

# Projet Génie Logiciel

## Documentation de conception

ENSIMAG 2ème année - 2022-2023

---

### Groupe 1 - équipe 3

Jorge Luri Vañó  
Nils Depuille  
Vianney Vouters  
Virgile Henry  
Logan Willem

# I. Architecture Fournie

## 1) Principe de base du compilateur

La commande “decac” exécute le programme principale du compilateur, dans la classe “DecacMain”. Cette classe s’occupe de lire et d’interpréter les options. Ensuite, le programme va créer des instances du compilateurs, (“DecacCompiler”), et lancer la compilation.

La classe du compilateur contient principalement un “IMAPogram”, qui est une classe représentant le programme final IMA. Le compilateur va prendre en entrée le fichier à compiler, créer des instances du lexer et du parser généré avec ANTLR4, qui vont créer et renvoyer un “AbstractProgram”. Nous reviendrons sur cette classe lors des différentes classes de l’arbre.

Ensuite, le compilateur va appeler la vérification contextuelle sur le programme, qui va être faite de façon récursive sur l’arbre créé.

Finalement, le compilateur va appeler la génération du code, qui va écrire dans le programme IMA. Cette génération est elle aussi faite de façon récursive. Et enfin, le programme IMA sera écrit dans un fichier.

## 2) Les classes de l’arbre :

Un programme est représenté dans toute la compilation comme un arbre. La classe principale abstraite de cet arbre, “Tree”, implémente les méthodes générales utilitaires pour manipuler les nœuds. Chaque partie du programme va surcharger cette classe, selon sa fonctionnalité dans le langage déca.

La classe principale est un “AbstractProgram”, qui représente l’intégralité du programme. Elle est composée d’une liste de déclaration de classe et d’un bloc de code principal.

Le bloc de code principal, “AbstractMain”, peut être soit un bloc de code concret, “Main”, ou vide : “EmptyMain”. Un bloc de code “Main” contient une liste de déclarations de variables et une liste d’instructions.

Une déclaration de classe contient une liste de champs (“ListDeclField”) et une liste de méthodes (“ListDeclMethod”).

Enfin, une déclaration de méthode contient une liste de paramètres, et un corps de méthode. Ce corps de méthode est abstrait car peut être de deux types : du code déca, ou du code en assembleur. Une méthode en déca contient, comme le bloc principal, une liste de déclaration de variables et une liste d’instructions.

### 3) Les Instructions

Les instructions représentent une grande partie du fonctionnement du langage déca. On a une classe abstraite pour représenter ces instructions, "AbstractInst". Cette classe contient énormément de méthode abstraite pour la génération du code, la vérification contextuelle.

Les différentes instructions que l'on a en déca sont :

- "NoOperation" : pas d'opération, ou l'instruction vide.
- "IfThenElse" : l'instruction conditionnelle, qui contient d'autres blocs d'instructions pour les corps des branches "then" et "else", ainsi qu'une condition
- "While" : une boucle conditionnelle, qui elle contient également d'autres instructions en plus de sa condition.
- "Print / Println" : une instruction pour afficher une valeur dans la console.

Ces instructions se basent également sur une autre classe importante du langage, les expressions, dans la classe "AbstractExpr".

### 4) Les Expressions

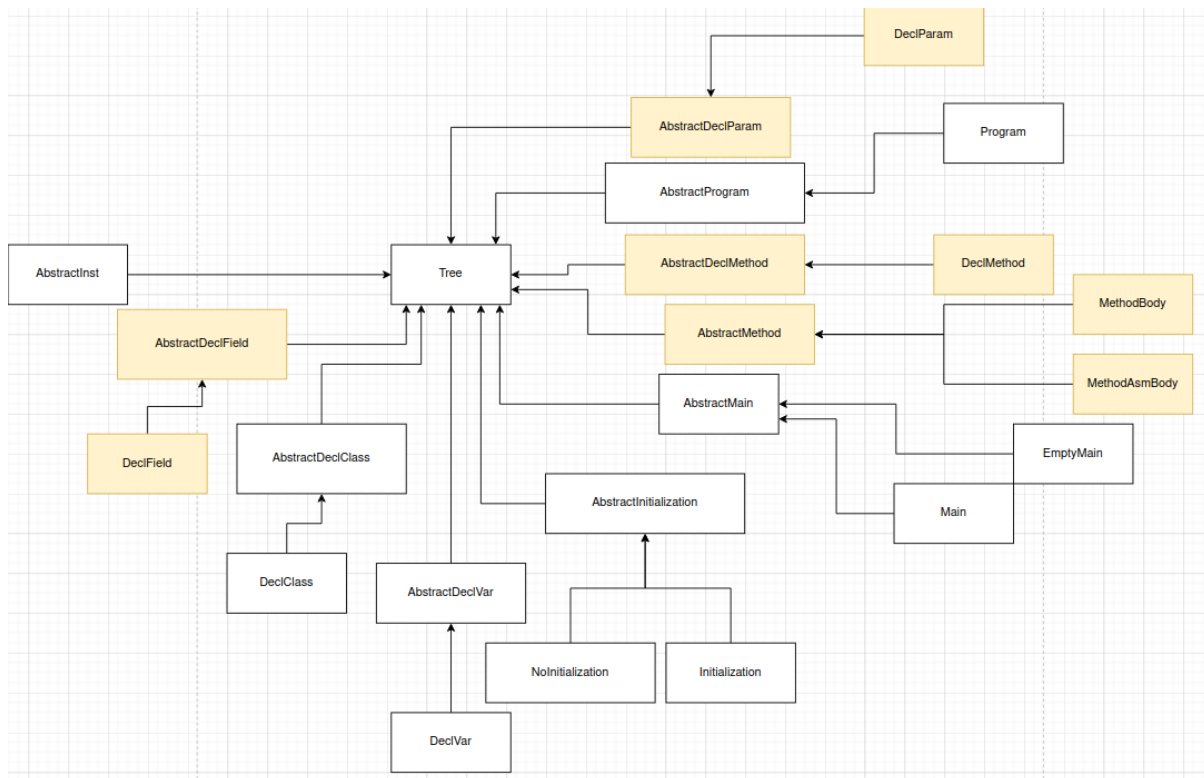
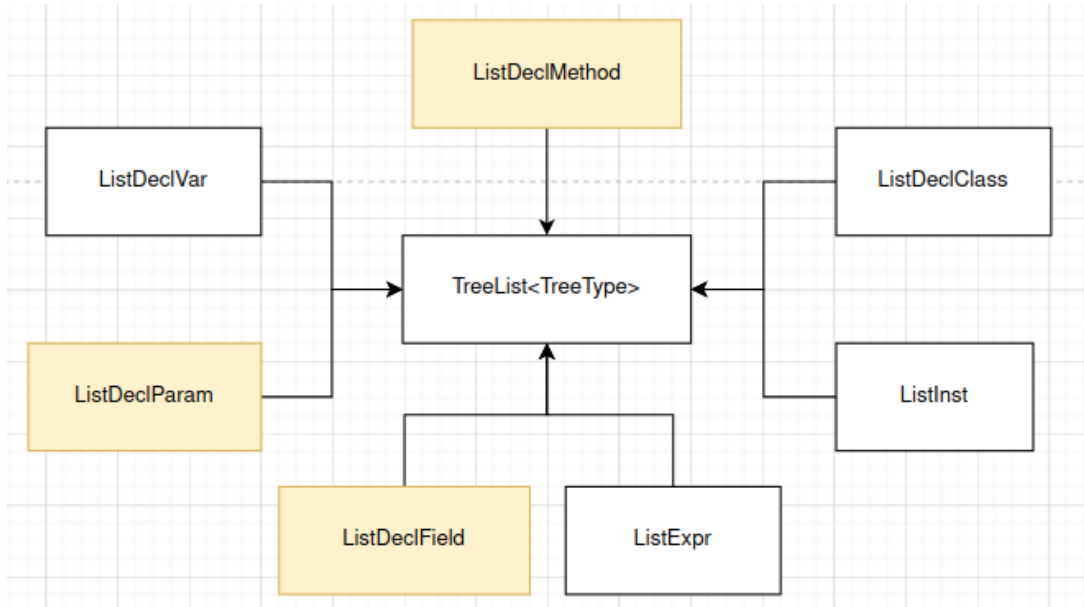
Les expressions sont l'ensemble des nœuds de l'arbre qui renvoient des valeurs. Cela va des simples littéraux aux opérations jusqu'à l'assignation de variables.

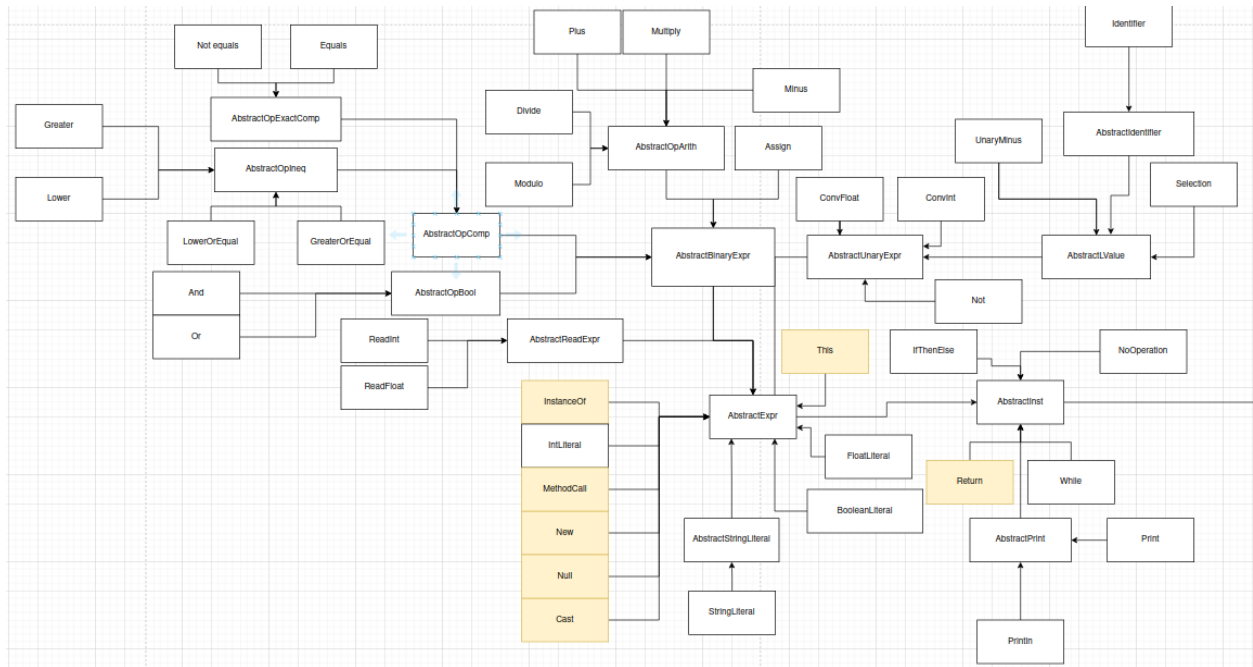
L'ensemble des expressions est

- "FloatLiteral", "IntLiteral", "StringLiteral", "BooleanLiteral" qui possèdent leur valeur en field
- "AbstractUnaryExpr" : Avec les opérations qui ne prennent qu'une seule opérande, comme les Not, ConvFloat
- "AbstractBinaryExpr" : Avec les opérations binaires arithmétiques et de comparaisons
- "AbstractReadExpr" : qui contient les ReadInt et ReadFloat

## 5) Schémas:

Voici les schémas pour représenter toutes les classes utiles de l'arbre, et leur hiérarchie.





## II. Classes Ajoutées

Avec l'implémentation de l'objet dans le langage déca, on a du rajouter des classes dans l'arbre abstrait qui représentaient certaines parties du programme avec objet, notamment :

Instructions Abstraites:

- "Return" : mot clé pour renvoyer une valeur depuis une fonction.

Expressions Abstraites:

- "InstanceOf" : savoir si une classe est une instance d'une de ses classes filles.
- "MethodCall" : Appel de méthode.
- "New" : Crée un nouvel objet de la classe précisée.
- "Null" : Représente la valeur nulle pour les objets.
- "Cast" : Pour effectuer un transtypage sur un objet.
- "This" : Pour référencer une classe depuis une de ces méthodes.

Nouveaux Noeuds :

- "AbstractDeclField / DeclField" : Déclaration d'un champ d'une classe.
- "AbstractDeclParam / DeclParam" : Déclaration des paramètres d'une méthode.
- "AbstractDeclMethod / DeclMethod" : Déclaration d'une méthode

- “AbstractMethodBody / MethodBody / MethodAsmBody” : Corps d’une méthode

Nouvelles listes de Noeuds :

- “AbstractListField” : Liste de déclaration de champs.
- “AbstractListMethod” : Liste de déclaration de méthodes.
- “AbstractListParams” : Liste de déclaration de paramètres d’une méthode.

## III. Génération du code

### 1) Introduction

La génération du code étant beaucoup moins guidée que la partie A et B, il a fallu écrire nous même une bonne partie de l’architecture.

Toutes les méthodes qui génèrent du code sont nommées “codeGen[...]”. Ainsi, on retrouve la génération du code pour les méthodes, instructions, expressions, etc...

Le principe de base est que la génération du code est appelée récursivement depuis le programme. La génération du code du programme va appeler la génération du code des classes et du bloc principal, etc.


La génération concrète d’assembleur est donc faite à plusieurs endroits.

### 2) Contexte de génération du code

Nous avons créé des classes et des méthodes pour simplifier la génération du code. La plus grosse implémentation est la classe “CompilerContextBlock”, qui représente un contexte de code pour la génération du code.

On va désigner par contexte une zone de code où les registres et la pile locale sont considérés comme non utilisés au début. Cela peut être le bloc principal, ou un corps de méthode.

Le compilateur va alors avoir une liste de contexte de génération, et on pourra créer et terminer des contextes.



Un contexte de génération permet de gérer l'utilisation des registres, ainsi que la taille maximale de la pile. Il contient donc entre autres deux méthodes, "allocateRegister" et "freeRegister" qui permettent de récupérer un registre non utilisé et de le libérer. Cela permet de ne pas se soucier de l'utilisation des registres lors de la génération du code: à chaque fois que l'on a besoin d'un registre, on le demande au contexte courant. Si aucun registre n'est disponible, le contexte va se charger de sauvegarder un registre sur la pile, et le restaurera lorsque le registre sera libéré. Il ne sera pas réellement désigné comme libre, mais reprendra son utilisation précédente.

Cela ajoute quelques contraintes cependant: tout registre alloué doit être libéré: on a en effet rajouté une vérification qui s'assure que lorsqu'un contexte est terminé, tous les registres sont libres. Sinon, la compilation échouera. Cela nous permet surtout à nous de s'assurer que l'on utilise correctement les registres, et qu'il n'y a pas eu d'oubli de libération. De plus, la pile doit être au même niveau lors de l'allocation d'un registre et de sa libération, car l'allocation peut avoir sauvegardé un registre sur la pile.

Enfin, la génération du code assembleur est déportée dans le contexte. Lorsqu'un contexte se termine, le code de ce contexte est accolé au contexte précédent. Cela permet de facilement ajouter du code en début de chaque contexte, très utile pour générer le code de méthodes par exemple.


### **3) Génération du code des expressions**

La génération de code des expressions doit être légèrement différente du reste. En effet, une expression renvoie forcément une valeur. Ainsi, la fonction de génération du code des expressions prend en argument un registre, qui est le registre dans lequel l'expression en question doit renvoyer sa valeur. Si ce registre est nul, alors l'expression mettra sa valeur sur la pile.

Cette décision a énormément simplifié la génération du code. En effet, cela permet encore une fois de la garder simple : par exemple, les immédiats ont juste à se charger dans le registre donné, de même, les identifiants ont juste à charger la valeur à leur adresse dans le registre donné. Les fonctions sont donc courtes et claires.

De la même façon, les expressions plus complexes, telles que les opérations binaires n'ont qu'à allouer des registres, appeler la génération des expressions sur leur opérandes en leur passant ces registres, puis rajouter une ligne pour leur opération, et charger le résultat dans le registre demandé.

L'assignation est légèrement particulière. Comme on assigne à une "AbstractLValue", on va créer une méthode de génération du code pour ces L-Values, qui aura elle aussi un registre en



argument, sauf que cette fois ce sera le registre qui contient la valeur à assigner à la L-Value. Ensuite, les L-Value vont implémenter cette méthode (sélections, champs et identifiants). Encore une fois, le code reste finalement très simple pour l'assignation : on utilise le registre donné (puisque l'assignation est une expression), on génère le code de l'expression du membre de droite, puis on appelle la génération du code de la L-Value.

## IV. Conclusion

Finalement, on a essayé en construisant notre architecture par dessus celle fournie d'en respecter les principes, en restant simple et concis. On a factorisé au mieux le code quand c'était nécessaire, et on a rajouté des classes utilitaires qui ont grandement contribué à la simplicité finale du code.

De plus, le code final est facilement extensible, pour ajouter plus de fonctionnalités au langage déca. Cela demande quand même à ce que les contraintes décrites de la génération du code soient respecté, mais on s'en est assuré en mettant des exception à la compilation si elles ne le sont pas.