
Projet Génie Logiciel

Compilateur Deca

---

# Documentation Conception

---

*Auteurs :*
M. Benjamin ARGENSON
M. Jean-Baptiste LEFOUL
M. Thomas POUGET-ABADIE
M. Robinson PRADA MEJIA
M. Alexandre PROY

*Encadrants :*
Pr. Philippe BODIGLIO
Pr. Catherine ORIAT

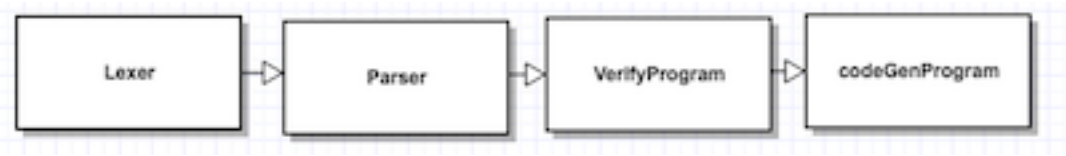
Version du 26 janvier 2017

# Table des matières

# 1 Architecture du projet

## 1.1 Architecture globale

Pour rappel, la compilation s'effectue en trois étapes que nous pouvons représenter à travers le graphique ci-dessous.




Dans une première étape, le fichier .deca est donné en entrée du lexer afin de le réduire à un ensemble de tokens pouvant ensuite être donnés au parser. Ce parser va ensuite construire un arbre de syntaxe abstraite et non décoré.

Au cours de la deuxième étape, nous effectuons une vérification contextuelle de cet arbre de syntaxe abstraite avec un ensemble de méthode "verify". Ces méthodes vont également permettre de décorer l'arbre.

Au cours de la troisième étape, nous effectuons la génération de code en revisitant les différents noeuds de l'arbre mais cette fois-ci en faisant appel à un ensemble de méthodes "codeGen".

Il est important de noter que le lexer ainsi que le parser font partie d'un framework nommé antlr et les fichiers ne sont donc pas des .java. Il est inutile de passer plus de temps sur ces deux fichiers et donc sur la première étape de la compilation puisque leur fonctionnement est expliqué dans le PDF du sujet et nous n'avons que complété ces fichiers sans en modifier le fonctionnement global.

## 1.2 Liste des classes

### 1.2.1 Paquetage Tree

Ce paquetage permet de constuire l'arbre de syntaxe abstraite à partir d'un fichier .deca. La classe **Tree.java** est la classe mère à toutes les classes du paquetage.

Afin de pouvoir prendre en compte le corps des méthodes, nous avons implémenté les classes suivantes :

- **AbstractMethodBody.java**

Cette classe étend la classe **Tree.java**.

- **MethodBody.java**

Cette classe étend la classe **AbstractMethodBody.java**.

- **MethodBodyASM.java**

Cette classe étend la classe **AbstractMethodBody.java**.

Les deux classes décrites précedemment permettent de prendre en compte les deux principales manières d'écrire le corps d'une méthode. La première est la plus classique et consiste en une liste de déclarations de variables puis d'instructions tandis que la deuxième permet d'écrire directement du code en langage d'assemblage dans le corps d'une méthode.

Afin de pouvoir convertir des entiers en flottants, nous avons implémenté la classe ci-dessous :

- **ConvInt.java**

Cette classe étend **AbstractUnaryExpr.java** et est implémentée de manière similaire à **ConvFloat.java**.

Pour la déclaration de paramètres d'une méthode, nous avons implémenté les classes suivantes :

- **AbstractDeclParam.java**

Cette classe étend la classe **Tree.java**.

- **DeclParam.java**

Cette classe étend **AbstractDeclParam.java** et implémente les différentes méthodes de vérifications contextuelles ainsi que les méthodes de génération de code en langage d'assemblage vis-à-vis de la déclaration de paramètres.

- **ListDeclParam.java**

Cette classe étend une TreeList<AbstractDeclParam>.

Pour l'appel des méthodes, nous avons implémenté la classe ci-dessous :

- **MethodCall.java**

Cette classe étend **AbstractExpr.java**.

Pour instancier des objets d'une classe, nous avons implémenté la classe suivante :

- **New.java**

Cette classe étend **AbstractExpr.java**.

Pour pouvoir faire un **return** au sein d'une méthode, nous avons implémenté la classe suivante :

- **Return.java**

Cette classe étend **AbstractInst.java**.

Pour pouvoir appeler l'objet au sein d'une méthode issue de la même classe (**this**),

nous avons implémenté la classe ci-dessous :

- **This.java**

Cette classe étend **AbstractLValue.java**.

Pour pouvoir caster des objets, nous avons implémenté la classe ci-dessous :

- **Cast.java**

Cette classe étend **AsbtractBinaryExpr.java**

Pour la définition d'un objet **null**, nous avons implémenté la classe ci-dessous :

- **NullLiteral.java**

Cette classe étend la classe **AbstractExpr.java**.

Toutes les classes implémentées ci-dessus découlent directement des différentes règles de grammaire permettant de générer du code .deca et représentent des noeuds dans l'abre de syntaxe abstraite. Dans chacune de ces classes, nous implémentons des méthodes permettant d'effectuer les vérifications contextuelles propres à la deuxième étape de compilation ainsi que des méthodes de génération de code .ass pour la troisième étape de la compilation.

### 1.2.2 Paquetage Context

Ce paquetage permet d'effectuer les vérifications contextuelles propres à la deuxième étape de la compilation. Pour ce faire, nous devons pouvoir garder en mémoire un ensemble d'informations lors de la visite des différents noeuds de l'arbre syntaxique.

Pour l'environnement global contenant la définition des différentes classes déclarées ainsi que les différents types du langage, nous avons créé une structure de donnée avec la classe ci-dessous :

- **EnvironmentType.java**

Cette classe est analogue à **EnvironmentExp.java** mais représente l'environnement global d'un programme .deca.

Pour un objet de type **null**, nous avons implémenté la classe suivante :

- **NullType.java**

Cette classe étend la classe **Type.java**.

Pour le type **void**, nous avons implémenté la classe suivante :

- **VoidType.java**

Cette classe étend la classe **Type.java**.

### 1.2.3 Paquetage Codegen

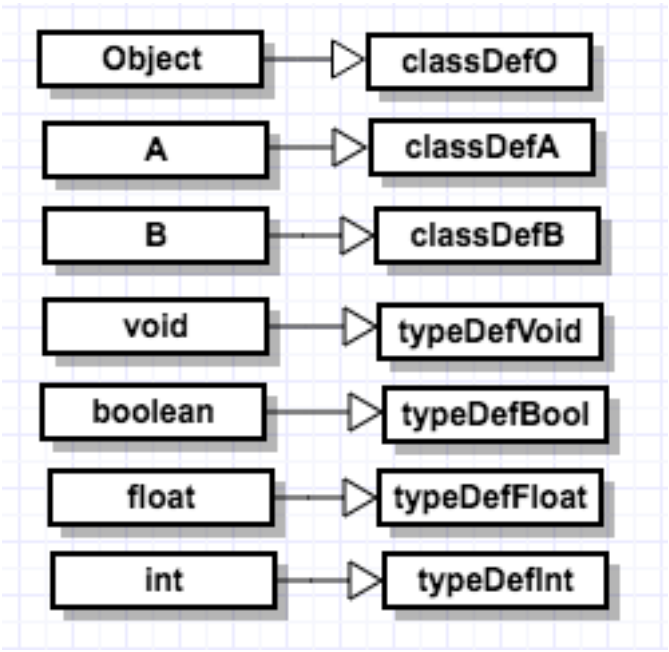
Nous avons décidé de ne pas utiliser ce paquetage.

# 2 Spécifications du code

## 2.1 Paquetage Context

La spécification la plus importante du paquetage **Context** est notre implémentation de l'environnement global avec la classe **EnvironmentType.java**.

De manière analogue à la classe **EnvironmentExp.java**, notre classe contient une HashMap permettant de lier des symboles à des instances de **TypeDefinition.java**. Cette structure de donnée peut se représenter de la manière suivante :




Dans l'instance de **EnvironmentType.java**, nous retrouvons les symboles des différentes classes ayant été déclarées ainsi que les différents types. À ces symboles sont associées des instances de **TypeDefinition.java**.

Pour les classes, l'instance de **ClassDefinition.java** contient un attribut qui est instance de **EnvironmentExp.java**.

C'est donc à travers la classe **EnvironmentType.java** que nous parvenons à créer une variable globale contenant les différents environnments permettant d'effectuer les vérifications contextuelles.

## 2.2 Paquetage Tree

### 2.2.1 Vérifications contextuelles

Nous disposons désormais d'une classe **EnvironmentType.java** dont l'instance contient tous les environnements. Cette variable est importante notamment lors de la deuxième étape de la compilation mais également lors de la troisième étape. Pour effectuer les vérifications contextuelles, nous visitons les différents noeuds de l'arbre de syntaxe abstraite. Il nous faut donc pouvoir accéder à cette variable depuis n'importe quel noeud.

Pour ce faire, notre instance de **EnvironmentType.java** est un attribut de la classe **DecacCompiler.java** qui est passé en paramètre de toutes les méthodes de vérifications contextuelles ainsi qu'en paramètre de toutes les méthodes de génération de code.

Lorsque nous visitons les noeuds, cette variable est modifiée. En effet, lors d'une déclaration de classe, nous ajoutons un symbole associé à une instance de **ClassDefinition.java** et constituant le nouvel environnement de cette classe. Chaque classe contient donc son propre environnement.

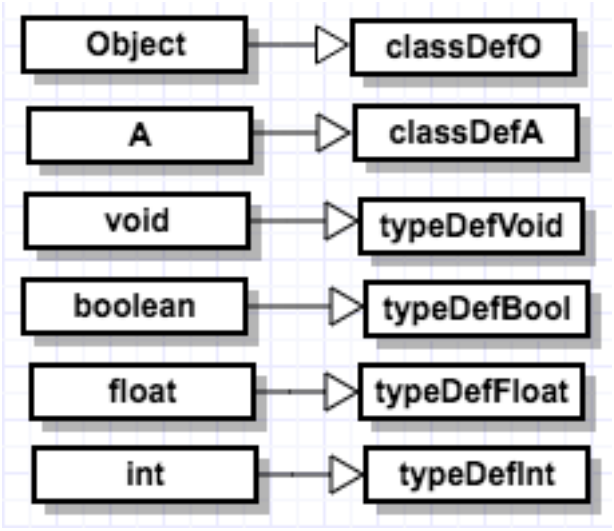
Il est important de noter que nous ne gardons en mémoire ni les environnements propres aux différentes méthodes définies au sein des classes ni l'environnment de la partie **main**.

Ainsi, lors des vérifications contextuelles du corps d'une méthode, nous créons une instance de **EnvironmentExp.java** que nous passons en paramètre des différentes méthodes de vérifications contextuelles. Il en est de même pour la partie **main** du fichier .deca.

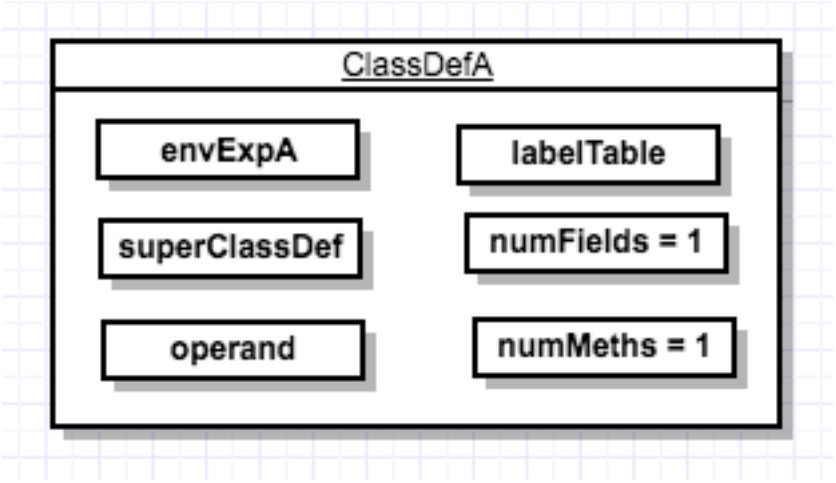
Afin d'illustrer le processus de vérifications contextuelles, étudions le programme suivant :

```
class A {

    int x;

    void hello() {
    }

}
```

Après vérifications contextuelles, nous obtenons un environnement global de la forme suivante :

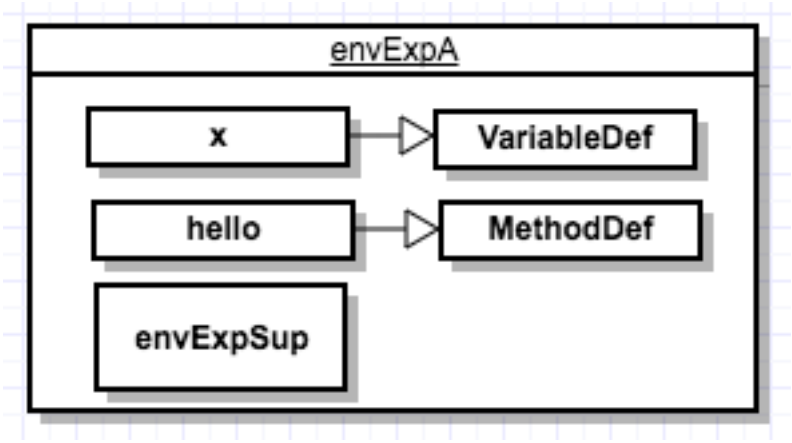



En étudiant de plus près **classDefA** nous obtenons la représentation suivante :




Par rapport au code initial, nous avons choisi de rajouter les attributs **operand** ainsi que **labelTable**.
L'attribut **operand** nous permet de stocker l'adresse de la classe. Ainsi, la **ClassDefinition** d'une classe nous permet d'accéder facilement à son emplacement au sein de la table des méthodes.
L'attribut **labelTable** joue un rôle similaire pour les méthodes. Les labels servent d'adresse pour le corps des méthodes et les stocker dans la **ClassDefinition** nous donne un accès facile à celles-ci.

En étudiant de plus près **envExpA** nous obtenons la représentation suivante :




Ci-dessus, envExpA contient un symbole pour la méthode **hello** associé à une **MehodDefintion** ainsi qu'un symbole pour **x** associé à une **VariableDefinition**. De plus, envExpA possède également comme attribut l'environnement de la classe mère qui est ici la classe **Object** étant donné que la classe **Object** est mère de toute classe et que A n'étend aucune autre classe.

### 2.2.2 Génération de code

Pour ce qui est de la génération de code, chaque classe du paquetage **Tree** dispose de méthodes **CodeGen** et/ou **CodeGenPrint**. Au cours de la troisième étape de compilation, nous parcourons l'arbre de syntaxe abstraite décoré de la même manière que lors des vérifications contextuelles mais cette fois-ci en appelant les différentes méthodes de génération de code.

Considérons le programme ci-dessous à titre d'exemple :

```
class A {

    int x;

    void hello() {

        println("hello");

    }

}
```

```
{
    A a = new A();
    a.x = 2;
    a.hello();
}
```

Nous commençons tout d'abord par la classe **Program.java**.

- **codeGenListDeclClass**

Cette méthode est appelée en premier sur l'attribut **classes** de cette classe. Cette méthode permet de construire la table des méthodes, ceci se faisant au tout début du programme .ass.

- **codeGenMain**

Cette méthode est la deuxième à être appelée. Elle permet de générer le code correspondant au bloc main du programme .deca.

- **codeGenListDeclClassMethods**

Cette méthode permet de générer le code correspondant aux corps des méthodes.

Ainsi, le code .ass contiendra tout d'abord le code permettant la création de la table des méthodes. Ensuite, il contiendra le code propre au bloc main du programme .deca. A la fin se trouve le corps des méhodes avec leurs labels. Pour terminer, la méthode **CodeGenProgram** crée les labels permettant de gérer certains messages d'erreur qui se trouvent à la fin du fichier .ass.

Commençons par étudier la méthode **CodeGenListDeclClass** de la classe **ListDeclClass.java**. Cette méthode appelle **CodeGenDeclClass** sur chaque instance de **DeclClass.java** contenue dans la TreeList. **CodeGenDeclClass** appelle à son tour **CodeGenDeclMethods** qui permet la construction de la table des méthodes.

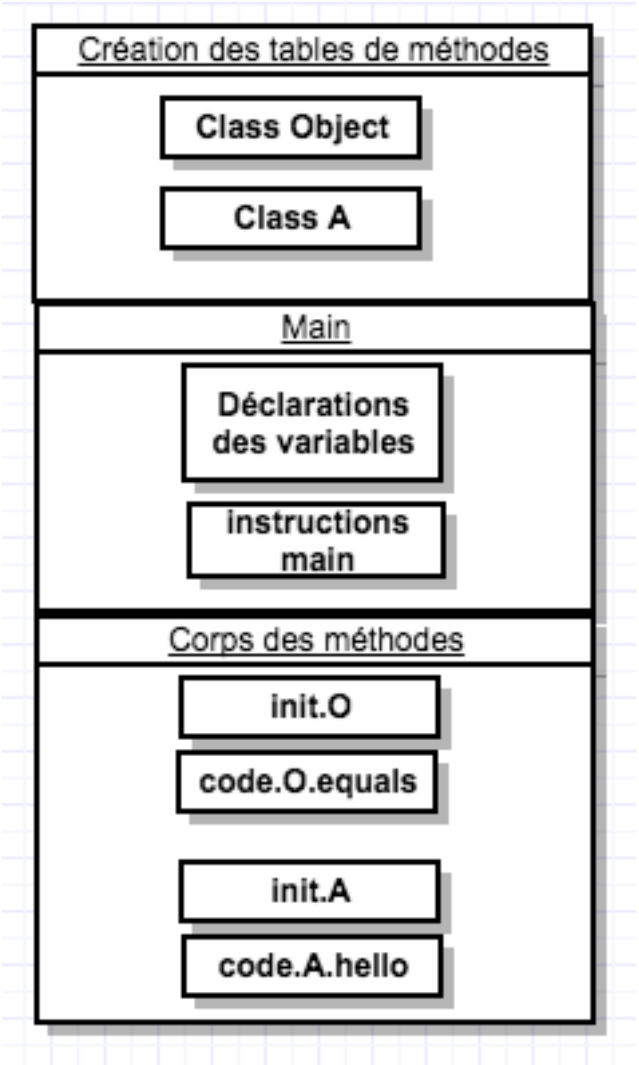
Ensuite, la méthode **codeGenMain** appelle **codeGenListDeclVar** puis **codeGenListInst** sur ses attributs **declVariables** et **insts** respectivement.

**CodeGenListDeclVar** appelle **CodeGenDeclVar** sur chaque instance de **DeclVar.java** contenue dans la TreeList. Cette méthode stocke l'adresse de la variable dans l'instance de **VariableDefinition.java** associé à l'attribut **VarName** de **DeclVar.java**. Ainsi, cette

adresse sera accessible facilement à partir de l'environnement du bloc **main**. **CodeGenDeclVar** appelle **CodeGenInitialization** sur son attribut initialization qui est instance de **Initialization.java** et non pas **NoInitialization.java** dans notre exemple.

Les méthodes de génération de code s'appelle de cette manière à tour de rôle afin de créer la totalité du fichier .ass.

Le fichier .ass résultant sera organisé de la manière ci-dessous :




Une convention et spécificité de notre compilateur à garder en tête est notre utilisation des registres ou plutôt notre faible utilisation de registres. Pour illustrer ce principe, nous pouvons considérer l'expression suivante :

```
2 + 3
```

Pour une telle instruction, la méthode **CodeGenExp** de **AbstractBinaryExpr.java** va appeler la méthode **CodeGenExp** de **IntLiteral.java** pour **2** et **3**. Les méthodes **CodeGenExp** ont la caractéristique de "pusher" leurs résultats sur la pile à la fin de leur exécution plutôt que de stocker ces résultats dans des registres. Ainsi, lors de la fin de l'exécution de **CodeGenExp**, nul besoin de savoir dans quel registre le résultat a été stocké. Il suffit de "popper" ce qui est à la tête de la pile dans un registre au choix.

ENSIMAG
681, rue de la Passerelle
38400 Saint-Martin-d'Hères