1. Descriptif des tests

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 première étapes se vérifie avec celle de la dernière.

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
```

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()`.

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 (voir 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

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. Ainsi il y a 3 personnes qui vérifient le code ainsi que les tests ce qui permet minimiser les erreurs et les oublis.

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		60 %		44 %	568	714	831	1 924	274	366	15	49
fr.ensimag.deca.tree		89 %		86 %	75	441	93	1 022	45	326	5	66
fr.ensimag.deca.context		49 %		42 %	72	109	99	180	67	102	7	21
fr.ensimag.deca		80 %		73 %	23	75	40	222	5	36	0	6
fr.ensimag.ima.pseudocode		76 %		80 %	30	84	45	180	26	74	5	26
fr.ensimag.ima.pseudocode.instructions		55 %		n/a	28	62	51	111	28	62	23	54
fr.ensimag.deca.tools		90 %		87 %	2	17	4	41	1	13	0	3
fr.ensimag.deca.codegen		98 %		50 %	2	19	1	62	0	17	0	4
Total	4 970 of 16 570	70 %	419 of 986	57 %	800	1 521	1 164	3 742	446	996	55	229

5. Méthodes de validation autres que le test

Nous utilisons l'intégration continue de gitlab avec les pipeline pour assurer une vérification de la rétrocompatiblité du code automatiquement.

À chaque "push" sur une branche feature, la pipeline vérifie la bonne compilation du code java et lance les tests grace à la commande `mvn test` dans un environnement à part. Si la compilation ou les tests échouent nous en sommes informés par email.