ENSIMAG

Grenoble INP

Conception

Projet GL

Groupe gl 49

Valentin OLLIER

Victor SAUNIER

Sofien CHENGUEL

Wael FEGUIRI

Hamza HASSOUN

Janvier 2020

Table des matières

1 Introduction

Ce document détaille les choix de conception et les démarches abordées afin de fournir un compilateur programmée essentiellement en Java et qui a pour dessein de compiler tout programme Deca avec ou sans objet.
Cependant notre compilateur présente des limitations qui n'ont pas pu être résolues. Il est possible de compiler et produire un code assembleur de tout programme sans objet en Deca. Par contre, pour la partie objet, l'implémentation demeure limitée. Il est possible pour l'utilisateur de créer des classes avec leurs attributs et méthodes cependant il n'est pas possible de les modifier suite à des bugs dans l'implémentation de notre partie objet du compilateur.

2 Architecture du compilateur

2.1 Analyse lexicale et syntaxique

Le compilateur a pour but de traduire tout programme Deca en un programme pouvant être éxécutée par la machine.
La première étape consiste donc à effectuer une analyse lexicale et syntaxique du programme fourni. Il est impératif de vérifier que le fichier fourni est bien un fichier Deca en faisant l'analyse lexicale de ce dernier.
Cette première étape permet de générer des **Tokens** qui permettront par la suite de construire un arbre via l'étape d'analyse syntaxique.

2.1.1 DecaLexer.g4

Dans le répertoire [**src/main/antlr4/fr/ensimag/deca/syntax**] un fichier **DecaLexer.g4** est fourni et est à compléter pour réaliser l'étape d'analyse lexicale.
Dans ce fichier nous avons codé la liste des mots et caractères reconnaissables par le language Deca ; ainsi que les caractères spéciaux réservés.

2.1.2 DecaParser.g4

Dans le répertoire [**src/main/antlr4/fr/ensimag/deca/syntax**] un fichier **DecaParser.g4** est fourni et est à compléter pour réaliser l'étape d'analyse syntaxique et retourné un arbre.
Dans ce fichier nous avons utilisé les fonction [**setLocation**] et [**tree.add**] afin de retourner un arbre et d'ajouter sa Location à partir des **Tokens** retournés du Lexer.
La prochaine étape dans le processus de conception de notre compilateur est donc l'analyse contextuelle qui va permettre la décoration de l'arbre fourni par les étapes d'analyse lexicale et syntaxique.

2.2 Analyse contextuelle

2.2.1 Tree

Le package Tree était un package très largement écrit, avec bon nombre de fichiers noeuds déjà présents. Pour la partie du compilateur sans objet, nous avons ajouté des fonctions de vérification dans chaque fichier du package. Ces fonctions ont pour rôle de vérifier le respect de la grammaire au niveau du noeud de l'arbre dans lequel on se trouve, et parcourir ses fils à la recherche d'éventuelles erreurs contextuelles.

2.2.2 DecacCompiler

Le décacompiler déjà fourni nous permet de vérifier le respect des règles de grammaire et de décoration de l'arbre pour le compilateur sans objet et avec objet. Pour se faire, il réalise l'étape de parsing, pour laquelle il vérifie que les localisations des noeuds sont bien implémentées et à l'aide de la fonction decompile, génère un programme correct en deca à partir du programme deca de base. Ensuite, il vérifie la bonne décoration de l'arbre après la vérifictaion contextuelle. Ainsi, en plus de réaliser la compilation d'un programme deca, le compileur nous permet aussi de vérifier que l'arbre est bien réalisé en premier lieu.

De plus, nous avons ajouté des options au compilateur le rendant modulable, pour travailler sur la vérification comme le parsing.

2.2.3 EnvironmentType

Enfin, nous avons décidé de créer une classe EnvironmentType, qui nous a permis de référencer les types présents dans le programme. Un objet de cette classe a été ajouté dans le compilateur, et permet d'ajouter au cours de la vérification les classes à cet environnement, en plus des types prédéfinis.

2.3 Génération de code

2.3.1 CodeGenInst et CodeGenPrint

Dans cette partie nous avons codés les instructions nécessaires pour obtenir un code assembleur généré dans un fichier de suffixe **".ass"** à partie d'un fichier source Deca.
CodeGenInst : Toutes les classes dans le répertoire [**src/main/java/fr/ensimag/-deca/tree**] possèdent la méthode codeGenInst afin de générer le code propre aux instructions.
CodeGenPrint : Toutes les classes dans le répertoire [**src/main/java/fr/ensimag/-deca/tree**] possèdent la méthode codeGenPrint pour la génération de code propre à un **print.**

2.3.2 Register Manager

Afin de pouvoir générer du code assembleur l'implémentation du Regiter Manager est primordiale pour gérer les registres que l'on utilise.
Nous faisons donc appel aux méthodes implémentées dans cette classe dans les fichiers du package Tree .
Ces méthodes seront détaillées dans la section **Implémentation du compilateur**

3 Implémentation du compilateur

3.1 Etape B : Vérification contextuelle

3.1.1 Implémentation de EnvironmentType

Pour l'environnement des types, le choix de conception de plus crucial que nous avions à faire était la structure de données à utiliser pour pouvoir récupérer correctement les bons types, et s'assurer que ces derniers étaient définis.

Pour ce faire, nous avons décider de créer un dictionnaire, référençant la définition d'un type, suivant le symbole qui lui est associé. Ainsi, nous avons utilisé une HashMap, très pratique pour réaliser un dictionnaire. Cet environnement nous permet donc de savoir directement si un objet est défini ou non. Par contre, à chaque appel de cette

fonction, nous devons recréer un symbole à partir du symbole renvoyé par le programme, qui visiblement diffèrent.

Ainsi, la méthode suivante est la plus importante pour l'environnement des types :
-put :
Cette méthode prend en argument **SymbolTable.Symbol key, Definition def** et nous permet de rajouter un symbole avec la définition associée, ou renvoyer une erreur de double définition si celle-ci est déjà présente.

3.1.2 Implémentation de EnvironmentExp

Nous avons créé notre EnvironmentExp des variables de la même manière, avec une HashMap associée à une variable. Celle-ci nous permet de récupérer au fur et à mesure les variables définies dans l'environnement de travail. Ici, la méthode importante est la méthode suivante :
-create :
Cette méthode prend en argument **SymbolTable.Symbol key, ExpDefinition def** et nous permet de rajouter un symbole avec la définition associée, ou renvoyer une erreur de double définition si celle-ci est déjà présente.

3.1.3 Décoration de l'arbre abstrait

La décoration de l'arbre abstrait se fait assez facilement, en ajoutant des fonctions permettant d'ajouter une définition à chaque noeud de l'arbre. Par la suite, et notamment pour les classes d'objets liés au compilateur complet, nous avons ajouté des fonctions **-iterChildren** et **-prettyPrintChildren** , qui permettent d'afficher l'arbre correctement décoré.

3.2 Etape C : Génération de code

3.2.1 Register Manager

Comme mentionnée précédemment cette classe a pour dessein de gérer les registre utilisés.
Voici donc les principales méthodes de la classe RegisterManager :

-manageRegisters :
Cette méthode prend en argument **DecacCompiler compiler** et renvoie un registre libre en vérifiant si on a utilisé le nombre maximum de registres permis (15) ou pas.
Dans le cas où aucun registre n'est libre l'instruction **compiler.addInstruction(new POP(Register.getR(lastRegused)))** permet de libérer le dernier registre utilisé et de le manipuler à nouveau.

-getLastRegused :
Cette méthode nous permet d'obtenir le numéro (valeur entière) du dernier registre utilisé.

-affectReg :
Cette méthode prend en argument **DecacCompiler compiler** et nous renvoie un entier pour pouvoir effectuer l'affectation d'un registre.

-setLastRegused :
Cette méthode prend en argument un entier et permet donc de choisir manuellement le dernier registre que l'on souhaite utiliser et qui sera la valeur de l'entier en argument de notre méthode.

-isFree :
Cette méthode prend en argument un entier qui correpond à un numéro de registre et retourne un booléen permettant de savoir si ce registre est libre ou pas.

-freeReg (int i, DecacCompiler compiler) :
Cette méthode permet de libérer le registre i.

3.2.2 Codage des instructions

Partie sans Objet :
L'implémentation des instructions de génération de code pour le language Deca sans Objet a été principalement réalisée en codant les méthodes codeGenPrint et codeGenInst dans chacune des classes correspondantes.
Mais dans le but de factoriser notre code, des méthodes ont été crées dans les classes Abstract et auxquelles nous avons fait appel dans les classes fille ont permis de réaliser la génération de code assembleur.

Quelques exemples :

–AbstractOpArith : Pour les opérations arithmétiques nous avons utilisé la méthode **codeExpInt**. Avec la méthode getOperatorName() nous déterminons le type d'opération (addition , soustraction, multiplication, division ou modulo) et nous appliquons les instructions nécessaires.
Ensuite dans les classes héritières **Plus, Minus, Modulo, Multiply et Divide** nous faisons appel à **codeExpInt** dans **codeGenInst**.

–AbstractOpBool : Pour les opérations booléennes nous avons utilisé la méthode **codeOpBool**. Avec la méthode getOperatorName() nous déterminons le type d'opération (AND ou OR) et nous appliquons les instructions nécessaires.
Ensuite dans les classes héritières **And et Or** nous faisons appel à **codeOpBool** dans **codeGenInst**.

–AbstractOpCmp : Pour les opérations de comparaison nous avons utilisé la méthode **codeOpCmp**. Avec la méthode getOperatorName() nous déterminons le type d'opération (Greater, GreaterOrEqual, Lower, LowerOrEqual, Equals, NotEquals) et nous appliquons les instructions nécessaires.
Ensuite dans les classes héritières **Greater, GreaterOrEqual, Lower, LowerOrEqual, Equals, NotEquals** nous faisons appel à **codeOpCmp** dans **codeGenInst**.

Partie avec Objet :
Pour l'implémentation des instructions de génération de code concernants la partie objet, de nouvelles méthodes ont été créées dans les classes correspondantes.

public void codeGenDeclClass(DecacCompiler compiler)
Cette méthode implémentée dans la classe DeclClass a permis de coder les instructions nécessaires lors de la déclaration d'une nouvelle classe dans un programme Deca.

public void codeGenDeclFields(DecacCompiler compiler)
Cette méthode implémentée dans la classe DeclClass a permis de coder les instructions nécessaires lors de la déclaration des champs d'une classe. Elle gère donc l'initialisation de la déclaration des attributs et méthodes.

public void codeGenDeclField(DecacCompiler compiler)
Cette méthode implémentée dans la classe DeclField permet donc de générer le code d'initialisation d'un champ déclaré dans une classe.

public void codeGenDeclClassTable(DecacCompiler compiler)
Cette méthode implémentée dans la classe AbstractDeclClass gère la génération de code de la table des classes et super-classes.

public void codeGenMethod(DecacCompiler compiler)
Cette méthode implémentée dans la classe DeclMethod génère le code asembleur d'une méthode dans une classe.

public void codeGenVar(DecacCompiler compiler)
Cette méthode implémentée dans la classe DeclVar génère le code assembleur d'une variable du programme en cours.

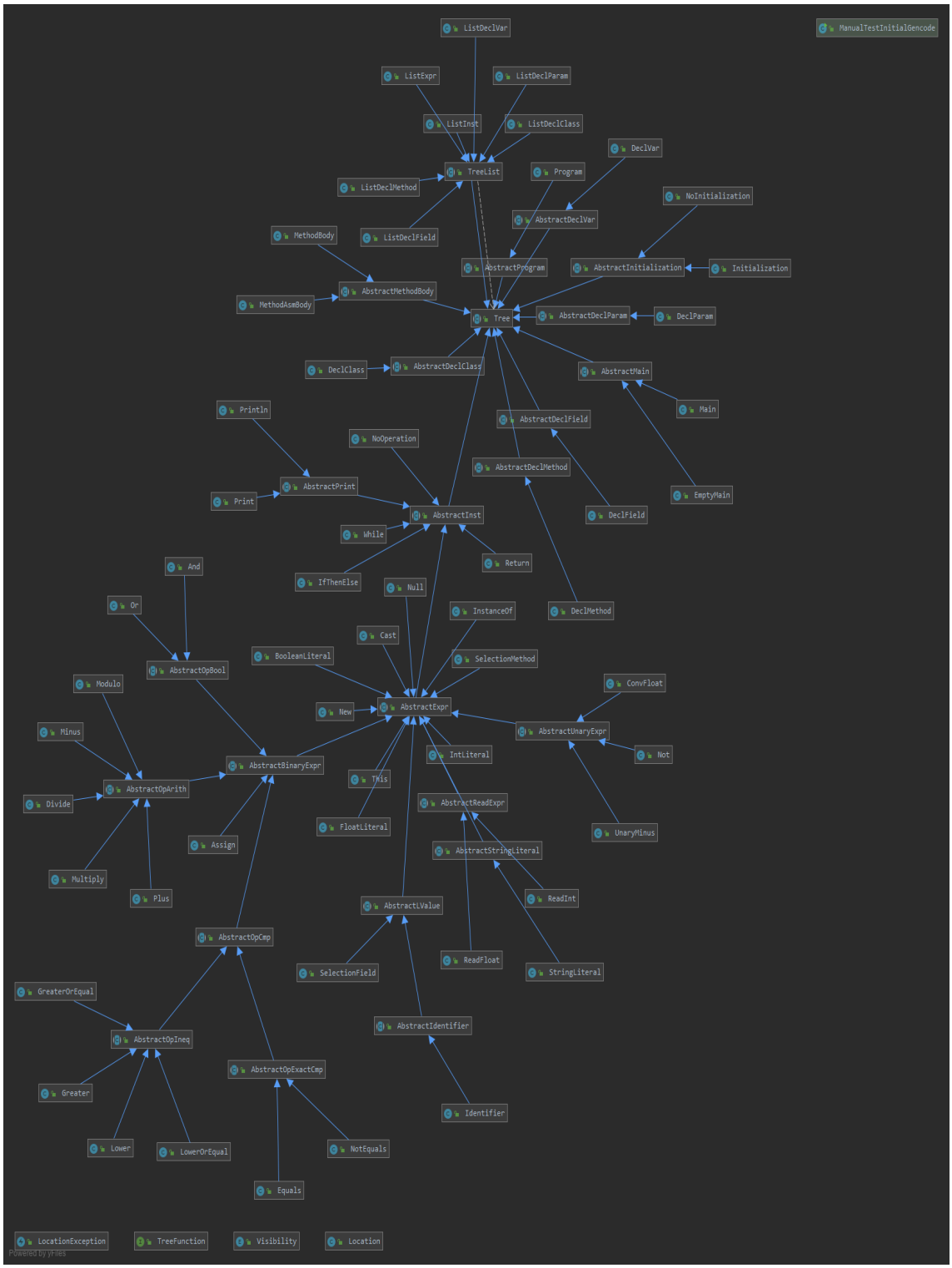
public void codeGenDeclBody(DecacCompiler compiler)
Cette méthode implémentée dans la classe MethodBody génère le code assembleur d'un corps de méthode.

3.2.3 Classe Objet

Cette clase du package **codegen** situé dans [**src/main/java/fr/ensimag/deca**] a été implémentée afin de pouvoir la méthode equals de object.
La méthode **public static void codeGenEq(DecacCompiler compiler)** permet donc d'aboutir à cette fin.

4 Annexe

4.1 Java Class Diagramm of Tree Package



4.2 Exemples d'insctructions en deca et des fichiers assembleurs fournis par le compilateur

int x ; println(0.5*(x*x)) ;

```
; start main program
; Main program
; Beginning of main instructions :
LOAD 0, R2
STORE R2, 1(GB)
WNL
HALT
; end main program
```

class A {
int n ;
int m ;
}
class B {
A a ;
int p ;
}

```
; start main program
; Début du programme
STORE R0, 1(GB)
STORE R0, 2(GB)
TSTO 3
BOV pile_pleine
ADDSP 3
; Main program
HALT
; initialisation des champs de la classe A
init.A :
LOAD 0, R2
LOAD -2(SP), R1
STORER 2, 1(R1)
LOAD 0, R3
LOAD -2(SP), R1
STORER 3, 2(R1)
RTS
; initialisation des champs de la classe B
init.B :
LOAD null, R4
LOAD -2(SP), R1
STORER 4, 1(R1)
LOAD 0, R5
LOAD -2(SP), R1
STORER 5, 2(R1)
RTS
; Methode Equals
```

$code.Object.equals$:
$PUSHR2$
$PUSHR3$
$LOAD-2(LB), R2$
$LOAD-3(LB), R3$
$CMPR2, R3$
$BEQtrue$
$ADD0, R0$
$BRAfin$
$true$:
$ADD0, R0$
fin :
$BRAfin.Object.equals$
$WSTR"Erreur : sortiedelaméthodeequalssansreturn"$
WNL
$ERROR$
$fin.Object.equals$:
$POPR3$
$POPR2$
RTS
$pile_pleine$:
$WSTR"Erreur : lapileestpleine"$
WNL
$ERROR$
$dereferencement_null$:
$WSTR"Erreur : dereferencementdenull"$
WNL
$ERROR$
tas_plein :
$WSTR"Erreur : allocationimpossible, tasplein"$
WNL
$ERROR$
$;endmainprogram$