ENSIMAG

Projet Génie Logiciel

Groupe 13

Documentation de Conception du compilateur pour le Langage Deca

Réalisé par :

Benjelloun Otman
Birée Thomas
Boulouz Amine
El Goumiri Rida
Loginova Aleksandra

Année Universitaire : 2024/2025

Contents

1 Analyseur Lexical et Syntaxique pour le Langage Deca

Ce document présente en détail l'analyseur lexical (*lexer*) du langage Deca, développé avec **ANTLR4**. L'analyse lexicale est une étape cruciale dans le processus de compilation, consistant à transformer le code source en une suite d'unités lexicales (*tokens*). Ces *tokens* alimentent ensuite l'analyseur syntaxique (*parser*). Le Lexer joue un rôle fondamental en identifiant des motifs syntaxiques tels que les mots-clés, les symboles, les identifiants et les littéraux.

Dans le contexte du compilateur Deca, le Lexer est la première étape de traitement du code source. Il se situe avant l'analyse syntaxique et la vérification contextuelle. Cette hiérarchie garantit une structure claire et modulaire.

Pour la partie A, l'analyseur lexical a été développé à l'aide d'un fichier `.g4`, utilisé par l'outil **ANTLR4**. Ce fichier contient les définitions des règles lexicales du langage Deca et se trouve à l'emplacement suivant :

`src/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`

1.1 Lexer

La configuration du lexer précise que le code généré sera en Java et que notre lexer hérite de la classe `AbstractDecaLexer`.

1.1.1 Mots réservés

Les mots réservés définissent la structure du langage Deca et incluent des éléments comme les mots-clés (`class`, `if`, `return`, etc.). Ces règles sont disponibles dans le fichier `DecaLexer.g4`.

1.1.2 Symboles et opérateurs

Les symboles et opérateurs (tels que {, `+`, `-`, `==`) permettent de construire des expressions et des structures syntaxiques. Les définitions correspondantes sont disponibles dans le fichier `DecaLexer.g4`.

1.1.3 Littéraux

Les littéraux incluent les nombres entiers, flottants, et les chaînes de caractères. Par exemple, les règles pour les nombres incluent la prise en charge des entiers (`123`) et des flottants (`-123.45`). Ces définitions sont disponibles dans le fichier `DecaLexer.g4`.

1.1.4 Identifiants

Les identifiants (comme les noms de variables ou de méthodes) suivent une règle définie pour inclure des lettres, chiffres, et certains caractères spéciaux (`$`, `_`). Cette règle est disponible dans le fichier `DecaLexer.g4`.

1.1.5 Commentaires

Les commentaires mono-lignes (commençant par `//`) et multi-lignes (délimités par `/* ... */`) sont ignorés par le lexer. Les règles correspondantes se trouvent dans `DecaLexer.g4`.

1.1.6 Inclusion de fichiers

Le lexer supporte la directive `#include`, qui permet d'inclure des bibliothèques ou des définitions externes. Cette fonctionnalité est définie dans `DecaLexer.g4`.

1.1.7 Remarques Importantes

- L'ordre des règles dans `DecaLexer.g4` est crucial car ANTLR applique le principe "premier arrivé, premier servi".
- Les mots-clés doivent être définis avant les règles pour les identifiants afin d'éviter les conflits.
- Les fragments (`fragment`) sont des règles auxiliaires utilisées uniquement dans les règles principales.
- Les espaces et retours à la ligne sont ignorés grâce à une règle spécifique dans `DecaLexer.g4`.

1.2 Parseur

L'analyseur syntaxique (Parser) est responsable de l'analyse de la structure du programme Deca et de la construction de l'arbre abstrait. Il utilise les tokens fournis par l'analyseur lexical pour construire une représentation hiérarchique du programme. Le fichier est disponible :

`src/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`

Le parser est configuré pour :

- Générer du code Java
- Hériter de la classe AbstractDecaParser
- Utiliser le vocabulaire défini dans DecaLexer

1.2.1 Programme

Un programme est composé d'une liste de classes suivie d'un bloc principal (main).

1.2.2 Programme Principal

Le programme principal peut être vide ou contenir un bloc d'instructions.

1.2.3 Structure d'un Bloc

Un bloc en Deca est une unité syntaxique fondamentale qui contient des déclarations de variables et des instructions.

Cette règle définit la structure d'un bloc qui :

- Est délimité par des accolades (`OBRACE` et `CBRACE`)
- Contient une liste de déclarations (`list_decl`)
- Contient une liste d'instructions (`list_inst`)

1.2.4 Instructions et Expressions

La règle `list_inst` gère les instructions.

1.2.5 Expressions et Assignations

Ces règles définissent la gestion des expressions complexes et des assignations avec des vérifications de validité supplémentaires pour garantir la cohérence syntaxique.

1.2.6 Règle : `type`

Cette règle traite les types dans le langage Deca. Elle retourne un identifiant correspondant au type défini.

- La règle utilise la sous-règle `ident` pour extraire un identifiant valide.
- Elle vérifie que l'identifiant n'est pas nul et assigne sa localisation dans le code source.
- Retourne un `AbstractIdentifier`.

1.2.7 Règle : `literal`

Cette règle gère les valeurs littérales comme les entiers, flottants, chaînes, booléens, ainsi que les valeurs spéciales `null` et `this`.

- Chaque branche correspond à un type de littéral (entier, flottant, chaîne, etc.).
- Les exceptions sont levées pour les valeurs hors des limites ou malformées.
- Retourne un `AbstractExpr`.

1.2.8 Règle : `ident`

Cette règle extrait et retourne un identifiant sous forme d'`AbstractIdentifier`.

- Vérifie que l'identifiant est valide.
- Crée un symbole unique pour cet identifiant via le compilateur Deca.
- Retourne un `AbstractIdentifier`.

1.2.9 Liste des Classes

- Cette règle génère une liste de déclarations de classes (`ListDeclClass`).
- Chaque classe est extraite à l'aide de la règle `class_decl`.
- Les classes sont ajoutées de manière incrémentale à la liste.

1.2.10 Déclaration d'une Classe

- La règle `class_decl` définit la structure d'une classe, avec les éléments suivants :
  - Un nom de classe obligatoire, défini par `name`.
  - Une classe parente optionnelle, spécifiée par `superclass`. Si aucune classe parente n'est fournie, `Object` est utilisée par défaut.
  - Un corps de classe, défini par `body`, contenant les champs et les méthodes de la classe.
- La méthode utilise la classe parente par défaut (`Object`) si aucune extension n'est spécifiée.
- Les champs et méthodes du corps de la classe sont extraits via `getFields()` et `getMethods()`.

1.2.11 Extension de Classe

- Cette règle définit l'extension d'une classe en permettant d'ajouter une superclasse.
- Si le mot-clé `EXTENDS` est présent, l'identifiant suivant représente la superclasse.
- Si aucune extension n'est spécifiée, la classe parente par défaut est `Object`, créée comme un identifiant avec la localisation `BUILTIN`.
- La présence de la superclasse par défaut `Object` garantit la cohérence avec la hiérarchie des classes, même lorsque l'extension est omise.

1.2.12 Corps de Classe

- Le corps de la classe (`class_body`) peut contenir les éléments suivants :
  - Des déclarations de méthodes (`decl_method`), ajoutées à la liste `methods`.
  - Des ensembles de champs (`decl_field_set`), dont chaque champ est extrait et ajouté individuellement à la liste `fields`.
- Les déclarations de champs et de méthodes sont regroupées respectivement dans des instances de `ListDeclField` et `ListDeclMethod`.
- Une fois tous les éléments traités, les listes de champs et de méthodes sont associées à l'objet `ListDeclClassBody` via `setFields` et `setMethods`.
- Cette structure garantit une séparation claire entre les champs et les méthodes, facilitant ainsi les étapes ultérieures du processus de compilation.

1.2.13 Déclaration des Champs et des Méthodes

1.2.14 Ensemble de Champs

- Un ensemble de champs (`decl_field_set`) est défini par :
  - Une visibilité (`visibility`), comme `public` ou `protected`.
  - Un type (`type`) associé à tous les champs de cet ensemble.
  - Une ou plusieurs déclarations de champs (`list_decl_field`).
- La règle `list_decl_field` prend en paramètre le type et la visibilité afin de générer les déclarations individuelles correctement.
- Chaque déclaration de champ est extraite de `list_decl_field` et ajoutée à la liste `ListDeclField`.
- Les informations finales sont encapsulées dans une instance de `ListDeclField`, qui regroupe toutes les déclarations de champs.

1.2.15 Visibilité des Champs

- La visibilité par défaut est `PUBLIC`.
- Le mot-clé `PROTECTED` peut être utilisé pour indiquer une visibilité protégée.

1.2.16 Liste de Champs

- La règle `list_decl_field` permet de déclarer plusieurs champs dans une même instruction.
- Chaque champ est traité par la règle `decl_field`, qui prend en paramètre :
  - `typeIdent` : le type des champs.
  - `vis` : la visibilité des champs.
- La liste des champs est stockée dans une instance de `ListDeclField`.
- Chaque champ extrait par `decl_field` est validé avec un `assert` et ajouté à la liste via `add`.

1.2.17 Déclaration d'un Champ

- Un champ est défini par :
  - Un type, passé en paramètre (`typeIdent`).
  - Une visibilité, également passée en paramètre (`vis`).
  - Un identifiant (`ident`) représentant le nom du champ.
- Une initialisation optionnelle peut être ajoutée à l'aide du symbole `EQUALS`, suivi d'une expression (`expr`).
- Si aucune initialisation n'est spécifiée, une instance de `NoInitialization` est utilisée par défaut.

- La déclaration finale est encapsulée dans une instance de `DeclField`, qui contient :
  - Le type (`typeIdent`).
  - L'identifiant (`i.tree`).
  - L'initialisation (`NoInitialization` ou `Initialization`).
  - La visibilité (`vis`).
- La position du champ est définie avec `setLocation`, basée sur le début du type (`i.start`).

1.2.18 Déclaration d'une Méthode

- Une méthode est définie par les éléments suivants :
  - Un type de retour (`t=type`).
  - Un identifiant (`i=ident`) représentant le nom de la méthode.
  - Une liste de paramètres (`lp=list_params`).
- Le corps de la méthode peut être :
  - Un bloc standard (`block`) contenant des déclarations et des instructions. Ces éléments sont encapsulés dans une instance de `MethodBody`.
  - Un bloc assembleur (`ASM`) défini par une chaîne de texte multi-ligne (`multi_line_string`). Il est encapsulé dans une instance de `MethodAsmBody`.
- La méthode complète est représentée par une instance de `DeclMethod`, qui regroupe :
  - Le type de retour (`t.tree`).
  - Le nom de la méthode (`i.tree`).
  - La liste des paramètres (`lp.tree`).
  - Le corps de la méthode (`body`).
- La position de la méthode est définie avec `setLocation`, en utilisant la position du type (`t.start`).

1.2.19 Liste des Paramètres

- La règle extrait une liste de paramètres .
- Chaque paramètre est obtenu à l'aide de la règle `param`.

1.2.20 Chaînes Multilignes

- La règle traite les chaînes simples et les chaînes multilignes.
- Elle retourne le texte de la chaîne et sa localisation dans le code source.

1.2.21 Conclusion de l'analyse syntaxique

Cette documentation présente les principales règles de l'analyseur syntaxique Deca. En suivant cette structure et en s'appuyant sur des outils comme ANTLR, il est possible de construire et valider efficacement le parser du compilateur Deca.

Ces règles définissent la structure des classes dans le langage Deca, en intégrant la gestion des superclasses, des champs et des méthodes. Elles permettent de construire l'arbre syntaxique correspondant à une hiérarchie de classes.

1.2.22 Gestion des Erreurs

Le parser inclut des vérifications systématiques pour garantir la cohérence de la grammaire. Par exemple :

- `assert($list_decl.tree` $\neq$ `null)` : Valide la présence d'une liste de déclarations.
- `assert($inst.tree` $\neq$ `null)` : Vérifie la validité d'une instruction.

2 Vérification contextuelle et décoration de l'arbre

2.1 Contexte

Le langage Deca repose sur une syntaxe contextuelle qui définit les programmes " bien typés ". Ces programmes doivent être compilés en un programme assembleur respectant la sémantique du programme source. Réciproquement, tout programme Deca mal typé doit être rejeté par le compilateur avec un message d'erreur approprié.

La vérification contextuelle est essentielle pour garantir cette cohérence. Elle s'effectue en trois passes distinctes sur le programme, chacune ayant un objectif spécifique :

- **Passe 1** : Vérification des noms des classes et de la hiérarchie des classes. Cette étape permet de gérer les références croisées dans les déclarations.
- **Passe 2** : Vérification des déclarations de champs et des signatures des méthodes. Cette étape prend en compte les dépendances entre méthodes pouvant être récursives.
- **Passe 3** : Vérification des corps de méthodes, instructions, expressions, et initialisations.

Chaque passe utilise une grammaire attribuée définie sur la syntaxe abstraite du programme. Les programmes bien typés sont ceux dont l'arbre de syntaxe abstraite est accepté successivement par chacune des grammaires attribuées.

2.2 Objectifs

L'objectif de la vérification syntaxique contextuelle est double :

- Garantir qu'un programme Deca valide produit un comportement conforme à sa sémantique lors de son exécution.
- Rejeter systématiquement les programmes mal typés avec des messages d'erreur pertinents.

Pour atteindre ces objectifs, la vérification contextuelle repose sur :

- La gestion des environnements de types et des identificateurs (`env_types` et `env_exp`).
- La définition rigoureuse des relations entre types, sous-types et opérateurs.
- La compatibilité entre les déclarations, les signatures et les corps des méthodes.

Ces étapes assurent que chaque composant du langage est vérifié de manière systématique et cohérente.

2.3 Organisation des Sous-Répertoires

Le répertoire `src/main/java/fr/ensimag/deca` contient deux sous-répertoires principaux, `/context` et `/tree`, chacun ayant un rôle distinct dans la vérification contextuelle et la manipulation de la syntaxe abstraite du langage Deca.

2.3.1 Sous-répertoire `/context`

Le sous-répertoire `/context` contient les classes et interfaces nécessaires pour la vérification contextuelle du programme Deca. Ces classes définissent les structures de données et les algorithmes permettant de vérifier la conformité des programmes aux règles de typage et aux contraintes sémantiques du langage.

Principaux fichiers dans `/context` :

- `EnvironmentType.java` et `EnvironmentExp.java` : Ces classes implémentent les environnements de types et d'expressions, essentiels pour la gestion des symboles et des définitions dans les différentes passes de vérification.
- `Definition.java`, `Type.java`, `ClassDefinition.java` : Ces fichiers définissent les structures utilisées pour représenter les définitions des types, des classes, et des méthodes.
- `TypeDefinition.java`, `FieldDefinition.java`, `MethodDefinition.java` : Spécialisations des définitions pour les champs, les méthodes et les types.
- `ClassType.java`, `BooleanType.java`, `IntType.java`, `FloatType.java`, etc. : Implémentations des différents types du langage Deca.

Ce répertoire est donc dédié à la vérification contextuelle et au typage, en assurant la gestion correcte des environnements, des sous-types, et des relations entre les différents éléments du langage.

2.3.2 Sous-répertoire `/tree`

Le sous-répertoire `/tree` contient les classes nécessaires à la représentation et à la manipulation de l'arbre de syntaxe abstraite (AST) du programme Deca. Ces classes définissent les différents nœuds de l'arbre, leur structure, ainsi que leurs attributs hérités et synthétisés.

Principaux fichiers dans `/tree` :

- `AbstractExpr.java`, `AbstractInst.java`, `AbstractDeclMethod.java` : Classes abstraites représentant les nœuds de haut niveau (expressions, instructions, déclarations).
- `Assign.java`, `While.java`, `IfThenElse.java` : Classes concrètes représentant des instructions spécifiques.
- `ListDeclClass.java`, `ListInst.java`, `ListExpr.java` : Classes pour gérer les listes de déclarations, d'instructions, ou d'expressions.
- `IntLiteral.java`, `FloatLiteral.java`, `BooleanLiteral.java` : Représentation des littéraux du langage.
- `MethodCall.java`, `Selection.java` : Représentation des appels de méthode et des sélections de champs.

Ce répertoire est donc responsable de la représentation et de la navigation dans l'AST. Les classes définies ici permettent également d'appliquer les règles contextuelles en transmettant et en évaluant les attributs hérités et synthétisés.

2.4 Exemples :

- Dans `Identifier.java`, on trouve quatre méthodes de vérification spécifiques :
  - `verifyExpr`: Vérifie qu'un identifiant est valide en tant qu'expression.
  - `verifyMethod`: Vérifie qu'un identifiant est valide en tant que méthode.
  - `verifyType`: Vérifie qu'un identifiant est valide en tant que type.
  - `verifyRValue`: Vérifie qu'un identifiant peut être utilisé comme une valeur assignable (rvalue) tout en respectant les règles de typage.
- Dans `Assign.java`, une méthode `verifyExpr` vérifie la compatibilité entre les types du côté gauche (lvar) et droit (rexpr) d'une assignation.
- La méthode commence par vérifier que l'opérande gauche (`leftOperand`) correspond à une variable existante dans l'environnement local (`localEnv`).
- Elle appelle ensuite `verifyRValue` sur l'opérande droit (`rightOperand`) pour vérifier récursivement l'expression de droite.
- Si une conversion implicite est nécessaire (par exemple, de `int` vers `float`), un nœud `ConvFloat` est ajouté à l'AST.
- Si les types sont incompatibles et aucune conversion n'est possible, une exception `ContextualError` est levée avec un message d'erreur explicite.
- Enfin, la méthode marque la variable du côté gauche comme initialisée.

2.5 Environnement de Type et environnement d'expression

2.5.1 `EnvironmentType.java`

Description : Ce fichier implémente un environnement contenant les types du langage Deca. Cet environnement est utilisé pour gérer les types prédéfinis et ceux définis par l'utilisateur.

Utilité :

- **Prédéfinition des types standards :** Les types comme `int`, `float`, `boolean`, `void` et `Object` sont ajoutés par défaut à l'environnement via la méthode `declareClass`.
- **Gestion des définitions de types :** La méthode `defOfType` permet de rechercher la définition d'un type spécifique.
- **Vérification des relations de sous-types :** Les méthodes `isSubType` et `isSubClass` vérifient si un type ou une classe est un sous-type d'un autre, conformément aux règles de polymorphisme.
- **Utilisation dans la vérification contextuelle :** Pendant les passes contextuelles, les types déclarés par l'utilisateur sont ajoutés à cet environnement pour permettre les vérifications comme la compatibilité des types ou la déclaration correcte des classes.

2.5.2 `EnvironmentExp.java`

Description : Ce fichier implémente un environnement d'expressions, représenté comme une liste chaînée de dictionnaires, qui associe des identificateurs à leurs définitions dans différents contextes (par exemple, des superclasses ou des méthodes).

Utilité :

- **Gestion hiérarchique des environnements :** Chaque instance d'`EnvironmentExp` peut pointer vers un environnement parent, ce qui permet de gérer la portée et l'héritage des identificateurs.
- **Recherche de définitions :** La méthode `get` permet de rechercher un identificateur dans l'environnement courant et, si nécessaire, dans ses environnements parents.
- **Gestion des définitions spécifiques :** Les méthodes `getAsField` et `getAsMethod` permettent de rechercher des identificateurs spécifiques comme des champs ou des méthodes.
- **Insertion de nouvelles définitions :** La méthode `declare` permet d'ajouter une nouvelle définition à l'environnement, avec une gestion des doublons via une exception `DoubleDefException`.
- **Utilisation dans la vérification contextuelle :** Cet environnement est essentiel pour vérifier la déclaration et l'utilisation correcte des variables, champs et méthodes dans le code utilisateur.

2.6 Deca sans Objet

2.6.1 Chaîne d'éxecution de la vérification contextuelle

La vérification de l'arbre abstrait se fait à partir de l'appel à la méthode `verifyProgram`. La chaîne de vérification débute. Un appel est fait aux méthodes `verifyClasses` et `verifyMain` sur les attributs `classes` et `main` de l'instance de `Program`. L'arbre est ainsi parcouru.

La vérification pour le main, (et donc à fortiori le Deca sans-objet) se fait en une seule passe. On vérifie la liste des déclarations des variables, et ensuite la liste des instructions.

2.6.2 Vérification de la liste des déclarations des variables

Pour chaque `DeclVar` de `ListDeclVar`, `verifyDeclVar` est invoqué.

Remarque : l'appel à la méthode `newGlobalVar` de la classe `DecacCompiler` incrémente l'attribut `globalVarCount` de cette même classe, cet attriut sera utilisé lors de la génération de code pour gérer les éventuels débordements de piles (Cela sera abordé avec en détails dans la section de génération de code).

Création et affectation de la définition de variable dans l'environnement actuel (en l'occurence ici, l'environnement global) dans le cas où celle-ci n'a pas déjà été déclarée. Une erreur adéquate est produit au cas échéant.

Vérification de l'expression représentant le nom de variable(le mécanisme de vérification de `AbstractExpr` et des classes qui en dérivent sera détaillé ultérieurement).

Vérification d'une éventuelle initialisation de la variable, et gestion d'une variable balise de l'état d'initialisation (ou non) de la variable dans sa `VariableDefintion` (cela nous permettra de lever un warning lorsqu'une variable non initialisée, et pour laquelle aucune valeur n'est assignée, est utilisée).

2.6.3 Vérification d'`AbstractExpr`

La classe `AbstractExpr` est parent de plusieurs classes, dont on distinguera trois catégories différentes : les littéraux, les identifieurs et les opérations, qu'elles soient unaires, binaires ou de comparaison. (il y a en fait une autre catégorie, celle des mots clé réservés du langage Deca, qu'on abordera ultérieurement).

Les littéraux

Ces littéraux implémentent chacun la méthode `verifyExpr`, qui ne font que renvoyer le type du littéral (e.x `compiler.environmentType.INT` pour `IntLiteral`)

Les identifieurs

Un nouvel identifieur est créé à chaque fois que le parser en détecte un, il est cependant nécessaire de préserver l'unicité de la relation entre le symbole de l'identifier (son "nom") et la définition initialisée à sa déclaration. La méthode `declare` de la classe `EnvironmentExp` s'en charge (c'est d'ailleurs ici qu'on se charge de la gestion de l'erreur de définition double, comme on garantit l'unicité du symbole).

Ainsi, la méthode `verifyExpr` de la classe `Identifier` fait correspondre la définition appropriée et se l'attribue. Elle fait cela pour le type également.

Vérification des opérations (opérations arithmétiques prises comme exemple)

La vérification des opération arithmétiques est effectuée directement dans la classe parent `AbstractOpArith`. On vérifie, "récursivement", l'opérande de droite de l'expression, puis l'opérande gauche. On atteint toujours un identifieur ou un littéral, qui renvoient leurs types, il s'en suit l'attribution des types adéquats (avec prise en compte des conversions implicites d'entiers en flottants par ajout des nodes `ConvFloat` lorsque cela est nécessaire).

On gère ici l'erreur d'incompatibilité des types des opérandes des opérations arithmétiques. Les vérifications des opérations logiques et des comparaisons se font de façon analogue.

Remarque : Ici on remarque aussi un appel à la méthode `setPossibleOverflow` de la classe `DecacCompiler`. On notifie le compilateur de la possibile présence d'une erreur overflow sur les opérations arithmétiques lorsque cela est nécessaire. Nous nous pencherons sur la gestion des erreurs runtime des programmes compilés en détails dans la partie de génération de code.

2.6.4 Vérification de `ListInst`

Similairement à `ListDeclVar`, `ListInst` dispose de la méthode `verifyListInst` qui invoque `verifyInst` pour chacune des instances d'`AbstractInst` concernées.

Remarque : la classe `AbstractExpr` dispose en fait elle aussi d'une méthode `AbstractInst`. Cette dernière ne fait cependant qu'appeler la méthode `VerifyExpr`.

2.6.5 Vérification d'`AbstractPrint`

La vérification des instructions `Print` et `Println` se fait dans la classe parent `AbstractPrint`. Elle s'occupe simplement de vérifier que chacune des `AbstractExpr` de son attribut `ListExpr arguments` est d'un type accepté par ces méthodes, elle lève une erreur appropriée sinon.

2.6.6 Vérification de `IfThenElse`, `While`, `NoOperation`

Comme son nom l'indique, la vérification de `NoOperation` ne fait rien. Celles de `IfThenElse` et `While` vérifient, quant à elles, que leurs conditions sont bien des booléens, et invoque `verifyListInst` pour les instructions des blocs if/while. Un choix d'implémentation additionnel a été effectué au niveau de la classe `IfThenElse` : on représente les branches else if par une `Arraylist` d'instances de classes `IfThenElse` (dont l'attribut `elseBranch` est une instance de `IfThenElse` qui n'aura, par construction, aucune instruction).

La vérification de `While` est analogue.

2.7 Partie avec Objet

Dans cette partie nous expliquerons la partie du code qui s'occupe de gérer les classes, leur héritage et le objets issus de ces classes.

2.7.1 Vérification des Classes

La vérification contextuelle des classes dans Deca se fait en plusieurs étapes (ou passes). Chaque étape est dédiée à un aspect particulier de la vérification, comme la vérification des déclarations des classes, des membres (méthodes et champs), et du corps des classes. Voici une explication détaillée du processus :

2.7.2 Vérification des Déclarations des Classes

La classe `DeclClass` joue un rôle central dans la déclaration et la vérification des classes. Elle s'appuie sur une série de méthodes pour exécuter les trois passes de la syntaxe contextuelle.

Pass 1 : Vérification des Classes (`verifyListClass`). Cette passe vérifie les déclarations des classes et initialise leurs environnements.

- Elle parcourt la liste des classes déclarées en appelant `verifyClass` sur chaque classe.
- Une fois les classes vérifiées, elle passe à la vérification des membres (`verifyListClassMembers`).

Pass 2 : Vérification des Membres des Classes (`verifyListClassMembers`). Cette passe vérifie les membres de chaque classe, y compris leurs méthodes et champs.

- Elle appelle `verifyClassMembers` sur chaque classe pour vérifier les signatures des méthodes et les déclarations des champs.
- Après avoir vérifié les membres, elle passe à la vérification du corps des classes (`verifyListClassBody`).

Pass 3 : Vérification du Corps des Classes (`verifyListClassBody`). Cette passe vérifie le corps de chaque classe, en s'assurant que les définitions des champs et des méthodes respectent les règles contextuelles.

2.7.3 Vérification des Champs (`DeclField`)

Les champs des classes sont vérifiés via la méthode `verifyListFieldBody` définie dans `DeclField`. L'objectif est de vérifier chaque champ déclaré dans une classe, en appelant `verifyFieldBody` sur chaque champ.

Validation des Types des Champs (`verifyFieldType`).

- Vérifie que le type d'un champ est valide (par exemple, un champ ne peut pas être de type `void`).
- Ajoute le champ à la définition de la classe et gère les redéfinitions avec une exception `DoubleDefException`.
- Met à jour le statut d'initialisation du champ si une valeur par défaut est définie.

2.7.4 Vérification des déclarations de méthodes (`verifyDeclMethod`)

La déclaration des méthodes dans Deca est gérée par la méthode `verifyDeclMethod`, qui vérifie leur validité contextuelle et leur conformité avec les méthodes des classes parentes (en cas de redéfinition). Cette méthode est définie dans la classe `DeclMethod`.

Objectifs de `verifyDeclMethod` :

- Vérifier que le type de retour de la méthode est valide.
- Valider la liste des paramètres et construire la signature de la méthode.
- Vérifier la conformité des redéfinitions de méthodes (override).
- Ajouter la méthode à l'environnement de la classe en gérant les doublons.

Étapes principales :

1. Récupérer l'environnement des membres de la classe parente (`parentEnvExp`) et initialiser un nouvel environnement pour la méthode (`methodEnvExp`).
2. Vérifier le type de retour de la méthode avec `returnIdentifier.verifyType`.
3. Construire la signature de la méthode en vérifiant les paramètres via `params.verifyListDeclParam`.
4. Si une méthode du même nom existe dans la classe parente, vérifier :
    - La correspondance entre les types de retour.
    - La correspondance entre les signatures.
5. En cas de validité, associer un label unique à la méthode.
6. Ajouter la méthode à l'environnement de la classe et lever une exception en cas de doublon (`DoubleDefException`).

Explications des Points Importants :

- **Vérification du Type de Retour :** Le type de retour est vérifié via `returnIdentifier.verifyType`, et tout type invalide (comme `void` dans un contexte interdit) lève une exception `ContextualError`.
- **Construction de la Signature :** La signature est construite en validant les types des paramètres avec `verifyListDeclParam`.
- **Gestion des Redéfinitions :** Si une méthode existe dans la classe parente :
  - Les types de retour doivent correspondre exactement.
  - Les signatures doivent être identiques.
  - Une exception est levée en cas de non-conformité.
- **Ajout dans l'Environnement :** La méthode est ajoutée à l'environnement de la classe via `classDef.getMembers().declare`, et une exception `DoubleDefException` est levée en cas de duplication.

3 Génération de code

3.1 Description de l'étape

La génération de code (*étape C*) consiste à transformer l'arbre syntaxique abstrait décoré en un programme en langage d'assemblage exécutable par la machine abstraite IMA. Cette étape intervient après les analyses syntaxique (A) et contextuelle (B), et elle s'appuie sur les décorations ajoutées à l'arbre pour garantir que le code produit est à la fois cohérent et optimisé.

Les principaux objectifs de cette étape sont :

- **Traduire les instructions et expressions Deca** en séquences d'instructions d'assemblage IMA.
- **Gérer efficacement les ressources machine**, notamment les registres et la pile.
- **Inclure la gestion des erreurs d'exécution**, comme les divisions par zéro ou les dépassements de pile.
- **Construire la table des méthodes** pour les programmes avec objets.

3.2 Liste des répertoires concernés

Les classes et fichiers impliqués dans cette étape sont regroupés dans les répertoires suivants :

- `src/main/java/fr/ensimag/deca/tree/` : Contient les classes permettant de parcourir l'arbre abstrait pour chaque type d'instruction et d'expression. Ce répertoire est particulièrement crucial pour les méthodes `codeGen`, car des classes implémente la génération de code pour un type spécifique de nœud de l'arbre syntaxique.
- `src/main/java/fr/ensimag/deca/codegen/` : Regroupe les classes spécialisées dans la génération du code d'assemblage
- `src/main/java/fr/ensimag/deca/ima/pseudocode/` : Définit les instructions et structures de données pour représenter un programme en assembleur IMA.

3.3 Étapes principales de l'implémentation

3.3.1 Initialisation

Avant de parcourir l'arbre syntaxique, le compilateur initialise plusieurs structures :

- **La table des registres** : Un tableau `avaRegs[]` pour suivre l'état des registres (libres ou occupés).
- **La pile** : Alloue de l'espace pour les variables temporaires ou globales.
- **La table des méthodes** : Structure regroupant les adresses des méthodes pour les programmes avec objets.

3.3.2 Parcours de l'arbre abstrait

Chaque nœud de l'arbre syntaxique est visité pour générer les instructions correspondantes :

- **Expressions** : Les littéraux, identifiants, et opérations sont traduits en instructions telles que `LOAD`, `ADD`, ou `MUL`. Les conversions implicites (e.g., `ConvFloat`) sont gérées directement dans l'arbre.
- **Instructions** : Les structures de contrôle (e.g., `if`, `while`) sont traduites en blocs de code utilisant des étiquettes pour gérer les sauts conditionnels.
- **Appels de méthodes** : Les arguments sont empilés, et un saut est effectué à l'adresse de la méthode. Les étiquettes associées à chaque méthode sont définies dans la table des méthodes.

3.3.3 Détails des classes du répertoire `tree`

Le répertoire `tree` contient les classes représentant les différents nœuds de l'arbre syntaxique abstrait et gérant la génération de code pour ces derniers.

`AbstractExpr` Cette classe abstraite sert de base aux expressions de l'arbre. Sa méthode `codeGenPrint` génère les instructions nécessaires pour évaluer une expression et afficher son résultat, avec un traitement distinct pour les entiers et les flottants.

`AbstractOpArith` Gère les opérations arithmétiques binaires (`+`, `-`, `*`, etc.). La méthode `codeGenInst` génère le code pour évaluer les deux opérandes, appliquer l'opération et gérer les éventuels débordements.

`AbstractOpCmp` Implémente les comparaisons binaires (`==`, `<`, `>`, etc.). `codeGenInst` évalue les deux opérandes et génère l'instruction `CMP`, suivie d'une instruction conditionnelle pour déterminer le résultat.

`And` Gère l'opérateur logique `&&`. La méthode `codeGenInst` implémente une évaluation paresseuse, évitant d'évaluer l'opérande droit si l'opérande gauche est `false`.

`Assign` Gère les assignations (`=`). `codeGenInst` évalue l'expression à assigner et la stocke dans la variable ou le champ cible, en tenant compte des conversions implicites.

`BooleanLiteral` Représente les valeurs `true` et `false`. `codeGenInst` charge la valeur correspondante (`1` ou `0`) dans un registre ou sur la pile.

`Cast` Gère les conversions explicites de type (e.g., `int` vers `float`). La méthode `codeGenInst` applique l'instruction appropriée (`FLOAT` ou `INT`) selon le type cible.

`ConvFloat` Effectue une conversion implicite d'`int` vers `float`. `codeGenInst` génère l'instruction `FLOAT` pour effectuer cette conversion, en optimisant l'utilisation de la pile et des registres.

`DeclClass` Gère la déclaration des classes, y compris la génération de la table des méthodes (VTable) et l'initialisation des champs. `codeGenTable` et `codeGenClassInit` assurent respectivement la gestion de l'héritage et l'initialisation des champs et méthodes.

`DeclField` Gère la déclaration des champs d'une classe. `codeGenDeclField` initialise les champs explicitement, tandis que `codeGenDeclFieldDummy` génère une initialisation par défaut (e.g., `0` pour les entiers, `false` pour les booléens). Les instructions `LOAD` et `STORE` sont utilisées pour gérer les valeurs et leur emplacement en mémoire.

`DeclMethod` Représente une méthode déclarée dans une classe. `codeGenMethodBody` génère le corps de la méthode, en créant un label unique pour la méthode, en gérant les paramètres, et en produisant le code des instructions du corps. Elle assure également la sauvegarde et la restauration des registres, ainsi que la gestion des retours pour les méthodes non-void.

`DeclParam` Gère la déclaration des paramètres dans une méthode. `codeGenDeclParam` génère le code pour charger le paramètre dans un registre ou une position appropriée sur la pile. Elle vérifie également que le type du paramètre est valide et qu'il n'y a pas de doublons dans la liste des paramètres.

`DeclVar` Représente la déclaration d'une variable locale. La méthode `codeGenDeclVar` attribue une position mémoire pour la variable dans la pile ou le registre global (`GB`) et génère le code pour son initialisation, si nécessaire. Si la variable est initialisée, le code correspondant à l'expression d'initialisation est généré, et la valeur est stockée dans l'adresse allouée à la variable.

`Identifier` Représente un identifiant dans le langage Deca, qui peut correspondre à une variable, un champ, une méthode, ou une classe. La méthode `codeGenInst` gère la récupération de la valeur associée à l'identifiant et son stockage dans un registre ou sur la pile. La méthode `codeGenPrint` gère l'affichage de l'identifiant, en utilisant les instructions appropriées selon son type (`int`, `float`, etc.).

`IfThenElse` Gère les instructions conditionnelles `if`, `else if` et `else`. La méthode `codeGenInst` génère les instructions nécessaires pour évaluer la condition, sauter aux blocs correspondants (`then` ou `else`) et terminer proprement la structure avec une étiquette `end_if`.

`InstanceOf` Gère l'opérateur `instanceof`, qui vérifie si une expression est une instance d'un type donné. La méthode `codeGenInst` génère un code en boucle pour comparer la table des méthodes (`VTable`) de l'expression avec celle de la classe cible et de ses superclasses, jusqu'à obtenir une correspondance ou atteindre la classe `null`.

`IntLiteral` Représente une constante entière dans Deca. La méthode `codeGenInst` génère le code pour charger la valeur entière dans un registre ou sur la pile, selon les ressources disponibles. `codeGenPrint` produit le code nécessaire pour afficher cette valeur à l'aide de l'instruction `WINT`.

`ListDeclClass` Gère la génération de code pour les classes.

- `codeGenListClassTable` : Génère les tables virtuelles (VTables) pour les classes, en commençant par la classe `Object`, puis les classes définies par l'utilisateur. Met à jour le compteur de pile pour refléter la taille des VTables.
- `codeGenListClassMethods` : Génère le code des méthodes pour toutes les classes, y compris la méthode `Object.equals` qui compare les adresses des objets.

`Main` Gère la génération de code pour le programme principal.

- `codeGenMain` :
  - Génère le code pour la déclaration des variables dans le bloc principal, en ajoutant un commentaire et en initialisant la gestion d'un éventuel dépassement de pile.
  - Ajoute le code pour les instructions du bloc principal après les déclarations de variables.

`MethodAsmBody` Représente un corps de méthode contenant du code assembleur personnalisé.

- `codeGenInst` :
  - Ajoute des commentaires indiquant le début et la fin du code assembleur.
  - Parcourt chaque ligne de code assembleur fourni, ajoutant des commentaires pour documentation dans le code généré.

`MethodBody` Représente le corps d'une méthode, contenant des déclarations de variables locales et des instructions.

- `codeGenInst` :
  - Génère le code pour les déclarations de variables locales via `decls.codeGenListDeclVar`.
  - Génère le code pour les instructions via `insts.codeGenListInst`.
  - Ajoute des commentaires pour délimiter le début et la fin du corps de la méthode dans le code généré.

`MethodCall` Représente l'appel d'une méthode avec ses arguments sur un objet.

- `codeGenInst` :
  - Alloue de l'espace sur la pile pour les paramètres, y compris l'instance (paramètre implicite).
  - Génère le code pour évaluer l'adresse de l'objet appelant et la place dans la pile.
  - Évalue les arguments un par un en générant leur code et les place dans la pile.
  - Charge l'adresse de la table des méthodes (`VTable`) de l'objet.

- Appelle la méthode en utilisant l'instruction `BSR` avec l'offset de la méthode dans la `VTable`.
- Libère l'espace de la pile réservé aux arguments après l'appel.
- Si la méthode a un type de retour non `void`, place le résultat dans `R0`.

`New` Gère l'instanciation d'un nouvel objet avec le mot-clé `new`.

- `codeGenInst` :
  - Alloue la mémoire pour une nouvelle instance de la classe en utilisant l'instruction `NEW`, avec un espace calculé comme le nombre de champs + 1 (pour la table des méthodes).
  - Charge l'adresse de la `VTable` de la classe dans le registre.
  - Stocke l'adresse de la `VTable` dans le premier champ de l'instance.
  - Appelle la méthode d'initialisation de la classe `init.<ClassName>` en utilisant l'instruction `BSR`.
  - Après l'appel, l'adresse de l'instance nouvellement allouée est laissée dans un registre pour un usage ultérieur.

`Not` Gère l'opération unaire logique `!`, utilisée pour inverser une valeur booléenne.

- `codeGenInst` :
  - Génère le code pour l'opérande et récupère sa position (registre ou pile).
  - Si aucun registre n'est disponible, utilise la pile pour effectuer l'opération :
    - Charge la valeur dans un registre temporaire, la pousse sur la pile, et la récupère pour exécuter l'opération `NOT`.
    - Utilise deux étiquettes (`not_true` et `not_end`) pour gérer les cas `true` et `false`.
  - Si un registre est disponible, effectue l'opération directement dans le registre :
    - Compare la valeur à `true` (`1`) en utilisant une instruction conditionnelle `BEQ`.
    - Charge `0` si l'opérande est `true`, ou `1` sinon.
  - Met à jour la position du résultat (`lastExprPos`) avec le registre ou la pile.

`Or` Gère l'opération logique `||` avec court-circuit.

- `codeGenInst` :
  - Génère le code pour l'opérande gauche.
  - Si l'opérande gauche est `true`, saute directement à une étiquette `or_true`.
  - Sinon, évalue l'opérande droit.
  - Si l'opérande droit est `true`, saute également à `or_true`.
  - Si aucune des deux conditions n'est remplie, charge `0` (faux).
  - Charge `1` (vrai) à l'étiquette `or_true`.

`Program` Gère la génération de code pour un programme complet, incluant les classes et le bloc `main`.

- `codeGenProgram` :
  - Génère la table des VTables via `codeGenListClassTable`.
  - Génère le code du `main` via `codeGenMain`.
  - Ajoute une instruction `HALT`.
  - Génère le code des méthodes des classes via `codeGenListClassMethods`.
  - Ajoute la gestion des erreurs grâce à `genCodeErrors`.

`ReadFloat` Gère la lecture d'une entrée utilisateur en tant que nombre flottant (`float`).

- `codeGenInst` :
  - Utilise l'instruction `RFLOAT` pour lire un `float`.
  - Ajoute un contrôle d'erreur pour les entrées/sorties via `BOV` (branch on overflow).
  - Si un registre temporaire est disponible :
    - Charge la valeur lue dans le registre via `LOAD`.
    - Met à jour la position de la dernière expression.
  - Sinon :
    - Empile la valeur lue sur la pile via `PUSH`.
    - Met à jour la position de la dernière expression pour indiquer l'adresse sur la pile.

`ReadInt` Gère la lecture d'une entrée utilisateur en tant qu'entier (`int`).

- `codeGenInst` :
  - Utilise l'instruction `RINT` pour lire un entier.
  - Ajoute un contrôle d'erreur pour les entrées/sorties via `BOV`.
  - Si un registre temporaire est disponible :
    - Charge la valeur lue dans le registre via `LOAD`.
    - Met à jour la position de la dernière expression.
  - Sinon :
    - Empile la valeur lue sur la pile via `PUSH`.
    - Met à jour la position de la dernière expression pour indiquer l'adresse sur la pile.

`Return` Gère l'instruction de retour (`return`) dans une méthode.

- `codeGenInst` :
  - Évalue l'expression de retour (`ret`) en appelant `codeGenInst` sur celle-ci.
  - Charge la valeur de l'expression dans le registre `R0` via l'instruction `LOAD`.
  - Met à jour la position de la dernière expression (`Register.setLastExprPos`) pour refléter la valeur dans `R0`.
  - Effectue un saut (`BRA`) vers l'étiquette de fin de méthode, définie par `HelperInfo`.

`Selection` Gère les accès aux champs ou méthodes d'un objet via une sélection (`object.field`).

- `codeGenInst` :
  - Génère le code pour évaluer l'expression de l'objet (`object`) et charge son adresse dans un registre.
  - Vérifie les accès pour éviter les déréréférencements de null (`CMP` suivi d'un saut conditionnel vers `null_dereference_error`).
  - Calcule l'offset du champ (ou méthode) sélectionné via un `RegisterOffset` en fonction de l'index du champ.
  - Charge la valeur du champ sélectionné dans le registre cible à l'aide de l'instruction `LOAD`.

`This` Représente le mot-clé `this`, utilisé pour référer à l'instance courante d'une classe dans Deca.

- `codeGenInst` :
  - Charge l'adresse de l'instance courante depuis `-2(LB)` (position réservée pour `this` dans le contexte d'exécution) dans un registre inutilisé.
  - Met à jour `lastExprPos` pour refléter l'emplacement du registre utilisé pour `this`.
  - Vérifie la disponibilité des registres pour optimiser l'exécution.

`UnaryMinus` Représente l'opérateur unaire `-` (négation arithmétique) dans Deca.

- `codeGenInst` :
  - Évalue l'opérande et récupère sa position dans un registre ou sur la pile.
  - Si l'opérande est sur la pile (`SP`) :
    - Charge la valeur dans un registre fixe (`R1`).
    - Applique l'instruction `OPP` pour inverser le signe.
    - Pousse la valeur résultante sur la pile.
    - Met à jour le compteur de pile avec `StackCount.countPush()`.
  - Si l'opérande est dans un registre :
    - Applique directement l'instruction `OPP` au registre contenant l'opérande.

3.3.4 Gestion des erreurs

Plusieurs étiquettes sont insérées dans le code pour gérer les cas d'erreurs :

- **Division par zéro** : Un saut vers une routine d'erreur est ajouté en cas de division par zéro. Le message affiché est : `division by zero`.
- **Dépassement de pile** : Géré par un compteur incrémenté lors de chaque appel de fonction. En cas de dépassement, le message `Stack Overflow` est affiché.
- **Dépassement du tas** : Si la mémoire pour créer un nouvel objet est insuffisante, le message `Heap Overflow Error` est levé.
- **Erreur d'entrée/sortie** : Si un type en entrée n'est pas compatible, le message `Input/Output error` est affiché.
- **Déréférencement de null** : Lorsqu'une variable nulle est assignée ou déréférencée, le message `Dereferencement null` est levé.