Documentation de conception

Primo, le projet se scinde en deux parties nodales: le répertoire ***main*** contenant les fichiers relatifs au fonctionnement effectif du compilateur et le répertoire ***test*** contenant les fichiers permettant de vérifier la cohérence et la complétude du travail effectué. En plus de ces deux répertoire est présent un fichier *pom.xml* qui contient les dépendances externes ainsi que des paramètres de configurations.

- Concentrons nous maintenant sur le répertoire ***main***.

Le répertoire ***antlr4*** contient les fichiers *DecaLexer.g4* et *DecaParser.g4,* complétés par le programmeur, qui sont des fichiers de configurations donnés à *antlr* afin de faire analyse syntaxique et lexical du code source *deca* donné en entrée.
Les répertoires **assembly** et **bin** contient le script sh *decac* qui compile un programme *deca* donné en entrée.

Nous allons maintenant aborder la partie centrale du compilateur, le répertoire ***java*** regroupe les fichiers .java:
fr.ensimag.deca comporte les fichiers principaux java du compilateur. Est notamment présent *DecaMain.java,* contenant la fonction *main* de java. Ce dernier fait ensuite appel à *DecaCompiler* pour lancer le processus de compilation du programme deca via *doCompile.* C'est dans ce dernier fichier que sont lancées les étapes A, B et C.
De plus, sont présents dans **fr.ensimag.deca.tree** les fichiers *.java* nécessaires à la construction de l'arbre abstrait du programme deca avec des classes abstraites pour les non terminaux et non abstraites pour les terminaux. Ces différentes classes possèdent les méthodes de vérification, de compilation et de décompilation du programme deca à compiler en assembleur. Fut rajoutées par la suite des méthodes de décompilation vers un programme Java pour la génération de ByteCode Java à partir du programme source *deca*.

fr.ensimag.ima.pseudocode comporte les fichiers *.java* relatif à IMA et nécessaire à la génération d'un code capable de fonctionner sur une machine dotée du système IMA.

- Passons désormais au répertoire ***test:***

Dans le répertoire **deca** sont disposés tous les fichiers de test *.deca* classé selon la partie testé: Analyse syntaxique (syntax), Analyse Contextuelle (Context) ou génération de code assembleur (CodeGen).

A chacun des fichiers .deca est associé un fichier script .sh qui permet d'exécuter ces tests sur les différentes étapes de la compilation.
La construction de ces scripts est toujours la même:
Voici un exemple pour un script de test de l'étape B.

```
#! /bin/bash

##################
#Init (don't modify)
SCRIPT_PATH=./src/test/script/
. ${SCRIPT_PATH}/init-script.sh
##################




PROGRAM_PATH="$TESTS_PATH/syntax/invalid/provided"
PROGRAM_NAME="chaine_incomplete"
EXPECTED_OUTPUT="FAIL"




##################
#End (don't modify)
. ${SCRIPT_PATH}/end-script.sh
##################
```

Un script init-script.sh initialise le test.
Ensuite, le test .deca en question est sujet à la compilation.
Enfin, un script end-script.sh ferme le test.

Il existe par ailleurs des tests systèmes, des scripts plus dense qui lance la compilation sur plusieurs fichiers *.deca* d'un coup, tel que *common-tests.sh* par exemple.

- Implémentation de **EnvironmentExp**

Champs :

En plus d'un dictionnaire (Map) associant à un Symbol sa définition, on dispose d'un champ/attribut "parentEnvironment", qui permet par référence successive à un parent, de créer une liste chainée d'instances de classes "EnvironmentExp".

Méthodes :

Les méthodes **get** et **contains** sont récursives, elles cherchent d'abord dans l'environment exp sur lequel on utilise ces méthodes puis elles cherchent dans les environements au dessus dans l'hiérarchie.

La méthode **UnionDisjointe** implémente l'opération portant le même nom décrite dans le poly. Dans un premier temps on vérifie si les deux environnements ont une clé en commun, si c'est le cas on renvoie une erreur contextuelle. Sinon on ajoute le contenu de l'environment en paramètre dans celui sur lequel on a appelé la méthode.

- Classes crées pour implémenter la partie objet

Comme pour les classes de la partie sans-objet qui sont fournies, on modélise chaque non terminal par une classe abstraite, et les terminaux par des classes qui étendent une classe abstraite.

Les classes sont toutes construites de façon similaire, en voici la méthode générale :

-En gardant à l'esprit que chaque classe représente un nœud de l'arbre, les enfants de ce nœud vont correspondre à des champs de la classe.

-Dans les constructeurs on vérifie si les arguments sont non nuls sauf dans les cas où ils sont optionnels comme par exemple l'extension class dans une déclaration de classe qui n'est pas forcément précisée (mise à Object dans ce cas lors de la décoration à l'étape B)

-En plus des méthodes suggérées par les enseignants on implémente une méthode “decompileToJava” utile pour l’extension BYTE. On implémente également des méthodes (notamment “getDVal”) utiles pour le codeGen de l’étape C.