Documentation de validation

Descriptif et organisation des tests:

Les tests sont contenus dans le package prénommé ***test*** du projet. Ils sont divisés en trois grandes catégories: *Syntax*, *Context* et *CodeGen*. La catégorie *Syntax* contient les processus de test en lien avec l'étape A, c'est-à-dire l'analyse syntaxique et lexicale.
Context, les processus en lien avec l'étape B, c'est-à-dire l'analyse contextuelle et la décoration de l'arbre.
Finalement, *CodeGen* contient les processus en lien avec l'étape C, la génération de code assembleur.CodeGen
Le répertoire ***test/deca*** contient les programmes de test *.deca* des trois étapes A, B et C. Associé à celà, le répertoire ***test/script*** contenant les script *.sh* permettant l'exécution des tests et leur automatisation.

Les tests de l'étape B:

Nous allons désormais nous pencher sur les fichiers *.deca* de test de l'étape B contenus dans ***test/deca/context*** puisqu'ils contiennent la substance des tests, les scripts n'étant que des outils pour les exécuter.
Dans cette partie, les tests sont essentiellement unitaires. Dans les tests contenues dans ***/invalid*** il s'agit de cibler une erreur possible par fichier en sachant qu'une même règle de grammaire peut possiblement invoquer plusieurs erreurs. Ainsi, plusieurs fichiers tests peuvent cibler une même règle de grammaire, d'où la dénomination *test_rule3_33arithmOp.deca* et *test_rule3_33Equals.deca* par exemple pour la règle (3.33).
Concernant les tests ***invalids*** la concision est de mise puisqu'il est préférable que les tests unitaires soient les plus courts possible. En effet, l'objectif est de cibler une seule erreur en particulier, un test trop long serait susceptible de générer plusieurs erreurs dont seule la première serait visible par le développeur. Les messages d'erreurs explicites permettent alors de noter si la bonne erreur a été levée.

Les tests ***valid*** quant à eux peuvent être légèrement plus conséquent puisque peu important la taille du test ce dernier est censé passer. Les tests sont alors échelonnés et non regroupés par règle de grammaire comme fait pour les tests invalides.
Concrètement, les tests ont été écrits et exécutés manuellement, en premier lieu, en alimentant le contenu des tests au fur et à mesure que les fichiers .*deca* sont créés. Par exemple, concernant la partie Object d'abord à été écrits et vérifié le test qui instancie simplement une class vide (*testObject-initClass.deca*) avant de considérer le test qui fait appel à "extends" (*testObject-extendsSimple.deca*).

De manière générale, les tests de l'étape B cherchent à vérifier que la syntaxe est correcte, le programme deca n'est pas exécuté. C'est uniquement à partir de l'étape C que le fonctionnement algorithmique est mis à l'épreuve. Ainsi, les test deca contenant des *println("ok test passed"),* par exemple, sont considérablement pertinents lors de l'étape C qui va stricto sensu exécuter le code assembleur généré et afficher les sorties.

<u>Automatisation des tests:</u>

L'automatisation des tests est essentiellement basé sur des scripts .sh. Par exemple, *run-all-added-tests.sh* est un script bash qui lance l'ensemble des script de test .sh des étapes B et C.

Chaque test Deca est associé à un script shell. Ce script réalise essentiellement les opérations suivantes :

- il vérifie qu'ima est bien installé là où il doit être, ou, à défaut, l'installe ;
- il tente de compiler le programme Deca, et compare le résultat (succès ou échec) au résultat de compilation attendu ;
- s'il s'agit d'un test de l'étape C, il l'exécute avec ima, et compare le résultat obtenu au résultat attendu ;
- s'il s'agit d'un test de l'étape C (à quelques exceptions près), il le compile en bytecode et compare le résultat obtenu au résultat attendu.

run-all-added-tests.sh lance tous ces tests et les affiche en deux parties :

- les tests qui devraient fonctionner, seules les erreurs s'affichent (à noter que les tests avec des flottants affichent souvent des "fausses" erreurs pour le bytecode, à cause de la comparaison des résultats comportant des flottants)
- des tests qui devraient renvoyer des erreurs à l'étape B

Couverture des tests:

Pour s'assurer de la couverture de nos tests nous n'avons pas utilisé Jacoco, nous avons toutefois cherché à maximiser la couverture de nos tests en cherchant à faire en sorte que nos tests couvrent tous les chemins de l'arbre représentant la grammaire du langage Deca.
De plus, pour chaque erreur contextuelle décrite dans le poly nous avons créé un test nous assurant que notre compilateur la relève.

Enfin, nous avons fait des tests de cas subtiles tel que celui où une déclaration de variables dans une méthode et un paramètre de cette même méthode ayant le même symbole pour nous assurer que notre compilateur ne fait pas la confusion entre deux symboles identiques mais de nature différente.