Document de conception

GL 03

M. Migné*E. Tellier†A. El Ahmar‡A. Benmoussa§
A. Fuentes¶Y. Amoura‖J. Vandeputte**
F. Constans††

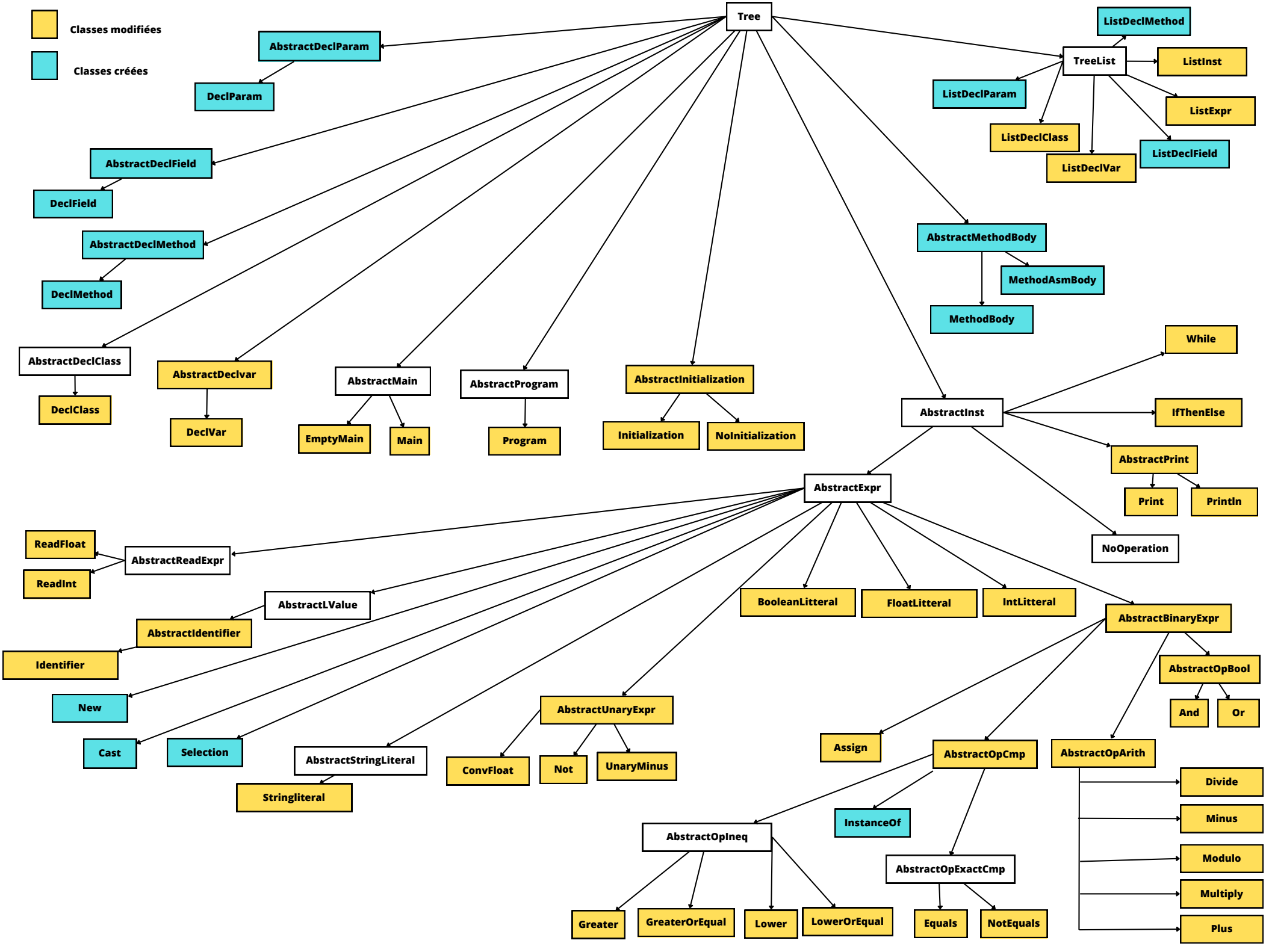
21 juin 2023

*Matthieu.Migne@grenoble-inp.org>
†Emilien.Tellier@grenoble-inp.org
‡Anas.Elahmar@grenoble-inp.org
§Alae.Benmoussa@grenoble-inp.org
¶Aubin.Fuentes@grenoble-inp.org
‖Yanis.Amoura@grenoble-inp.org
**Jules.Vandeputte@grenoble-inp.org
††Florence.Constans@grenoble-inp.org

Table des matières

1 Architecture du compilateur

La figure ci-dessous montre l'architecture des nœuds des classes de l'AST. Les flèches montrent les relations d'héritage, les classes bleues montrent les classes créées et les classes jaunes montrent des classes qui ont été modifiées. Nous avons aussi créé la classe-outil Label Manager, elle permet de générer des étiquettes pour une instruction telle que *if* ou *while*, ainsi que la classe MethodTable pour permettre de faire des accès mémoire en assembleur aux méthodes.


Classes modifiées
Classes créées
Tree
AbstractDeclParam
DeclParam
AbstractDeclField
DeclField
AbstractDeclMethod
DeclMethod
AbstractDeclClass
DeclClass
AbstractDeclVar
DeclVar
AbstractMain
EmptyMain
Main
AbstractProgram
Program
AbstractInitialization
Initialization
NoInitialization
ListDeclMethod
TreeList
ListInst
ListDeclParam
ListExpr
ListDeclClass
ListDeclField
ListDeclVar
AbstractMethodBody
MethodAsmBody
MethodBody
AbstractInst
While
IfThenElse
AbstractPrint
Print
Println
NoOperation
AbstractExpr
ReadFloat
AbstractReadExpr
ReadInt
AbstractLValue
AbstractIdentifier
Identifier
New
Cast
Selection
AbstractStringLiteral
Stringliteral
AbstractUnaryExpr
ConvFloat
Not
UnaryMinus
BooleanLitteral
FloatLitteral
IntLitteral
AbstractBinaryExpr
AbstractOpBool
And
Or
Assign
AbstractOpCmp
AbstractOpArith
Divide
Minus
Modulo
Multiply
Plus
InstanceOf
AbstractOpIneq
AbstractOpExactCmp
Greater
GreaterOrEqual
Lower
LowerOrEqual
Equals
NotEquals

2 Spécifications sur le code du compilateur

2.1 Spécifications des classes représentant des nœuds

Les spécifications du code du compilateur sont les mêmes que rédigées en amont dans le cahier des charges.
On peut tout de même diviser en trois partie le fonctionnement de ce compilateur :

— Etape A : L'analyse lexicale du programme en entrée se fait dans DecaLexer.g4 compilé en Java, présentant le lexique défini dans le cahier des charges. Ensuite, la construction de l'arbre se fait grâce au programme DecaParser.g4 compilé en Java. Ces fichiers existant déjà au départ du projet, nous n'avons fait que les compléter en suivant ce qui avait déjà été fait. L'analyse syntaxique vérifie de manière assez classique la syntaxe du programme et si celle-ci respecte ce qui est défini par le cahier des charges. Par exemple si un opérateur binaire contient bien deux expressions et un unaire n'en a qu'un, ou si un assignement de variable contient bien un identifiant de variable.

— Etape B : L'analyse contextuelle est beaucoup plus poussée que l'analyse syntaxique car elle ajoute certaines contraintes de contexte au programme entrant. En plus de vérifier que pour chaque noeud de l'arbre syntaxique on utilise les bons types, et pour indiquer quelle est l'erreur de contexte lors d'une comparaison entre une chaîne de caractère et un entier par exemple. Cette étape vérifie aussi les tests de précondition définis dans le cahier des charges comme la division par 0.
Cette analyse contextuelle est réalisée dans les surcharges de la méthode "verifyExpr".

— Etape C : La génération de code a demandé de modifier toutes les classes finales (qui n'ont pas de classe enfant). Globalement le code écrit a été placé le plus haut possible dans l'architecture présentée dans le schéma ci-dessus, de manière à ne pas répéter le même code dans des classes ayant un parent commun. Ainsi certaines parties de l'architecture, notamment la partie concernant les opérations arithmétiques n'ont quasiment aucune génération de code dans leur classe finale, tout étant fait en appelant la méthode de génération de code du parent. Certaines opérations étant plus singulières ont demandé une attention particulière pour définir les cas et effets sur la génération de code IMA. C'est le cas par exemple pour Modulo pour la partie arithmétique et pour OR qui a demandé plus de réflexion que de développement.
Cette génération de code est réalisée dans les méthodes par codeGen*, avec * variant selon la branches de l'architecture où se trouve le nœud en cours.

2.2 Spécifications des classes utilitaires

En plus de l'architecture des nœuds de l'arbre syntaxique, nous avons également mis en place une classe utilitaire appelée LabelManager, qui nous permet de générer des étiquettes en fonction de l'instruction en cours de traitement. Cette classe agit comme un service, offrant différentes méthodes en fonction de la valeur attendue. Nous avons ajouté plusieurs accesseurs à la classe DecacCompiler afin d'obtenir des variables clés nécessaires au bon fonctionnement du compilateur. Par exemple, pour différencier les étiquettes lorsqu'il y a deux nœuds du même type dans le programme d'entrée, nous avons introduit un compteur incrémenté qui est ajouté à l'étiquette, garantissant ainsi la génération d'une étiquette unique à chaque fois. De plus, le nombre maximal de registres obtenu à partir de la classe de gestion des options du compilateur (qui a également été modifiée) peut être accessible de manière globale par le compilateur. Cette information est particulièrement utile lors des opérations arithmétiques pour déterminer si nous avons dépassé le nombre d'opérations intermédiaires autorisées, ce qui nécessiterait l'utilisation de la pile.
En complément des fonctionnalités existantes, nous avons développé une classe utilitaire appelée MethodTable qui permet de gérer une table des méthodes. Cette classe est utilisée pour associer des définitions de méthodes à des étiquettes correspondantes. La classe MethodTable est conçue pour faciliter la gestion des définitions de méthodes et de leurs étiquettes correspondantes. Elle offre des fonctionnalités pratiques pour accéder rapidement aux étiquettes des méthodes lors de la génération du code. Cette classe est utilisée en tant que composant essentiel dans l'implémentation de la partie objet de notre compilateur.
De manière générale, ces classes utilitaires sont conçues pour gérer les paramètres globaux ou fournir des services qui ne font pas partie intégrante des fonctionnalités du compilateur. En déléguant ces responsabilités à des classes dédiées, nous garantissons une conception modulaire et organisée.

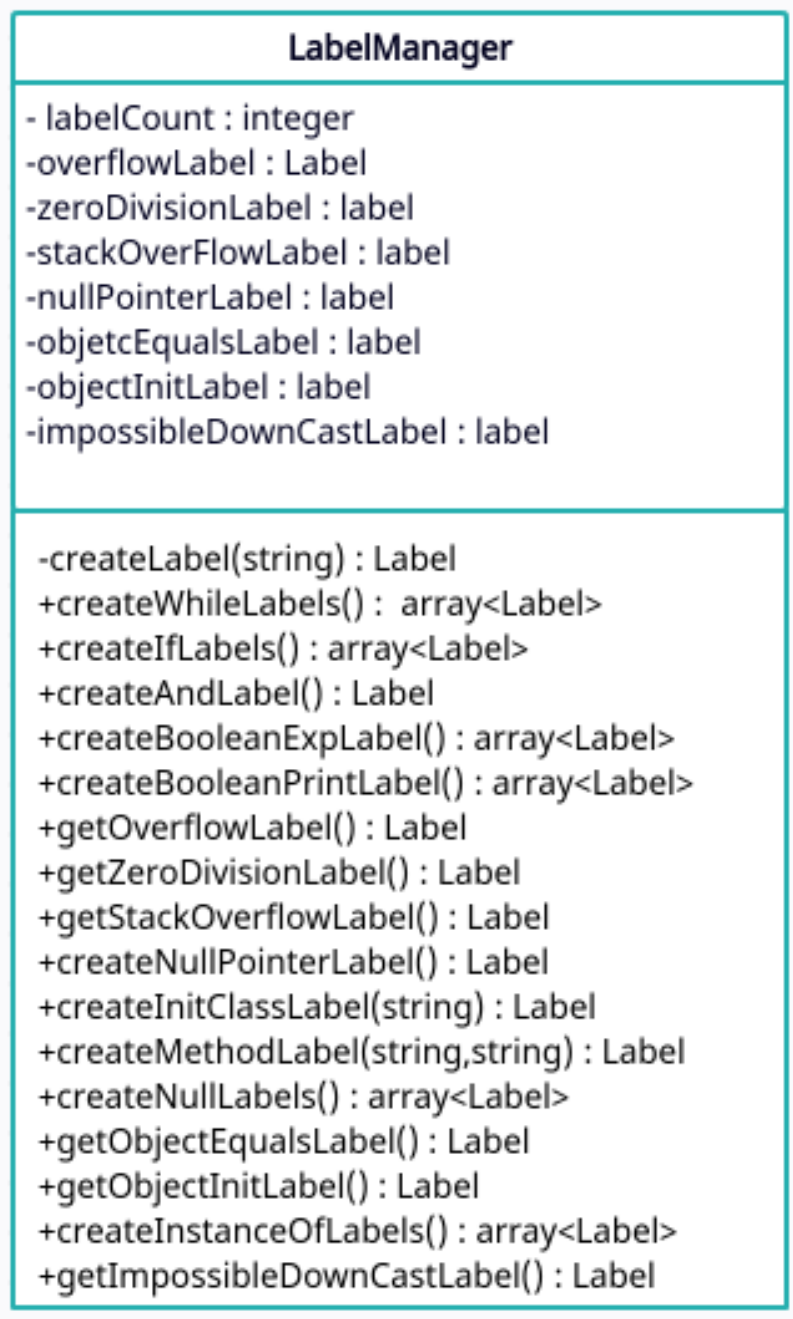


FIGURE 1 – Diagramme de classe de la classe-utilitaire LabelManager

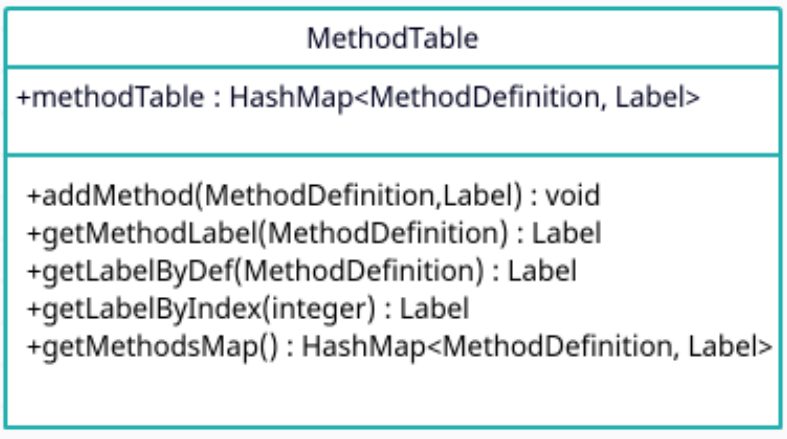


FIGURE 2 – Diagramme de classe de la classe-utilitaire MethodTable

3 Algorithmes et structures de données

3.1 Partie sans objet

Pour la première partie du compilateur (partie sans objet), nous avons implémenté les fonctionnailtés de base du compilateur.

L'implémentation des variables correspond à une grande partie du travail effectué (elles peuvent être déclarée...). L'assignation est aussi rendu possible.

Enfin, nous avons aussi implémentés les opérations arithmetiques et logiques.

3.2 Partie objet

Voici une documentation de conception pour l'implémentation de la partie objet du compilateur du langage Deca :

La partie objet du compilateur Deca étend la fonctionnalité de base en prenant en charge les concepts de la programmation orientée objet, tels que les classes, les objets, l'héritage et les méthodes. Voici quelques points clés concernant l'implémentation de la partie objet :

1. Utilisation d'une HashMap : Pour gérer les expressions et les associer à des étiquettes (labels), nous utilisons une structure de données HashMap. Cette structure nous permet d'effectuer des recherches efficaces et de récupérer rapidement les expressions correspondantes à partir de leurs étiquettes.

2. Gestion des registres : Dans la partie objet, nous utilisons tous les registres de 0 à 16. Les valeurs retournées par les méthodes sont stockées dans le registre R0, tandis que le registre R2 est réservé pour stocker la valeur de retour lorsqu'une méthode se termine.

3. Utilisation de push et pop réguliers : Pour conserver les registres dont nous avons besoin pour les comparaisons ultérieures, nous utilisons régulièrement les instructions push et pop. Ces instructions nous permettent de sauvegarder les registres sur la pile et de les récupérer ultérieurement lorsque nous en avons besoin.

Ces points clés représentent certains aspects importants de l'implémentation de la partie objet du compilateur Deca. Ils garantissent une gestion efficace des expressions, une utilisation optimale des registres et la préservation des valeurs de retour lors de l'exécution des méthodes.