Documentation de Conception
Compilateur Deca

Roboam Guillaume Raballand Cyprien Talbi El Mehdi
Charles-Mennier Matéo Altieri Aubin

GL1, GL01, Équipe 1
17 janvier 2025

Contents

1 Introduction

Ce document présente la documentation de conception du projet de compilateur Deca. Il offre une vue d'ensemble de l'architecture, des composants clés, des interfaces, ainsi que des choix techniques majeurs réalisés au cours du développement. L'objectif principal est d'aider les développeurs à maintenir ou faire évoluer le compilateur en fournissant des informations supplémentaires par rapport à celles initialement fournies.

2 Architecture du Code

2.1 Arbre Abstrait

L'architecture repose sur une hiérarchie de classes représentant l'arbre abstrait. Chaque nœud de l'arbre correspond à une classe spécifique, et ces classes sont organisées selon les concepts du langage Deca. L'arbre abstrait est conçu pour refléter fidèlement la grammaire et servir de base aux phases suivantes du compilateur.

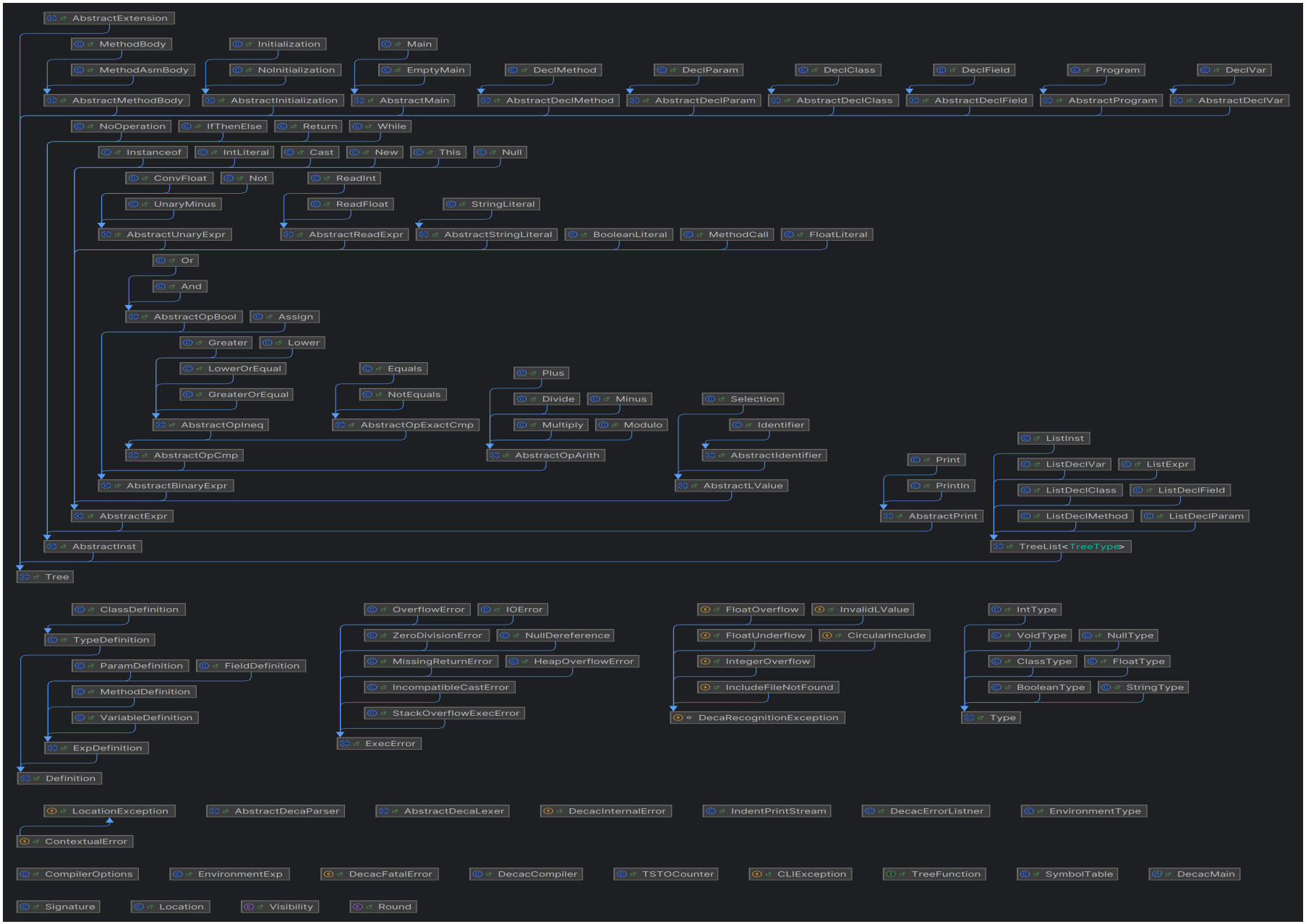

Figure 1: Diagramme de relations entre les fichiers du dossier deca

3 Spécifications Supplémentaires

3.1 Gestion des Environnements d'Expression

Pour gérer l'empilement des environnements d'expression (`EnvironmentExp`):

- Utilisation d'une `HashMap` pour représenter l'environnement courant.
- Référence au parent pour permettre une recherche hiérarchique.

Exemple de code pour la méthode `get` :

```
public ExpDefinition get(Symbol key) {
    ExpDefinition res = currentEnvironment.get(key);
    if (res == null && parentEnvironment != null) {
        return parentEnvironment.get(key);
    }
    return res;
}
```

3.2 Gestion des Registres

- **Registre R0** : Utilisé pour des calculs temporaires sans sauvegarde nécessaire.
- **Autres registres** : Débutent à R2 pour les calculs complexes, augmentant dans l'ordre croissant.
- Problèmes potentiels : Risques d'erreurs lorsque des registres sont mal attribués.

3.3 Classes Définies

- `ClasseDefinition` :
  - Ajout d'un tableau `Label[]` représentant la table des méthodes.
  - Champ `operand` contenant l'adresse de la classe (début de la table des méthodes).

3.4 Méthodes Clés

- `getDVal` : Associe une `DVal` à chaque expression (`null` pour binaires, littéraux pour `Literal`, etc.).
- `codeExp` : Calcule la valeur d'une expression dans un registre donné.
- `codeGenBool` : Génère le code pour le contrôle de flux ($\langle \text{Code}(C, \text{vrai}, E) \rangle$).

3.5 Gestion des "Not Yet Implemented"

Les méthodes non implémentées sont définies au sommet de la hiérarchie, puis override dans les sous-classes. Cela déclenche une exception si une méthode requise n'est pas override.

3.6 Gestion des "Missing Return"

Pour gérer les `return` manquants :

- Ajout d'un argument contenant le label de la méthode.
- Alternative envisagée : Placer ce label dans un champ global au compilateur, mais cela a introduit des problèmes de génération de code.

3.7 Ordre de Génération du Code

1. Table des méthodes.
2. Programme principal.
3. Initialisation des objets.
4. Code des méthodes.
5. Gestion des erreurs d'exécution.

3.8 Gestion des Caractères Spéciaux

- Support pour `\n`, `\r`, `\t` dans les `print`, extensible aux caractères UTF-8 mais pas implémenté.
- Ajout de Pseudo-Instructions : `RUTF8` et `WUTF8` pour gérer les caractères UTF-8.

4 Structures de Données Utilisées

- `StackOverflowCounter` : Suivi de l'utilisation maximale de la pile.
- `ExecError` : Classe abstraite pour gérer les erreurs d'exécution. Exemple :

```
public class HeapOverflowError extends ExecError {
    public static final HeapOverflowError INSTANCE = new
       HeapOverflowError();

    public HeapOverflowError() {
        super(new Label("heap_overflow_error"), "Error:␣Heap␣
           Overflow");
    }
}
```