Document de conception

GL 03

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

12 juin 2023

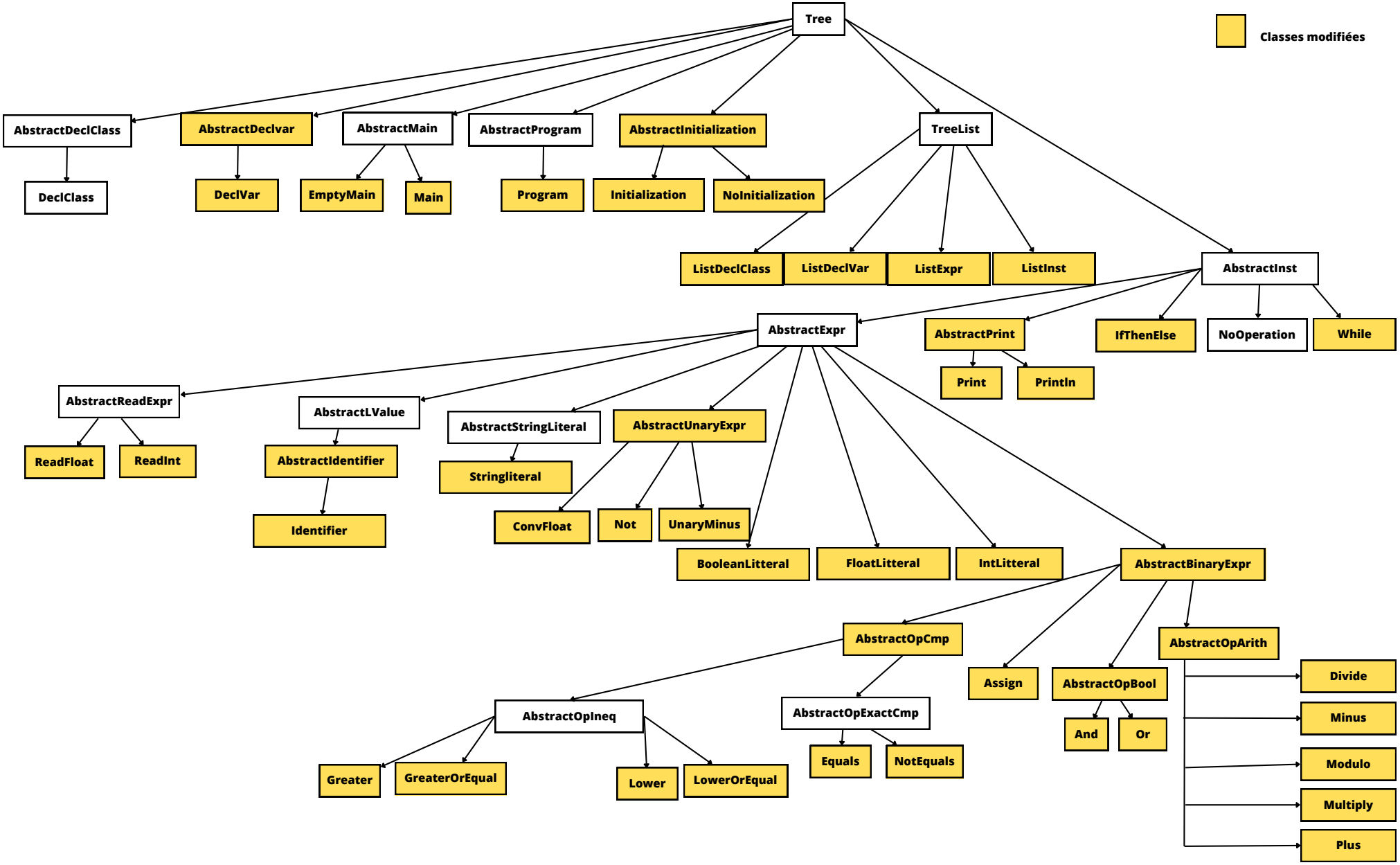
*Matthieu.Migne@grenoble-inp.org>
†Emilien.Tellier@grenoble-inp.org
‡Anas.Elahmar@grenoble-inp.org
§Alae.Benmoussa@grenoble-inp.org
¶Aubin.Fuentes@grenoble-inp.org
‖Yanis.Amoura@grenoble-inp.org
**Jules.Vandeputte@grenoble-inp.org
††Florence.Constans@grenoble-inp.org

Table des matières

Cette documentation concerne uniquement Deca sans objet pour le rendu intermédiaire 2

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, et les classes jaunes montrent des classes qui ont été modifiées. Pour deca sans objet nous n'avons créé aucune classe correspondant à un nœud de l'AST. Seule la classe-outil Label Manager a été créée, elle permet de générer des étiquettes pour une instruction telle que if ou while.


Classes modifiées
Tree
AbstractDeclClass
AbstractDeclVar
AbstractMain
AbstractProgram
AbstractInitialization
TreeList
DeclClass
DeclVar
EmptyMain
Main
Program
Initialization
NoInitialization
ListDeclClass
ListDeclVar
ListExpr
ListInst
AbstractInst
AbstractExpr
AbstractPrint
IfThenElse
NoOperation
While
Print
Println
AbstractReadExpr
AbstractLValue
AbstractStringLiteral
AbstractUnaryExpr
ReadFloat
ReadInt
AbstractIdentifier
Stringliteral
Identifier
ConvFloat
Not
UnaryMinus
BooleanLitteral
FloatLitteral
IntLitteral
AbstractBinaryExpr
AbstractOpCmp
AbstractOpArith
Assign
AbstractOpBool
Divide
Minus
Modulo
Multiply
Plus
AbstractOpIneq
AbstractOpExactCmp
And
Or
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

De manière externe à l'architecture des nœuds de l'arbre syntaxique, nous avons également une classe d'outil créée LabelManager qui nous permet de générer des labels selon l'instruction en cours de génération. Cette classe fonctionne comme un service en proposant différentes méthodes selon la valeur attendue. Différents accesseurs ont été ajouté dans la classe DecacCompiler pour obtenir des variables clés du compilateur pour fonctionner. Par exemple pour différencier les labels si l'on a deux fois un nœud de même type dans le programme d'entrée, on a ajouté un compteur incrémenté qui est collé au label pour en générer un unique à chaque fois. Aussi le nombre de registres maximal obtenu par la classe de gestion des options du compilateur (elle-aussi modifiée) est obtenu par un accesseur global à tout le compilateur. On en a en effet besoin lors des opérations arithmétiques pour savoir si l'on a dépassé ce nombre d'opérations intermédiaires et qu'il faut donc utiliser la pile.
De manière générale, ces classes "outils" sont utiles pour des paramètres globaux ou pour des services qui ne sont pas des fonctionnalités intrinsèques au compilateur et qui sont donc délégués

3 Algorithmes et structures de données