

PROJET GÉNIE LOGICIEL 2023-2024

DOCUMENTATION DE CONCEPTION

Ingénieur 2ème année

Professeur encadrante:

Karine ALTISEN

Equipe projet :

- Stéphane KOUADIO
- Julian COUX
- Breno MORAIS
- Loan GATIMEL
- Hugo MERCIER

SOMMAIRE

I. Introduction.....	2
1.1 Objectif du document.....	2
1.2 Comprendre le compilateur.....	2
II. Architectures du Compilateur.....	3
2.1 Structure Globale.....	3
2.2 Dépendances entre Classes.....	4
III. Spécifications Additionnelles du Code.....	6
3.1 Modifications apportées.....	6
3.2 Conventions de Codage.....	8
IV. Perspective d'évolution.....	9
4.1 Limites du compilateur.....	9
V. Conclusion.....	10

I. Introduction

1.1 Objectif du document

Bienvenue dans la documentation de conception du compilateur Deca, un projet ambitieux visant à mettre en place une solution robuste pour la compilation du langage Deca, une variante de Java. Cette documentation s'adresse aux développeurs qui, à l'avenir, auront la charge de maintenir, améliorer, ou simplement comprendre le fonctionnement interne de ce compilateur.

Notre objectif principal dans cette documentation est de fournir une vision d'ensemble claire de l'organisation générale de l'implémentation. Vous découvrirez les différentes architectures qui composent le compilateur, les spécifications additionnelles du code, ainsi que les algorithmes et les structures de données personnalisés que nous avons intégrés pour répondre aux besoins spécifiques du langage Deca.

Cette documentation ne cherche pas à reproduire des listings de code, mais plutôt à offrir des informations complémentaires aux documents fournis par les enseignants. Vous y trouverez des explications sur les choix de conception, les justifications derrière les modifications apportées, et une vue d'ensemble détaillée des éléments clés qui constituent notre solution.

Nous espérons que cette documentation servira de ressource précieuse pour ceux qui auront la tâche stimulante de travailler sur le compilateur Deca à l'avenir. Bonne lecture et bon développement !

1.2 Comprendre le compilateur

L'objectif de notre compilateur est de générer un fichier assembleur compatible avec la machine virtuelle IMA à partir d'un programme Deca donné. La compilation s'effectue en trois étapes distinctes :

1. **Étape A - Analyse Syntaxique :**

- L'analyse syntaxique vise à déterminer la conformité du programme sur le plan de la syntaxe.
- Le compilateur crée un arbre syntaxique représentant la structure du programme, assurant ainsi son écriture correcte.

2. **Étape B - Analyse Contextuelle :**

- Cette étape se concentre sur la vérification des aspects contextuels tels que la gestion des variables, des types et des classes.

- En cas de réussite des vérifications, le compilateur enrichit l'arbre syntaxique en ajoutant des informations contextuelles, telles que les types de variables ou la localisation de leur définition.
3. **Étape C - Génération du Code Assembleur :**
- Une fois l'arbre enrichi, le compilateur procède à la génération du code assembleur correspondant.
 - Le fichier assembleur résultant est prêt à être exécuté par la machine virtuelle IMA, concrétisant ainsi la transformation réussie du programme Deca initial.

Parallèlement, une extension a été ajoutée à notre compilateur. La bibliothèque Math.h, qui permet d'inclure les principales fonctions trigonométriques aux programmes Deca des utilisateurs. Avec une documentation décrivant les procédés utilisés et la précision des fonctions implémentées. Mais aussi cette documentation décrit les choix de rapport entre précision et performance réalisés.

II. Architectures du Compilateur

2.1 Structure Globale

Avant d'explorer en profondeur le fonctionnement interne du compilateur, il est crucial de comprendre la structure des fichiers et leur organisation. Dans le cadre de ce projet, une organisation méthodique s'impose pour assurer une hiérarchie de dossiers cohérente.

Le répertoire principal est divisé en deux axes majeurs : les fichiers du répertoire "main", dédiés à la compilation, et les fichiers du répertoire "test", qui ont été essentiels pour tester et valider notre compilateur.

Dans la branche "main", plusieurs éléments méritent une attention particulière :

-Fichiers antlr4 : Ces fichiers sont cruciaux pour l'étape A de la compilation. Parmi eux, Lexer.g4 et Parser.g4 jouent un rôle clé. Le lexer, responsable de l'analyse syntaxique, définit toutes les syntaxes spécifiques autorisées en Deca ainsi que leur interprétation. Le Parser, chargé de l'analyse lexicale, vérifie la construction correcte du programme, s'assurant entre autres que chaque type est suivi d'une variable et que les parenthèses s'ouvrent et se ferment correctement. C'est également dans ce programme que le premier arbre syntaxique est créé.

-Fichiers Java : Ces fichiers sont essentiels pour les étapes B et C de la compilation. Sous la branche "/tree/", on retrouve l'ensemble des fichiers .java qui décrivent des actions spécifiques du langage. Une architecture particulière a été

adoptée entre les différentes classes afin de maximiser la factorisation du code (voir Annexe n°1). Cette approche nous a permis de regrouper les fichiers par famille, tels que ceux dédiés à la vérification des déclarations de variables, aux opérations mathématiques, aux comparaisons, etc.

Ces fichiers sont le cœur même de la compilation. Mis à part le Lexer et le Parser, la plupart des fichiers qui ont été et qui pourront être modifiés, se trouvent ici. Pour chacun de ces fichiers, on trouvera les méthodes utilisées pour l'étape B (verifyXYZ), celles de l'étape C (codeGen) et de nombreuses autres pour rendre la communication entre classes possible.

-Fichiers Deca de TRIGO : Ces fichiers servent à l'intégration de l'extension TRIGO au compilateur. Ces fichiers servent de bibliothèque avec Math.h que les utilisateurs peuvent utiliser dans leur programme Deca pour utiliser des fonctions tel que *cos* et *sin*. Ces fichiers se trouvent dans le répertoire :

src/main/resources/include/math.deca

Ces fonctions sont : *sin*, *cos*, *asin*, *atan* et *ulp*. D'autres fonctions sont implémentés dans ce fichier, et reconnaissables par le signature "-" devant leur intitulés. Elles servent principalement à l'écriture des fonctions trigonométriques citées précédemment. Ces fonctions sont aussi documentées dans ce document. Et une meilleure description de leur précision et réalisation est disponible dans la documentation de l'extension.

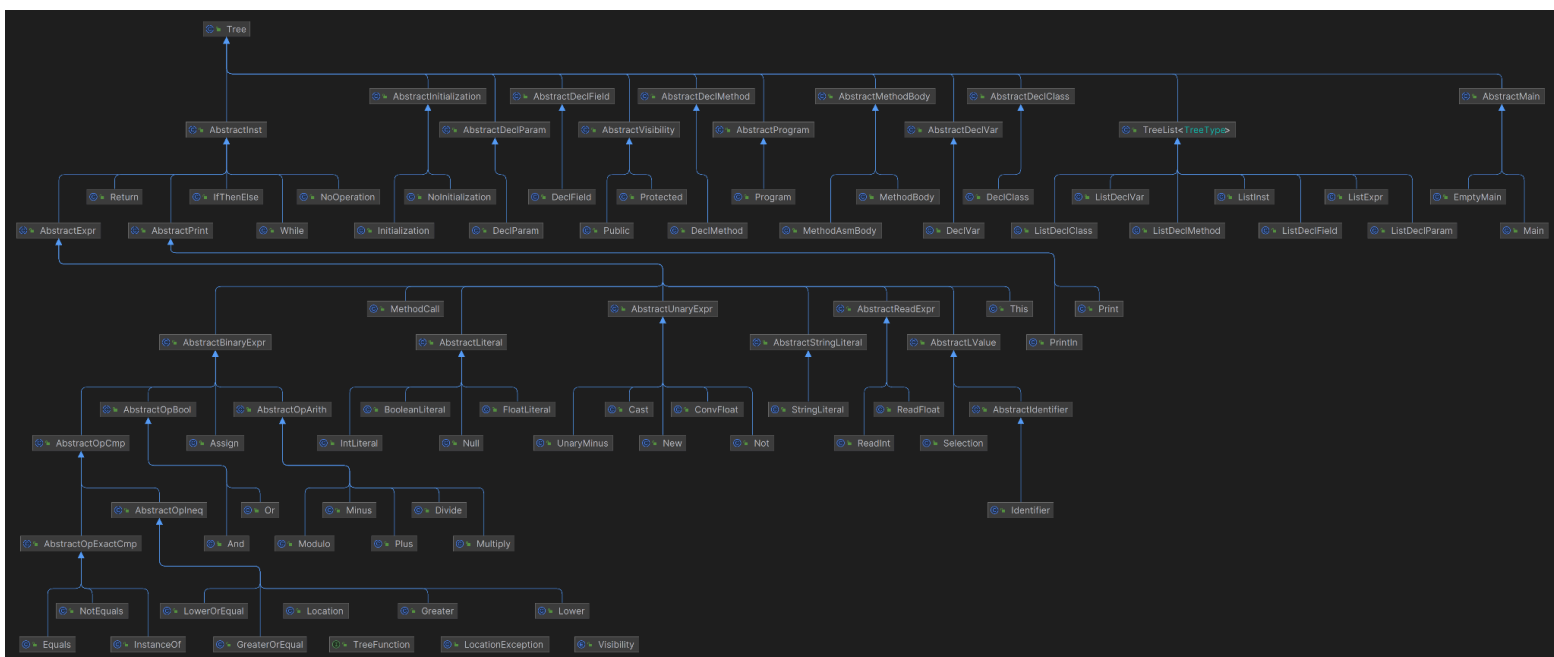
Dans la branche "**test**", plusieurs éléments méritent d'être soulignés :

-Fichiers de test en Deca : Ces fichiers sont variés et répartis en quatre sections distinctes : les tests d'analyse lexicale, syntaxique, contextuelle et de génération de code. Chacune de ces sections comporte deux types de tests : les tests de validation, destinés à confirmer le bon fonctionnement du compilateur, et les tests d'invalidation, visant à vérifier la capacité du compilateur à détecter et remonter une erreur lorsque nécessaire. Ce choix de tests variés est crucial pour garantir la robustesse du compilateur dans différentes situations.

Scripts de test : Les scripts de tests jouent un rôle fondamental au cours du développement du compilateur. Ces scripts permettent des tests partiels du compilateur, offrant la possibilité de cibler spécifiquement certaines étapes de la compilation pour des évaluations plus précises. Par exemple, si des incertitudes subsistent quant à l'étape A (analyse syntaxique), l'utilisation d'un script (*./lexer_test.py*) avec un fichier Deca en tant que paramètre facilite la réalisation de tests ciblés. En plus de ces scripts, d'autres sont disponibles et sont essentiels pour le processus de débogage, comme indiqué dans la Documentation de Validation.

2.2 Dépendances entre Classes

Le diagramme de dépendance entre les classes offre une vue d'ensemble de l'architecture entre les fichiers de la branche "tree", mettant en évidence l'utilisation de l'héritage pour maximiser la factorisation du code. Chaque classe hérite indirectement de la classe de base "Tree". De plus, pour chaque type d'instruction ou de méthode, une classe abstraite est établie, servant de point de départ à d'autres classes spécifiques. Cette approche garantit une organisation claire et modulaire, favorisant la réutilisation du code.



Annexe n°1 : Diagramme de classe

Dans notre architecture, il est essentiel que les différentes étapes puissent communiquer ou du moins échanger des informations. En effet, comme nous l'avons vu, le programme Deca traverse successivement les 3 étapes de compilation. Ces 3 étapes doivent être capables de traiter les mêmes données, sans en perdre, pour que la compilation se fasse sans encombre.

Tout d'abord, l'étape A qui crée l'arbre syntaxique, est utilisée par l'étape B qui va pouvoir le décorer avec l'analyse contextuelle. (avec quelle variables) L'échange est notamment possible avec les variables **compile**, **localEnv** et **currentClass** (parfois seulement **compile**).

Une fois que l'arbre a été enrichi par l'étape B, il est transmis à l'étape C. Cette dernière se charge de le convertir en un fichier assembleur prêt à être exécuté par

IMA. Cette coordination efficace entre les étapes garantit une manipulation cohérente des données tout au long du processus de compilation.

III. Spécifications Additionnelles du Code

3.1 Modifications apportées

Selon le développement du projet, il a été nécessaire d'ajouter de plus en plus de méthodes, d'attributs et même de classes différentes pour le bon fonctionnement du compilateur. Certains d'entre eux étaient triviaux, se limitant à la propagation de l'information à travers l'arbre abstrait. Par exemple, l'ajout de l'attribut `currentClass` dans la classe `Assign` pour lui permettre d'avoir un comportement spécifique en fonction de son emplacement dans le code, que ce soit dans la fonction principale ou dans une méthode.

Par conséquent, la plupart des attributs ajoutés ont suivi cette logique. D'autres exemples incluent les compteurs dans les classes `And` et `Or` pour éviter la répétition des étiquettes utilisées dans les branches. L'ajout de méthodes a été crucial pour déterminer la structure du projet. Pour les étapes B et C, plusieurs fonctions ont été ajoutées pour la vérification et la génération de code, cherchant toujours à abstraire autant que possible et à tirer parti de l'héritage de classe.

Nous avons utilisé la famille de fonctions "verify" pour la vérification et la décoration de l'arbre abstrait, les adaptant à chaque nœud spécifique.

```
/**
 * Pass 2 of [SyntaxeContextuelle]. Verify that the class members (fields and
 * methods) are OK, without looking at method body and field initialization.
 */
1 implementation   ▲ Julian Coux *
protected abstract void verifyClassMembers(DecacCompiler compiler, ClassDefinition superClass,
                                           ClassDefinition classe, int index)
    throws ContextualError;

/**
 * Pass 3 of [SyntaxeContextuelle]. Verify that instructions and expressions
 * contained in the class are OK.
 */
1 implementation   ▲ Julian Coux *
protected abstract void verifyClassBody(DecacCompiler compiler, EnvironmentExp env_exp, ClassDefinition classe)
    throws ContextualError;
```

Annexe n°2 : Fonctions de l'étape B

Pour la plupart des cas, il a suffi d'ajouter les fonctions "verify" pour que l'étape B fonctionne parfaitement. Cependant, pour l'étape C, des modifications supplémentaires ont été nécessaires.

DecacCompiler

Les ajouts à la classe DecacCompiler étaient les suivants :

```
public void addErrorCheck(Label errorLabel) {
```

```
public void addFirst(Instruction instruction) {
```

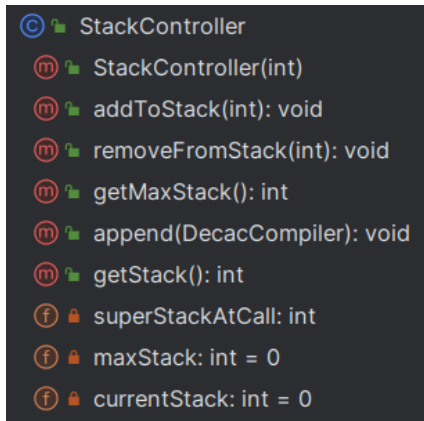
```
7 usages
private StackController stackController;
2 usages  ➤ Breno Morais
public void initStackController(int stack) { stackController = new StackController(stack); }
11 usages  ➤ Breno Morais +1
public void addToStack(int i) { stackController.addToStack(i); }
10 usages  ➤ Breno Morais +1
public void removeFromStack(int i) { stackController.removeFromStack(i); }
4 usages  ➤ Breno Morais +1
public int getMaxStack() { return stackController.getMaxStack(); }
2 usages  ➤ Breno Morais +1
public void append(DecacCompiler compiler) {...}
2 usages  ➤ Breno Morais
public int getStack() { return stackController.getStack(); }
```

Les deux premières fonctions ont pour objectif de fournir un contrôle accru aux autres classes sans perdre la sécurité de la centralisation, nécessaire pour assurer la mise en œuvre correcte des indicateurs. `addErrorCheck` ajoute des instructions de vérification d'erreur en fonction des options du compilateur, tandis que `addFirst` était nécessaire pour ajouter le test de pile pour chaque bloc après le calcul pendant la génération de code. Nous considérons cette solution comme nécessaire, car le nombre d'opérations push et pop dépend fortement des opérations et fonctions utilisées pendant la génération de code.

Pour maintenir ce contrôle avec succès et cohérence, nous avons créé la classe `StackController`, connectée à un compilateur. Comme chaque bloc de code devait avoir son propre test de pile, nous avons décidé de créer un "mini programme" pour chaque bloc, maintenant le contrôle de sa propre pile. Après la génération du code, nous utilisons la fonction `addFirst` pour ajouter les tests de pile avec les valeurs calculées. Enfin, le "mini programme" est incorporé au programme principal à l'aide de la fonction `append` du compilateur. Nous avons opté pour la classe `DecaCompiler` au lieu de `IMAPProgram`, car même si la classe `IMAPProgram` s'adapte mieux au niveau sémantique, la `DecaCompiler` possède des méthodes et des attributs

extrêmement utiles, en plus d'être utilisée par toutes les autres méthodes de l'étape C, permettant son utilisation dans les deux situations.

La classe StackController en elle-même était simple, ne conservant que le niveau actuel de la pile à ce moment-là et enregistrant le maximum utilisé.



```
StackController
StackController(int)
addToStack(int): void
removeFromStack(int): void
getMaxStack(): int
append(DecacCompiler): void
getStack(): int
superStackAtCall: int
maxStack: int = 0
currentStack: int = 0
```

Un problème que nous avons rencontré dans notre code est que ce test ne prend en compte que les opérations de la pile à ce niveau, sans tenir compte des opérations des méthodes appelées. Nous avons envisagé une solution possible, qui consisterait à stocker l'utilisation de la pile de chaque méthode, ainsi que la taille de la pile au moment de l'appel de la méthode, et à ne prendre cette information qu'à la fin et à l'incorporer aux tests. Cependant, en raison du manque de temps, nous n'avons pas pu mettre en œuvre cette solution.

Les seules autres classes créées étaient l'interface BooleanValue et la classe MethodName. L'interface BooleanValue indique que cette classe contient la méthode codeGenNot, utilisée pour générer le code chargeant la valeur inverse de cet objet. La classe MethodName stocke les noms des méthodes ainsi que les noms des classes auxquelles appartiennent ces méthodes, créée pour implémenter l'héritage des méthodes en Deca. Dans ce cas, le nom de la méthode est comparé aux méthodes de la superclasse, ajoutant celles qui ne sont pas écrasées, tout en maintenant toujours la classe dans laquelle cette méthode a été définie.

3.2 Conventions de Codage

Nous avons suivi la convention des Types Abstraits de Données, adoptant la pratique cohérente de définir des classes abstraites et structurer le code de manière à utiliser la superclasse la plus abstraite possible. Cela a abouti à la normalisation de la communication entre les classes. De plus, nous avons opté pour l'utilisation de CamelCase pour nommer les variables, les classes et les méthodes, en privilégiant la lisibilité et l'auto-explicabilité du code.

Il est à noter l'application de la boucle `foreach` dans toutes les itérations sur les collections, étant un choix récurrent. Cependant, dans les cas où il était nécessaire de parcourir la collection en sens inverse, des difficultés ont été rencontrées. Normalement, nous aurions utilisé la méthode `reverse` sur un clone de la liste, mais cette fonction n'était pas disponible dans notre version du JDK, nous obligeant à utiliser une boucle `"for i"` dans certaines situations.

Malheureusement, nous reconnaissons que l'utilisation des casts aurait pu être améliorée. Dans certaines situations, nous avons constaté que des classes en apparence déconnectées auraient dû partager une interface. Cependant, ce besoin n'a été identifié que dans des phases plus avancées du projet, où l'introduction de ces interfaces aurait pu accroître considérablement la complexité du système. Dans ce contexte, nous avons décidé que l'économie de temps et de complexité offerte par l'utilisation de casts serait plus avantageuse que d'atténuer les risques. Naturellement, nous avons mis en œuvre des procédures pour traiter les erreurs de casting, garantissant que toutes les situations soient gérées correctement.

Nous avons adopté le `Logger` comme outil pour créer des messages de débogage et de traçage. Cet outil s'est avéré particulièrement utile lors de l'étape B, lorsque nous nous familiarisons encore avec le fonctionnement et la structure de l'arbre abstrait. À travers le `Logger`, nous avons pu comprendre de manière efficace la dépendance entre les nœuds, en identifiant tous les problèmes et leurs origines.

IV. Perspective d'évolution

4.1 Limites du compilateur

En ce qui concerne les améliorations possibles de notre compilateur, une piste envisageable serait l'optimisation des performances afin d'accroître son efficacité et sa rapidité. Cette optimisation pourrait se concentrer sur divers aspects tels que la génération de code intermédiaire, l'optimisation du code généré, ou encore l'utilisation judicieuse des ressources matérielles.

Parallèlement, une extension significative du compilateur pourrait consister à introduire la prise en charge de nouvelles fonctionnalités du langage. Cela pourrait inclure des aspects tels que la manipulation de chaînes, l'intégration d'une boucle `'for'`, ou encore l'ajout de fonctions spécifiques comme `'readStr()'`. En élargissant ainsi la palette des fonctionnalités, nous permettons aux développeurs d'exprimer leurs idées de manière plus riche et diversifiée.

Une autre piste à explorer serait l'amélioration des options disponibles dans notre compilateur, voire l'introduction de nouvelles fonctionnalités. Cela pourrait inclure des fonctionnalités d'optimisation spécifiques, des paramètres de personnalisation avancés, ou d'autres outils facilitant le processus de développement.

La gestion des erreurs constitue également un aspect crucial à perfectionner. En mettant l'accent sur des messages d'erreur plus explicites et en fournissant des suggestions pertinentes pour corriger les erreurs, nous renforçons l'efficacité du processus de débogage.

Pour garantir la fiabilité du compilateur, une autre direction prometteuse serait d'étendre nos scripts de tests à l'ensemble du processus de compilation. Cela permettrait d'automatiser la validation, assurant ainsi une couverture complète pour toutes les parties du compilateur. Une validation automatique renforcée contribuerait à la robustesse du compilateur, garantissant une qualité constante du code généré.

En somme, ces pistes d'évolution pour notre compilateur visent à renforcer ses performances, enrichir ses fonctionnalités, améliorer sa flexibilité et garantir une fiabilité constante, offrant ainsi un outil de développement plus efficace et complet.

V. Conclusion

Avec la conception complète du compilateur Deca, nous avons établi une base solide pour un développement logiciel efficace et innovant. Cette documentation met en exergue les principes clés de notre approche, soulignant notre engagement envers la performance et la fiabilité.

Nous remercions notre équipe dévouée et encourageons la communauté à continuer d'explorer et de contribuer au succès du compilateur Deca. Nous sommes donc reconnaissant de l'administration de l'Ensimag pour la mise en place de ce projet et pour leur engagement à façonner un avenir où la technologie et la durabilité vont de pair.