GRENOBLE INP UGA Ensimag

Document de conception

GL 22 - Groupe 5

Reda Kabbaj

Youssef Hannat

Youssef Elaasri

Yakoub Dehbi

Jihad Hammoud

25 janvier 2024

1 Introduction :

Ce document de conception est destiné à un développeur qui souhaiterait maintenir et/ou faire évoluer le compilateur. C'est pourquoi, nous utilisons un langage technique afin de mieux appréhender la conception de l'implémentation de notre compilateur. Nous vous invitons à bien lire attentivement les différentes sections du document où nous détaillerons toutes les étapes, allant de l'analyse lexicale du fichier source à l'analyse syntaxique qui produit un arbre abstrait, dont on devrait vérifier le contexte, enrichir et décorer, et on finit par la génération du code assembleur afin qu'il soit exécuté par la machine ima.

2 Analyse lexicale :

2.1 Architecture :

Le fichier **src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4** est écrit en ANTLR, et est destiné à générer l'analyseur lexical à partir d'une suite de lexèmes définis dans ce fichier. Ainsi, jusqu'à cette étape, notre compilateur prend en entrée un code source écrit en Deca, et le transforme en une suite de lexèmes, tout en identifiant des erreurs lexicales si elles existent, en levant une erreur provoquée par ANTLR. Cette suite de lexèmes est ensuite donnée en entrée de l'analyseur syntaxique qui donnera en sortie un arbre abstrait, et qui sera détaillé en section 3.

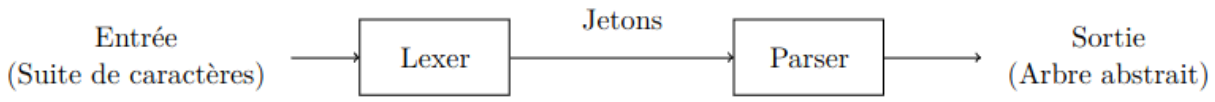

FIGURE 1 – Shéma du Lexer et du Parser

3 Analyse syntaxique :

3.1 Architecture :

Le fichier **src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4** est écrit en ANTLR, et est compilé en un fichier Java décrivant l'analyseur syntaxique de notre compilateur. Ce fichier permet de construire la grammaire hors-contexte en implémentant des règles définissant la syntaxe d'un fichier Deca syntaxiquement correct. Ainsi, on use de l'ensemble des lexèmes définis produits par l'analyseur lexical pour construire à partir de ces lexèmes et de ces règles un arbre abstrait primitif, qui sera décoré par la suite dans l'étape de verifications contextuelles (Voir Figure 1).

4 Verifications contextuelles :

4.1 Répertoires concernés :

- **src/main/java/fr/ensimag/deca/tree/*** : Ce répertoire contient plusieurs classes, chaque classe représente un non-terminal ou un terminal de la grammaire attribuée, ce qui permet d'implémenter le parcours de l'arbre abstrait.
- **src/main/java/fr/ensimag/deca/context/*** : Ce répertoire contient les classes Java qui nous seront utiles pour les vérifications contextuelles, à priori toutes les définitions possibles (FieldDefinition, VariableDefinition, MethodDefinition...) et les types possibles (IntType, FloatType, ClassType...).

  De plus, on définit aussi dans ce répertoire un environnement de type EnvironmentType.java et un environnement d'expression EnvironmentExp.java.

4.2 Structure et quelques précisions :

L'étape des vérifications contextuelles est cruciale pour notre compilateur. Elle joue un rôle important en assurant la sémantique de Deca et en donnant sens au code fourni.

Vu la complexité de cette étape, elle est effectuée en 3 passes :

— La première passe permet de vérifier le nom et la hiérarchie de classes.

— La deuxième passe permet de vérifier les champs et la signature des méthodes.

— La troisième passe permet de vérifier le corps des méthodes et les déclarations et les instructions du programme principal.

Afin de stocker ces définitions pour une utilisation ultérieure, il est nécessaire d'envisager deux structures de données qui définissent l'environnement de types et l'environnement d'expressions.

— EnvironmentType : Il s'agit d'une classe ayant comme attribut un dictionnaire qui associe à chaque symbole défini dans la classe src/main/java/-fr/ensimag/deca/tools/SymbolTable.java, un TypeDefinition qui est aussi une classe qui définit les définitions des types prédéfinis comme Int, Float, String, et les autres types comme Class.

— EnvironmentExp : Il s'agit d'une classe ayant comme attribut un dictionnaire qui associe à chaque symbole défini dans la classe src/main/java/-fr/ensimag/deca/tools/SymbolTable.java, un ExpDefinition qui est aussi une classe qui définit les définitions des variables, paramètres, champs et méthodes.

La décoration et l'enrichissement de l'arbre abstrait se fait au fur et à mesure des trois passes effectuées. En effet, il s'agit de définir les définitions et les types

de nos structures de données préalablement stockées dans les environnements EnvironmentType et EnvironmentExp, grâce aux méthodes setDefinition et setType.

Au cours de ces 3 passes, une erreur contextuelle est levée lorsque l'arbre abstrait ne respecte pas les règles de la grammaire attribuée, et dans ce cas le programme s'arrête en laissant un message d'erreur précisant la cause et l'endroit de l'erreur.

4.3 Liste de classes implémentées et leurs dépendances :

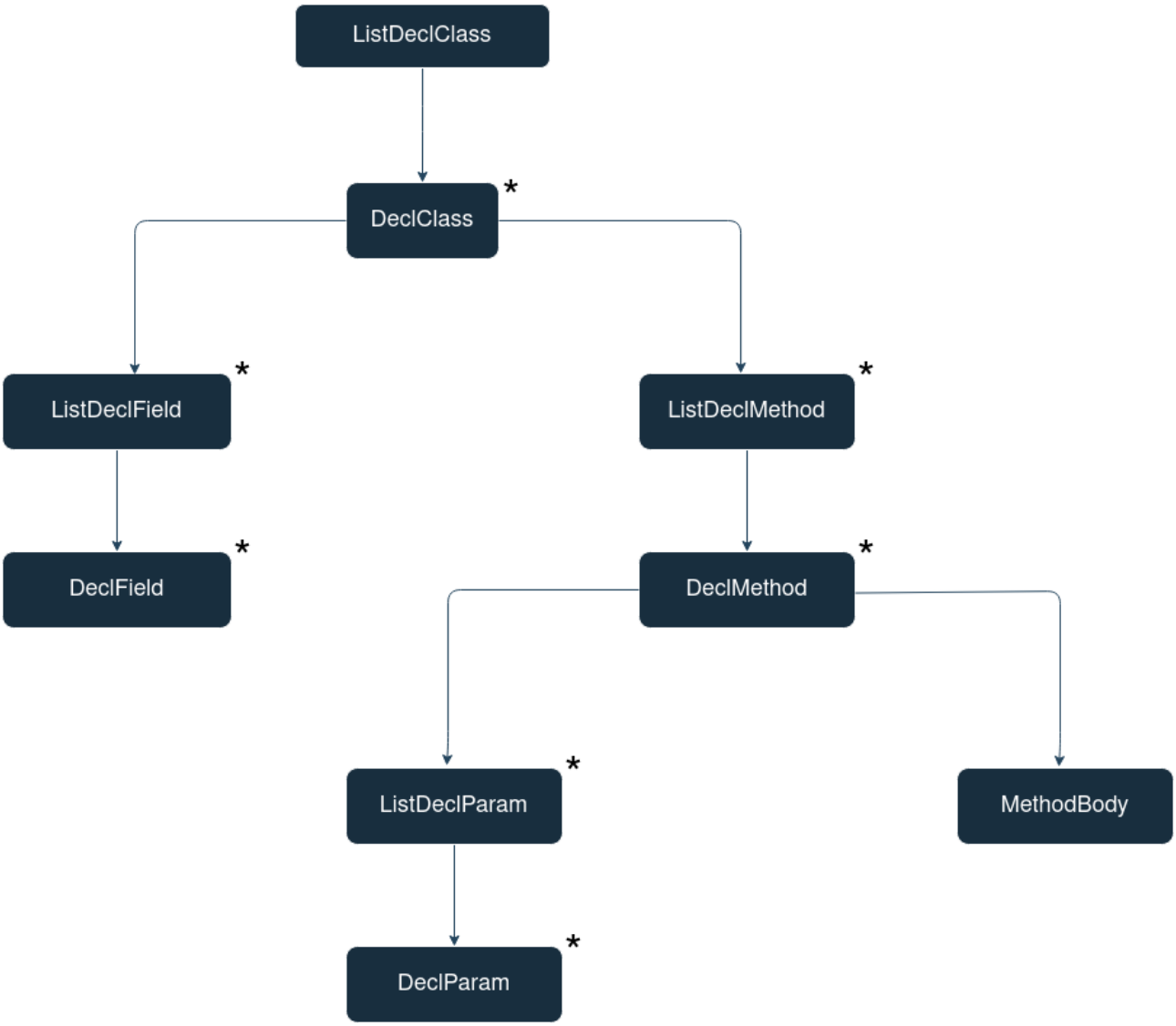

FIGURE 2 – Liste des classes et leurs dépendances

La figure 2 ci-dessus montre les dépendances des classes qu'on a ajoutées pour la partie **avec objet** et qu'on détaillera ci-dessous :

— **ListDeclClass :** Cette classe permet d'implémenter une liste éventuellement vide de classes définies par la classe **DeclCLass**.

— **DeclClass :** Cette classe permet d'implémenter la déclaration d'une classe, avec comme attributs la liste des champs ListDeclField et la liste des méthodes ListDeclMethod.

— **ListDeclMethod :** Cette classe permet d'implémenter une liste éventuellement vide de méthodes définies par la classe **DeclMethod**.

— **DeclMethod :** Cette classe permet d'implémenter la déclaration d'une méthode, avec comme attributs la liste des paramètres ListDeclParam et le corps de la méthode MethodBody.

— **ListDeclParam :** Cette classe permet d'implémenter une liste éventuellement vide de paramètres définies par la classe **DeclParam**.

— **DeclParam :** Cette classe permet d'implémenter la déclaration d'un paramètre.

— **MethodBody :** Cette classe permet d'implémenter le corps d'une méthode, avec comme attributs la liste de déclaration de variables ListDeclVar et la liste d'instructions ListInst.

L'intérêt de ces classes réside dans un premier temps dans l'implémentation de nos structures de données qui ont élé laissés incomplètes dans la partie **avec objet**, et dans un deuxième temps, dans la réalisation de l'étape B qui se compose des vérifications contextuelles de notre arbre abtrait, de sa décoration et de son enrichissement.

Pour mieux appréhender notre conception de cette partie, on vous propose dans la Figure 3 un schéma qui met en exergue les sous-parties de cette étape, leurs dépendances, l'utilisation de ces classes ajoutées et leurs intérêts :

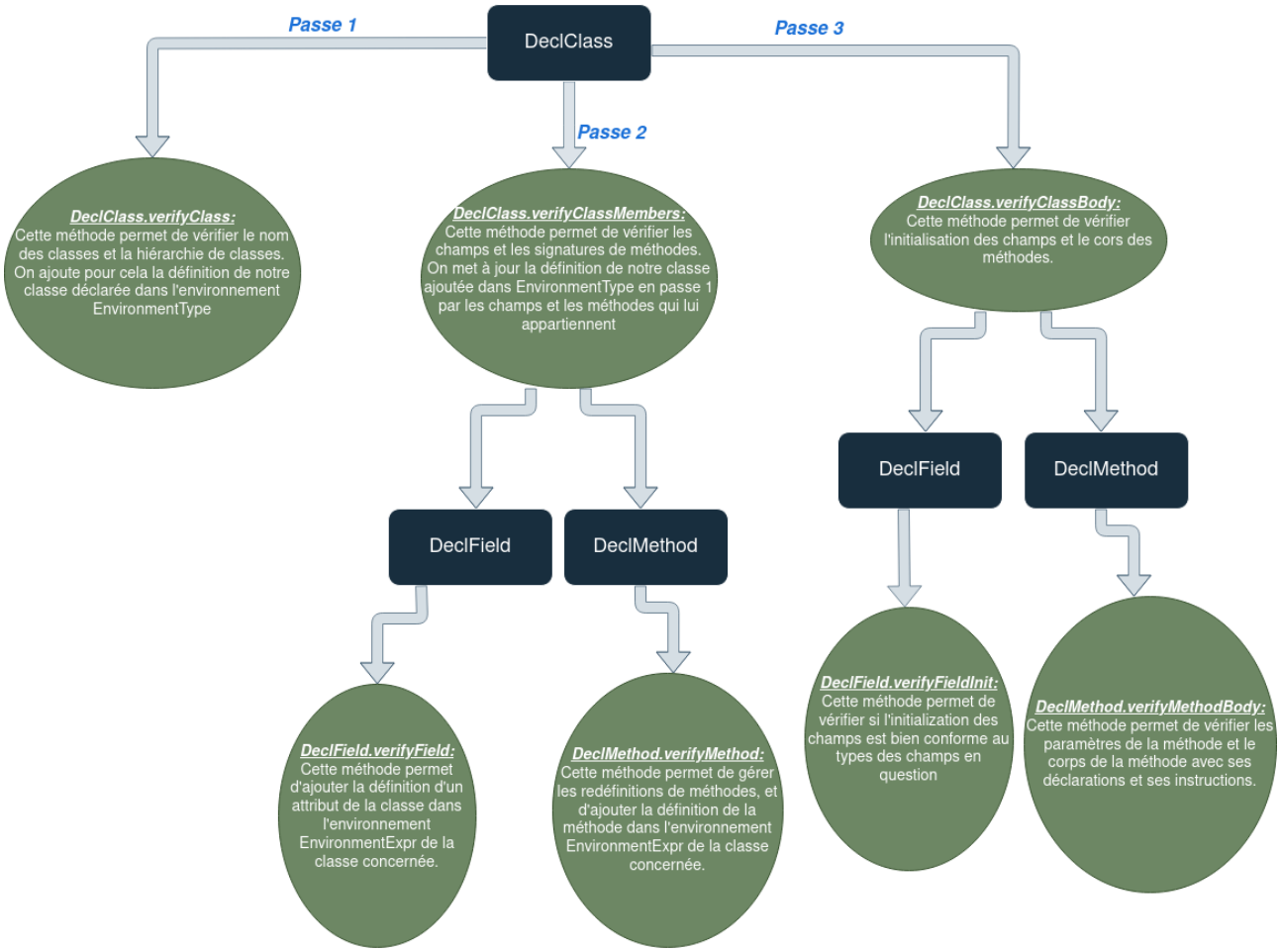

FIGURE 3 – Les étapes des vérifications contextuelles

Afin de rendre nos classes utilisables par l'utilisateur, il est nécessaire d'ajouter d'autres classes qui permettent d'implémenter d'autres fonctionnalités nécessaires à la manipulation des classes, comme la sélection de champs, l'appel de méthodes à titre d'exemple.

Ci-dessous la liste des classes Java qu'on a conçu ajouter avec leurs intérêts :

— **This :** Cette classe hérite de AbstractExpr et permet de faire référence à l'objet de la classe dans laquelle on se trouve. On vérifie ainsi que this n'est pas utilisé en dehors d'une classe.

— **New :** Cette classe hérite de AbstractExpr et permet de déclarer un nouvel objet d'une classe donnée. On vérifie que le type statique et le type dynamique de l'objet déclaré sont bien conformes à certaines spécifications.

— **Selection :** Cette classe hérite de AbstractLValue et permet de sélecionner un champ d'un objet associé à une classe donnée. On vérifie que l'objet sélectionné est bien de type Class et qu'on sélectionne effectivement un champ de visibilité PUBLIC et non pas PROTECTED.

— **MethodCall :** Cette classe hérite de AbstractExpr et permet l'appel d'une méthode, en vérifiant qu'on appelle bien une méthode existante et avec le nombre adéquat de paramètres.

— **Return :** Cette classe hérite de AbstractInst et permet de retourner une valeur au sein d'une méthode. On vérifie qu'on retoune bien une valeur qui n'est pas de type Void et que la valeur retournée est bien du type attendu.

— **Instanceof :** Cette classe hérite de AbstractExpr et peremt de vérifier si un objet donné est bien une instance d'une classe donnée.

— **Cast :** Cette classe hérite de AbstractExpr et permet de convertir un objet d'un type donné vers un autre type donné. On vérifie que cette conversion

est bel et bien possible.

— **Null :** Cette classe hérite de AbstractExpr et permet d'assigner la valeur null à un objet de type Class. Il s'agit d'un literal de type NullType (comme le literal IntLiteral de type IntType).

5 Génération du code assembleur :

5.1 Introudction :

La génération de code dans notre compilateur est une étape cruciale qui transforme l'arbre syntaxique abstrait (AST) décoré, obtenu à la suite de l'analyse sémantique, en code assembleur destiné à la machine IMA. Cette documentation vise à fournir une compréhension approfondie du processus de génération de code, des structures de données utilisées et des mécanismes de gestion d'erreur.

5.2 Architecture et spécifications :

Processus de parcours de l'arbre :

Lors de la génération de code, nous parcourons l'arbre en commençant par la classe "Program", descendant ensuite dans les méthodes, les blocs et les expressions. L'expression est lue de droite à gauche pour faciliter l'évaluation correcte des opérations.

Exemple :

Pour l'expression *1 + 2 + 3*, l'arbre est construit comme *plus(plus(1, 2), 3)*, permettant une évaluation correcte de gauche à droite.

Gestion des registres et de la pile (Classe "Stack") :

La classe "Stack" joue un rôle essentiel dans la gestion des registres et de la pile pendant la génération de code. Voici les principaux attributs et méthodes de la classe :

— **addrCounter :** addrCounter : Compteur de l'emplacement mémoire pour les déclarations de variables, de classes et de méthodes.

— **numberOfRegisters :** Nombre maximal de registres (par défaut 16, modifiable par l'option -r X).

— **maxTSTO :** Variable pour suivre l'utilisation maximale de la pile.

— **counterTSTO :** Compteur d'utilisation de la pile, incrémenté lors des déclarations, décrémenté lors des opérations "push" et "pop".

— **currentRegister :** Registre courant pouvant être utilisé, avec une vérification pour éviter les dépassements.

— **maxRegister :** Utilisé dans le codeGen des méthodes pour calculer le TSTO et éviter des passes inutiles.

Gestion des erreurs (Classe "ErrorHandler") :

La classe "ErrorHandler" est dédiée à la gestion des erreurs d'exécution. Toutes les erreurs sont enregistrées à la fin du code assembleur généré, optimisant ainsi la taille du fichier. Les erreurs sont générées en fonction des vérifications effectuées lors de la génération de code.

Personnalisation et optimisations :

Un programmeur a la flexibilité de personnaliser l'arbre entre les étapes B et C. Pour modifier la génération de code, il peut ajuster la méthode codeGenInst dans la classe appropriée. Les classes "Stack" et "ErrorHandler" sont mises à disposition pour faciliter la gestion des registres, de la pile et des erreurs.

6 Algorithmes utilisés :

6.1 Parcours de l'Arbre :

Le parcours de l'arbre est réalisé en profondeur d'abord, assurant une navigation logique. Chaque nœud déclenche la méthode codeGenInst correspondante, facilitant la génération de code.

6.2 Génération de Code pour les Expressions :

Les algorithmes de génération de code pour les expressions suivent une logique de lecture de droite à gauche. Cela assure une évaluation correcte des opérations. Les exemples concrets ci-dessous illustrent cette approche :

Exemple 1 : Addition Simple (1 + 2) :

```
LOAD #1, R2
ADD #2, R2
HALT
```

*Exemple 2 : Expression Complex (a + b) * (c - d) :*

```
LOAD a(GB), R2
SUB b(GB), R2
LOAD c(GB), R3
ADD d(GB), R3
MUL R2, R3
LOAD R3, R2
HALT
```

6.3 Gestion de registres et de la pile :

La classe "Stack" joue un rôle crucial dans la gestion des registres et de la pile. Les mécanismes de suivi des registres et d'optimisation de la pile assurent une

utilisation efficace des ressources. Voici quelques détails spécifiques à la gestion des registres :

Exemple : Exemple de gestion des registres :

```
LOAD a(GB), R2
SUB b(GB), R2
LOAD c(GB), R3
ADD d(GB), R3
MUL R2, R3
LOAD R3, R2
HALT
```

6.4 Gestion des erreurs :

La classe "ErrorHandler" est dédiée à la gestion des erreurs d'exécution. Des exemples spécifiques pour le Compilateur Deca sont fournis, couvrant des situations telles que la déréférenciation nulle et les erreurs de pile. *Exemple : Exemple de gestion des erreurs :*

```
dereferencing.null:
WSTR "Error: Déréférencement nul"
WNL
ERROR

stack_overflow_error:
WSTR "Error: Débordement de la pile"
WNL
ERROR
```

6.5 Exemple de code généré :

Exemple : Appel de la fonction IloveAssembly

```
ADDSP #1
LOAD IloveAssembly(GB), R2
STORE R2, 0(SP)
LOAD 0(SP), R2
CMP #null, R2
BEQ dereferencing.null
LOAD 0(R2), R2
BSR 2(R2)
SUBSP #1
```