
└── 📂 provided/
```

Création des fichiers modèles (semi-automatisable)

Dans 📂 `create_modele/`, on a plusieurs scripts bash.

Le script 🔨 `create-lexer-modele.sh`, par exemple, lance le script 🔨 `test_lex` sur tous les fichiers sources deca qui concernent le lexer (i.e. appartenant au dossier 📂 `src/test/deca/syntax/lexer`). Ce script 🔨 `test_lex` créé le fichier texte qui contient la sortie générée par le lexer. Ces sorties sont placées dans le dossier 📂 `modele`.

Les autres scripts de ce dossier fonctionnent de la même manière, mais prennent en entrée des fichiers sources deca concernant leur phase respective.

⚡ **Note importante :** Il est important de noter que rien ne garantit, à ce niveau, que les sorties soient correctes, même lorsque les fichiers deca sont justes ! C’est pourquoi nous avons passé en revue ces sorties avant de les considérer comme étant des modèles de tests. Cela justifie pourquoi il ne faut pas, sans raison impérative, relancer les scripts de 📂 `create_modele/`, et aussi pourquoi on dit que ce procédé est semi-automatisable. Cela aurait pour conséquence d’effacer les sorties dont le contenu a été vérifié.

Les sorties correspondent donc au comportement du lexer (respectivement du parser) attendu pour un fichier source deca donné ! Ceci permettra donc de faire des tests de non-régression, car nous savons à présent comment notre lexer (respectivement parser) doit se comporter pour une entrée donnée.

Dossier contenant les modèles

Le dossier 📂 `modele/`, contient les modèles de sortie générés par les scripts cités ci-dessus. Ces modèles de sorties sont utilisés pour vérifier que la sortie générée par le compilateur est correcte et conforme aux spécifications. Il contient les sous-dossiers :

```
📂 src/test/script/modele.
├── 📂 codegen/
├── 📂 compilations-options/
├── 📂 context/
├── 📂 divers/
├── 📂 lexer/
└── 📂 parser/
```

De la même manière, le dossier 📂 `modele-extension` contient les modèles de référence pour l'extension du langage.

Si vous souhaitez prendre part au développement du projet, c'est ici qu'il vous faudra placer les nouveaux tests que vous créez. Pour plus d'informations, veuillez vous référer au document "Documentation de conception".

Ces dossiers sont importants pour assurer la qualité, la fiabilité et la sécurité du compilateur.

Dossier contenant les launchers

Ce dossier contient les scripts bash qui lancent les fichiers Java concernant des passes différentes de la compilation. Le script 🔨 `test_synt`, par exemple, exécute le fichier Java 📚 `ManualTestSynt.java`, qui lance le parser, à l'aide de l'instruction `` `parser.parseProgramAndManageErrors` ``.

Script tests_launcher.py

Son rôle

Le script Python 🐍 `tests_launcher.py` joue un rôle crucial dans la validation du compilateur Deca en offrant une flexibilité supplémentaire dans la manière de lancer les tests. Il permet de sélectionner précisément les dossiers et les fichiers de tests à exécuter, ainsi que d'inclure ou d'exclure des fichiers spécifiques du processus de test. De plus, il fournit une présentation claire et esthétique des résultats de test dans le terminal et permet de consulter les sorties de test plus en détail dans un fichier de log 📜 `tests.log`, facilitant ainsi le débogage. Ce script Python est donc un outil indispensable pour assurer la qualité, la fiabilité et la sécurité du compilateur Deca en effectuant des tests de manière plus efficace et flexible.

Sa sobriété

Ce script permet d'économiser de l'énergie en permettant de lancer seulement les tests concernant la partie sur laquelle l'on est en train de travailler. Cela signifie qu'au lieu d'exécuter tous les tests à chaque fois, on peut cibler uniquement les tests pertinents pour la fonctionnalité spécifique que l'on est en train de développer ou de tester. Cela permet de gagner du temps et de l'énergie, car on ne perd pas de temps à exécuter des tests qui ne sont pas pertinents pour la tâche en cours. Une fois que l'on a fini de développer et de tester la partie spécifique, on peut alors lancer tous les tests d'intégration pour s'assurer que tout fonctionne correctement ensemble.

Autres méthodes de validation

Review de code

La revue de code est une méthode importante de validation utilisée pour assurer la qualité, la fiabilité et la sécurité du code. Dans notre projet, nous avons utilisé cette méthode en effectuant des revues de code par les pairs. Cela a permis de bénéficier d'un retour critique sur notre code et de s'entraider lorsque l'on en avait besoin. Cette méthode a été efficace pour détecter les erreurs, les incohérences et les problèmes potentiels dans le code, ainsi que pour améliorer la qualité, la lisibilité et la maintenabilité du code. Elle a également contribué à renforcer la collaboration et la communication dans l'équipe.

Document *Issues*

Parmi les autres méthodes de validation utilisées dans notre projet de Génie Logiciel, nous avons mis en place un système de gestion des problèmes en utilisant un tableur “Issues” (Google Spreadsheet). Ce système permet aux testeurs de faire des demandes de corrections ou d'améliorations sur les fonctionnalités du logiciel, et permet aux développeurs de suivre et de gérer ces demandes. L'utilisation d'un tableur permet d'avoir une vue d'ensemble claire du travail restant à faire et de suivre efficacement l'avancée des corrections et des améliorations. Cette méthode a aidé à améliorer la communication et la collaboration entre les testeurs et les développeurs, en permettant une gestion efficace des problèmes et une meilleure qualité du logiciel.

Retour suivi client

Nous avons utilisé les retours des réunions de suivi avec le client pour guider notre développement en termes de priorités et pour corriger les erreurs et les bugs relevés par le client. Ces réunions nous ont permis de prendre en compte les besoins et les attentes du client, ainsi que les erreurs et les choses à fixer pour le sprint suivant. Cela nous a aidé à améliorer la qualité du logiciel en y apportant des modifications et des améliorations en fonction des retours du client.