PROJET GL

# DOCUMENT DE VALIDATION:

Grenoble INP-Ensimag

Amine ANGAR
Dy EL ALEM
Mouad TARBOUI
Oussama KADDAMI
Youness LEHDILI

# Validation:

## Parte A:

Pour la partie analyse lexicale et syntaxique - lextest.sh : effectuant test_lex sur les tests valides

- synttest.sh : effectuant test_synt sur les tests valides

- lextestinv.sh : effectuant test_lex sur les tests invalides

- synttestinv.sh : effectuant test_synt sur les tests invalides

Pour lancer un test unitaire sur la partie lexicale, il suffit de lancer à partir de la racine du projet la commande $test\_lex$ (si $test\_lex$ est ajouté au PATH) sinon faire:

src/test/script/launchers/test_lex

Pour lancer un test unitaire sur la partie syntaxique il suffit de lancer à partir de la racine du projet la commande test_synt (si test_synt est ajouté au PATH), sinon faire :

src/test/script/launchers/test_synt

Pour lancer les scripts de tests automatiques faire depuis la racine du projet:

./src/test/script/<nom_du_script>

Les tests écrits recouvrent l'ensemble des règles de la grammaire de deca. Les exemples des erreurs levées sont dans le document d'utilisateur.

## Partie B:

Pour la partie B on a implémenté plusieurs tests pour tester les différentes fonctionnalités du langage , ces tests sont organisés en fichiers par étape dans le fichier valid/ pour les tests valides et invalid/pour les tests invalides. On peut tester ces différents tests d'une manière unitaire par la commande test_context fichier ou lancer les script automatiques qui testent tous les fichiers ./src/test/script/context-test.sh pour les tests valides et ./src/test/script/context-test-inv.sh pour les tests invalides.Pour la validation de cette partie on a aussi procédé plusieurs fois au déboggage pour corriger les erreurs qui apparaissaient souvent au cours du projet, plusieurs techniques de déboggage ont été utilisés, parfois juste en observant plus le code d'autre fois à l'aide d'Eclipse et en faisant des prints pour tracer le chemin vers l'erreur.Ces techniques ont étés la plupart du temps et efficaces et ont permis de corriger la totalité des erreurs pour délivrer finalement une partie B complètement fonctionnelle.

## Partie C:

Tout au long du projet, nous avons consacré une part significative de notre temps à effectuer des tests afin d'intégrer une partie dans l'ensemble du compilateur. Nous avons adopté une stratégie de découpage des incréments "hello world", "Sans objet", "Langage essentiel" et "Langage complet" en sous incréments ce qui nous a permis d'avancer en parallèle et de manière synchroniser, ainsi que de pouvoir à chaque fois tester l'ensemble du compilateur sur le sous incréments. Concernant la partie C, nous avons défini pour le langage sans Objet des tests unitaires qui ont permis de tester cette partie indépendamment des autres parties du projet. Ensuite, après que chaque membre ait fini sa partie(concernant le sous-incrément) nous procédions aux tests sur l'ensemble du compilateur.

**Tests unitaires:**

Afin de tester la partie de génération du code de manière indépendante, nous avons implémenté différents tests unitaires dans le répertoire : "src/test/java/fr.ensimag.deca.tree" en plus du "testManualTestInialGenCode.java" fourni. Dans ces tests nous définissions les objets nécessaires à l'exécution des fonctions objet de tests et nous procédions aux tests indépendants. Ces tests nous permettaient d'une part de tester de manière indépendante la partie en question, et d'autre part de savoir comment l'arbre décoré doit être conçu afin que la partie génération de code s'exécute correctement. Ceci a été donc pertinent et nous a permis de trouver le point qui posait un problème quand le compilateur ne marchait pas.

**Tests d'intégration:**

Ensuite, après la fin de chaque incrément,nous implémentions des tests ".deca". Ces tests permettent donc de tester l'ensemble du compilateur. Nous avons essayé de tester au moins une seule fois chaque fonction implémentée afin de vérifier son bon fonctionnement comme prévu. Ensuite, nous essayons de trouver des tests qui peuvent poser des problèmes à notre compilateur et corriger les éventuels problèmes.

Après l'écriture de l'ensemble des tests et l'exécution de notre compilateur sur le code du test, nous faisons des vérifications du code assembleur générer pour vérifier que le code généré est en adéquation avec nos attentes.

Enfin, nous exécutons le code assembleur généré par le biais d'ima. En cas de bug subsistant, la procédure de débogage s'effectue sur plusieurs niveaux. On commence par les options de debug spécifiées

dans le poly ainsi qu'en utilisant des prints que ce soit dans le code source du compilateur ou dans le code assembleur généré.

Vous trouverez les tests associés à la partie génération de code dans le répertoire

src/test/deca/codegen/valid/.

Ce répertoire contient des sous-répertoires classifiant les différents tests. Vous pouvez lancer le script des tests depuis le répertoire Projet_GL à l'aide de la commande:

"./src/test/script/gencodetest.sh".

Ce script permet de créer les fichiers assembleurs associés aux fichiers sources ainsi que d'exécuter ces derniers avec une redirection de sortie vers des fichiers ".res". Après la génération de ces derniers, vous pouvez les supprimer en lançant la commande:

./src/test/script/gencodeclean.sh.

Nous avons aussi généré des tests pour vérifier le bon fonctionnement des exceptions dictées dans la partieSemantique. Vous trouverez ces tests dans le répertoire :

src/test/deca/codegen/invalid/.

Concernant les tests associés aux fonctions "read", ils sont présents dans le sous-répertoire src/test/deca/codegen/interactive, puisqueleurs sorties dépendent des paramètres d'entrée renseignés par l'utilisateur. Nous avons intégré la base des tests dans le fichier pom.xml pour les tests valides.

## Gestion des risques:

| Risque | Conséquence | Solution |
|---|---|---|
| Indisponibilité d'un membre de groupe pour une durée | Retard du projet sur les taches attribuées à ce membre | Voir si ces taches ne sont pas indispensables dans le court terme, dans ce cas attribuer une partie de ces taches à un autre membre. |
| Pister sur une fausse voie par rapport aux spécifications du poly | Produit un décalage quant à la réalisation des incréments du projet | Demander à notre encadrant sur riot. |
| Problème avec git (conflits de fichiers par exemple) | Temps perdu à résoudre les conflits | Prévoir une réunion zoom entre les membres concernés par le conflit (par exemple deux membres qui modifient le même fichier) |
| Différend entre deux membres de groupe travaillant sur une même partie | Risque de retard sur cette partie | Préférer la communication et le discours pour résoudre le problème le plus vite possible |
| Incapacité d'un membre à réaliser une tache | Grand retard sur la partie concernée | Demander de l'aide à un autre membre ou à l'encadrant. |
| Démotivation d'un membre de l'équipe | Retard sur la partie concernée | Demander de l'aide à un autre membre |

## Gestion des rendus:

En ce qui concerne les rendus, nous organisons une réunion Zoom un ou deux jours avant la date du rendu pour discuter là où on est ainsi que pour savoir si des membres ont besoin d'aide. La veille du rendu on parle sur le groupe si tout va bien et on organise une réunion si besoin. On s'assure que le code compile et que le rendu est bien respecté puis on effectue un commit sur git. Pour le rendu intermédiaire par exemple nous avons prévu un document dans le répertoire qui donne les détails de ce rendu.