Sommaire

1. Descriptif des tests

1.1 Types de tests

A chaque étape de développement nous avons ajouté des tests `.deca` pour vérifier le bon comportement des features, et assurer une rétrocompatibilité du code.

Des tests de comportement de types valides, et invalides vérifient les trois étapes "syntax", "context" et "codegen" d'une feature.

À noter que nous avons choisi de prioriser les tests invalides pour les étapes "syntax" et "context", et les tests valides et invalides pour l'étape "codegen", car la validité des deux premières étapes se vérifie avec celle de la dernière.

1.2 Header des tests

```
// Description:
//  Calcul de racine d'un polynôme par dichotomie
//  On cherche x tq exp(x)=2 donc Ln(2) mais en approximant
//   exp par son developpement à l'ordre 7
//
// Resultats:
//  6.93148e-01 = 0x1.62e448p-1
//
// Historique:
//    cree le 25/04/2022
```

1.3 Organisation des tests

Les tests à partir du "rendu intermédiaire 2" sont rangés dans des répertoires qui correspondent à leur fonctionnalité. Par exemple:

```
decac:
└── [syntax|context|codegen]:
    ├── valid:
    │   └── sprint:
    │       ├── feature1:
    │       │   ├── test1
    │       │   ├── test2
    │       │   └── test3
    │       │
    │       └── feature2:
    │             ├── test1
    │             ├── test2
    │             └── test3
    │
    └── invalid:
     └── sprint:
           ├── feature1:
           │   ├── test1
           │   ├── test2
           │   └── test3
           │
           └── feature2:
                 ├── test1
                 ├── test2
                 └── test3
```

Le répertoire `/codegen` contient spécifiquement un répertoire `/interactive`, qui recense les fichiers `.deca` faisant usage de `readInt()`, ou de `readFloat()`.

1.4 Objectifs des tests

L'objectif des tests est atteint lorsque tous les tests renvoient "PASSED", cela signifie que les features fonctionnent correctement. Si un test renvoie "FAILED", la feature ne fonctionne pas de la manière attendue.

Les tests sont rédigés en même temps que la feature (voire avant), de cette manière on peut corriger le code du programme pour valider les tests.

On peut attendre un résultat sous 4 formats :

- Une erreur de syntaxe ou contextuelle ⇒ le résultat attendu est stocké dans un fichier [- .lis-] et on y retrouve le nom du fichier deca testé, ainsi que la localisation de l'erreur au sein du fichier deca.

- Une erreur à l'exécution du code assembleur ⇒ un fichier [- .expected-] indique l'erreur attendue. (se référer à la classe ErrorCatcher)
- Un arbre syntaxique décoré (ou non dans le cas d'un test de syntaxe) d'un code deca conforme ⇒ le résultat attendu est stocké dans un fichier [+ .lis+]
- Le résultat de l'exécution d'un script IMA sans erreur ⇒ le résultat attendu est stocké dans un fichier [+ .expected+] et si le script IMA attend une action de l'utilisateur, toutes les entrées nécessaires au fonctionnement du script IMA sont stockées dans un fichier **.in**

2. Scripts de tests

Pour lancer tous les tests du projet il faut exécuter les commandes suivantes : `mvn clean` `mvn test`

Pour lancer un test spécifique il faut utiliser la commande : `testLauncher.sh sprint/nomFeature`

Plusieurs scripts ont été créés pour assurer la couverture de tests :

- testLauncher.sh ⇒ script qui lance individuellement les tests d'une feature.
- wrapperTestLauncher.sh ⇒ script qui fait appel à testLauncher.sh pour tester toutes les features d'un sprint, ainsi que l'option de décompilation.
- testDecacOptions.sh ⇒ script qui permet de vérifier le bon fonctionnement de toutes les options decac.
- getCoverage.sh ⇒ script appelé après la génération du rapport Jacoco, pour mettre à jour notre badge de couverture de tests, et rendre accessible le rapport Jacoco à jour.

3. Gestion des risques et gestion des rendus

3.1 Gestion des risques

Pour éviter des erreurs lors du développement des tests, nous avons décidé de travailler par groupes de deux personnes. De plus une personne est responsable du "code review", au moment de "merge" les branches. De cette manière, il y a 3 personnes qui vérifient le code ainsi que les tests, ce qui permet minimiser les erreurs et les oublis.

3.2 Gestion des rendus

La procédure pour vérifier les tests avant un rendu :

- cloner le master sur une machine propre (sans autre projet GL) dans un dossier "$HOME/Projet_GL".
- ajouter dans le fichier .bashrc :

```
JAVA_HOME=/usr/lib/jvm/java-16-openjdk-amd64
M2_HOME=/opt/maven
MAVEN_HOME=$M2_HOME
PATH=$M2_HOME/bin:"$PATH"
PATH=/matieres/3MM1PGL/global/bin:"$PATH"
PATH=$JAVA_HOME/bin:"$PATH"
PATH=$HOME/Projet_GL/src/main/bin:"$PATH"
PATH=$HOME/Projet_GL/src/test/script:"$PATH"
```

```
PATH=$HOME/Projet_GL/src/test/script/launchers:"$PATH"
export PATH
```

- relire le bash : `source ~/.bashrc`
- vérifier les commandes suivantes :

```
which mvn ⇒ /opt/maven/bin/mvn
which java ⇒ /usr/lib/jvm/java-16-openjdk-amd64/bin/java
which ima ⇒ /matieres/3MM1PGL/global/bin/ima
```

- tester ima avec : `ima -v`
- aller dans le projet, et lancer :

```
mvn clean
mvn compile
mvn test-compile
```

- lancer les tests avec : `mvn test` (voir 2. Scripts de tests pour plus de détails).

4. Résultats de Jacoco

Il est possible de générer un rapport Jacoco à partir de la commande suivante :

```
mvn clean jacoco:prepare-agent install jacoco:report
```

Jacoco nous permet d'assurer la couverture de nos tests en analysant toutes les instructions java appelées ou non lors de l'exécution des tests.

Voici le résumé de notre rapport Jacoco :

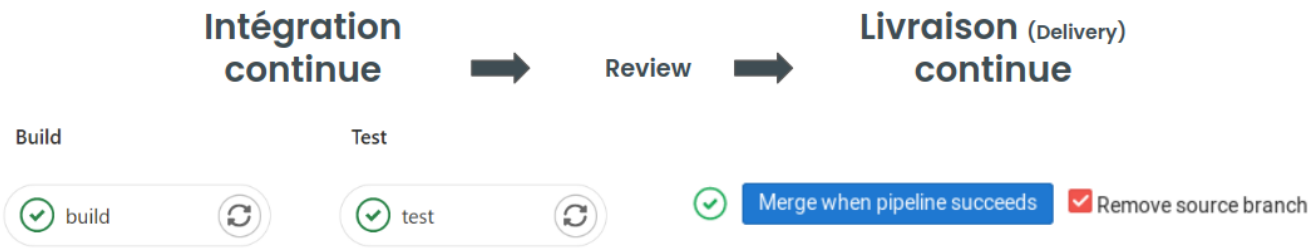
Deca Compiler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
fr.ensimag.deca.syntax		77 %		56 %	567	787	445	2 030	245	372	1	49
fr.ensimag.deca.tree		93 %		89 %	67	655	89	1 820	22	438	0	86
fr.ensimag.deca		81 %		74 %	20	77	34	218	2	37	0	6
fr.ensimag.ima.pseudocode		83 %		80 %	22	86	29	185	18	76	1	26
fr.ensimag.ima.pseudocode.instructions		70 %		n/a	20	62	36	111	20	62	16	54
fr.ensimag.deca.context		93 %		92 %	18	128	18	226	15	109	0	21
fr.ensimag.deca.codegen		90 %		91 %	3	34	7	111	2	28	0	3
fr.ensimag.deca.tools		90 %		87 %	2	17	4	41	1	13	0	3
Total	3 394 of 22 079	84 %	410 of 1 357	69 %	719	1 846	662	4 742	325	1 135	18	248

5. Méthodes de validation autres que le test

Nous utilisons l'intégration continue de gitlab avec les pipelines, pour assurer une vérification de la rétrocompatibilité du code automatiquement.

À chaque "push" sur une branche feature, la pipeline vérifie la bonne compilation du code java, et lance les tests grâce à la commande `mvn test` dans un environnement à part. Si la compilation ou les tests échouent nous en sommes informés par email. Si tout se passe bien et que la branche a été "review", elle est automatiquement "merge" sur notre branche de développement.



La pipeline est exécutée sur un serveur ubuntu personnel dédié au projet.

6. Tests unitaires

En utilisant la technique **boîte noire**, nous avons écrit des tests unitaires pour chaque fonctionalité. Plus précisément:

```
── decaObjet
   ├── cast
   ├── class
   ├── emptyClass
   ├── extends
   ├── extension
   ├── fields
   ├── general
   ├── include
   ├── instanceof
   ├── method
   ├── multi_line_comment
   └── protected
── general
── intermediaire
── intermediaire2
   ├── arithmetiques
   ├── boolean
   ├── cmp-int-float
   ├── convFloat
   ├── elseif
   ├── mots_reserves
   ├── printx
   ├── test-complementaire
   └── wrongValue
── provided
── renduInitial
```