Document de conception

Auteurs :
Mehdi El Idrissi El Fatmi
Khadija El Adnani
Alaa Jennine
Badr El Khoundafi
Yassine El Kouri

22/01/2025

January 24, 2025

Contents

1 Introduction

Le projet consiste à développer un compilateur pour le langage Deca, avec une extension ARM pour générer du code assembleur adapté à cette architecture. Ce document détaille les étapes d'implémentation, les choix techniques, et les résultats obtenus.

2 Contexte et objectifs

Le langage Deca est conçu pour apprendre les concepts de compilation, en fournissant une syntaxe simple et un modèle extensible. Le compilateur doit :

- Analyser et valider la syntaxe et la sémantique d'un programme Deca.
- Générer un arbre abstrait décoré.
- Produire un code assembleur adapté à l'architecture ARM.

3 Architecture du compilateur

Le compilateur est divisé en plusieurs parties principales :

Analyse lexicale

3.1 Analyse lexico- syntaxique

Cette partie permet de faire la verification lexicale et syntaxique du programme et nous donner en sortie un arbre syntaxique .

3.2 Analyse contextuelle

3.2.1 Introduction

Dans le cadre de l'analyse contextuelle pour le langage Deca, cette étape joue un rôle crucial dans la vérification de la validité contextuelle d'un programme. Contrairement à l'analyse syntaxique, qui s'assure uniquement que le programme respecte la grammaire définie, l'analyse contextuelle se concentre sur les aspects sémantiques en appliquant un ensemble de règles contextuelles rigoureuses.

Ces règles permettent de garantir que le programme respecte les contraintes propres au langage Deca, comme la déclaration préalable des variables, la compatibilité des types dans les expressions, et la portée des identificateurs. Une fois ces vérifications effectuées, l'arbre syntaxique généré lors de l'étape précédente est enrichi avec des informations supplémentaires, telles que les types des expressions, les références aux déclarations associées, et les portées des variables.

3.2.2 Hiérarchie du code: Classes et dépendances

L'analyse contextuelle est structurée autour de plusieurs classes centrales qui s'intègrent au modèle global du compilateur. Voici une description générale des classes impliquées :

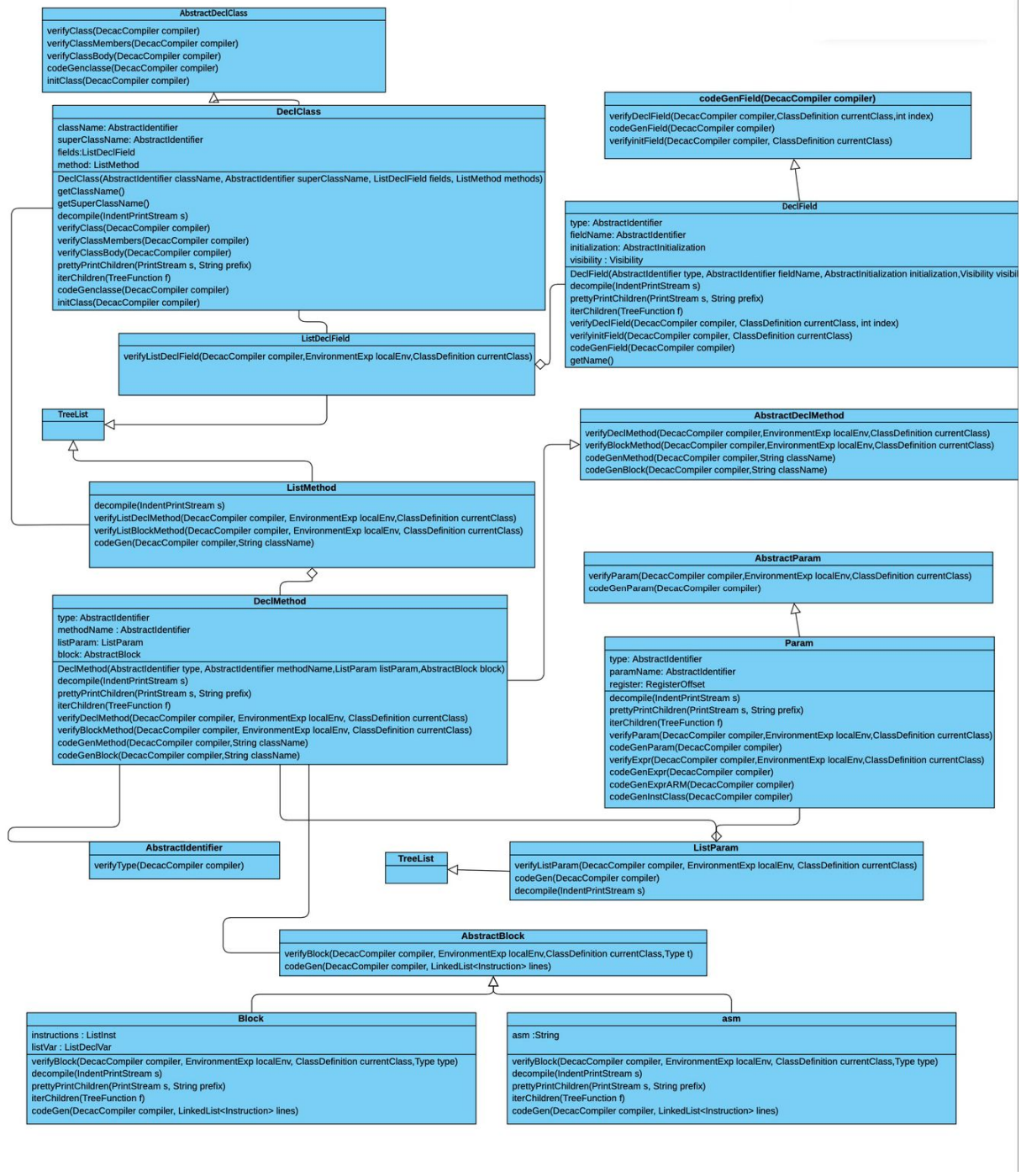

Figure 1: Diagramme UML representant le hiearchie des classes.

EnvironmentExp : Cette classe représente un environnement lié aux définitions de variables et méthodes. Elle implémente un dictionnaire associant les noms des identificateurs à leurs définitions (`ExpDefinition`). Chaque instance d'`EnvironmentExp` est liée à un environnement parent, permettant ainsi de modéliser les portées imbriquées (par exemple, une classe et ses superclasses).

- **Fonctionnalités principales :**
  - Ajouter des méthodes et des variables (à travers `add` et `declare`).

- Récupérer une définition via la méthode `get`, qui effectue une recherche dans l'environnement courant et, si nécessaire, dans les environnements parents.
- Gérer les exceptions de déclarations multiples (`DoubleDefException`).

EnvironmentType : Cette classe gère les types prédéfinis et les définitions de classes dans le langage Deca. Elle inclut initialement des types standards, tels que `int`, `float`, `boolean`, `void`, ainsi que le type `Object`, ajouté lors de l'implémentation de la partie orientée objet. Elle permet également l'ajout de nouveaux types via des déclarations de classes utilisateur.

- **Fonctionnalités principales :**
  - Stocker les types prédéfinis dans une table de symboles (à l'aide de `HashMap<Symbol, TypeDefinition>`).
  - Fournir une méthode pour récupérer une définition de type via un symbole (`defOfType`).
  - Inclure des objets pour chaque type prédéfini (par exemple, `VOID`, `INT`, `FLOAT`, etc.), facilitant leur accès direct.

Lors de l'implémentation de la partie B, nous avons adopté une approche itérative en débutant par le langage Deca sans objet. Le paragraphe suivant détaillera les classes nécessaires à l'implémentation de cette partie :

Langage sans objet

Dans la version sans objet, plusieurs fonctions ont été implémentées pour assurer le respect des règles de la grammaire et du bon fonctionnement du compilateur. Voici une description détaillée des principales fonctions :

- `verifyExpr` :
  - Rôle : Vérifie la validité des expressions en respectant les règles de la grammaire.
- `verifyRvalue` :
  - Rôle : Valide les expressions utilisées comme valeurs dans les affectations.
- `verifyCondition` :
  - Rôle : Vérifie les conditions des structures `if` et des boucles `while`.
- `verifyProgram` :
  - Rôle : Valide le programme dans son ensemble en se limitant à la fonction `main`.
- `verifyMain` :
  - Rôle : Vérifie la liste des instructions et des déclarations dans la fonction `main`.

Ainsi, pour la grammaire sans objet, seul le passe 3 est nécessaire. Par conséquent, nous nous limitons à vérifier uniquement la liste des déclarations et la liste des instructions à l'aide des fonctions `verifyX`. Voici un exemple illustratif des fonctions appelées pour un programme.

```
{
    int x = 6;
    boolean estVrai = true;
}
```

Pour ce programme Deca sans objet :

- La fonction `verifyListDeclVar` est appelée pour analyser la liste des déclarations de variables.
  - Cette fonction appelle `verifyDeclVar` pour chaque déclaration individuelle.
    - `verifyDeclVar` effectue les vérifications suivantes :
      - Appelle `verifyType` pour s'assurer que le type spécifié pour la variable est valide.
      - Appelle `verifyInitialisation` pour valider que la variable est correctement initialisée (ou n'a pas d'initialisation: NoInitialisation).
      - `verifyInitialisation` fait appel à `verifyExpr` pour vérifier que l'expression assignée est conforme au type attendu et correcte et enrichir l'arbre si necessaire.

Ce processus garantit que toutes les déclarations respectent les règles de la grammaire et du typage.

Langage avec objet

1. Organisation générale et classes principales

L'implémentation du compilateur s'articule autour de plusieurs classes clés, chacune ayant un rôle bien défini pour assurer la gestion des méthodes, champs et appels dans le langage Deca orienté objet. AbstractDeclMethod,AbstractDeclFieldet AbstractBlock. Ces trois classes heritent de tree.

- **DeclMethod** :
  - Cette classe représente la déclaration d'une méthode dans une classe et herite de AbstractDeclMethod.
  - **Éléments notables** :
    - Gestion des signatures des méthodes, y compris leur validation et leur compatibilité avec les méthodes redéfinies dans les classes mères conformement aux regles deca.
    - Attribution de numéros d'ordre aux méthodes pour leur référencement dans la table des méthodes.
- **DeclField** :

  - Cette classe permet de déclarer les champs (à la fois leurs types et leurs initialisations) dans les classes, et herite de AbstractDeclField.
  - **Éléments notables** :
    * Validation du type de chaque champ, en s'assurant que les restrictions (par exemple, pas de type void) sont respectées.
    * Gestion de la visibilité (étendue publique ou protégée) des champs dans les classes et leurs sous-classes.

- **Dot** :
  - Cette classe représente une expression d'accès à un membre d'une classe (exemple : objet.membre).
  - **Éléments notables** :
    * Validation que l'expression à gauche du . est bien un objet valide (classe ou instance).
    * Recherche dans les classes et super-classes pour récupérer la définition du membre accédé.
    * Gestion des restrictions d'accès pour les membres déclarés comme protected .

    NB : Cette classe est équivalente à `Selection`, que nous n'avions pas identifié comme déjà implémentée initialement, ce qui justifie le développement de cette fonction.

- **PointObject** La classe `PointObjet` gère l'accès à une méthode d'un objet via la notation pointée (`instance.method()`).

- **CallMethod** :
  - Cette classe modélise l'appel d'une méthode sur un objet ou dans un contexte de classe.
  - **Éléments notables** :
    * Validation des arguments passés à la méthode : correspondance en type et en nombre avec la signature attendue.

- **AbstractBlock** La classe Block représente le corps d'une méthode.

  Les classes `Block` et `Asm` héritent de la classe abstraite `AbstractBlock`, mais elles remplissent des rôles distincts dans le compilateur :
  - `Block` :
    * Représente le corps d'une méthode dans le langage Deca.
    * Contient une suite de déclarations de variables locales (`listVar`) suivie d'instructions (`instructions`).
  - `Asm` :
    * Représente une section de code assembleur directement intégrée.
    * Contrairement à `Block`, elle ne contient ni déclarations de variables ni instructions de haut niveau.

New :

- Cette classe permet de créer une nouvelle instance d'une classe donnée via la syntaxe `new ClassName()`.

InstanceOf :

- Cette classe implémente l'opérateur `instanceof`, qui vérifie si une expression est d'un type de classe donné.

L'implémentation des classes `ListMethod` et `ListDeclField` s'est ensuite avérée indispensable.

Spécifications

Quelques modifications ont été apportées aux fonctions verifyX afin de respecter les contraintes de la partie orientée objet. De plus, certaines fonctions ont été implémentées lorsque cela s'est avéré nécessaire, comme la méthode equals pour la classe Signature.

3.3 Vérification contextuelle

La vérification contextuelle dans l'implémentation du compilateur est organisée en trois passes distinctes, chacune étant gérée par une fonction spécifique dans la classe abstraite `AbstractDeclClass` :

- **Pass 1 :**Gérée par la méthode `verifyClass`, cette première passe assure la validation de la déclaration d'une classe, sans examiner son contenu. Elle vérifie que la super-classe existe et est valide, crée une définition pour la classe actuelle, et met à jour l'environnement global des types en y intégrant cette nouvelle classe. Elle garantit également que la classe n'est pas déjà déclarée et associe les définitions adéquates aux identifiants de la classe et de sa super-classe.
- **Pass 2 :**Gérée par la méthode `verifyClassMembers`, cette seconde passe contextuelle vérifie les membres de la classe (champs et méthodes) en se limitant à leurs déclarations, sans examiner les corps des méthodes ni les initialisations des champs. La méthode `verifyClassMembers` appelle `verifyListDeclField` pour valider les déclarations des champs et `verifyListDeclMethod` pour celles des méthodes. La méthode `verifyDeclField` s'assure qu'un champ possède un type valide (non `void`), qu'il n'est pas déjà déclaré dans l'environnement local ou dans une super-classe, et l'ajoute à l'environnement de la classe. Enfin, `verifyDeclMethod` vérifie le type de retour des méthodes, leur compatibilité avec une éventuelle redéfinition, et les déclare de manière unique dans l'environnement local.
- **Pass 3 :** Gérée par la méthode `verifyClassBody`. Cette dernière passe s'assure que les instructions et expressions contenues dans la classe sont correctes. Cela inclut la vérification du corps des méthodes a l'aide de la methode verifyBlock et des initialisations de champs a l'aide de verifyInitField.

Chaque passe est essentielle pour garantir la cohérence et la validité contextuelle des classes dans le langage Deca, en s'appuyant sur une séparation claire des responsabilités.

3.4 Génération de code Assembleur pour la machine abstraite ima

3.4.1 Introduction

La génération du code image (IMA) représente une étape clé dans le processus de compilation, où le code source est transformé en une forme exécutable ou en une représentation intermédiaire propre à une architecture cible. Cette phase vise à traduire les structures de données et les instructions générées précédemment dans l'arbre abstrait décoré en un code optimisé, généralement sous forme de langage assembleur ou d'instructions machine. Dans le cadre de notre projet, cette étape s'inscrit dans un processus global d'implémentation d'un compilateur pour le langage Deca, avec une extension pour générer du code assembleur adapté à l'architecture ARM. La conception de la génération du code image se repose sur une analyse détaillée des instructions et sur la création d'un environnement capable de manipuler les registres et autres éléments spécifiques à l'architecture cible. Ce document décrit la conception et les choix effectués pour assurer une génération de code efficace et conforme aux exigences du projet.

3.4.2 Hiérarchie du code

La génération de code IMA repose sur une architecture modulaire et hiérarchique, permettant de traduire efficacement les arbres abstraits décorés en instructions machine. Chaque instruction est représentée par une classe spécifique, organisée selon des relations d'héritage pour maximiser la réutilisabilité et la clarté. Cette structure assure une séparation claire entre la logique des instructions et leurs opérandes, tout en facilitant l'extension du système pour intégrer de nouvelles fonctionnalités ou instructions.

3.5 Détails de la méthode `codeGen`

La méthode `codeGen` joue un rôle central dans la génération de code pour les opérations arithmétiques. Elle peut être résumée comme suit :

- **Objectif principal** : Traduire une opération abstraite (issue de l'arbre syntaxique décoré) en instructions machine compatibles avec l'architecture IMA.
- **Gestion des opérandes** :
  - Les opérandes peuvent être de différents types (`DVal`, `DAddr`, variables, constantes, etc.).
  - Des cas spécifiques sont traités, comme lorsque l'un ou les deux opérandes sont stockés en mémoire (offsets).
- **Optimisation des registres** :

    - La méthode gère intelligemment les registres disponibles (notamment `GPRegister`) pour minimiser les accès à la pile et optimiser les performances.
    - Si un registre est marqué comme *offset*, il est récupéré ou utilisé correctement dans les instructions.
- **Modularité via le constructeur** :
    - Une classe abstraite, comme `constructeur`, permet d'encapsuler la logique spécifique à une opération.
    - Par exemple, pour une addition, la classe `constructeurADD` génère l'instruction `ADD`.

3.6 Différences avec la génération de code pour la division

La génération de code pour l'opération de division se distingue par :

- **Gestion des erreurs** : Une vérification explicite est ajoutée pour éviter la division par zéro (`CMP` suivi de `BEQ`) avec un saut vers un label d'erreur (`division_by_zero_error`).
- **Conversion des types** : Les entiers sont convertis en flottants (`FLOAT`) avant la division pour assurer la compatibilité des types.

3.7 Génération de code pour l'initialisation

La méthode `codeGenInit` se distingue des autres opérations arithmétiques par l'ajout de l'instruction `STORE`, qui permet de stocker le résultat de l'opération dans une adresse mémoire. Le processus se déroule ainsi :

- Les opérandes gauche et droite sont d'abord générés (`codeGenInit`).
- Un registre est associé pour effectuer l'opération arithmétique.
- L'opération est ensuite réalisée à l'aide de la méthode `codeGen`, avec un constructeur spécifique (par exemple, `constructeurADD`).
- Enfin, le résultat est stocké dans l'adresse mémoire courante avec l'instruction `STORE`.

Ce processus est utilisé pour des initialisations telles que `x = y + z`.

3.8 Détails de la méthode `codeGenPrint`

La méthode `codeGenPrint` génère le code pour effectuer une opération arithmétique entre deux opérandes (`DVal`) tout en préparant le résultat pour l'impression via le registre `R1`.

- **Objectif principal :** Assurer le traitement des deux opérandes, exécuter l'opération définie par le `constructeur`, et charger le résultat dans `R1`.
- **Cas traités :**

    - Si le registre utilisé (`reg`) ou l'opérande gauche (`leftOperand`) est un `offset`, des instructions telles que `POP` et `LOAD` sont utilisées pour récupérer les valeurs nécessaires.
    - Si `leftOperand` est une adresse mémoire (`DAddr`), elle est chargée dans le registre pour effectuer l'opération.
    - Si l'un des opérandes est une variable, un traitement spécifique garantit une gestion correcte des valeurs.
- **Résultat final :** Le registre `R1` contient le résultat de l'opération, prêt à être imprimé.

3.9 Génération de code pour les instructions conditionnelles et les boucles

La méthode `codeGenInst` génère du code pour les instructions conditionnelles, en traitant la condition `if` et les branches `then` et `else`. Elle gère également l'ajout d'étiquettes pour la structure de contrôle.

- `Label elseLabel, endIfLabel, bodyLabel` sont générés pour identifier les différentes sections du bloc conditionnel.
- La condition est traitée via `codeGenInstrCond`, qui analyse les opérandes gauche et droite.
- Si une branche `else` existe, elle est traitée après la branche `then`, avec une gestion de saut (BRA) vers l'étiquette de fin.
- L'étiquette de fin est utilisée pour marquer la fin de la condition.

La méthode `codeGenInstrCond` gère les opérateurs logiques `AND` et `OR` en générant les instructions conditionnelles nécessaires :

- Si `OR` est activé, la condition est évaluée avec un compteur et des étiquettes associées.
- Si `AND` est activé, l'opérande gauche et droit sont traités pour évaluer la condition et gérer les étiquettes.
- Des étiquettes temporaires (`nouvLabel`) sont générées pour gérer la logique des `OR` et `AND`.

3.10 Génération du code pour les classes et les méthodes

La génération du code pour les classes et les méthodes est divisée en deux parties principales : la gestion des classes et la gestion des méthodes. Les étapes impliquent la création de la table des méthodes et des champs pour chaque classe.

3.10.1 Méthode `codeGenClasses`

La méthode `codeGenClasses` génère le code pour l'ensemble des classes, en associant une adresse pour chaque classe et en traitant les classes dans la liste.

- Une étiquette `labelClasses` est ajoutée pour marquer le début de la gestion des classes.
- Si la liste des classes contient des éléments, les adresses pour les objets sont initialisées avec `RegisterOffset`.
- Pour chaque classe, la méthode `codeGenclasse` est appelée pour générer le code spécifique de la classe.
- Une instruction `LOAD` et `STORE` est utilisée pour gérer l'adresse de la classe et ses méthodes.

3.10.2 Méthode `codeGenMethod`

La méthode `codeGenMethod` génère le code pour les méthodes d'une classe, en initialisant la table des méthodes et en associant les adresses aux méthodes de la classe.

- La méthode commence par générer des commentaires pour indiquer le début de la construction de la table des méthodes de la classe.
- Une adresse est associée pour la méthode `equals` de la classe `Object`.
- La table des champs est initialisée avec les attributs de la classe, et si la classe hérite d'une autre classe, les champs de la superclasse sont également pris en compte.
- La méthode `codeGenMethod` est appelée pour chaque méthode de la classe afin de générer le code de chaque méthode.
- Une table des champs est également construite, et les numéros des champs sont associés à chaque champ.

3.10.3 Génération du code pour les champs

La méthode `codeGenField` génère le code pour les champs d'une classe. Elle gère la création de l'adresse de chaque champ, son initialisation éventuelle, et la mise à jour de l'offset du registre correspondant.

- **Initialisation des champs :**
  - La méthode commence par appeler `setNbreField` pour incrémenter le nombre de champs.
  - L'adresse du champ est obtenue via `getRegisterClass`, qui renvoie un `RegisterOffset`.
  - Une instance de `FieldDefinition` est créée, associant le type, la localisation et l'offset du champ.
- **Gestion de l'initialisation :**

    - Si l'initialisation du champ est définie, `initialization.codeGenField` est appelée pour générer le code d'initialisation.
    - Sinon, une instruction `LOAD` est ajoutée pour initialiser le champ à une valeur par défaut (`0`).
- **Stockage de la valeur :**
    - Le code ajoute une instruction `LOAD` pour charger une adresse relative au champ, suivie d'une instruction `STORE` pour stocker la valeur dans l'adresse calculée.

3.11 Génération du code pour les instructions arithmétiques dans une méthode de classe

La méthode `codeGenInstClass` génère le code pour les opérations arithmétiques effectuées dans une méthode d'une classe. Elle traite les opérandes gauche et droit, effectue une addition et gère la libération des registres.

- **Initialisation des registres :**
    - Un registre est associé via `compiler.associerReg()` pour être utilisé dans les instructions suivantes.
- **Gestion des opérandes :**
    - La méthode génère le code pour les opérandes gauche et droite en appelant `codeGenInstClass` sur `getLeftOperand()` et `getRightOperand()` respectivement.
    - Ces opérandes sont chargés dans un registre pour être manipulés.
- **Vérification des registres :**
    - Si le registre n'est pas déjà dans la liste des registres utilisés (`compiler.registeres`), il est ajouté à cette liste.
- **Génération des instructions :**
    - La première instruction charge l'adresse de la variable `-2` dans le registre `reg` via une instruction `LOAD`.
    - La deuxième instruction charge la valeur de `leftOperand` dans le registre `reg`.
    - La troisième instruction ajoute la valeur de `rightOperand` au registre `reg`.
- **Libération du registre :**
    - Après l'exécution de l'instruction, le registre est libéré via `compiler.libererReg()` pour être réutilisé dans d'autres parties du code.

Génération de code ARM

L'extension ARM génère un code assembleur optimisé compatible avec l'architecture cible.

4 Choix techniques

Langage et outils

- Langage de développement : Java.
- Outils : `JavaCC` pour l'analyse lexicale et syntaxique.
- Gestion de projet : `Git`.

4.1 Gestion des erreurs

- **Erreurs lexicographiques :** Identification des caractères invalides.
- **Erreurs syntaxiques :** Rejet des structures malformées.
- **Erreurs sémantiques :** Détection des incohérences de types.

5 Défis rencontrés

- Gestion des dépendances dans l'arbre syntaxique.
- Optimisation du code généré pour ARM.
- Collaboration entre les membres de l'équipe sur les différentes parties du projet.

Résultats

Le compilateur produit un code assembleur ARM fonctionnel pour les programmes Deca valides. Voici un exemple de programme Deca et son code assembleur généré :

5.1 Code Deca

Listing 1: Exemple de code Deca

```
class Main {
    void main() {
        int a = 5;
        int b = 10;
        int sum = a + b;
        print(sum);
    }
}
```

5.2 Code ARM généré

Listing 2: Code assembleur ARM

```
MOV R0, #5
MOV R1, #10
ADD R2, R0, R1
BL print
```

6 Conclusion et perspectives

Le projet a permis d'implémenter un compilateur fonctionnel pour le langage Deca avec une extension ARM. Les prochaines étapes pourraient inclure :

- Ajout d'optimisations avancées (par ex. : élimination des redondances).
- Support pour d'autres architectures, comme RISC-V.

A Appendices

Structure du projet

```
src
   Lexer.java
   Parser.java
   SemanticChecker.java
   CodeGenerator.java
tests
   test_lexer.deca
   test_parser.deca
   test_codegen.deca
README.md
```