Documentation de conception

G7, gl35

January 2023

Contents

1 Introduction

Ce document présente les différents choix de conception adoptés lors de l'implémentation du compilateur Deca pour les différentes vérifications (lexicales, syntaxiques, contextuelles) ainsi que la partie de la génération du code. Il a pour but l'explication de l'architecture du code pour faciliter la maintenance ainsi que l'amélioration de notre compilateur.

2 Vérification lexicale et syntaxique

2.1 Généralités

Lors de la vérification lexicale, le compilateur vérifie bien que les "mots" ou Tokens du fichier .deca existent dans la lexicographie de deca.

Lors de la vérification syntaxique, le compilateur vérifie que le contenu du fichier .deca appartient à la grammaire du langage (vérification des règles) ce qui va permettre de construire l'arbre syntaxique abstrait par la suite.

2.2 Architecture

- `src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4` :
  Contient les Tokens du langage, en le modifiant on peut modifier le langage deca (ajout ou suppression).

- `src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4` :
  - Contient l'ensemble des règles de la grammaire cette dernière peut s'elargir par l'ajout d'autres règles.

- `src/main/java/fr/ensimag/deca/syntax`:
  - Ce package contient les erreurs levées en cas d'erreur.

- `src/main/java/fr/ensimag/deca/tree`:
  - Les classes pour le langage "Hello World" et "Sans objet" étaient fournies avec un très haut niveau d'abstraction. Par la suite, la construction des classes du langage "Avec objet" est faite de la même manière afin d'éviter les redondances et avoir un code factorisé au maximum.
  Chaque classe contient des méthodes pour l'affichage de l'arbre abstrait de chaque noeud (`prettyPrintChildren`,`iterChildren(TreeFunction)`) ainsi que la méthode `decompile(IndentPrintStream)` permettant la décompilation du fichier .deca donné.

La figure suivante présente l'ensemble des classes du package `tree` et leurs dépendances.

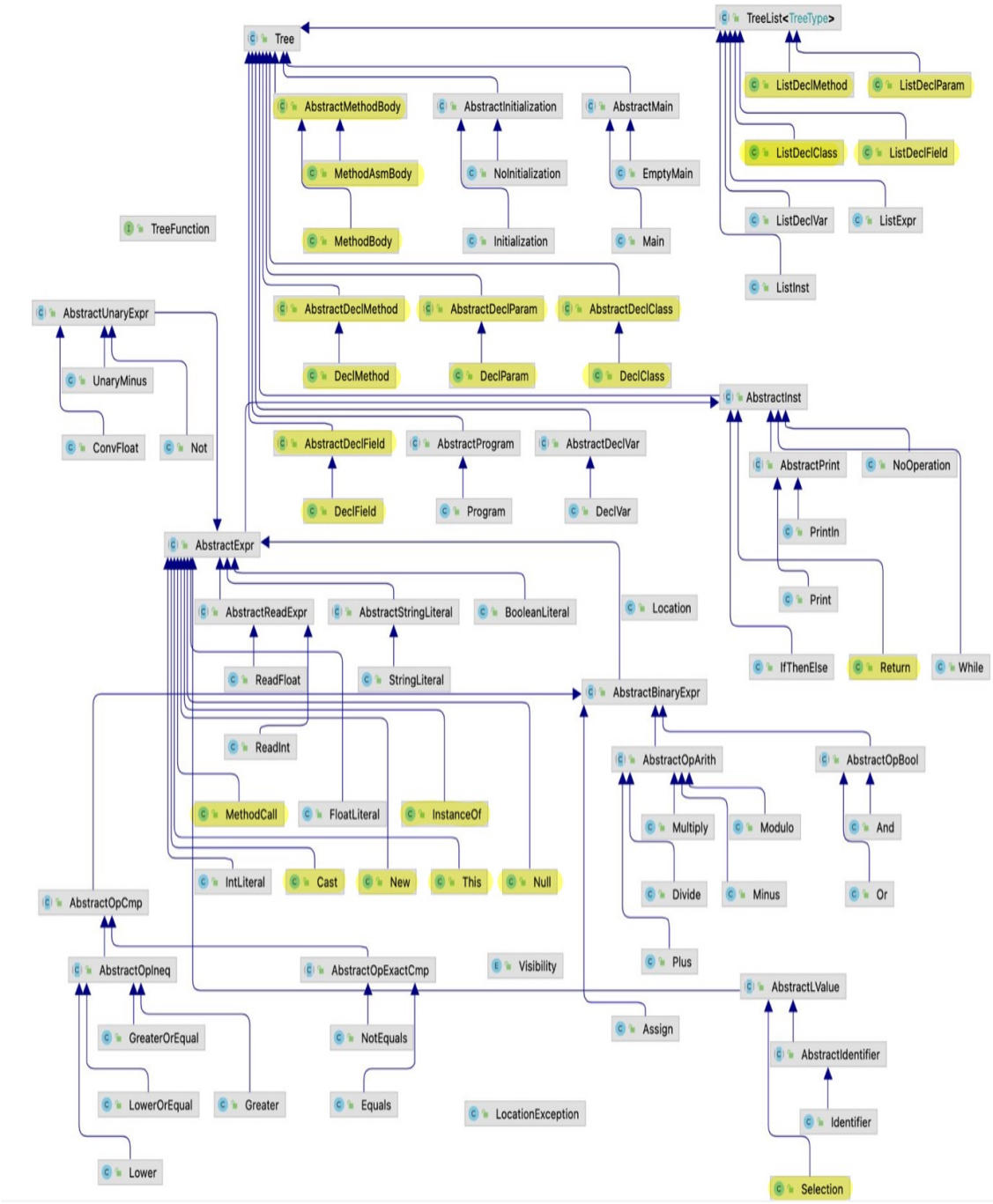

Figure 1: Les classes du package tree (En jaune les classes pour le langage "Avec Object")

3 Vérification contextuelle

3.1 Généralités

Dans cette section, on abordera tout ce qui est relative à la vérification contextuelle d'un programme. On s'intéresse à la vérification du code suivant les différentes règles en rapport avec le langage deca. La vérification contextuelle vient après une étape de vérification syntaxique, et elle a un double but: dériver notre programme à travers les règles adéquats et s'assurer que les conditions relatives à ces dernières sont respectées, mais aussi de construire un arbre décoré (contenant le typage des attributs et champs, les appels de méthodes, et le cast). Cette partie est lancée à travers la ligne suivante **prog.verifyProgram(compiler)**, qui présente une exécution normale dans le cas d'un programme contextuellement correcte, où renvoie l'exception ContextualError avec des précisions sur la règle non respectée dans ce cas-ci.

3.2 Environnement des types

L'environnement type est une structure qui permet de stocker les différents types. Stocker les types, se fait par l'association d'un nom à une définition du type donné. En matière d'implémentation la structure de données assurant ce travail est une map avec l'association d'un Symbol avec un Type Définition. Pour tout programme, lors de l'appel du constructeur de la classe **DecacCompiler**, un environnement type est créé avec les types **int**, **float**, **void**, **boolean** et **Object** de façon prédéfinie, et ensuite sur cet environnement s'additionnent les différents types définis par l'utilisateur dans le programme notamment les classes définies. La structure de données utilisée pour stocker ces correspondances est de la forme Map $(Symbol, TypeDefinition)$. La complexité algorithmique de la recherche et de l'insertion est en temps constants $\mathcal{O}(1)$. L'insertion s'effectue dans notre cas que pour les classes avec la fonction **addClass** de l'environnement type, prenant comme paramètre dans ce cas-ci le nom de la classe et la classe définition créée (Voir Passe 1 pour plus de détails). La recherche d'un élément peut s'effectuer de deux façons, si on a un accès à la **SymbolTable** présent dans le **DecacCompiler**, ou si on a un symbol on effectue la recherche avec la fonction **defOfType** présente dans l'environnement Type, qui renvoie une **TypeDefinition** dans le cas d'existence sinon **null**.

3.3 Environnement des expressions

L'environnement des expressions a une structure de données similaires à celle de l'environnement des types exposés plus haut. Mais une spécificité de plus pour celui-ci est la capacité d'empilement avec le chaînage possible avec l'attribut parentEnvironment dans la classe EnvironnementExp. Pour la déclaration d'une nouvelle association entre un symbol et une définition, on utilise la fonction **declare** qui s'exécute normalement dans le cas ou le symbol n'existe pas déjà, sinon elle renvoie l'exception DoubleDefException. Dans le cadre du langage deca, on a besoin de trois environnements des expressions deux liées avec les trois passes, et le troisième spécifique au Main de notre programme.

- Un environnement contenant l'ensemble des champs et méthodes pour chaque classe.
- Un environnement contenant l'ensemble des paramètres, variables et instructions locaux à une méthode.
- Un environnement associé au Main contenant les déclarations de variables et les instructions correspondant à cette partie du programme.

3.4 Passe 1

Lors de cette première passe, on effectue un parcours sur la liste des déclarations de classes remplis à travers la vérification syntaxique. Pour chaque classe, on effectue un appel sur la fonction **verifyClass** qui a des multiples rôles suivants le cas où on se situe:

1. Dans le cas où la classe définie n'étendrait pas d'une classe définie précédemment, on set la définition de l'attribut superclass dans la classe **DeclClass** à celle de la classe **Object**, et la location à **BUILTIN**.
2. Dans le cas où la classe définie étendrait d'une classe déjà définie, on accède à la définition et la location et le classType associée à celle-ci, on set ses différents attributs à l'attribut superclass.
3. Finalement, on vérifie que la définition de la superclass est non nulle, de type classe et le symbol de la nouvelle classe n'existe pas déjà dans l'environnement type. Si toutes ces conditions sont vérifiées, on effectue l'ajout dans l'environnement des expressions convenables avec la fonction **addClass**.

Figure 2: Illustration du fonctionnement de la première passe

3.5 Passe 2

Lors de cette deuxième passe, on effectue un parcours à l'intérieur de chaque classe, afin de vérifier les champs et méthodes définis à l'intérieur de celle-ci. Ceci s'effectue à travers la fonction **verifyClassMembers** qui effectue un appel de vérification sur la liste de déclaration de champs et de méthodes.

- Pour la déclaration des champs, on vérifie que le type du champ déclaré n'est pas void, et que le symbol utilisé n'existe pas déjà dans l'environnement des expressions empilées. Dans le cas où toutes ses conditions sont valables, on effectue l'ajout dans la hashmap de la classe currente.

- Pour la déclaration des méthodes, on vérifie la liste des paramètres et on récupère la signature si les paramètres sont bien typés. Si une méthode de même symbol est déjà définie dans une super classe, on renvoie une exception de type ContextualError, de même si le nouveau type de retour n'est pas un sous-type de celui déjà défini. Sinon, on effectue l'ajout de la méthode dans la hashmap de la classe courante avec les set type et définition adéquats.

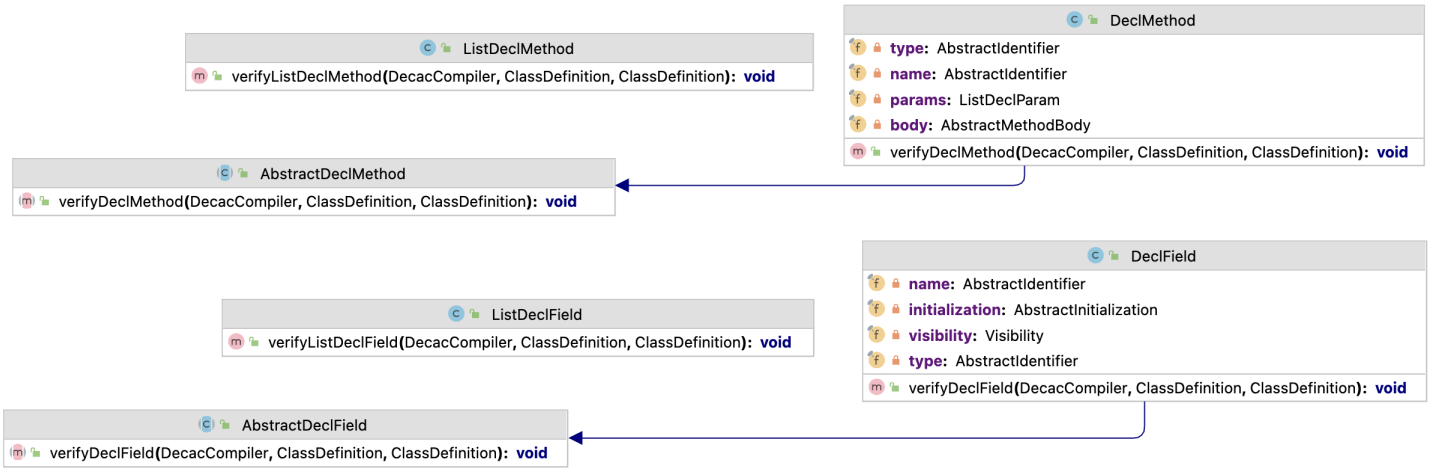

Figure 3: Illustration du fonctionnement de la deuxième passe

3.6 Passe 3

Lors de cette troisième passe, on effectue un parcours sur les différentes classes en appelant la fonction **verifyClassBody**. Cette fonction permet une vérification sur les fields (à travers la fonction **verifyListFieldInit**) en appelant les vérifications nécessaires dans le cas de l'initialisation. Elle permet aussi la vérification des corps des méthodes en appelant les vérifications des déclarations de variables et d'instruction (en utilisant la fonction **verifyListMethodBody**). Un ensemble de vérification exhaustive s'effectue ensuite pour s'assurer des règles de la passe 3 (les expressions, les opérations binaires et unaires, selection, les lvalues,...)

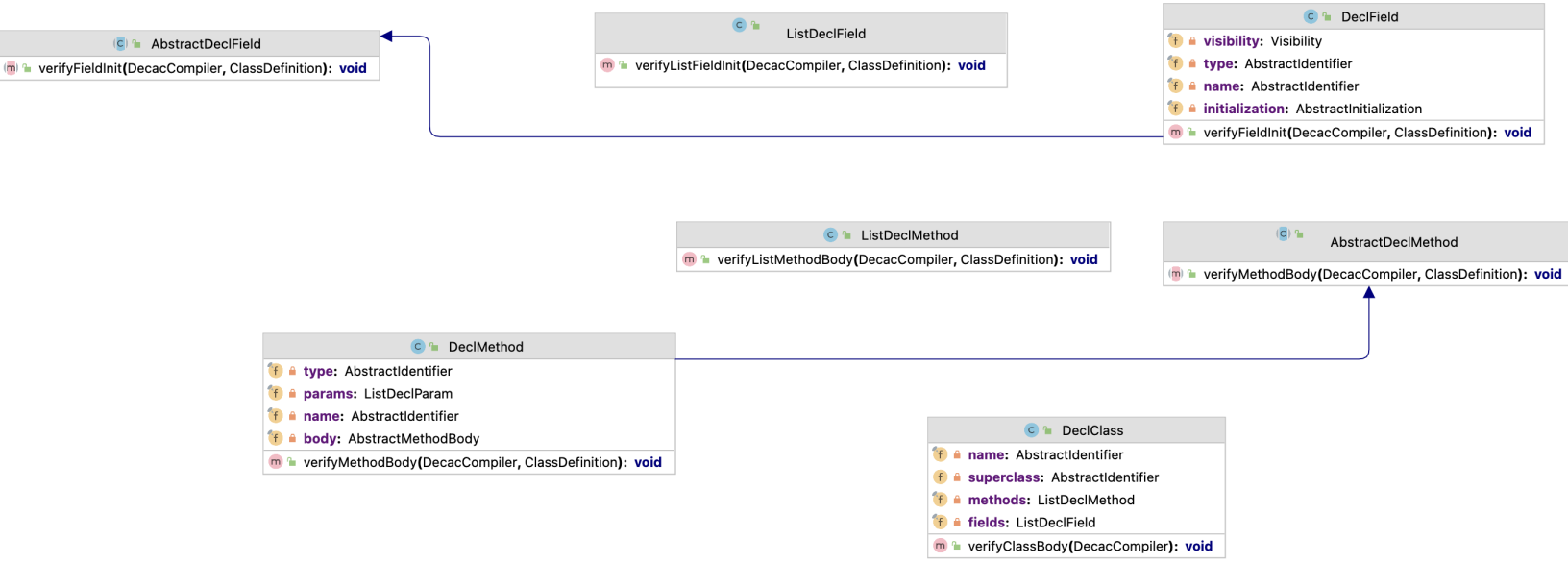

Figure 4: Illustration du fonctionnement de la première passe

4 Génération de code

Le but de cette étape est de générer le code assembleur à partir du résultat de l'étape B (arbre décoré associé à un programme Deca correct). Les différentes difficultés rencontrées lors du développement de cette partie sont: la gestion de la mémoire à travers la pile, le tas et les registres, la gestion des différentes erreurs à l'exécution ainsi que la génération des instructions assembleur.

4.1 Architecture:

•**src/main/java/fr/ensimag/deca/tree:** Les différentes classes des parcours des arbres auxquelles ont été ajouté les méthodes nécessaires à la génération de code.
•**src/main/java/fr/ensimag/deca/ima/pseudocode:** contient les classes associées au programme IMA et aux instructions assembleur.
•**src/main/java/fr/ensimag/deca:** Contient la classe du compilateur , à laquelle nous avons ajouté les différentes données nécessaires à la génération de code.
•**src/main/java/fr/ensimag/deca:** Contient des tests en Deca pour l'étape C.

4.2 Gestion de la mémoire:

Afin de garantir les différentes contraintes associées au compilateur et aux conventions de liaisons, nous avons eu besoin de stocker les données suivantes:
• Une variable entière stockant l'indice maximal du registre utilisable par le compilateur, il vaut par défaut 15 et peut être modifiée en utilisant l'option -r du compilateur.
• Un booléen permettant de supprimer ou non les différents tests à l'exécution associés aux erreurs 11.1 à 11.3. Il vaut par défaut false et peut être modifié avec l'option -n du compilateur.
• Une variable entière permettant de connaître le nombre de paramètre maximal des appels méthodes parmi un ensemble instructions.
• Une variable entière permettant de connaître le nombre de temporaires maximal à l'évaluation des expressions.
• Une variable entière permettant de connaître le nombre maximal de registres utilisé par une suite d'instructions afin de pouvoir sauvegarder ces registres avant l'exécution d'un bloc.

L'accés à ces différentes données nous permet de générer correctement l' instruction **TSTO** au début des différents blocs de code.

4.3 Génération du code assembleur:

Afin de pouvoir générer le code assembleur associé à un programmme Deca correct, nous avons ajouté les méthodes suivantes aux différents classes représentant l'arbre associé au programme Deca:
•**codeGenPrint(DecacCompiler compiler, boolean isHex)** cette méthode génére le code assembleur nécessaire à l'affichage d'un expression, le paramétre isHex permet de spécifier dans le cas où le type de l'expression est le type floattant, s'il faut l'afficher en décimal ou en hexadecimal, ce paramétre n'a pas d'effet sur l'affichage des entiers.
•**codeGenInst(DecacCompiler compiler)** cette méthode génére le code assembleur nécessaire aux différentes instructions.
•**codeExp(int n, DecacCompiler compiler)** cette méthode génére le code assembleur au calcul d'une expression et stocke son résultat dans le registre n, lors de ce calcul, la méthode ne peut utiliser les registres d'indice inférieur strictement à n, en dehors des registres *scratch* R0 et R1 car certaines expressions ont besoin de ces registres pour calculer leur valeur (appel méthode , lecture readInt() et readFloat()), ainsi la méthode se charge de

la gestion des temporaires à l'aide des instructions PUSH et POP, et en mettant à jour la variable entière permettant de connaître le nombre de temporaires maximal, et le registre maximal utilisé.
•**codeBranch(DecacCompiler compiler, boolean b, Label l)** cette méthode permet de générer le code nécessaire relatif aux branchements des structures de contrôle **if ... else if ... else ...** et **while**.
•**dval(DecacCompiler compiler)**: la méthode dval renvoie une **DVal** associée à l'expression afin de faciliter les calculs.
De plus, nous avons ajouté une méthode au compilateur permettant de changer d'EnvironementExp afin de modifier ce dernier lors de la génération de code des différentes méthodes. En effet, les adresses mémoires des différentes variables sont stockées dans EnvironementExp, d'où la nécessité de pouvoir le modifier.

4.4 Gestion des erreurs d'exécution:

Voici la liste des différentes erreurs pouvant survenir à l'exécution:
•L'erreur débordement de pile intervient lorsque la mémoire demandée par le programme pour les variables (globales ou locales aux méthodes), les temporaires, la table des méthodes, les appels de méthodes et la sauvegarde des registres est trop importante.
•L'erreur débordement de tas intervient lors de la création de nouvelles instances d'une classe avec **new**. Cette erreur signifie que la mémoire demandée au tas pour la création de nouvelle instance est trop importante.
•L'erreur entrée/sortie surgit lors de l'appel aux fonctions **readInt()** et **readFloat()**, lorsque le type de la saisie n'est pas compatible avec le type attendue.
•L'erreur débordement opération arithmétique surgit lors d'une opération de division/modulo entre entiers ou flottants lorsque le diviseur vaut 0, ou bien lors d'une opération arithmétique quelconque entre flottants lorsque le résultat n'est pas codable sur un flottant.
•L'erreur dereferencement de null intervient lorsque l'on essaye d'accéder à une méthode ou un champ d'un objet valant **null**.
•L'erreur cast impossible surgit lorqu'une opération de conversion ne peut aboutir.