

# Documentation de Conception

Projet Génie Logiciel — ENSIMAG

**Auteur :** Hamza Mountassir

**Équipe :** Groupe 8, équipe 43

**Members :**

Fatima Azzahra Ardan,

Mohammed EL ARABI,

Faical EL GOUIJ,

Hamza MOUNTASSIR,

Ayoub TOUATI

**Tuteur :** M. NICOLLIN Xavier

**Date :** 21 Janvier 2026

# 1 Introduction

Ce document constitue la documentation de conception du projet de Génie Logiciel. Il s'adresse à toute personne souhaitant maintenir et/ou faire évoluer le compilateur **decac** implémenté dans le cadre de ce projet.

Le compilateur est tout d'abord présenté de manière globale. Sont ensuite détaillées l'architecture générale du projet ainsi que les principaux choix de conception. Enfin, les algorithmes et les structures de données utilisés lors de l'implémentation du compilateur sont décrits.

Ce document n'a pas pour objectif de présenter des listings de code ni d'entrer dans les détails fins de l'implémentation. Il vise plutôt à fournir une aide à toute personne souhaitant comprendre l'organisation générale du compilateur et en poursuivre le développement.

## 2 Vue d'ensemble du compilateur

Le compilateur **decac** prend en charge les différents sous-ensembles du langage Deca, lequel est lui-même un sous-langage de Java.

Son implémentation est décomposée en trois étapes principales :

1. Analyse syntaxique ;
2. Analyse contextuelle ;
3. Génération de code et exécution de l'assembleur généré.

Les programmes appartenant au sous-ensemble *hello-world* sont supportés. Ils incluent les constructions élémentaires du langage ainsi que la génération et l'exécution du code associé sur la machine abstraite **IMA**.

Le sous-ensemble *sans objet* est également implémenté. Il couvre notamment les types primitifs, les expressions arithmétiques et booléennes, les structures de contrôle, ainsi que les déclarations et affectations de variables, conformément aux spécifications du langage **Deca** pour ce niveau. Ces fonctionnalités ont été testées et validées dans le cadre du projet.

Enfin, le compilateur prend en charge le langage complet avec objets. Les mécanismes de définition de classes, d'héritage, de méthodes et d'accès aux champs sont implémentés. Les fonctionnalités liées au typage dynamique, telles que les opérations de transtypage (*cast*) et les tests de type (*instanceof*), sont également disponibles. Leur bon fonctionnement est garanti pour les cas couverts par la campagne de tests.

Ainsi, le compilateur essentiel comme le compilateur complet permettent la compilation et l'exécution des programmes relevant des sous-ensembles du langage cités ci-dessus.

## 3 Architecture du projet

### 3.1 Organisation en packages

L'implémentation du compilateur **decac** repose principalement sur deux répertoires contenant les classes Java :

- **src/main/java**, qui regroupe le code de l'implémentation du compilateur ;
- **src/test/java**, qui contient les tests unitaires associés.

Ces deux répertoires sont organisés en plusieurs packages, décrits ci-dessous.

### 3.1.1 Packages de `src/main/java`

- `fr.ensimag.deca.tree` : contient les classes représentant les nœuds de l'arbre syntaxique abstrait (AST).
- `fr.ensimag.deca.syntax` : regroupe les classes nécessaires à l'analyse lexicale et syntaxique du langage.
- `fr.ensimag.deca.context` : contient les classes dédiées à l'analyse contextuelle et à la vérification des règles sémantiques.
- `fr.ensimag.deca.codegen` : regroupe les classes responsables de la génération du code cible.
- `fr.ensimag.deca.optim` : contient les classes liées à l'extension **OPTIM** du compilateur.
- `fr.ensimag.deca` : regroupe les classes centrales assurant le fonctionnement global du compilateur.
- `fr.ensimag.ima.pseudocode` : contient les classes représentant le pseudo-code de la machine abstraite **IMA**.
- `fr.ensimag.ima.pseudocode.instructions` : regroupe les classes correspondant aux différentes instructions **IMA**.

### 3.1.2 Packages de `src/test/java`

- `fr.ensimag.deca.syntax` : tests unitaires JUnit pour l'analyse lexicale et syntaxique.
- `fr.ensimag.deca.context` : tests unitaires dédiés à l'analyse contextuelle.
- `fr.ensimag.deca.tools` : tests unitaires relatifs à la génération de code.
- `fr.ensimag.deca.tree` : autres tests unitaires liés aux structures de l'AST.

## 3.2 Dépendances des classes

Pour le répertoire `src/main/java` :

Les dépendances des classes du package `fr.ensimag.deca.tree` sont liées aux relations de dérivation définies dans la grammaire abstraite. Si un élément du vocabulaire dérive d'un autre, alors la classe qui le représente hérite de la classe correspondant à cet élément. Les non-terminaux sont des classes abstraites, tandis que les terminaux sont des classes concrètes. Les constructeurs de ces classes sont utilisés au sein du **DecaParser** afin de construire l'arbre abstrait lors de l'analyse syntaxique.

Pour le package `fr.ensimag.deca.codegen`, il n'existe pas de relations d'héritage. Les instances de ces classes sont utilisées dans les classes du package `fr.ensimag.deca.tree` lors de la génération du code assembleur.

Pour le package `fr.ensimag.deca.context`, les classes peuvent être réparties en trois catégories : les sous-classes (directes ou indirectes) de la classe **Type**, les sous-classes de **Definition**, et celles qui n'héritent d'aucune classe particulière. Les relations d'héritage sont définies conformément aux spécifications du polycopié. Les instances de ces classes sont utilisées pour décorer l'AST et effectuer l'analyse contextuelle.

Pour le package `fr.ensimag.deca.syntax`, on trouve principalement des classes modélisant les erreurs levées lors de l'analyse lexicoo-syntaxique, comme **IncludeFileNotFoundException**. Les deux seules classes ne

représentant pas des erreurs sont **AbstractDecaLexer** et **AbstractDecaParser**, classes mères du lexer et du parser générés lors de l'étape A.

Pour le package `fr.ensimag.deca.optim`, les classes n'ont pas de relations d'héritage entre elles. Elles correspondent à différentes méthodes d'optimisation utilisées lors de la génération de code.

Les packages `fr.ensimag.ima.pseudocode` et `fr.ensimag.ima.pseudocode.instructions` contiennent respectivement les classes facilitant la construction du pseudo-code et celles représentant les instructions assembleur.

Les classes des packages de `src/test/java` ne présentent pas de dépendances structurelles : elles correspondent exclusivement à des tests unitaires JUnit.

## 4 Choix de conception

### 4.1 Conventions adoptées

Nous avons décidé de respecter les conventions de style de codage Java (par exemple : utilisation de verbes pour les noms de méthodes, majuscule initiale pour les noms de classes, etc.). De plus, nous respectons les principes d'encapsulation (attributs `private` ou `protected` si nécessaire pour l'héritage) ainsi que le polymorphisme.

### 4.2 Décisions de conception

Les décisions de conception concernent principalement l'étape C, à savoir la génération de code.

Nous avons choisi de créer plusieurs classes dédiées afin de factoriser le code et de faciliter la détection et la gestion des erreurs.

Une classe `RegManager` a été mise en place. Une instance de cette classe est stockée comme champ du compilateur `decac`. Elle permet de gérer l'allocation et la libération des registres, et de connaître à tout instant les registres disponibles. Des méthodes telles que `activateMethod` et `getFreeReg` permettent respectivement d'indiquer que l'on se trouve dans la génération de code d'une méthode et de récupérer un registre libre. Lors de la génération du code d'une méthode, les registres sont utilisés à partir de **R2** et sont sauvegardés/restaurés au début et à la fin de la méthode.

De manière similaire, une classe `StackManager` a été créée afin de gérer les opérations de *push* et *pop* sur la pile.

En complément, deux classes auxiliaires, `BinaryArithOpHelper` et `CompareBoolHelper`, ont été implémentées afin de mieux structurer la gestion des opérations arithmétiques et booléennes, notamment pour le choix des registres de chargement des opérandes.

Enfin, un `ErrorManager` a été implémenté. Il contient un type énuméré `ErrorType` représentant les différentes erreurs d'exécution, ainsi qu'une classe `ImaNormalizeAndCompare` qui sert à normaliser les float, afin de faciliter leur comparaison avec le contenu des fichiers `.expected`.

### 4.3 Justification des choix de conception

Ces classes ont été introduites afin d'externaliser la gestion des registres et de la pile en dehors du package `fr.ensimag.deca.tree`. Ce choix permet un meilleur contrôle global des ressources, notamment le suivi précis du nombre de *push* et de *pop* via le `StackManager`, ce qui est essentiel pour la détection des débordements de pile.

Les autres classes ont pour objectif principal de factoriser un code initialement conséquent et d'en faciliter la maintenance et le débogage.

## 5 Algorithmes et structures de données

### 5.1 Structures de données

Dans le `RegManager`, les registres **R2** à **R15** (et **R1** à **R15** dans le cas du code d'une méthode) sont modélisés à l'aide d'une `ArrayDeque`, utilisée comme une structure **LIFO**. Les registres sont empilés dans un ordre décroissant (les registres **R1** ou **R2** se trouvent en tête de pile). Cela permet de récupérer un registre libre via `removeFirst` et de le libérer via `addFirst`, garantissant une complexité en temps de  $O(1)$ .

Afin de déterminer les registres à sauvegarder et à restaurer lors de la génération du code des méthodes, un `TreeSet` est utilisé. Ce choix garantit l'absence de doublons. La méthode `compareTo` a été implémentée pour la classe `GPRegister`.

Pour la construction de la table des méthodes, seules les structures de données standards de Java (listes) sont utilisées.

### 5.2 Algorithmes utilisés

Le parser repose sur une logique récursive, conformément aux indications du polycopié. Les principaux algorithmes utilisés concernent l'optimisation du code : **Peephole Optimization** (dans une version étendue analysant l'ensemble du code), **SSA-like optimizations** telles que la propagation de constantes, la propagation de copies, l'élimination des écritures mortes (*Dead Store Elimination*), l'élimination de code mort (*Dead Code Elimination*) et le **Constant Folding**.

## 6 Conclusion

Ce document a présenté la conception générale du compilateur `decac`, en détaillant son architecture, ses principaux choix de conception ainsi que les structures de données et algorithmes employés. Les décisions prises visent à garantir une implémentation modulaire, maintenable et conforme aux principes de la programmation orientée objet. Cette documentation constitue une base solide pour la maintenance et l'évolution future du compilateur, notamment dans le cadre de l'ajout de nouvelles optimisations ou de fonctionnalités du langage Deca.