Documentation de Conception

OUAZZANI CHAHDI Mohammed
BENTOUIMOU Yassine
ABDELWAHID Amine
AIT DRISS Salma
LEMDAGHRI ALAOUI Ibrahim

27 janvier 2023

Table des matières

1 Introduction

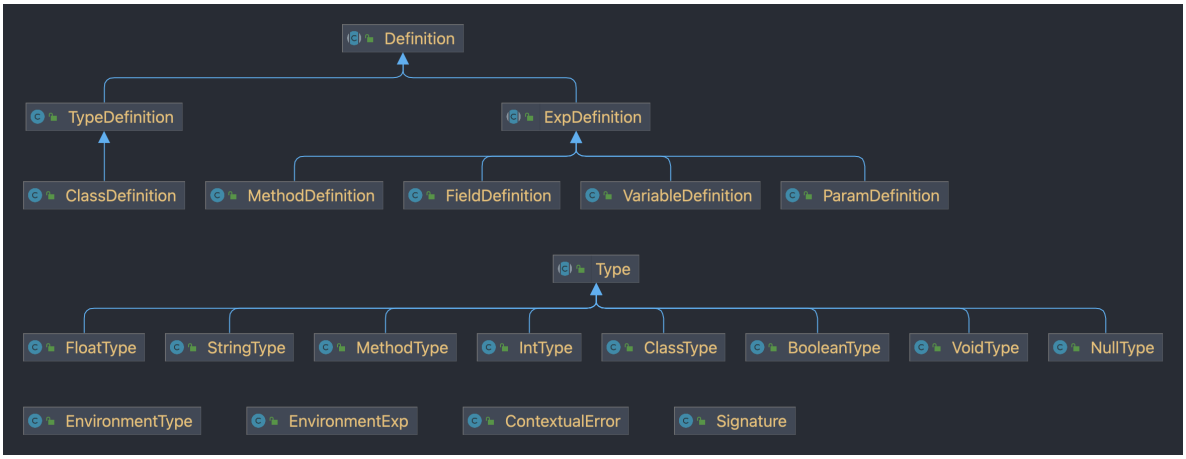
Ce projet est destiné à développer un compilateur du langage Deca (un langage qui ressemble au langage Java). L'implémentation est réalisée essentiellement par le langage JAVA. Chaque fichier source passe par 4 étapes (sauf détection d'une exception) : l'analyseur lexical, syntaxique, la contextualisation et finalement la génération du code assembleur. Ce dernier s'execute via un processeur purement virtuel, qui s'appelle ima, est construit pour rendre le développement du compilateur plus ou moins simple.

2 Les composants

L'analyseur lexical contient déjà les élements principales qu'un compilateur doit posséder et se trouve dans le fichier decaLexer.g4.

L'analyseur syntaxique decaParser.g4 suit la grammaire syntaxique fournie et instancie les objets et les types de bases sous forme d'un arbre prés à être parcouru pour la mise en place de l'analyse contextuelle et la génération du code. Pour cette étape, plusieurs éxceptions ont été définies dans le packetage src/main/java/deca/syntaxe : InvalidLitteralFloat, InvalidLitteralInt qui rejeteront les valeurs invalides des int et des floats, et bien d'autres qui sont utilisées par cette étape.

Les types de bases (**booleanType**, **IntType**, **FloatType**) de ce compilateur sont tous stockés dans le packetage src/main/java/deca/context. Pour déclarer des variables on utilise VariableDefinition. En ce qui concerne les classes du langage, le meme packetage contient les types concernés : (**ClassType**, **MethodType**...). Ces types sont tous utilisés pour instancier les objets correspondants pour définir leurs types et leurs définitions lors de la décoration de l'analyse contextuelle.

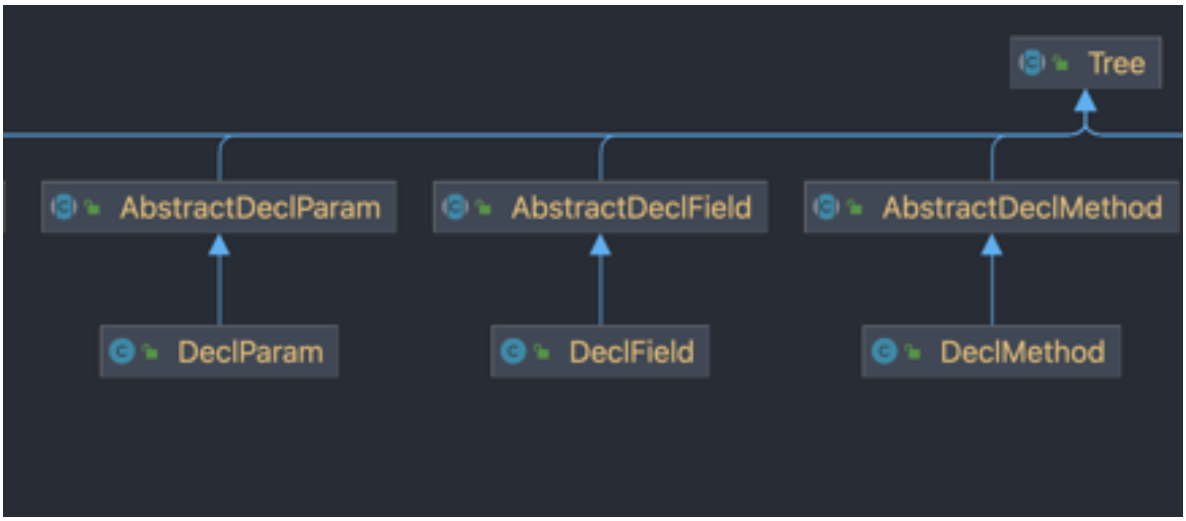


Le meme packetage contient deux types d'environment qui stockeront les éléments déclarés. En ce qui concerne l'**EnvironmentType**, il stockera dès le debut de la compilation du ficher source deca les types de bases (**int**, **float**, **boolean**, **objet**...) et aprés chaque déclaration de classe, cette dernière est ajoutée à l'environment s'il vérifie quelques contraintes. Ce fichier contient aussi quelques opérations et méthodes qui sont trés utilisées par d'autre fichier lors de l'étape de la contextualisation (**subType**, **empilement**).

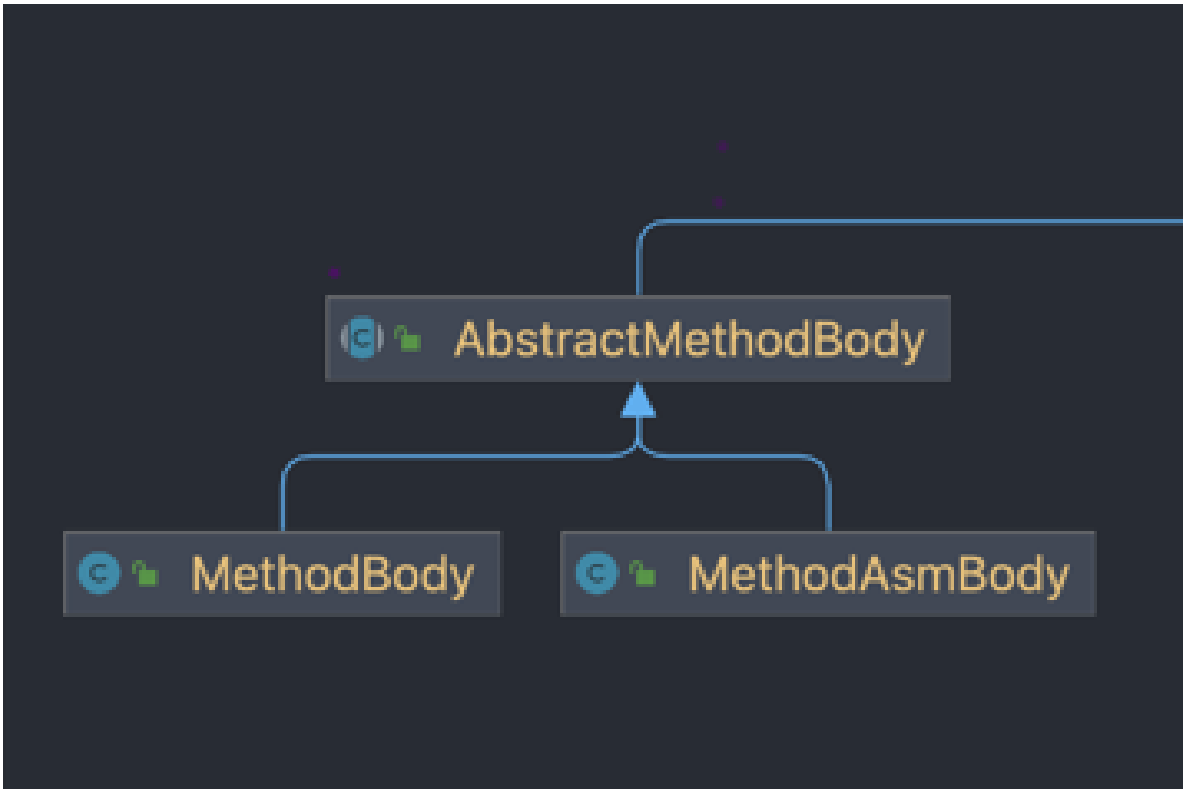
L'EnvironmentExp est un type d'environment qui stockera les instances définies localement que ce soit dans une méthode, une classe ou dans le programme principal (main), il contient aussi les opérations : **Empilement** et **UnionDisjoint**.

3 Declaration d'une classe

Aprés avoir détécter une déclaration d'une classe par le Parser, l'arbre géneré par cette étape doit contenir un objet de type DeclClass auquel est associé le nom, la superClass, la liste des champs(**ListeDeclField**) et la liste des méthodes(**ListDeclMethod**). La ListeDeclField contient les DeclField où on trouve les informations associées à la declaration du champ (la **visibilité**, **type**, le **nom** et l'**initialisation**).
La **ListDeclMethod** regroupe les élements de type **DeclMethod** dont le **type de retour**, le **nom**, la **ListDeclParam**, et puis **Body** y sont attachées.



Pour la **ListDeclParam**, qui elle aussi regroupe les élément de type **DeclParam**. Quant au Body, on distingue deux types qui héritent de la classe **AbstractMethodBody** :
MethodBody et **MethodAsmBody** qui traiteront les deux possibilités pour écrire le corps d'une méthode.



Au sein d'une méthode, outre que les opérations et les expressions fournies et utilisées dans le programme principal, on peut appeler **return** et **this** . La première expression est une instruction donc encapsulée par AbstractInst et la deuxième est une expression tout court suivi par un **dot**. Donc pour bien définir celle-ci, l'implementation de **selection** se trouve dans le meme packetage qui regroupe l'expression à gauche qui pourra être **this** ou une instance déclarée, et l'expression à droite qui devra être un champ ou une méthode que l'instance (ou this) doit posseder soit par déclaration ou par héritage. Il existe aussi d'autres instructions qui peuvent être utilisées en langage deca. Par exemple, il est possible d'instancier un objet à **null** ou le créer à l'aide de l'instruction **new**. On peut aussi faire appel à une methode dans le main ou dans une autre classe, sous certaines conditions, ou encore faire un cast entre 2 objets ou variable ou champs de types compatibles.

4 Gestion des registres

La gestion des registres est nécessaire pour minimiser les ressources utilisées lors de l'exécution, ainsi que implémenter l'option **-r** qui exige la compilation en limitant le nombre de registres utilisables.
Pour gérer ces contraintes, nous avons ajouté dans la classe **DecacCompiler** les attributs **currRegNum** qui présentent le prochain registre utilisable parmi les registres à usage général.
Nous présentons aussi les méthodes **useReg()** et **freeReg()** qui mettent à jour les registres utilisables. A chaque utilisation effective d'un registre, il faut appeler la méthode **useReg()**, et à chaque libération il faut appeler la méthode **freeReg()**.
L'utilisation d'un entier pour conserver toutes les informations utiles sur les registres permet d'optimiser le compilateur.
La méthode **useLoad()** aussi joue un rôle très important puisqu'elle nous permet de décider si nous devons utiliser les instructions **Load/Store** ou **Push/Pop** pour manipuler nos registres.

5 Gestion de la mémoire(Offset global et locale) :

Pour gérer la mémoire nous avons introduit dans la classe compilateur les attributs **offset** et **tempStack**. **Offset** est incrémenté lors de l'ajout des variables globales dans la pile, et la construction de la table des méthodes virtuelles. **TempStack** permet de suivre l'évolution de la pile vis-à-vis des variables temporaires.
Nous avons aussi implémenté la méthode **updateBlocRegMax()** qui permet de faire les calculs à la volée pour calculer les empilements nécessaires fait dans la mémoire.
Ainsi on ajoute l'instruction **TSTO** pour tester sur la possibilité de faire ces empilements en utilisant les méthodes **append()** et **addFirst()** offerte par le paquetage.

6 Gestion des erreurs

Pour gérer les instructions, nous avons ajouté un **HashMap errorsMap** dans le compilateur qui contient les étiquettes des erreurs avec leurs messages. L'utilisation de cette structure de données nous permet de ne stocker les étiquettes qu'une seule fois et ainsi optimiser notre utilisation des ressources. Cela nous permet aussi d'avoir des tests d'appartenance et des ajouts en temps constant.

7 Solutions pour les problèmes de branchement :

Pour assurer des branchements fonctionnels, nous avons eu besoin de l'unicité des étiquettes. Pour cela nous nous basons principalement sur la **Location** des lexèmes pour construire l'étiquette. Dans le cas où on peut avoir une confusion sur la **Location** des tokens, nous utilisons l'adresse de l'instance pour construire l'étiquette.

8 Les méthodes importantes :

Pour factoriser le code, nous avons introduit des méthodes qui sont utiles et qui ont diverses fonctionnalités. La méthode **codeGenLoad** par exemple, permet de retourner un registre contenant la valeur de l'expression, cette méthode permet de garantir la récursivité des opérations
Puisque les champs des classes n'ont pas une adresse fixe dans la mémoire. La méthode **setAdrFields()** permet de donner une valeur à un champ en utilisant un registre. On appelle cette méthode avant d'appeler les méthodes des opérations et des branchements qui utilisent les adresses de leurs opérandes.