Projet Génie Logiciel

Document de Conception

Réalisé par :
Mouez JAAFOURA, Ilyas MIDOU, Meriem KAZDAGHLI,
Salim KACIMI, Mehdi DIMASSI

Date : 23janvier 2025

Grenoble INP - Ensimag & Phelma
Université Grenoble Alpes

Table des matières

1 Introduction

Le projet Génie Logiciel consiste à développer en Java un compilateur pour le langage Deca, un langage minimaliste inspiré de Java. Ce projet s'inscrit dans un cadre pédagogique visant à approfondir la compréhension des langages de programmation et à améliorer les compétences en conception et développement logiciel.

Le projet est divisé en plusieurs étapes correspondant aux phases classiques de compilation :

— Analyse lexicale et syntaxique.
— Analyse contextuelle.
— Traduction en code machine.

Cette documentation de conception s'adresse à tout développeur souhaitant maintenir, corriger ou faire évoluer le compilateur Deca. Elle offre une vue d'ensemble claire de l'architecture et des choix de conception qui ont guidé le développement, afin de faciliter la compréhension du fonctionnement interne du compilateur et de permettre des évolutions futures.

Le document présente :

— Une description générale de l'architecture logicielle, incluant les principales classes et leurs interactions.
— Les structures de données et algorithmes clés employés dans le compilateur.
— Les spécifications supplémentaires mises en œuvre au-delà des exigences initiales.

2 Organisation générale de l'architecture

2.1 Organisation des Composants

Grammaire ANTLR

— Définit les règles syntaxiques de Deca dans le fichier `DecaParser.g4`.
— Génère des classes Java pour l'analyse syntaxique, permettant de construire l'arbre syntaxique abstrait (AST).

Classes pour l'AST

— Situées dans `src/main/java/fr/ensimag/deca/tree/`.
— Implémentent des nœuds spécifiques pour représenter les structures du programme (instructions, expressions, déclarations).

Gestion des Erreurs

— Des exceptions personnalisées sont utilisées pour signaler les erreurs syntaxiques (InvalidLValue, CircularInclude, etc.).

Diagramme de classes : UML voir annexe1

2.2 Classes associées

La classe `Tree` est la classe de base pour tous les nœuds de l'AST. Elle définit les comportements génériques que tous les nœuds doivent implémenter ou utiliser.

— `void verify(DecaCompiler compiler)` : Vérifie la validité sémantique du nœud.
— `void codeGen(DecaCompiler compiler)` : Génère le code machine correspondant au nœud.
— `String decompile()` : Reconstruit une représentation textuelle (source) à partir de l'arbre.

Les classes dérivées de `AbstractOpArith` modélisent les opérations arithmétiques binaires dans l'AST du langage Deca. Ces classes représentent des nœuds de l'arbre syntaxique abstrait pour des opérations telles que :

- **Addition (`Plus`)** : Représente l'opération d'addition entre deux expressions.
- **Soustraction (`Minus`)** : Représente l'opération de soustraction entre deux expressions.
- **Multiplication (`Multiply`)** : Représente l'opération de multiplication entre deux expressions.
- **Division (`Divide`)** : Représente l'opération de division entre deux expressions.
- **Modulo (`Modulo`)** : Représente l'opération de calcul du reste de la division entre deux expressions.

Classe `AbstractOpArith`

Cette classe abstraite est la base commune pour toutes les opérations arithmétiques binaires. Elle définit les éléments nécessaires pour gérer les calculs arithmétiques de manière générique et extensible.

Caractéristiques principales :

- **Constructeur** : Permet d'initialiser les deux opérandes (`leftOperand` et `rightOperand`) nécessaires à l'opération.
- `void verifyExpr(DecaCompiler compiler, EnvironmnetExp localEnv, ClassDefinition currentClass)` : Vérifie la validité des types des opérandes et effectue les conversions nécessaires pour garantir la conformité avec les règles du langage Deca.

Les classes `AbstractOpBool`, `And`, et `Or` représentent les opérateurs logiques binaires dans le langage Deca. Elles permettent de modéliser les expressions logiques comme `&&` (ET logique) et `||` (OU logique) dans l'arbre syntaxique abstrait (AST).

Ces classes s'intègrent dans la hiérarchie des expressions binaires, héritant de `AbstractBinaryExpr` .

Classe `AbstractBinaryExpr`

La classe abstraite AbstractBinaryExpr représente une expression binaire dans l'AST (Abstract Syntax Tree) du langage Deca. Elle sert de classe de base pour toutes les opérations binaires, telles que les additions, les comparaisons, et les opérations logiques. **Méthodes principales :**

- **Accès aux opérandes :**
  - `getLeftOperand` : Retourne l'opérande gauche.
  - `getRightOperand` : Retourne l'opérande droit.
  - `setLeftOperand` : Définit l'opérande gauche. Vérifie que l'opérande n'est pas `null`.
  - `setRightOperand` : Définit l'opérande droit. Vérifie que l'opérande n'est pas `null`.
- **Gestion de l'arbre syntaxique :**
  - `iterChildren` : Parcourt les nœuds enfants (opérandes gauche et droit).
  - `prettyPrintChildren` : Fournit une représentation lisible des nœuds enfants pour le débogage.
- **Affichage et décompilation :**
  - `decompile` :
    - Génère une représentation textuelle de l'expression binaire, incluant les opérandes et l'opérateur.
    - Produit un format lisible, par exemple : `(opérande_gauche opérateur opérande_droit)`.
  - `getOperatorName` (abstraite) : Retourne le symbole de l'opérateur (à définir dans les sous-classes).

Classe `AbstractReadExpr`

Représente les expressions `readInt()` et `readFloat()` du langage Deca. Ces expressions permettent de lire des valeurs à partir de l'entrée standard, en les interprétant respectivement comme des entiers ou des flottants.

Rôle principal :

- Fournir une base commune pour les deux types d'expressions de lecture.
- Simplifier l'implémentation en factorisant les comportements communs.

La classe ReadFloat

La classe `ReadFloat` représente l'expression `readFloat()` dans le langage Deca. Elle permet de lire une valeur flottante à partir de l'entrée standard et de la stocker dans une variable. Cette classe hérite de `AbstractReadExpr` et implémente les méthodes nécessaires pour :

- Vérifier l'expression (`verify()`).
- Décompiler l'expression en texte source (`decompile(IndentPrintStream s)`).
- Générer le code machine correspondant (`codeGen()`).
- Gérer les enfants dans l'arbre syntaxique abstrait (`getChildren()`).

La classe ReadInt

La classe `ReadInt` représente l'expression `readInt()` dans le langage Deca. Elle permet de lire une valeur entière à partir de l'entrée standard et de la stocker dans une variable.

Points importants :

- Hérite de `AbstractReadExpr`.
- Fournit les mêmes fonctionnalités que `ReadFloat`, adaptées à la lecture des entiers.

La classe AbstractOpCmp

La classe abstraite AbstractOpCmp représente les opérateurs de comparaison dans l'AST (Abstract Syntax Tree) du langage Deca. Elle étend la classe AbstractBinaryExpr pour gérer les expressions binaires et est utilisée comme base pour implémenter des opérateurs tels que $>=$, $<=$, $>$, $<$.

La méthode `verifyExpr` joue un rôle clé dans la gestion des expressions de comparaison en assurant la validité contextuelle et la compatibilité des types.

Fonctionnalités principales :

- Vérifie que les types des opérandes sont compatibles avec l'opération de comparaison.
- Implémente une conversion implicite de `int` vers `float` lorsque nécessaire.
- Retourne toujours un type `boolean` pour les expressions de comparaison valides.
- Gère les erreurs contextuelles avec des messages explicites si les opérandes ne sont pas compatibles.

Gestion des types avec ConvFloat : Si un des opérandes est de type int et l'autre de type float, la classe construit un nœud de conversion (ConvFloat) pour aligner les types et garantir la cohérence des calculs.

La classe AbstractOpExactCmp

La classe abstraite AbstractOpExactCmp est une spécialisation de la classe AbstractOpCmp. Elle représente les opérateurs de comparaison exacte dans l'AST (Abstract Syntax Tree) du langage Deca, tels que $==$ (égalité) et $!=$ (différence).

La classe AbstractOpIneq

La classe abstraite AbstractOpIneq hérite de la classe AbstractOpCmp. Elle représente les opérateurs d'inégalité dans l'AST (Abstract Syntax Tree) du langage Deca, tels que <, >, <=, et >=.

La classe AbstractInst

La classe abstraite AbstractInst représente une instruction générique dans l'AST (Abstract Syntax Tree) du langage Deca. Elle fournit les bases nécessaires pour implémenter des instructions spécifiques dans le compilateur, qu'il s'agisse d'instructions de contrôle de flux, d'affectations, ou d'appels de méthode.

Fonctionnalités principales : Définition des opérations communes pour toutes les instructions, notamment :

- Vérification contextuelle (verifyInst) des instructions selon les règles de la grammaire et les types.
- Génération de code assembleur (codeGenInst) correspondant à l'instruction.
- Décompilation pour produire une version textuelle lisible de l'instruction.

Méthodes principales

`verifyInst :` Implémente le non-terminal `inst` dans la passe 3 de l'analyse contextuelle. Vérifie la validité contextuelle de l'instruction en utilisant :

- `env_types` : pour l'environnement global des types.
- `env_exp` : pour l'environnement des variables locales.
- `class` : pour la classe actuelle (ou `null` si l'instruction est dans le bloc principal).
- `return` : pour le type de retour (ou `void` dans le bloc principal).

Lance une exception `ContextualError` en cas de non-conformité.

`codeGenInst :` Génère le code assembleur correspondant à l'instruction. Utilise les fonctionnalités du compilateur (`DecacCompiler`) pour produire le code machine. Lance une exception `CLIException` en cas d'erreur critique.

`decompileInst :` Produit une représentation textuelle de l'instruction. Ajoute des détails spécifiques (comme des points-virgules) si nécessaire pour respecter la syntaxe du langage source.

La classe IfThenElse

La classe IfThenElse hérite de AbstarctInst et représente une instruction conditionnelle complète dans l'AST (Abstract Syntax Tree) du langage Deca. Elle gère les blocs if, else if, et else.

Méthodes principales :

- `verifyInst` :
  - Vérifie que la condition est bien de type `boolean`.
  - Valide contextuellement les branches `then` et `else`.
  - Lance une exception `ContextualError` en cas d'erreur.
- `codeGenInst` :
  - Génère le code assembleur correspondant à la structure conditionnelle.
  - Utilise des étiquettes (`Label`) pour gérer les sauts entre les branches.
- `decompile` :
  - Reconstruit une représentation textuelle lisible de l'instruction.
- `iterChildren` et `prettyPrintChildren` :

— Parcourent les enfants (nœuds) de l'AST pour effectuer des traitements spécifiques.

La classe While

La classe `While`hérite de AbstarctInst et représente une structure de contrôle de boucle dans l'AST (Abstract Syntax Tree) du langage Deca. Elle modélise une boucle conditionnelle de la forme `while(condition) { body }`, où le corps est exécuté tant que la condition est vraie.
Méthodes principales :

- `verifyInst` :
  - — Vérifie que la condition est bien de type `boolean`.
  - — Valide la liste des instructions dans le corps de la boucle.
  - — Lance une exception `ContextualError` en cas de non-conformité.
- `codeGenInst` :
  - — Génère le code assembleur correspondant à la structure de boucle.
  - — Ajoute des étiquettes pour l'entrée de la condition et le début du corps de la boucle.
  - — Utilise des instructions de saut (`BRA`) pour gérer les itérations.
- `decompile` :
  - — Produit une représentation textuelle lisible de l'instruction `while`.
- `iterChildren` et `prettyPrintChildren` :
  - — Parcourent et affichent les enfants (nœuds) de l'AST pour effectuer des traitements spécifiques.

La classe AbstractPrint

La classe abstraite `AbstractPrint`hérite de AbstarctInst et représente les instructions de type `print` dans le langage Deca. Ces instructions permettent d'afficher des valeurs sur la sortie standard. La classe inclut une gestion pour l'affichage en format hexadécimal (`printx`) et sert de base pour des sous-classes telles que `Print`, `Println`

- `String getSuffix()` : Retourne le suffixe utilisé dans l'instruction (`""` pour `print`, `"ln"` pour `println`, etc.).

Affichage en Hexadécimal

Si `printHex` est vrai, l'instruction inclut un `x` dans la syntaxe de la décompilation.

La classe Return

La classe Return hérite de AbstractInst représente une instruction de retour dans l'AST (Abstract Syntax Tree) du langage Deca. Elle est utilisée pour renvoyer une valeur depuis une méthode ou une fonction, en respectant les contraintes de type imposées par la signature de celle-ci.
Méthodes principales :

- `verifyInst` :
  - — Vérifie que l'instruction `return` est conforme au type de retour spécifié par la méthode ou la fonction.
  - — Lève une exception `ContextualError` si :
    - — `return` est utilisé dans une fonction de type `void`.
    - — Le type de l'expression à retourner n'est pas compatible avec le type attendu.
  - — Convertit automatiquement un type `int` en `float` si nécessaire.
- `codeGenInst` :
  - — Génère le code assembleur pour l'instruction `return`.
  - — Évalue l'expression à retourner, la stocke dans le registre `R0`, restaure les registres, et insère l'instruction `RTS`.

- `decompile` :
  - Produit une représentation textuelle de l'instruction `return`.
- `iterChildren` et `prettyPrintChildren` :
  - Parcourent et affichent les enfants (nœuds) de l'AST pour effectuer des traitements spécifiques.

La classe NoOperation
La classe NoOperation hérite de AbstractInst représente une instruction vide ou une opération sans effet dans l'AST (Abstract Syntax Tree) du langage Deca **Méthodes principales :**

- `verifyInst` :
  - Implémentée pour respecter l'interface d'une instruction, mais aucune vérification contextuelle n'est requise.
- `codeGenInst` :
  - Implémentée pour respecter l'interface d'une instruction, mais elle ne génère aucune instruction en code machine.
- `decompile` :
  - Produit une représentation textuelle de l'instruction vide, qui correspond à un point-virgule (`;`).
- `iterChildren` et `prettyPrintChildren` :
  - Ces méthodes ne font rien, car `NoOperation` est un nœud feuille dans l'AST et n'a pas de sous-nœuds.

La classe AbstractExpr
AbstractExpr représente une expression dans l'AST (Abstract Syntax Tree) du langage Deca. Elle étend AbstractInst et sert de base pour implémenter des types d'expressions spécifiques. Une expression est tout élément ayant une valeur, comme une opération arithmétique, un littéral, ou une condition. Méthodes principales

`verifyInst :`

- Vérifie que l'expression respecte les règles contextuelles et renvoie son type.
- Implémente les non-terminaux expr et lvalue dans la passe 3 de l'analyse contextuelle.
- Utilise les environnements locaux et globaux pour valider les types et la portée.

`verifyCondition :`

- Vérifie qu'une expression utilisée comme condition est de type boolean.
- Lève une exception ContextualError si le type est incorrect.

`codeGenPrint :` Génère le code machine pour afficher l'expression. Gère les types (int, float, boolean) et l'option d'affichage en hexadécimal pour les flottants.

`decompileInst :` Produit une représentation textuelle de l'expression, avec un point-virgule si nécessaire.

`codeGenInst :` Génère le code pour évaluer l'expression, avec des étiquettes pour les conditions.
La classe TypeCasting
La classe TypeCasting héritant de AbstractExpr représente une opération de transtypage (cast) dans l'AST (Abstract Syntax Tree) du langage Deca. **Méthodes principales :**

- `verifyExpr` :
  - Vérifie que la conversion entre le type de la variable et le type cible (`newType`) est valide.
  - S'assure que les types impliqués sont compatibles ou que le transtypage est autorisé.
  - Lance une exception `ContextualError` en cas d'incompatibilité des types.
- `decompile` :
  - Produit une représentation textuelle de l'opération de transtypage, par exemple `(float) x`.
- `codeGenInst` :
  - Génère le code assembleur pour effectuer la conversion.
  - Gère les cas spécifiques :
    - Conversion d'un flottant (`float`) en entier (`int`) avec arrondi.
    - Conversion d'un entier (`int`) en flottant (`float`).
  - Utilise des étiquettes (`Label`) pour gérer l'arrondi des flottants.
- `iterChildren` et `prettyPrintChildren` :
  - Parcourent et affichent les nœuds enfants, à savoir `newType` et `variable`.

La classe New
La classe New héritant de AbstractExpr représente une expression de création d'objet ou de tableau dans le langage Deca. Elle modélise les instructions de la forme Class var = new Class() ;
Méthodes principales :

- `verifyExpr` :
  - Vérifie que l'identifiant correspond à une classe ou à un tableau défini dans l'environnement global des types (`EnvironmentType`).
  - Associe le type à l'expression si la vérification réussit.
  - Lance une exception `ContextualError` si l'identifiant n'est pas valide.
- `codeGenInst` :
  - Génère le code assembleur pour allouer de la mémoire pour l'objet ou le tableau.
  - Initialise la table des méthodes virtuelles (VTable) pour les objets.
  - Gère les dépassements de mémoire avec l'instruction `BOV`.
- `decompile` :
  - Produit une représentation textuelle de l'expression `new`, par exemple : `new Class()`.
- `prettyPrintNode` et `prettyPrintChildren` :
  - Fournissent une représentation lisible de l'expression pour le débogage ou l'affichage.
- `iterChildren` :
  - Parcourt les enfants de l'arbre (dans ce cas, l'identifiant).

La classe TableAllocation
La classe TableAllocation représente une allocation de tableau multidimensionnel dans le langage Deca. **Méthodes principales :**

- `verifyExpr` :

  - Vérifie que chaque dimension est de type entier (`int`).
  - Valide l'existence du type dans l'environnement global des types (`EnvironmentType`).
  - Crée un type de tableau correspondant aux dimensions spécifiées.
  - Associe le type du tableau à l'expression.
  - Lance une exception `ContextualError` si une vérification échoue.
- `getTailleExpr` et `setTailleExpr` :
  - Récupèrent ou définissent la taille (dimensions) du tableau.
- `codeGenInst` :
  - Génère le code pour allouer de la mémoire pour le tableau.
  - Initialise les éléments du tableau avec des valeurs par défaut en fonction de leur type :
    - `0` pour les entiers et booléens.
    - `0.0` pour les flottants.
    - `null` pour les références.
  - Gère les dépassements de mémoire avec l'instruction `BOV`.
- `decompile` :
  - Produit une représentation textuelle de l'instruction d'allocation (non implémentée).
- `prettyPrintNode` et `prettyPrintChildren` :
  - Fournissent une représentation lisible de l'expression pour le débogage ou l'affichage.

La classe AbstractSelectionMethodCall
La classe abstraite AbstractSelectionMethodCall représente un appel de méthode avec une sélection d'objet dans l'AST (Abstract Syntax Tree) du langage Deca. Elle permet d'accéder à une méthode ou un membre défini dans une autre classe à travers un objet. **Méthodes principales :**

- `getRightIdentifier` :
  - Retourne l'identifiant correspondant à la méthode ou au membre sélectionné.
- `getLeftExpr` :
  - Retourne l'expression représentant l'objet à partir duquel la méthode est appelée.
- `getTailleExpr` :
  - Retourne la taille associée à la définition de la méthode ou du membre sélectionné.
  - Vérifie le type de l'expression gauche (`leftExpr`) et s'assure qu'il correspond bien à une classe valide.
  - Utilise l'environnement local de la classe pour récupérer la définition du membre appelé.
- `setTailleExpr` :
  - Met à jour la taille associée à la méthode ou au membre sélectionné.
  - Suit une logique similaire à `getTailleExpr` pour localiser la définition, puis modifie la taille.
- `isSelectionCall` :
  - Indique que cette classe représente un appel de méthode via sélection (`true`).

La classe Selection
La classe Selection hérite de AbstractSelectionMethodCall représente une opération de sélection d'attribut ou de méthode sur un objet dans l'AST (Abstract Syntax Tree) du langage Deca
Méthodes principales :

- `verifyExpr` :
  - Vérifie que l'expression de gauche (`leftExpr`) est un objet valide et que l'attribut ou méthode existe dans la classe correspondante.
  - Assure que les attributs protégés sont accessibles uniquement dans les sous-classes appropriées.
  - Retourne le type de l'attribut ou méthode sélectionné.
  - Lance une exception `ContextualError` si l'attribut ou méthode est introuvable ou inaccessible.
- `codeGenInst` :
  - Génère le code assembleur pour accéder à l'attribut sélectionné.
  - Utilise un décalage d'enregistrement (`RegisterOffset`) pour accéder à l'attribut via son index.
- `codeGenCond` :
  - Génère le code conditionnel pour une sélection booléenne.
  - Compare la valeur de l'attribut sélectionné avec 0 et saute à l'étiquette appropriée selon la condition.
- `decompile` :
  - Produit une représentation textuelle de l'opération de sélection, par exemple `obj.attribut`.
- `prettyPrintChildren` :
  - Gère l'affichage lisible des nœuds enfants, y compris `leftExpr` et `rightIdent`.

La classe Assign

La classe Assign héritant de AbstractBinaryExpr représente une affectation dans l'AST (Abstract Syntax Tree) du langage Deca. Elle modélise les instructions de la forme lvalue = expr, où une valeur (expr) est assignée à une variable ou une structure (lvalue). **Méthodes principales :**

- `verifyExpr` :
  - Vérifie que les types des opérandes gauche et droite sont compatibles.
  - Gère les conversions implicites (par exemple, conversion de `int` en `float`).
  - Valide les tailles des listes dans le cas d'affectation à une structure de type tableau ou liste.
  - Lance une exception `ContextualError` en cas de types incompatibles ou de tailles incorrectes.
- `codeGenInst` :
  - Génère le code assembleur pour l'affectation.
  - Gère spécifiquement :
    - Les tableaux (`TypeTable`), en copiant chaque élément.
    - Les attributs d'objets (`Selection`).
    - Les listes d'éléments (`ListElement`).
  - Utilise des instructions `LOAD` et `STORE`.
- `codeGenCond` :
  - Produit une valeur booléenne (`0` ou `1`) pour les conditions.
  - Ajoute des instructions conditionnelles pour gérer les branchements (`BEQ`, `BNE`).

La classe AbstractIdentifier

La classe abstraite AbstractIdentifier représente un identifiant dans l'AST (Abstract Syntax Tree) du langage Deca. Elle est utilisée pour identifier des variables, des méthodes, des champs, ou des types dans un programme Deca **Méthodes principales :**

- **Accès à la définition :**
  - `getDefinition` : Retourne la définition générale associée à l'identifiant.
  - Méthodes spécialisées pour des types spécifiques de définitions :
    - `getClassDefinition`
    - `getFieldDefinition`
    - `getMethodDefinition`
    - `getVariableDefinition`
    - `getExpDefinition`
  - Ces méthodes effectuent un transtypage explicite et lèvent une exception si la définition n'est pas du type attendu.
- **Validation contextuelle :**
  - `verifyType` : Vérifie le type associé à l'identifiant dans l'environnement global des types (`env_types`).
  - Lance une exception `ContextualError` si le type n'est pas valide.
- **Gestion des tailles :**
  - `getTailleExpr` et `setTailleExpr` : Récupèrent ou mettent à jour la taille associée à une définition, comme une liste ou un tableau.
- **Gestion des types dynamiques :**
  - `setDynamicType` : Définit le type dynamique d'un identifiant dans un environnement donné.
  - `getDynamicType` : Récupère le type dynamique.

La classe Identifier
Elle hérite de AbstractIdentifier et implémente la gestion spécifique des identifiants, tels que les variables, les champs, les méthodes, ou les types. **Méthodes principales :**

- **Validation et récupération des définitions :**
  - `getDefinition` : Retourne la définition générale associée à l'identifiant.
  - Méthodes spécialisées pour les types spécifiques :
    - `getClassDefinition`
    - `getFieldDefinition`
    - `getMethodDefinition`
    - `getVariableDefinition`
    - `getExpDefinition`
  - `setDefinition` : Associe une définition à l'identifiant.
- **Validation contextuelle :**
  - `verifyExpr` :
    - Vérifie que l'identifiant est défini dans l'environnement local.
    - Associe la définition et le type à l'identifiant.
    - Lance une exception `ContextualError` si l'identifiant n'est pas trouvé.
  - `verifyType` :
    - Vérifie que l'identifiant est un type valide dans l'environnement global.
    - Associe la définition et le type à l'identifiant.
    - Lance une exception `ContextualError` si le type est introuvable.
- **Génération de code :**

- `codeGenInst` : Génère le code pour charger la valeur associée à l'identifiant.
- `codeGenCond` : Génère le code conditionnel pour comparer la valeur de l'identifiant.

– **Affichage et parcours :**

- `decompile` : Produit une représentation textuelle de l'identifiant.
- `prettyPrintNode` : Affiche les informations de l'identifiant sous une forme lisible.

La classe AbstractDeclClass

La classe abstraite AbstractDeclClass représente une déclaration de classe dans le langage Deca.

Méthodes principales :

– **Vérifications contextuelles :**

- `verifyClass` :
  - Correspond à la première passe de la vérification contextuelle.
  - Vérifie que la déclaration de la classe est correcte sans examiner son contenu.
  - Lance une exception `ContextualError` en cas de problème.
- `verifyClassMembers` :
  - Correspond à la deuxième passe de la vérification contextuelle.
  - Vérifie que les membres de la classe (champs et méthodes) sont correctement déclarés sans analyser les corps des méthodes ou les initialisations des champs.
  - Lance une exception `ContextualError` en cas de problème.
- `verifyClassBody` :
  - Correspond à la troisième passe de la vérification contextuelle.
  - Vérifie que les instructions et expressions contenues dans la classe sont correctes.
  - Lance une exception `ContextualError` en cas de problème.

– **Génération de code :**

- `genClassTable` :
  - Génère les tables des méthodes et des champs associées à la classe.
  - Utilise un label pour la méthode `equals` de l'objet parent (souvent `Object.equals`).
- `codegenClass` :
  - Génère le code associé à la déclaration de la classe, y compris les initialisations et les définitions des méthodes.
  - Lance une exception `CLIException` en cas de problème lors de la génération.

La classe DeclClass

La classe DeclClass hérite de AbstractDeclClass et représente une déclaration de classe dans le langage Deca

Méthodes principales :

– **Vérifications contextuelles :**

- `verifyClass` :
  - Vérifie l'existence de la superclasse et initialise les informations de type pour la classe.
  - Lance une exception `ContextualError` si une vérification échoue.
- `verifyClassMembers` :
  - Vérifie les déclarations des champs et méthodes sans analyser leurs définitions complètes.

  - `verifyClassBody` :
    - Vérifie les initialisations des champs et la validité des corps des méthodes.
- **Génération de code :**
  - `genClassTable` :
    - Génère les tables des méthodes et des champs pour la classe, incluant les méthodes héritées.
  - `codegenClass` :
    - Génère le code pour l'initialisation des champs et la définition des méthodes.
- **Affichage et parcours :**
  - `decompile` : Produit une représentation textuelle de la déclaration de classe.
  - `prettyPrintChildren` et `iterChildren` : Parcourent les éléments de la classe pour afficher ou traiter ses membres.

La classe AbstractDeclVar
La classe abstraite AbstractDeclVar représente une déclaration de variable dans l'AST (Abstract Syntax Tree) du langage Deca **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyDeclVar` :
    - Correspond à la gestion du non-terminal `decl_var` dans la passe 3 de l'analyse contextuelle.
    - Vérifie la déclaration de la variable en utilisant :
      - `env_types` pour accéder aux types définis globalement.
      - `localEnv` pour accéder à l'environnement local des variables.
      - `currentClass` pour accéder à la classe courante (ou `null` dans le bloc principal).
    - Assure que la variable n'est pas déjà définie dans l'environnement courant.
    - Lance une exception `ContextualError` ou `DoubleDefException` en cas d'erreur.
- **Génération de code :**
  - `codeGenDeclVar` :
    - Génère le code d'instruction pour allouer et initialiser la variable dans l'environnement global.
  - `codeGenDeclVarLocal` :
    - Gère spécifiquement l'allocation et l'initialisation des variables locales dans un environnement donné, basé sur un index.

La classe DeclVar
La classe DeclVar hérite de AbstractDeclVar représente la déclaration d'une variable dans le langage Deca **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyDeclVar` :
    - Vérifie que le type de la variable est valide (ni `null` ni `void`).
    - Vérifie l'initialisation de la variable en fonction de son type et de sa taille.
    - Ajoute la variable à l'environnement local (`localEnv`).
    - Gère les erreurs contextuelles, comme une redéfinition de la variable.

- **Génération de code :**
  - `codeGenDeclVar` :
    - Génère le code pour déclarer et initialiser une variable globale.
    - Alloue une adresse mémoire pour la variable et y applique l'initialisation.
  - `codeGenDeclVarLocal` :
    - Génère le code pour déclarer et initialiser une variable locale dans la pile.
    - Utilise un index pour associer une adresse locale à la variable.
- **Affichage et parcours :**
  - `decompile` : Produit une représentation textuelle de la déclaration.
  - `prettyPrintChildren` et `iterChildren` : Parcourent les composants de la déclaration pour un affichage ou un traitement.

La classe AbstractInitialization
La classe abstraite AbstractInitialization représente l'initialisation de variables, champs, ou autres éléments dans le langage Deca **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyInitialization` :
    - Implémente le non-terminal `initialization` de l'analyse contextuelle à la passe 3.
    - Vérifie la validité de l'initialisation par rapport au type attendu (`t`) et à la taille (`taille`) dans un environnement donné (`localEnv`).
    - Gère le contexte des classes via l'attribut `currentClass`.
    - Lance une exception `ContextualError` si l'initialisation n'est pas valide.
- **Génération de code :**
  - `codeGenInit` :
    - Génère le code machine pour initialiser une variable ou un champ avec une adresse mémoire (`addr`).
  - `codeGenInitField` :
    - Génère le code machine spécifique pour initialiser un champ, basé sur son type (`type`).
- **Détection d'absence d'initialisation :**
  - `isNoInitialization` :
    - Retourne `true` si aucune initialisation explicite n'est fournie.
    - Peut être redéfinie par des sous-classes pour signaler une initialisation existante.

La classe Initialization
La classe Initialization représente l'initialisation explicite d'une variable ou d'un champ dans le langage Deca. Elle étend la classe abstraite AbstractInitialization pour fournir une implémentation concrète **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyInitialization` :
    - Vérifie que le type de l'expression d'initialisation est compatible avec celui de la variable ou du champ.
    - Gère les conversions implicites (par exemple, `int` vers `float`) si nécessaire.
    - Vérifie l'égalité des dimensions si la variable est de type tableau.

    - Lance une exception `ContextualError` en cas de non-conformité.
- **Génération de code :**
  - `codeGenInit` :
    - Génère le code machine pour initialiser une variable ou un champ en mémoire avec l'expression donnée.
  - `codeGenInitField` :
    - Génère le code machine pour initialiser un champ dans le contexte d'une classe.
- **Gestion des initialisations :**
  - `isNoInitialization` :
    - Retourne `false` pour indiquer qu'une initialisation explicite est présente.
- **Affichage et parcours :**
  - `decompile` : Produit une représentation textuelle de l'initialisation, par exemple : `x = 42;`.
  - `prettyPrintChildren` et `iterChildren` : Parcourent l'expression d'initialisation pour un traitement ou un affichage.

La classe NoInitialization
La classe NoInitialization représente l'absence explicite d'initialisation pour une variable ou un champ. Par exemple, dans l'instruction int x;, la variable x est déclarée sans initialisation **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyInitialization` :
    - Vérifie que le type de la variable ou du champ sans initialisation est bien défini dans l'environnement global des types.
    - Lance une exception `ContextualError` si le type n'est pas reconnu.
- **Génération de code :**
  - `codeGenInit` :
    - Implémentation vide, car il n'y a aucune action spécifique pour une variable non initialisée.
  - `codeGenInitField` :
    - Génère le code machine pour initialiser un champ avec une valeur par défaut selon son type :
      - `0` pour les types `int` et `boolean`.
      - `0.0` pour les types `float`.
      - `null` pour les objets.
- **Affichage et parcours :**
  - `decompile`, `iterChildren`, et `prettyPrintChildren` :
    - Implémentations vides, car cette classe ne contient pas de données supplémentaires à afficher ou à parcourir.

La classe DeclField
La classe DeclField représente la déclaration d'un champ (ou attribut) dans une classe en Deca. **Méthodes principales :**

- **Décompilation :**

  - `decompile` :
    - Produit une représentation textuelle (source) de la déclaration du champ.
    - Affiche le nom du champ suivi de son initialisation, si elle existe.
- **Vérification contextuelle :**
  - `verifyDeclField` :
    - Vérifie que le champ est correctement déclaré dans le contexte de la classe.
    - Incrémente le nombre de champs de la classe et attribue un index au champ.
    - Ajoute la déclaration à l'environnement local de la classe.
    - Lève une exception `ContextualError` si le champ est défini plusieurs fois.
  - `verifyDeclFieldInit` :
    - Vérifie la validité de l'initialisation du champ, si elle est spécifiée.
    - Gère la taille et le type dynamique de l'attribut, en cas de tableaux ou de types complexes.
- **Génération de code :**
  - `codeGenField` :
    - Génère le code machine pour initialiser un champ de classe.
    - Alloue l'espace mémoire requis pour le champ.
    - Gère les cas d'initialisation explicite ou par défaut.
- **Affichage et parcours :**
  - `iterChildren` et `prettyPrintChildren` :
    - Permettent de parcourir les enfants de l'arbre syntaxique (le nom et l'initialisation du champ).

La classe DeclFieldSet

La classe DeclFieldSet représente un groupe de champs (ou attributs) de même type, déclarés simultanément dans une classe en Deca. **Méthodes principales :**

- **Vérification contextuelle :**
  - `verifyDeclFieldSet` :
    - Vérifie que les champs sont déclarés et conformes au contexte.
    - Valide le type des champs et les dimensions des tableaux.
    - Lève une exception `ContextualError` en cas d'incohérence.
  - `verifyDeclFieldSetInit` :
    - Vérifie l'initialisation des champs.
    - Garantit la conformité des types et dimensions pour chaque champ.
- **Décompilation :**
  - `decompile` :
    - Génère une représentation textuelle de la déclaration des champs.
- **Affichage et parcours :**
  - `iterChildren` et `prettyPrintChildren` :
    - Parcourent les enfants de l'arbre syntaxique (type et liste des champs).

3 Annexes

3.1 Diagramme UML

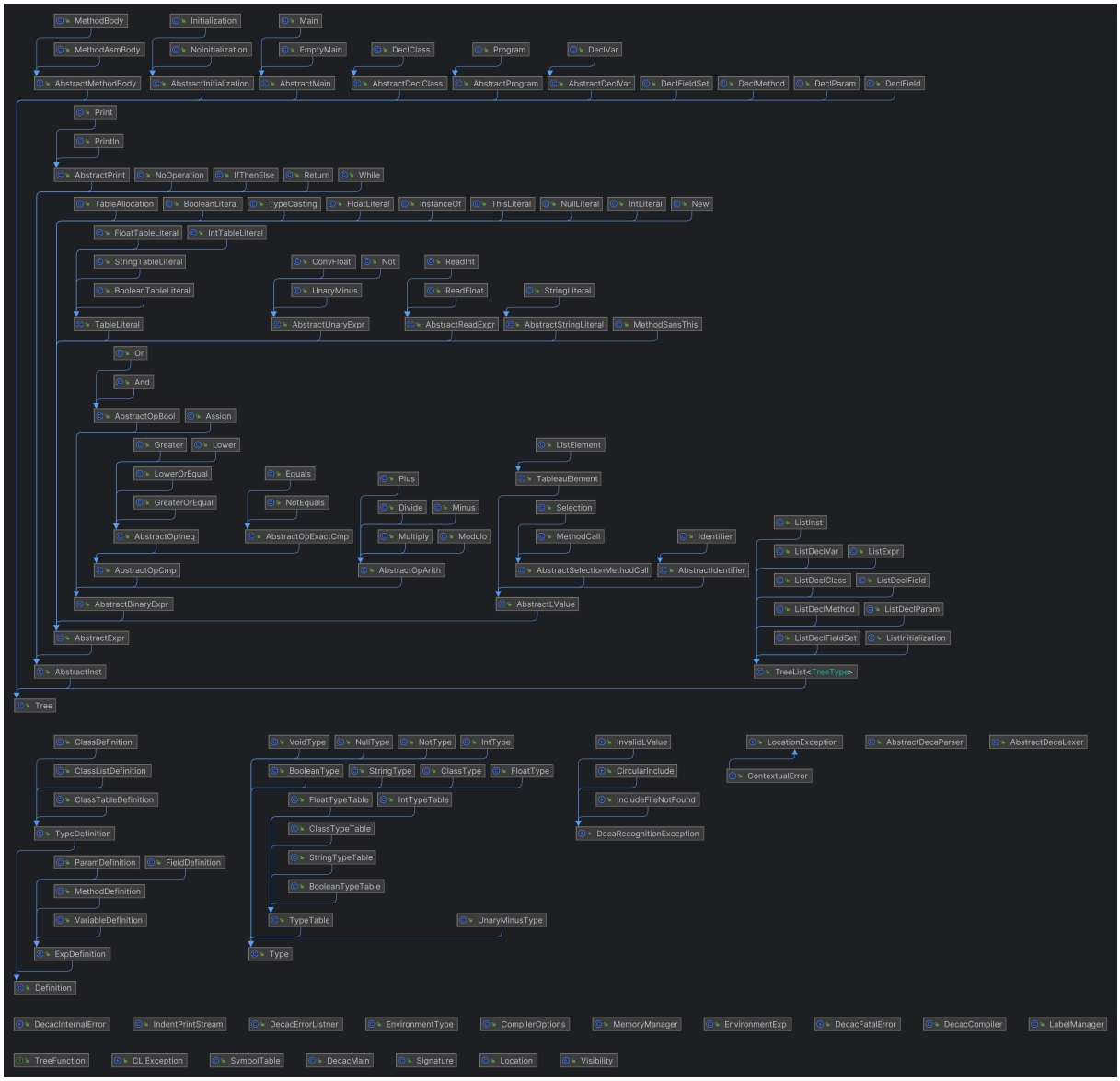

FIGURE 1 – Annexe.1