Equipe 27 - Mathis Lavigne, Adrien Kaufman, Berkan Kesman, Quentin Cousin, Christophe Ehret

Documentation de conception

Projet GL - Groupe 5 - Equipe 27

1. Architecture du projet

1.1. Package fr.ensimag.deca.syntax :

Ce package traite l'étape A du compilateur :

1.1.1. AbstractDecaLexer :

L'analyse lexicale représentée par *AbstractDecaLexer* transforme la suite de caractères du fichier d'entrée en suite de lexèmes. Ces lexèmes sont reconnus conformément à ce qui spécifié dans le fichier DecaLexer.g4 (répertoire src/main/antrlr4/fr/ensimag/deca/syntax)

1.1.2. AbstractDecaParser :

L'analyse syntaxique représentée par *AbstractDecaParser* transforme la suite de lexèmes en un arbre de syntaxe abstraite (tree). Ces suites de lexèmes sont transformées conformément aux règles définies dans le fichier DecaParser.g4 (même répertoire que DecaLexer.g4). Des exceptions peuvent être levées : *CircularInclude* pour une inclusion circulaire, *IntegerTooLarge* pour un entier trop grand, *InvalidLValue* pour une affectation incompatible, etc …

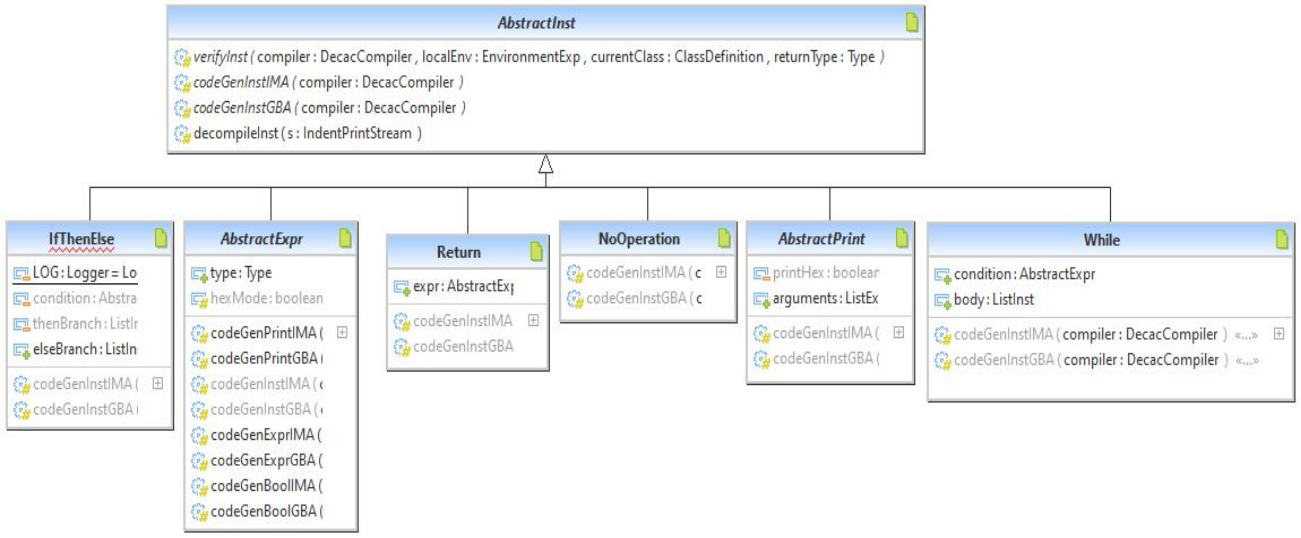
1.2. Package fr.ensimag.deca.tree :

Ce package permet de créer l'arbre de syntaxe abstraite, puis de la décorer afin de la transformer en une suite d'instructions assembleur. La classe Tree est la classe de base de ce package.

Nous allons commencer par décrire la classe *AbstractInst* et les classes qui en dérivent :

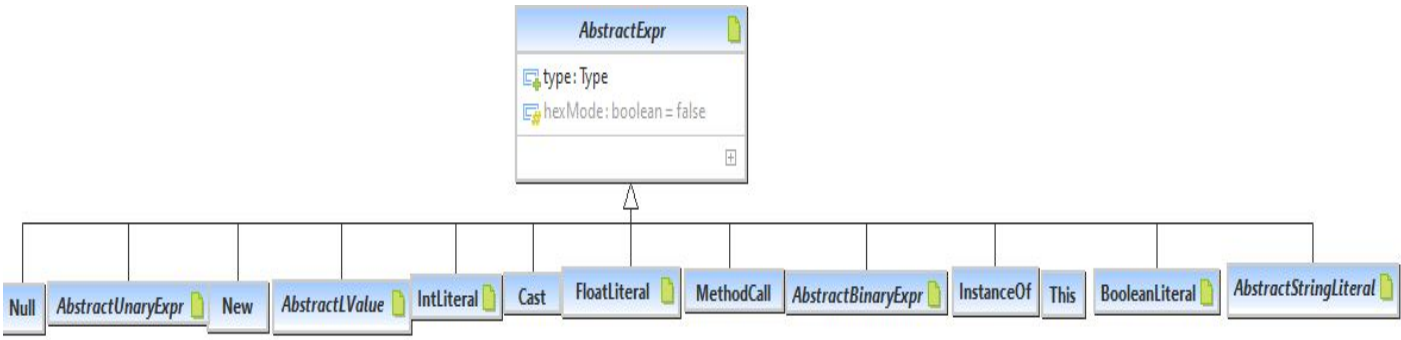
1.2.1. AbstractInst :

AbstractInst est la classe qui permet de représenter l'ensemble des instructions du programme. Plusieurs classes héritent de *AbstractInst* : *While*, *AbstractExpr*, *Return*, *AbstractPrint*, *NoOperation*, *IfThenElse*.



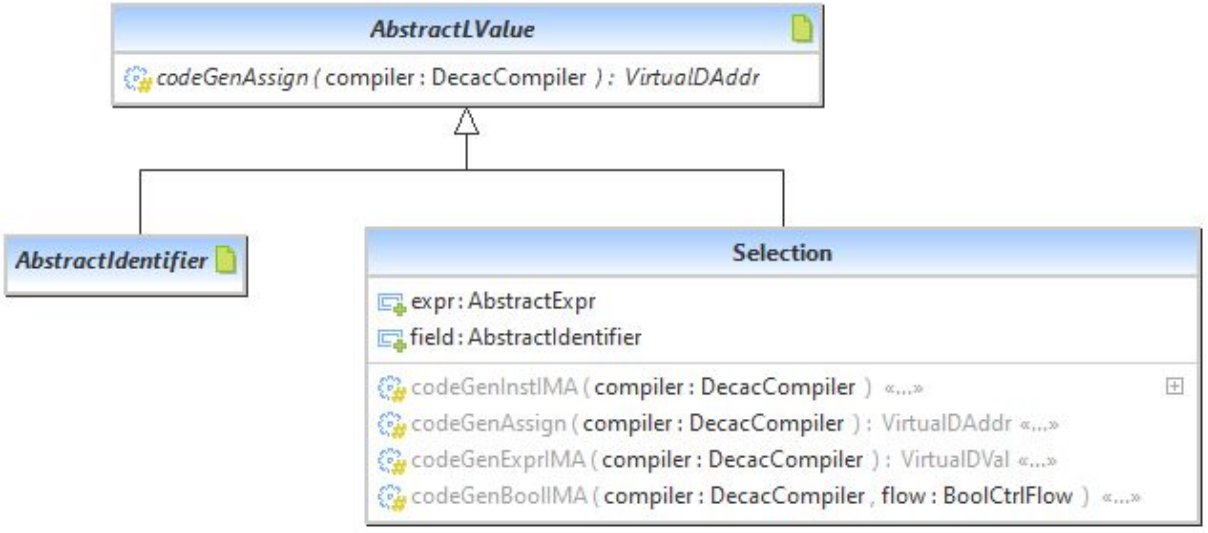
1.2.2. AbstractExpr :

Cette classe très générale, permet de représenter toutes les expressions du langage deca :



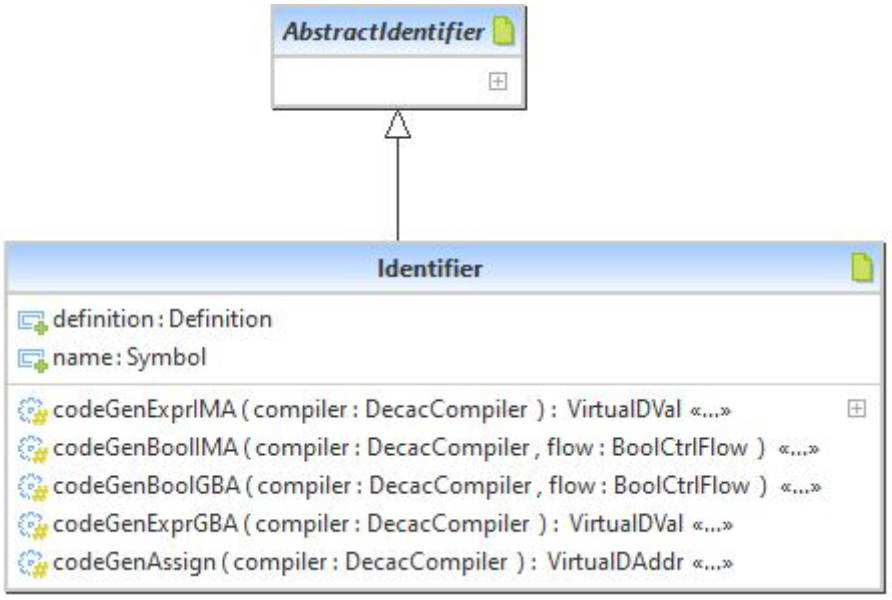
1.2.3. AbstractLValue :

Cette classe représente le côté de gauche d'une affectation. Deux classes héritent de *AbstractLValue* : *AbstractIdentifier* (identificateur) et *Selection* (pour une selection d'un champ ou d'une méthode).



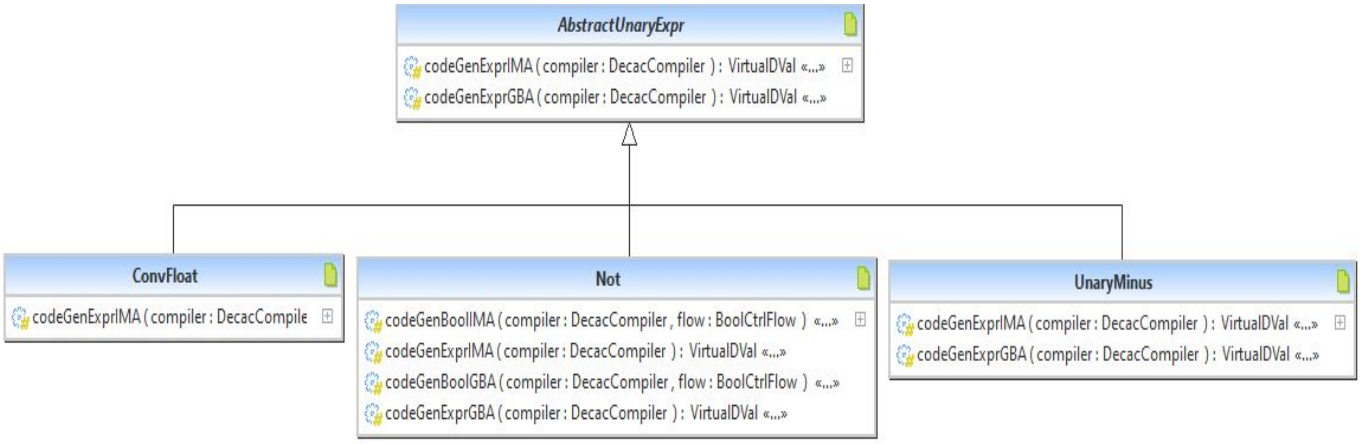
1.2.4. *AbstractIdentifier :*

AbstractIdentifier correspond à un identificateur. Sa classe concrète possède deux attributs : une classe *Symbol* qui correspond tout simplement à une chaîne de caractère ainsi qu'une classe *Définition* donnant des informations (comme le type) sur l'identificateur.



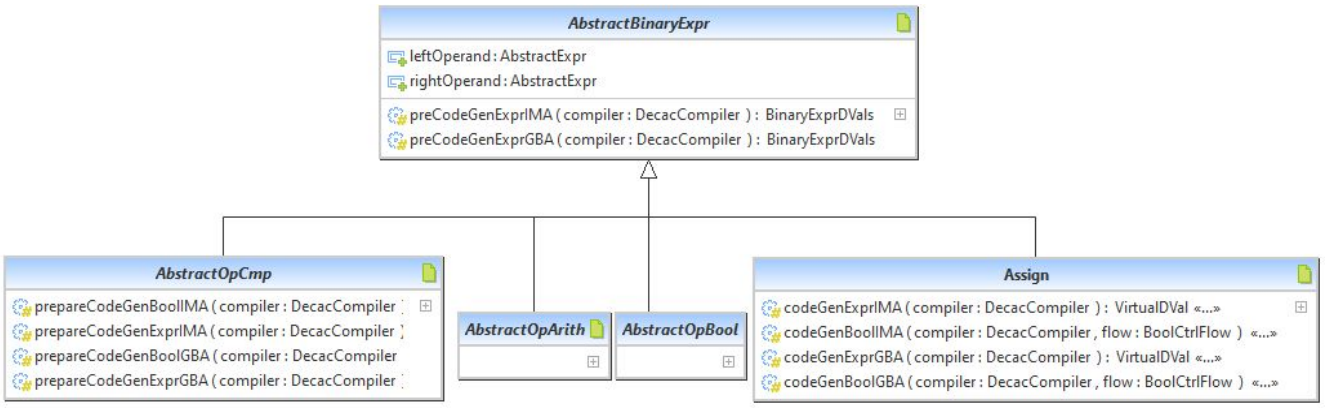
1.2.5. *AbstractUnaryExpr* :

Cette classe qui hérite de *AbstractExpr* représente les expressions unaires (*UnaryMinus*, *Not*, *ConvFloat*).



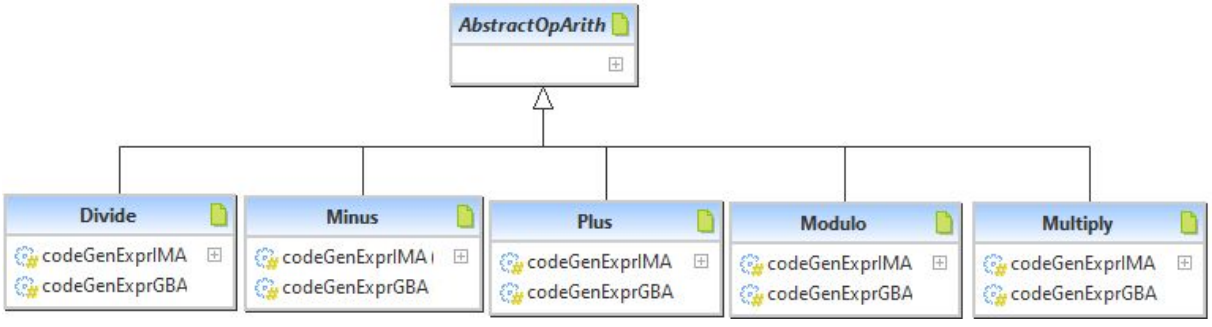
1.2.6. AbstractBinaryExpr :

Cette classe qui hérite de *AbstractExpr* permet de représenter les expressions binaires (les opérations arithmétiques, booléenne, les comparaisons ainsi que les affectations). Elle possède deux attributs de type *AbstractExpr* qui représentent l'opérande gauche et l'opérande de droite.



1.2.7. AbstractOpArith :

Cette classe représente les opérations arithmétiques (somme, différence, division, modulo, multiplication).

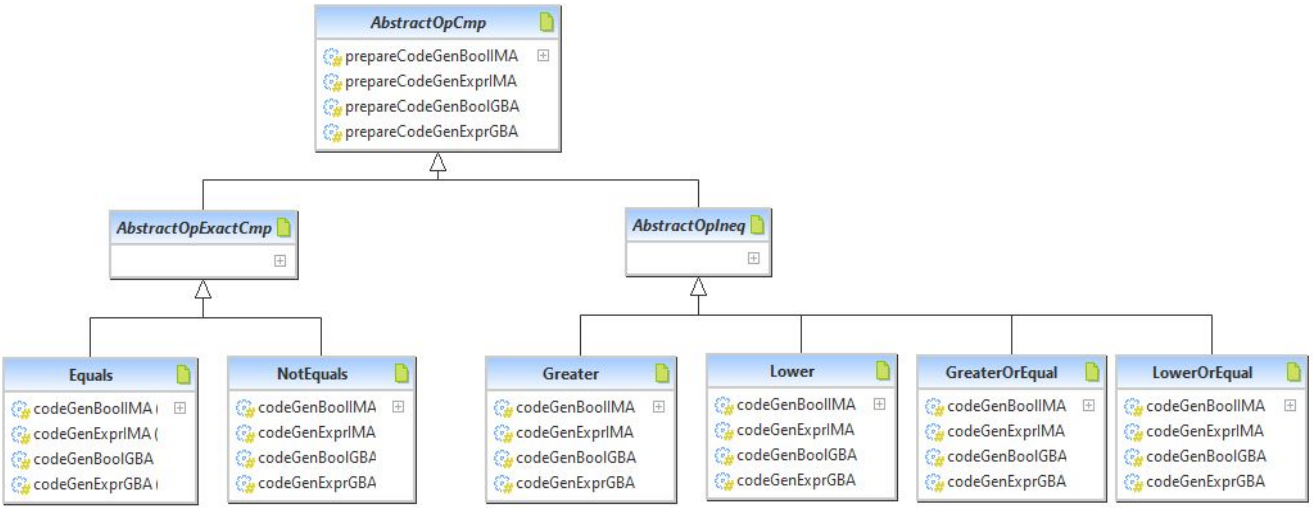


1.2.8. AbstractOpCmp :

Cette classe représente toutes les opérations de comparaison :

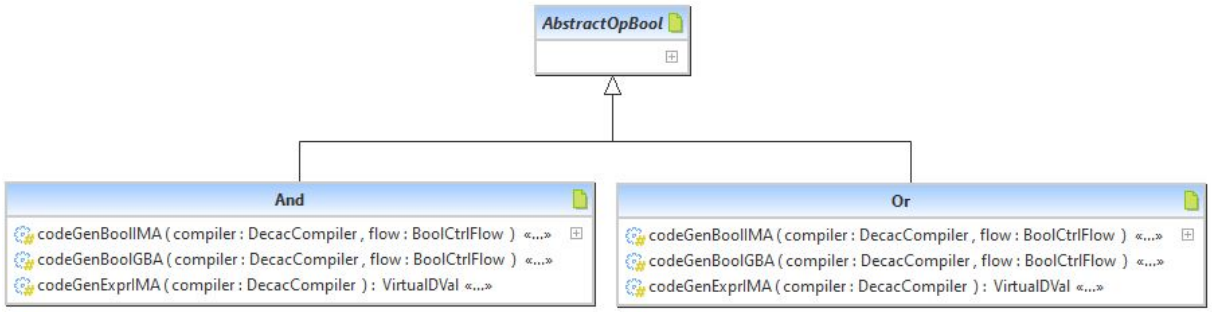
- égalité (*AbstractOpExactCmp* dont héritent Equals et *NotEquals*)

- inégalité(*AbstractOpIneq* dont héritent *Greater*, *GreaterOrEqual*, *Lower*, *LowerOrEqual*)



1.2.9. AbstractOpBool :

Cette classe représente les opérations booléennes AND et OR.

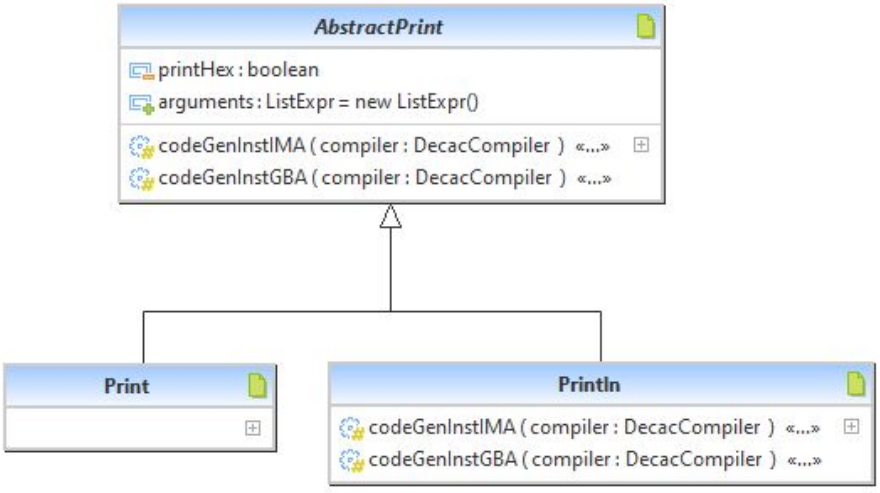


1.2.10. AbstractReadExpr :

Cette classe permet de lire un entier (*ReadInt*) ou un float(*ReadFloat*) au clavier.

1.2.11. AbstractPrint :

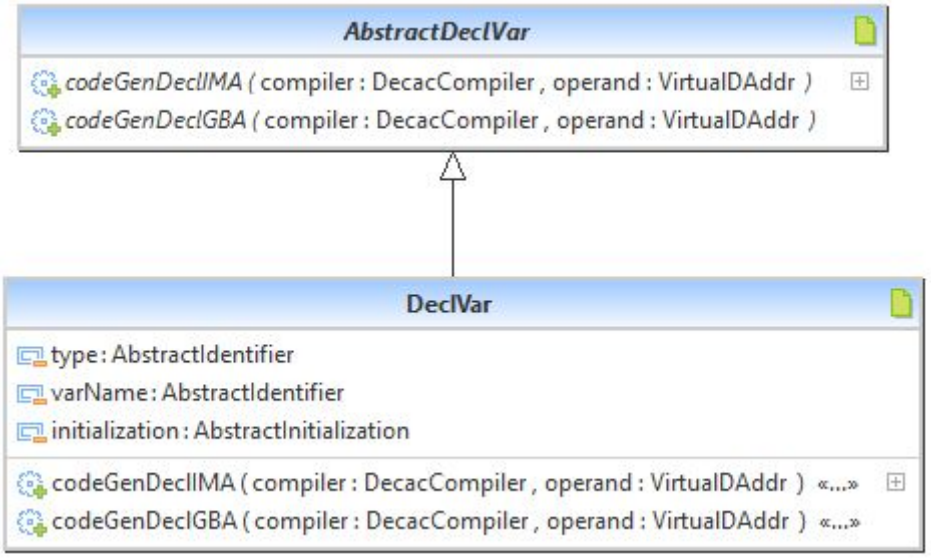
Cette classe permet l'affichage à l'écran. Elle possède deux attributs : un booléen qui affiche en hexadécimal s'il vaut true, et une liste des expressions à afficher. Deux classes héritent de *AbstractPrint*: *Print* et *Println* (pour un retour à la ligne) :



1.2.12. AbstractDeclVar :

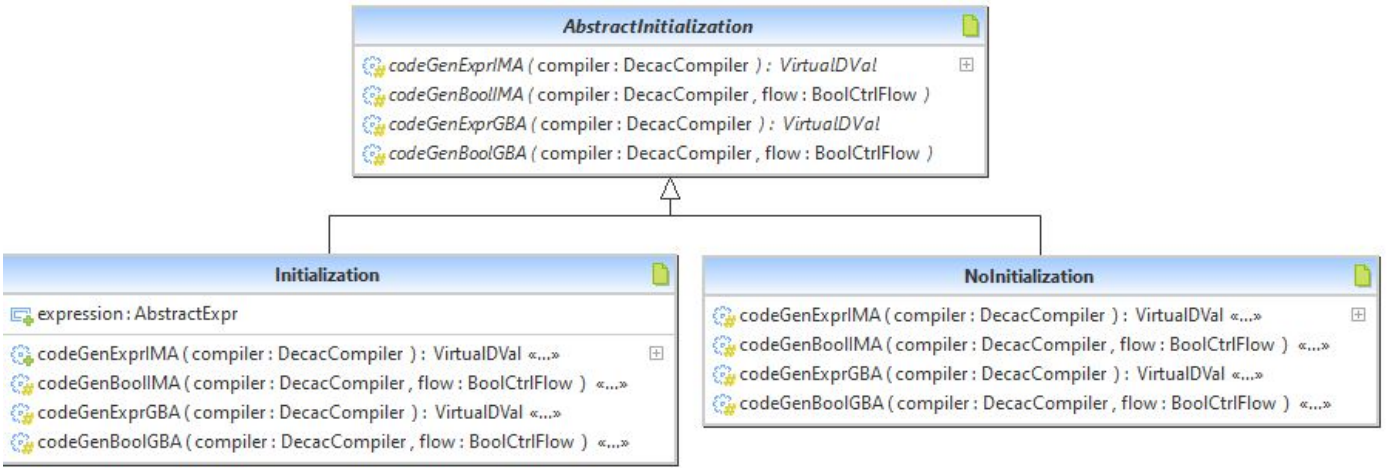
AbstractDeclVar est la classe qui permet de déclarer une variable. Il existe aussi *ListDeclVar* pour représenter une liste de déclaration de variables. La classe concrète qui hérite de *AbstractDeclVar* possède plusieurs attributs :

- *AbstractIdentifier* représentant le type de la variable.
- *AbstractIdentifier* représentant le nom de la variable.
- *AbstractInitialization* représentant l'initialisation de la variable.



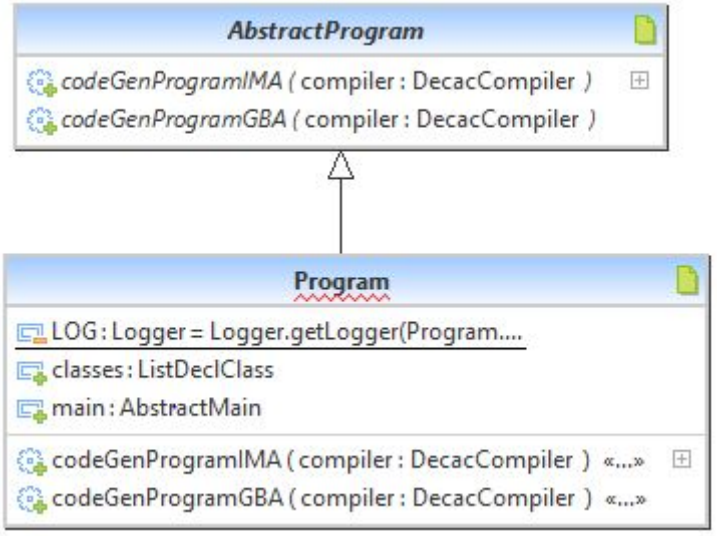
1.2.13. AbstractInitialization :

Cette classe permet de potentiellement initialiser un attribut ou une variable. Deux classes concrètes héritent de *AbstractInitialisation* : *Initialization* et *NoInitialization*.



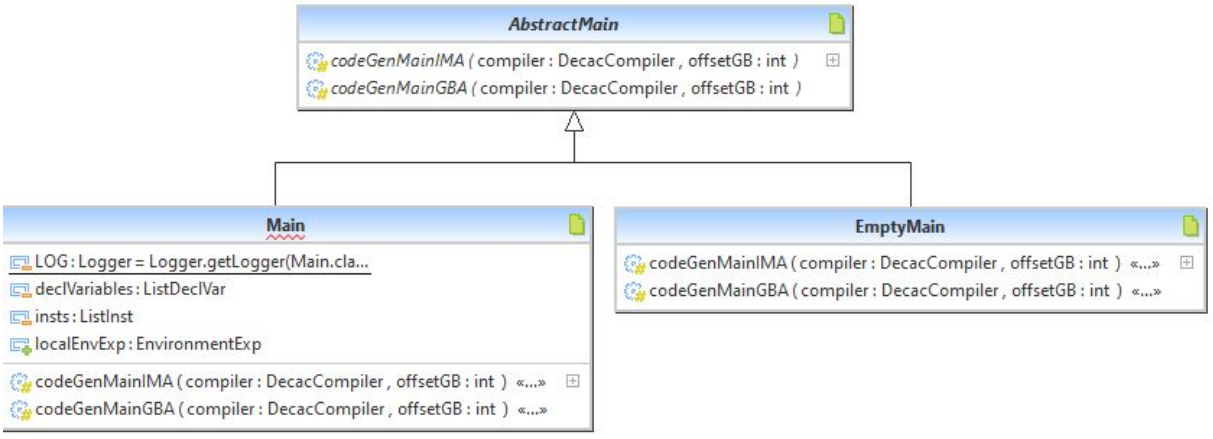
1.2.14. AbstractProgram :

C'est la classe à partir de laquelle commence le programme. La classe concrète *Program* possède deux attributs, conformément à la syntaxe abstraite : *ListDeclClass* et *AbstractMain*



1.2.15. AbstractMain :

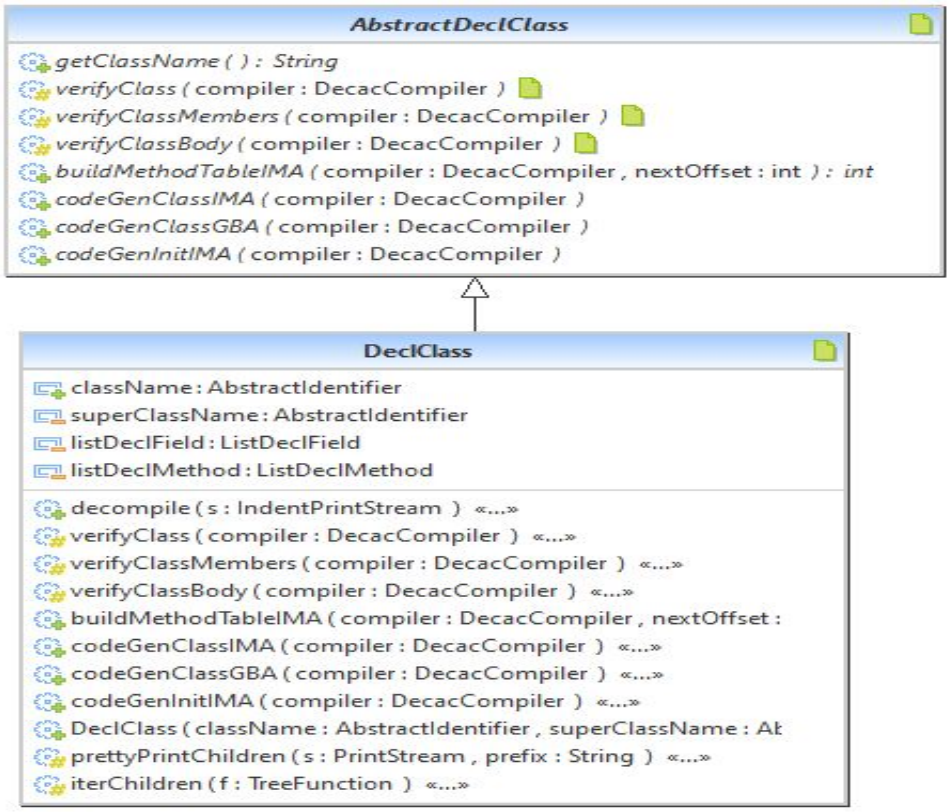
C'est d'ici que commence le programme principal. La classe *Main* possède trois attributs : une liste de déclaration de variable, une liste d'instruction ainsi qu'un environnement d'expression.



1.2.16. AbstractDeclClass :

AbstractDeclClass est la classe qui permet de déclarer des classes. Il existe aussi *ListDeclClass* pour représenter une liste de déclaration de classes. La classe concrète qui hérite de *AbstractDeclClass* possèdent plusieurs attributs :

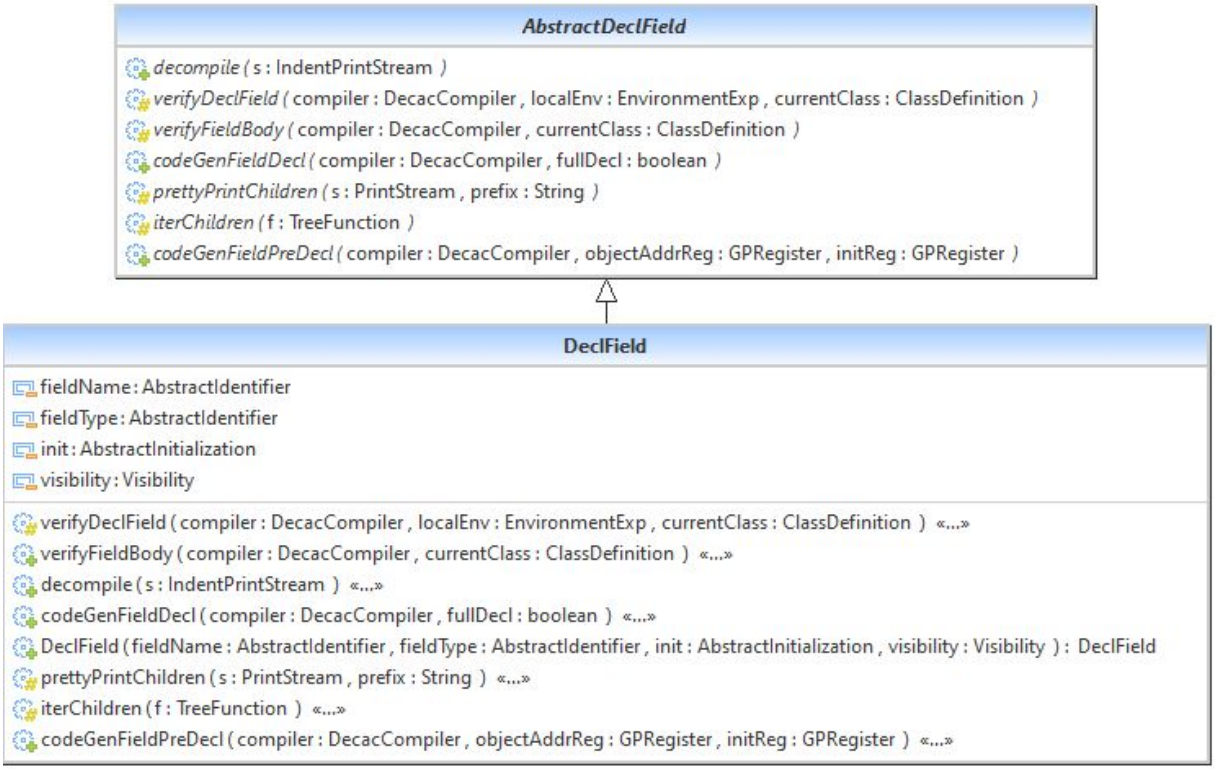
- *AbstractIdentifier* représentant le nom de la classe.
- *AbstractIdentifier* représentant le nom de la super classe.
- *ListDeclField* représentant l'ensemble des champs de la classe.
- *ListDeclMethod* représentant l'ensemble des méthodes de la classe.



1.2.17. AbstractDeclField :

AbstractDeclField est la classe qui permet de déclarer des champs. Il existe aussi *ListDeclField* pour représenter une liste de déclaration de champs. La classe concrète qui hérite de *AbstractDeclField* possèdent plusieurs attributs :

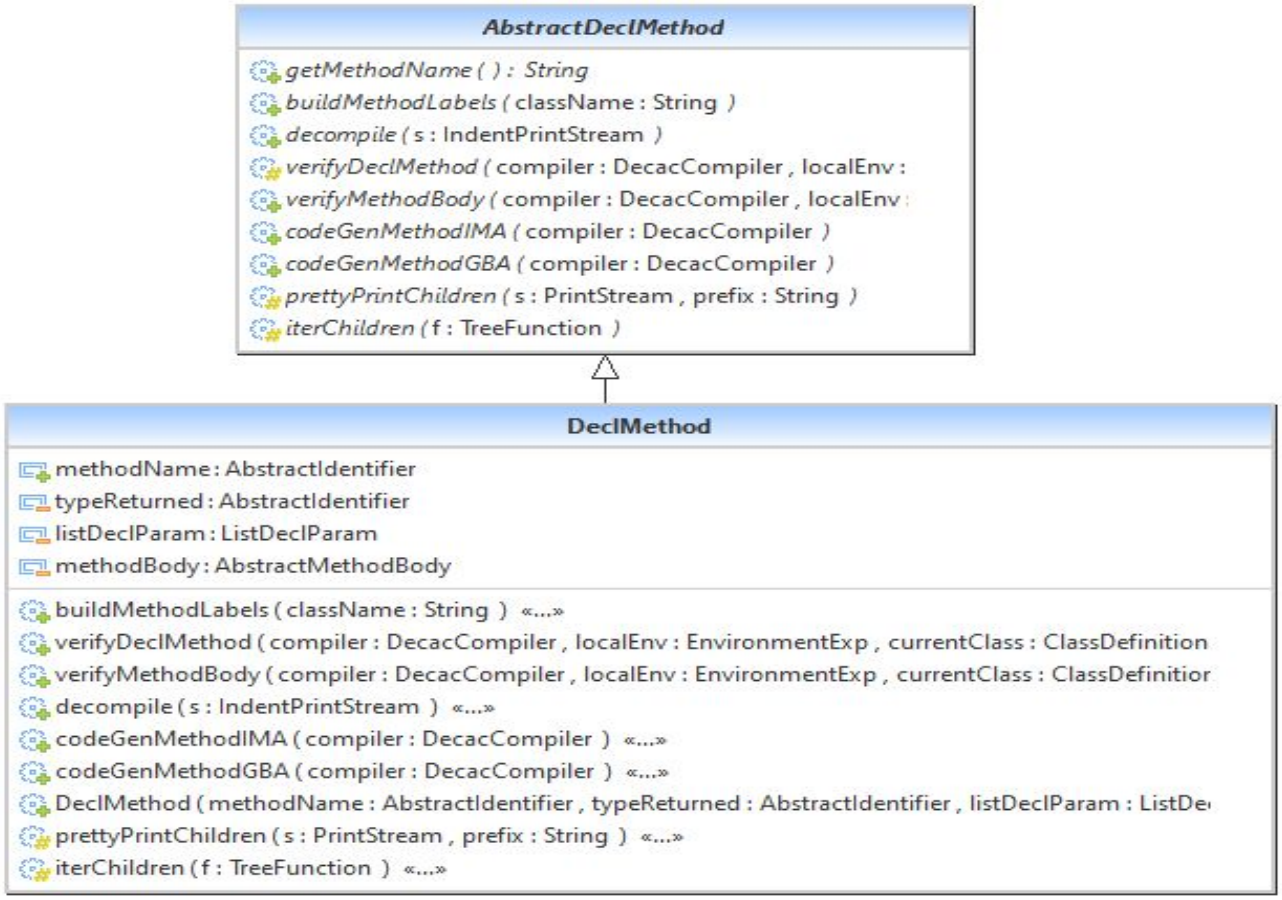
- *AbstractIdentifier* représentant le nom du champ
- *AbstractIdentifier* représentant le type du champ
- *AbstractInitialization* représentant l'initialisation du champ
- *Visibility* pour la visibilité du champ



1.2.18. AbstractDeclMethod :

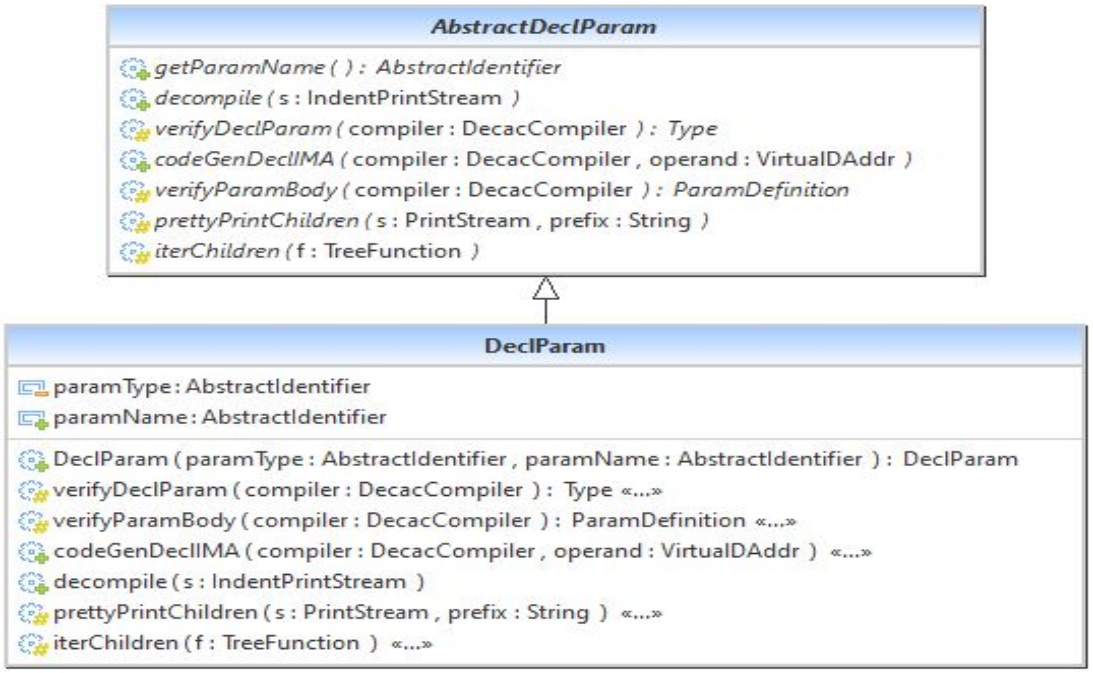
AbstractDeclMethod est la classe qui permet de déclarer des méthodes. Il existe aussi *ListDeclMethod* pour représenter une liste de déclaration de méthodes. La classe concrète qui hérite de *AbstractDeclMethod* possèdent plusieurs attributs :

- *AbstractIdentifier* représentant le nom de la méthode
- *AbstractIdentifier* représentant le nom du type retourné
- *ListDeclParam* représentant l'ensemble des paramètres
- *AbstractMethodBody* représentant le corps de la méthode



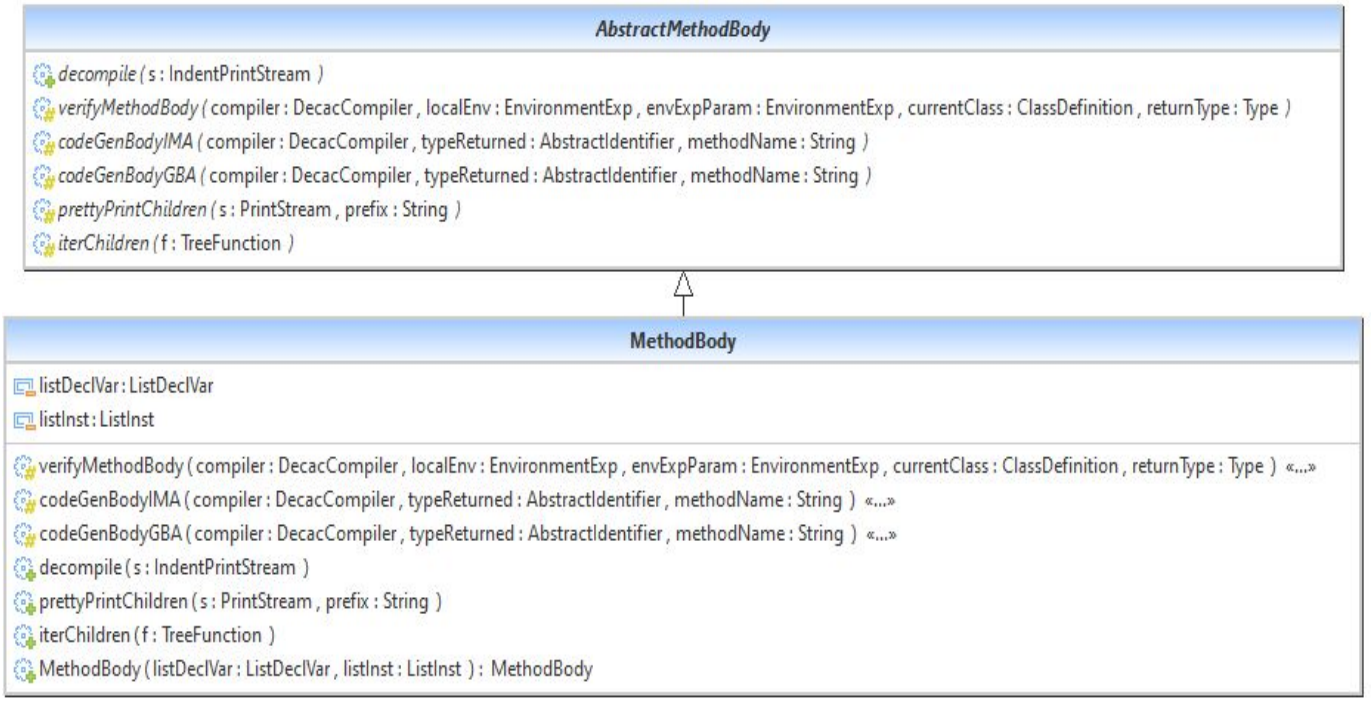
1.2.19. AbstractDeclParam :

AbstractDeclParam est la classe qui permet de déclarer des paramètres. Elle possède deux attributs : un identificateur de type ainsi qu'un identificateur de nom. Il existe aussi *ListDeclParam* pour représenter une liste de paramètres.



1.2.20. *AbstractMethodBody* :

Cette classe permet de représenter le corps d'une méthode. La classe qui hérite de *AbstractMethodBody* possède deux attributs : une liste de variable et une liste d'instruction.



1.2.21. *MethodAsm* :

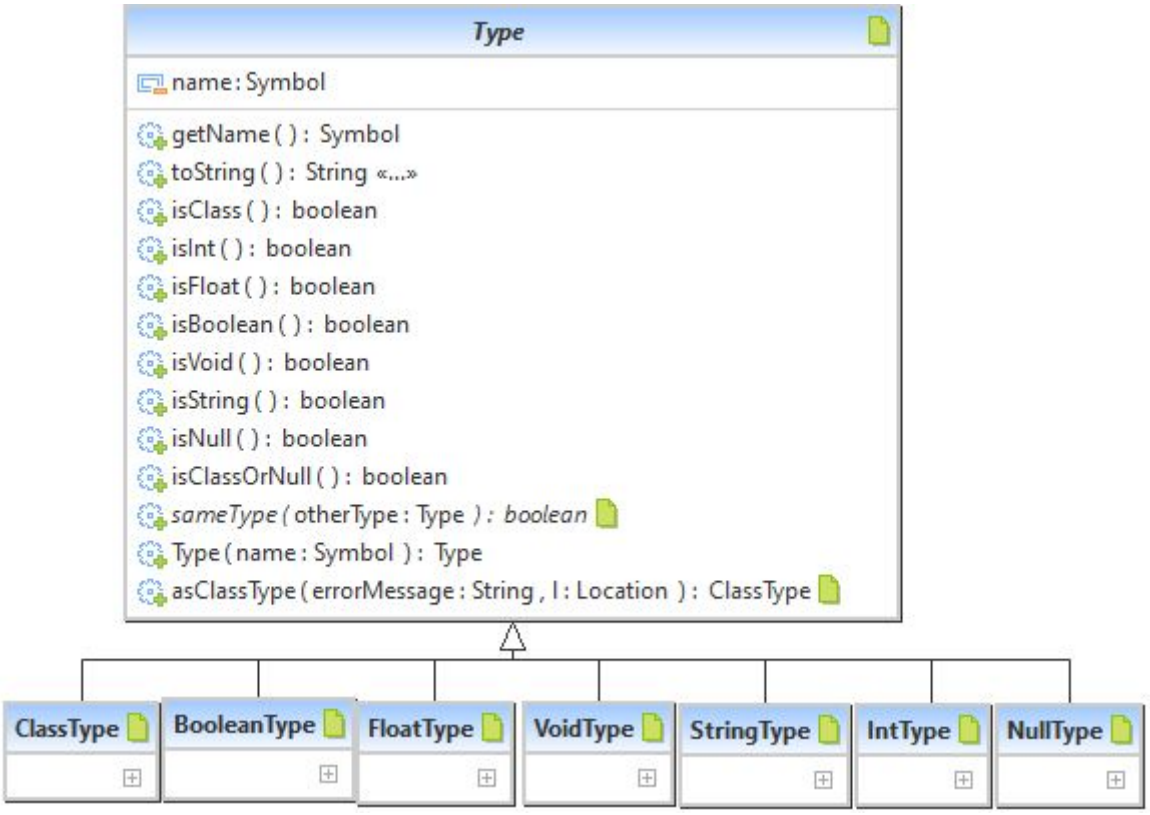
Cette classe permet d'écrire une méthode en Asm. Elle possède un attribut de type String.

1.3. *Package fr.ensimag.deca.context :*

Ce package contient toutes les classes nécessaires à l'étape B du compilateur : l'analyse contextuelle.

1.3.1. *Type :*

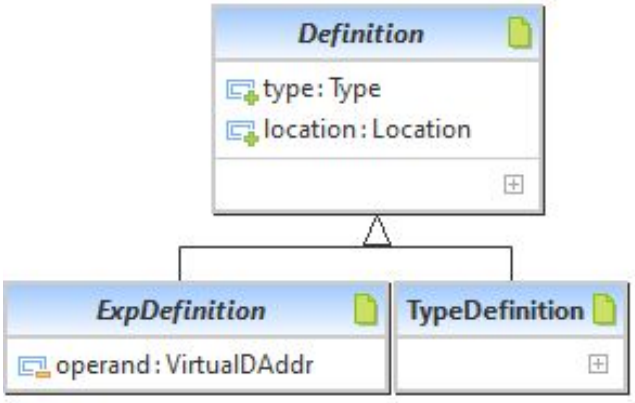
La classe *Type* permet d'ajouter des informations de typage aux classes du package tree. Elle contient un attribut Symbol qui correspond à une chaine de caractère représentant le type. Plusieurs classes héritent de Type : *BooleanType* représentant le type booléan, *ClassType* représentant le type class, *FloatType* représentant le type float , etc … :



1.3.2. Definition :

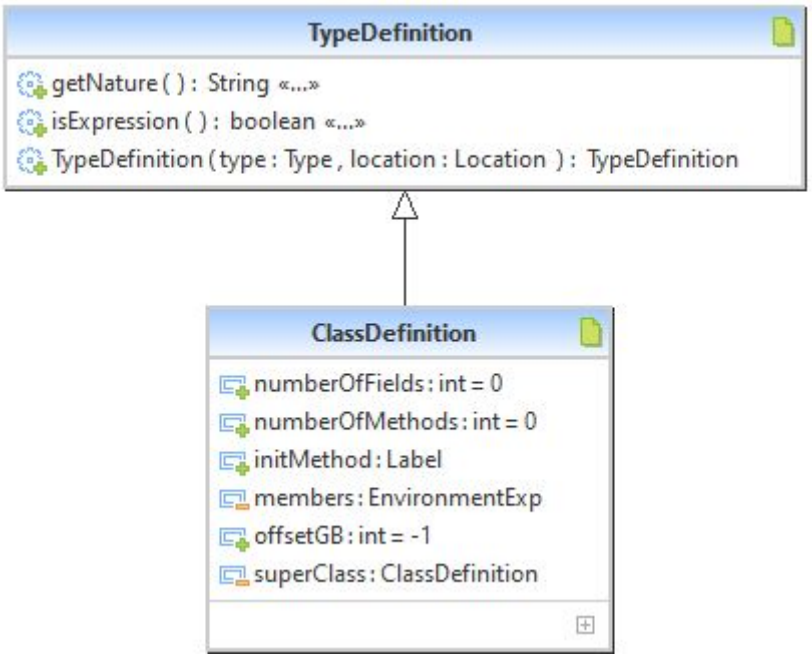
La classe *Definition* permet de donner une définition à un identificateur (définition de types, de variables, de classes, etc …)

Deux classes héritent de *Definition* : *ExpDefinition* et *TypeDefinition*.



1.3.3. TypeDefinition :

La classe *TypeDefinition* correspond à la définition de type. Il y'a une classe qui hérite de *TypeDefinition* : *ClassDefinition*



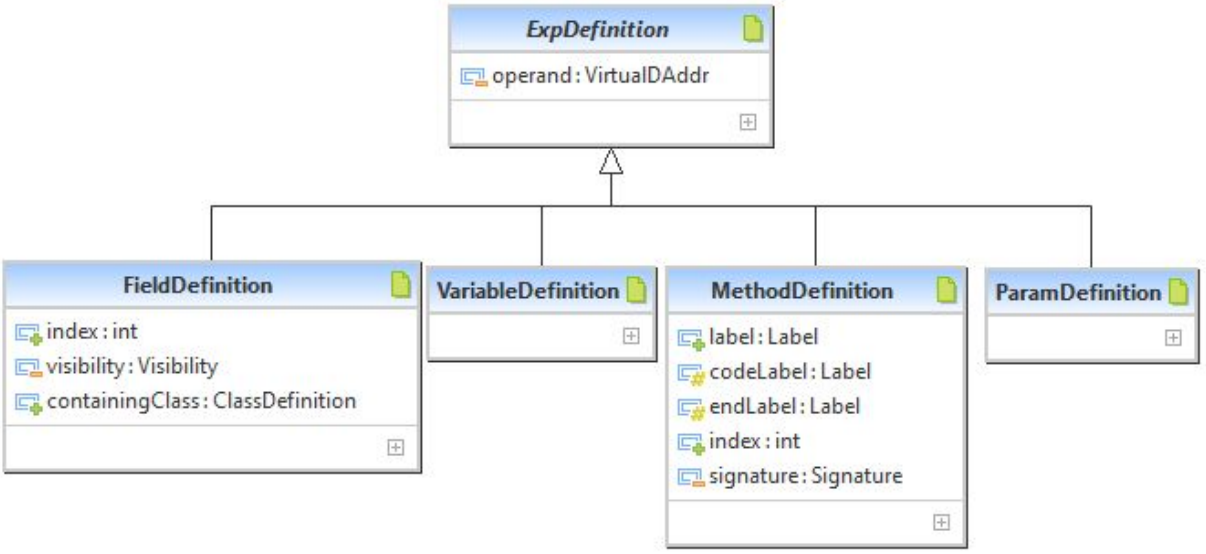
1.3.4. ClassDefinition :

ClassDefinition correspond à la définition d'une classe.

1.3.5. ExpDefinition :

ExpDefinition correspond à la définition d'une expression. Plusieurs classes héritent de *ExpDefinition* :

- *FieldDefinition* (correspond à la définition d'un champ)
- *MethodDefinition* (correspond à la définition d'une méthode)
- *ParamDefinition* (correspond à la définition d'un paramètre)
- *VariableDefinition* (correspond à la définition d'une variable)



1.3.6. *Signature :*

La classe Signature permet de stocker les arguments d'une méthode. Elle a donc un attribut : une liste de *Type* correspondant aux types des paramètres de la méthode.

1.3.7. *EnvironmentExp* :

Cette classe permet de stocker un dictionnaire qui associe à chaque identificateur sa définition, lorsque celle-ci est une instance de *ExpDefinition*.

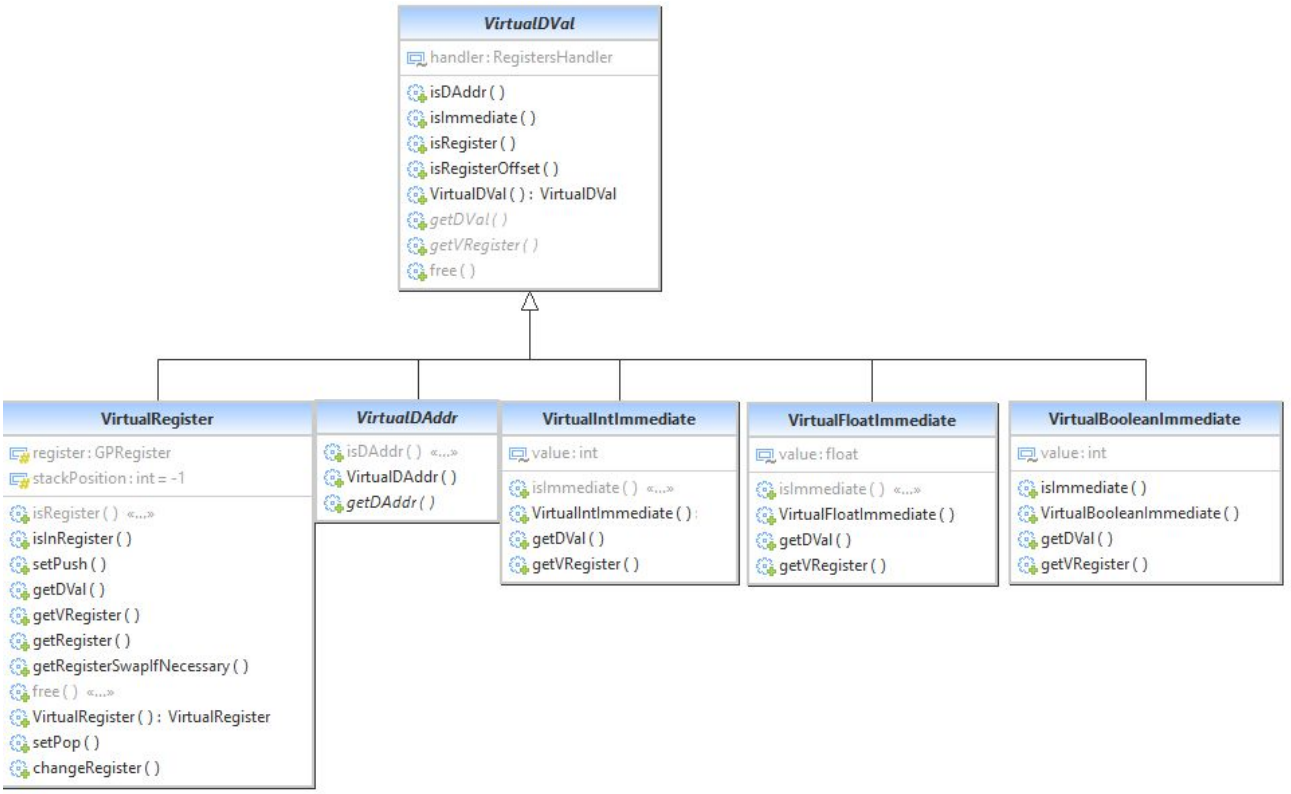
1.3.8. *TypeTools :*

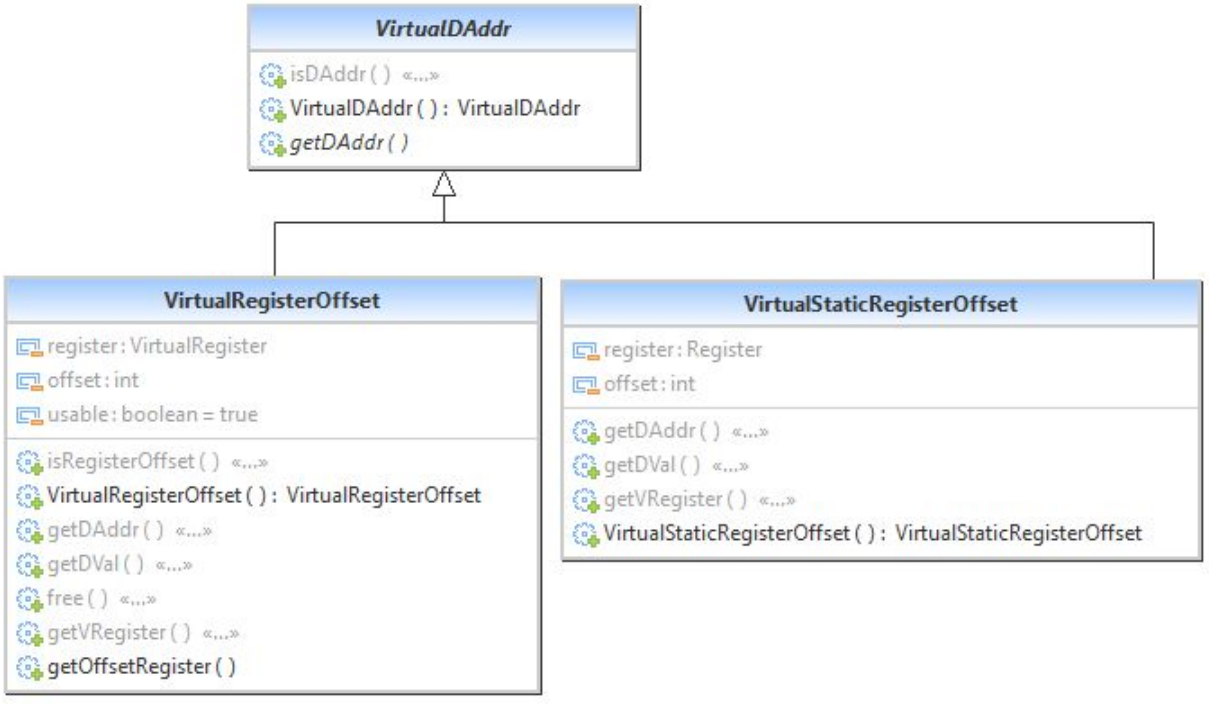
TypeTools est une classe utilitaire, elle contient des méthodes vérifiant la compatibilité pour l'affectation et la compatibilité pour la conversion.

1.4. *Package fr.ensimag.deca.codegen :*

1.4.1. *VirtualDVal*

VirtualDVal est une classe abstraite utilisée pour la gestion des registres. Cette classe abstraite permet de manipuler sans distinction des registres virtuels, des adresses virtuelles et des immédiats. Le fonctionnement de la gestion de registre est détaillé dans la section 2.3.2


VirtualDVal
handler : RegistersHandler
isDAddr ( )
isImmediate ( )
isRegister ( )
isRegisterOffset ( )
VirtualDVal ( ) : VirtualDVal
getDVal ( )
getVRegister ( )
free ( )
VirtualRegister
register : GPRegister
stackPosition : int = -1
isRegister ( ) «...»
isInRegister ( )
setPush ( )
getDVal ( )
getVRegister ( )
getRegister ( )
getRegisterSwapIfNecessary ( )
free ( ) «...»
VirtualRegister ( ) : VirtualRegister
setPop ( )
changeRegister ( )
VirtualDAddr
isDAddr ( ) «...»
VirtualDAddr ( )
getDAddr ( )
VirtualIntImmediate
value : int
isImmediate ( ) «...»
VirtualIntImmediate ( )
getDVal ( )
getVRegister ( )
VirtualFloatImmediate
value : float
isImmediate ( ) «...»
VirtualFloatImmediate ( )
getDVal ( )
getVRegister ( )
VirtualBooleanImmediate
value : int
isImmediate ( )
VirtualBooleanImmediate ( )
getDVal ( )
getVRegister ( )


VirtualDAddr
isDAddr ( ) «...»
VirtualDAddr ( ) : VirtualDAddr
getDAddr ( )
VirtualRegisterOffset
register : VirtualRegister
offset : int
usable : boolean = true
isRegisterOffset ( ) «...»
VirtualRegisterOffset ( ) : VirtualRegisterOffset
getDAddr ( ) «...»
getDVal ( ) «...»
free ( ) «...»
getVRegister ( ) «...»
getOffsetRegister ( )
VirtualStaticRegisterOffset
register : Register
offset : int
getDAddr ( ) «...»
getDVal ( ) «...»
getVRegister ( ) «...»
VirtualStaticRegisterOffset ( ) : VirtualStaticRegisterOffset

2. Implémentation du compilateur :

2.1. Vérification contextuelle :

2.1.1. Structure de données employés :

La vérification contextuelle s'effectue en parcourant l'arbre de syntaxe abstraite. Cette vérification se fait à l'aide de l'environnement d'expression et l'environnement de type, qui est lui définie dans le fichier *DecacCompiler*. Ces deux environnements sont implémentés à l'aide d'un dictionnaire (*HashMap*).

Chaque classe a son propre environnement d'expression.

C'est aussi dans le fichier *DecacCompiler* qu'est construit l'environnement de type prédéfini.

La construction de la méthode Equals est faite au début de la fonction *verifyProgram* de la classe *Program*.

2.2. Parcours de l'arbre :

Le parcours s'effectue en 3 passes :

2.2.1. Passe 1 :

La passe 1 a pour objectif de vérifier le nom des classes et leur hiérarchie.

Elle parcourt l'arbre à partir de la classe *Program* (avec la méthode *verifyListClass*) jusqu'à la déclaration de classe (méthode *verifyClass*).

Cette passe vérifie que la classe n'existe pas déjà, en vérifiant que le nom de la classe n'est pas déjà présent dans l'environnement de type du compilateur. Si ce n'est pas le cas, alors elle l'ajoute à l'environnement.

Elle vérifie aussi l'existence de la superclasse.

2.2.2. Passe 2 :

La passe 2 a pour objectif de vérifier les déclarations de méthodes et leurs signatures ainsi que les déclarations de champs.

La déclaration des champs se fait à l'aide de la méthode *verifyDeclField* . Le champ est ajouté à l'environnement d'expression de la classe courante, en vérifiant au préalable que le type du champ existe, que le nom ne soit pas déjà pris, etc …

L'initialisation des champs n'est pas traitée au cours de cette passe.

La déclaration des méthodes est faite dans la classe *ListDeclMethod* qui contient une liste de déclaration de méthodes.

On parcourt cette liste grâce à la fonction *verifyDeclMethod* qui s'assure que la déclaration est possible (par exemple, il ne faut pas que le nom de la méthode existe déjà pour désigner un champ).

La signature d'une méthode est construite dans la classe *ListDeclParam* avec *verifyListDeclParam*. Cette méthode parcourt la liste des paramètres avec *verifyDeclParam* et ajoute le type de chaque paramètre à la signature.

2.2.3. Passe 3 :

La passe 3 a pour objectif de vérifier le corps des méthodes, les expressions d'initialisation des champs ainsi que le programme principal.

Les champs sont initialisés dans la classe *DeclField* via *verifyFieldBody*, qui fait lui-même appel à la fonction *verifyInitialization* de *Initialization*.

Le corps d'une méthode est analysé dans une classe *MethodBody* avec la fonction *verifyMethodBody*. Cette fonction prend plusieurs arguments, dont en particulier l'environnement d'expression qui contient la liste des paramètres de la méthode à vérifier. Cette liste a été préalablement construite dans la classe *ListDeclParam* (via *verifyListParamBody*). A cet environnement est alors ajouté des potentielles déclarations de variables locales. Enfin les instructions de la méthode sont vérifiées dans cet environnement.

2.3. Génération de code

L'étape C est clairement l'étape la plus lourde en conception. Nous avons tout d'abord commencé par ajouter les fonctions de génération de code. Tous les nœuds de l'arbre ont une ou plusieurs méthodes dont le nom commence par *codeGen…* :

- *codeGenInst* : génère le code afin que le nœud soit interprété comme une instruction à part entière. Même si le nœud renvoie une valeur - après un calcul d'expression par exemple -, rien ne sera fait de celle-ci.
- *codeGenExpr* : génère le code afin que le nœud soit interprété comme une expression. Cette méthode renvoie le résultat de cette expression sous la forme d'une *VirtualDVal*, que ce soit un calcul arithmétique, un immédiat ou autre.
- *codeGenPrint* : génère le code afin d'afficher le nœud à l'écran. Typiquement, les instructions assembleur vont dépendre du type de l'expression à afficher.
- *codeGenBool* : génère le code afin qui va permettre de gérer le flot de contrôle des expressions booléennes. L'implémentation du flot de contrôle est détaillée dans une section dédiée.

D'autres fonctions *codeGen* plus spécifiques seront détaillées dans des sections dédiées.

Le suffixe de ce type de fonction est soit IMA, soit GBA. Celui-ci indique que les instructions générées seront interprétées respectivement soit par une machine virtuelle IMA, soit par une GameBoy Advance.

2.3.1. Gestion du flot de contrôle

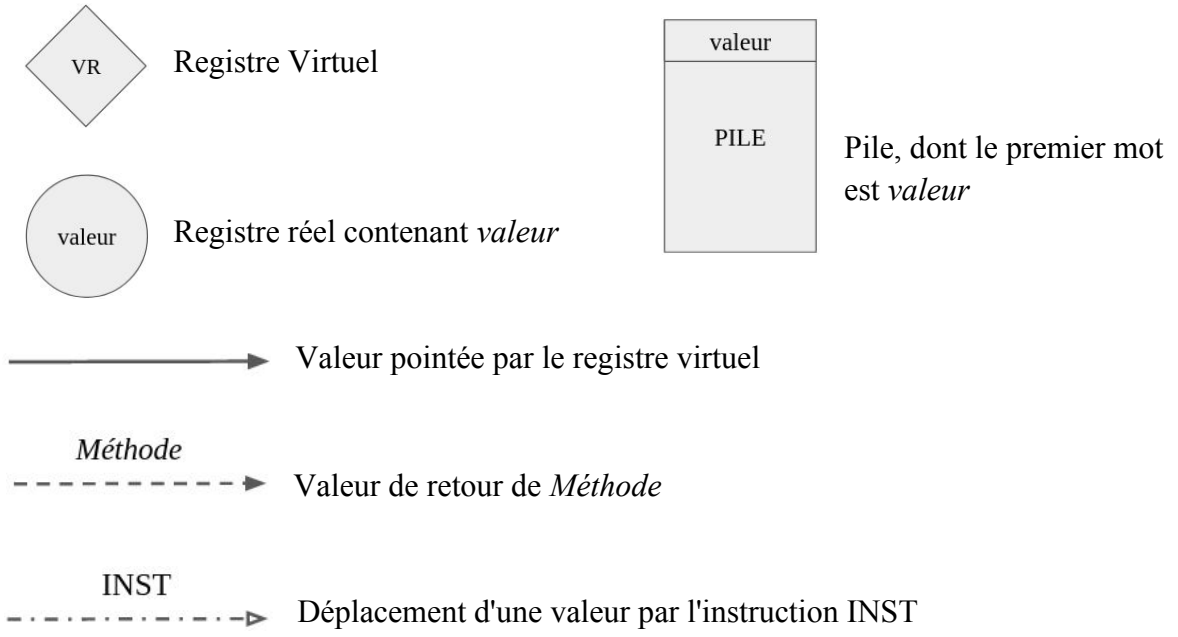
Lorsqu'une instruction doit agir de manière différente selon la valeur d'une expression booléenne, elle doit alors instancier un objet de type *BoolCtrlFlow* puis appeler *codeGenBool*. Cet objet contient un label, précisé par l'instruction, auquel l'expression booléenne doit brancher ou non selon sa valeur. Si la valeur de retour de la méthode *getBranchCond* est false (équivalent à b sur le poly), alors l'expression booléenne doit brancher vers le label, sinon, elle ne branche pas. La méthode *not* permet d'inverser la valeur de retour de cette méthode. La

valeur de retour initiale est définie par l'argument *branchCond* du contructeur de *BoolCtrlFlow*.

Pour illustrer le fonctionnement de cette implémentation, prenons par exemple l'instruction *While*. Lors de l'appel de la méthode *codeGenInst*, celle-ci va d'abord déclarer le label qui correspond au début de la boucle *While*. Puis, le corps de la boucle est inséré. Enfin, un objet de type *BoolCtrlFlow* est déclaré, son label est celui qui correspond au début du While et la valeur de retour de *getBranchCond* est initialement à "vrai" (~ b sur le poly). La méthode *codeGenBool* est ensuite appelée sur l'objet ainsi créé. Ainsi, tant que l'expression de condition est évaluée à vrai, le corps de la boucle est exécuté. Lorsque l'expression est évaluée à false, le branchement vers le début de la boucle n'est pas effectué, et le programme sort de la boucle.

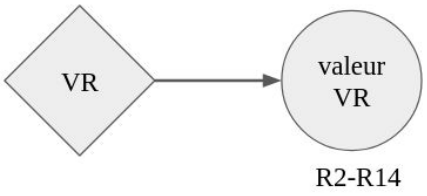
2.3.2. *Gestion des registres*

Un des points les plus importants de la génération de code est la gestion des registres. Au vu des problèmes et de la répétition de code que peut introduire une gestion manuelle des registres, nous avons décidé de déléguer cette tâche à des objets dédiés. Des schémas sont présents pour illustrer le principe de l'implémentation. En voici la légende :

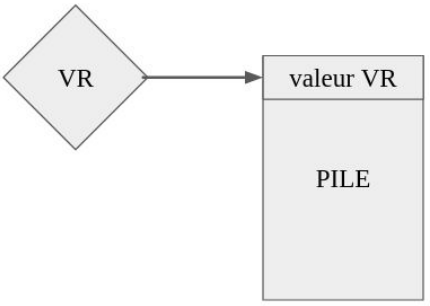


2.3.2.1. Registres virtuels

Au lieu de manier les registres manuellement, nous utilisons une interface sous la forme de la classe *VirtualRegister*. On peut voir cette interface comme un pointeur vers la valeur contenue dans le registre virtuel : soit la valeur pointée est effectivement stockée dans un registre réel qui lui est alloué (fig. 1), soit la valeur pointée est stockée dans la pile car tous les registres réels sont alloués à d'autres *VirtualRegister* pour stocker leurs valeurs (fig. 2).



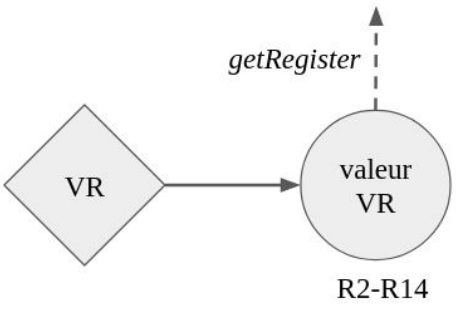
(fig.1) Registre virtuel dont la valeur pointée est stockée dans un registre réel alloué



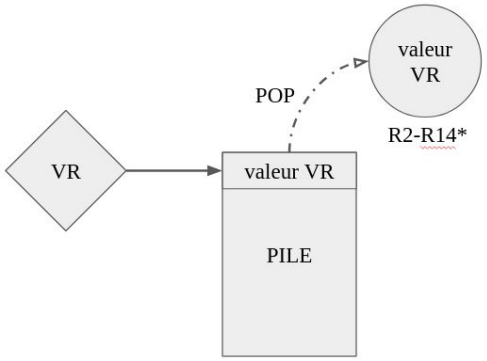
(fig.2) Registre virtuel dont la valeur pointée est empilée

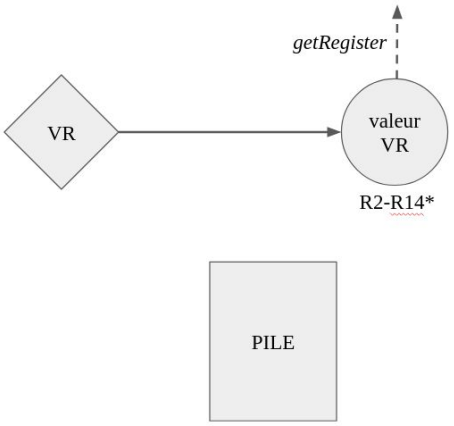
La méthode *getRegister* permet de récupérer un registre réel, sous la forme *GPRegister*, qui contient la valeur pointée par le *VirtualRegister*. On a alors deux cas de figure. Si la valeur est déjà stockée dans un registre réel, alors c'est celui-ci qui va être renvoyé par la méthode (fig. 3). Si la valeur est dans la pile, alors un nouveau registre réel est alloué et la valeur est dépilée (fig 4.1) et stockée dans le registre alloué qui est renvoyé par la méthode (fig. 4.2). Pour que ce dernier cas de figure puisse être effectué, il faut que deux conditions soient réunies. D'une part, il faut que la valeur pointée par le registre virtuel soit en haut de la pile. Cela impose donc que toutes les valeurs empilées par dessus par d'autres *VirtualRegister* soient dépilés avant. La cause de l'empilement de ces valeurs sera détaillée un peu plus tard dans ce chapitre. D'autre part, il faut qu'il y ait des registres réels qui ne soient pas alloués par d'autres *VirtualRegister*.

Cette dernière condition n'est pas nécessaire si l'argument *canBeR0* de la méthode est positionné à vrai. Dans ce cas là, la valeur sera dépilée si besoin dans le registre R0. Cette solution a cependant ses limites. En effet, R0 étant un registre scratch, nous avons pris la décision de ne jamais allouer R0 à un seul et unique *VirtualRegister*. Ainsi, la valeur pointée par celui-ci pourra être écrasée à tout moment lors de la génération de code d'une autre instruction. Cette solution est donc à utiliser uniquement si c'est la dernière fois que la valeur pointée par le *VirtualRegister* est utilisée. Ce dernier deviendra donc virtuellement inutilisable après ça.



(fig. 3) Retour de *getRegister* si un registre réel est alloué au registre virtuel



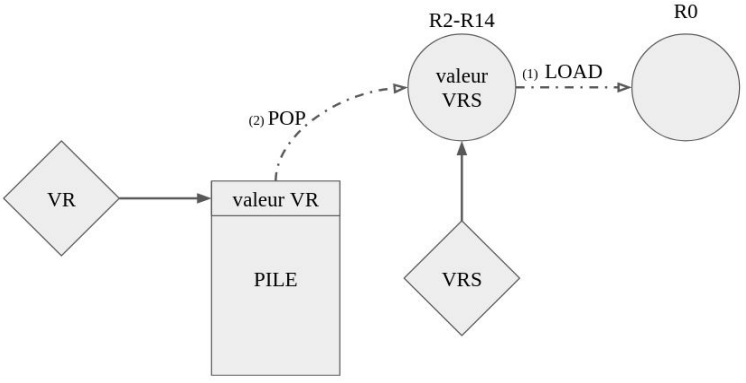


* Le registre alloué est R0 si l'argument *canBeR0* est vrai

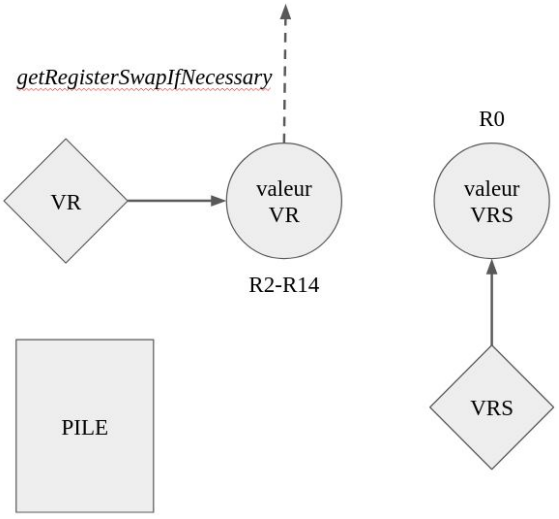
(fig. 4.1) Dépilement de la valeur pointée par le registre virtuel

(fig. 4.2) Allocation du registre réel et retour de *getRegister*

Il peut aussi y avoir des cas où l'on veut récupérer un registre réel contenant la valeur pointée par un *VirtualRegister* - appelé **VR** dans ce paragraphe - qui sera utilisable par la suite - qui n'est donc pas R0 - sans pour autant être sûr qu'il y ait des registres non alloués. On peut alors utiliser la méthode *getRegisterSwapIfNecessary*. Elle prend en argument un autre *VirtualRegister* - appelé **VRS** dans ce paragraphe - , qui lui doit absolument avoir sa valeur pointée dans un registre qui lui est alloué. Il y a donc deux cas de figure. Si il y a au moins un registre réel non alloué, alors la valeur pointée par **VR** est dépilée dans ce registre qui est ensuite retourné - ce qui est dans ce cas équivalent à la méthode *getRegister* (fig. 3). Si il n'y a plus aucun registre réel non alloué, la valeur pointée par **VRS** est alors chargée dans R0, puis la valeur pointée par **VR** est dépilée dans l'ancien registre alloué par **VRS** (fig. 5.1 et 5.2). De la même manière où la méthode *getRegister* avec *canBeR0* à true peut rendre inutilisable le *VirtualRegister* depuis lequel elle est appelée, le registre **VRS** devient virtuellement inutilisable car sa valeur pointée peut être chargée dans R0. Donc de la même façon, **VRS** ne doit pas être utilisé à posteriori. On a alors au final la valeur pointée par **VR** stockée dans un registre réel tandis que la valeur pointée par **VRS** est stockée dans le registre R0.



(fig 5.1) Déplacement des valeurs pointées par les différents registres virtuels



(fig 5.2) Allocation des registres et retour de la méthode *getRegisterSwapIfNecessary*

Le nombre de conditions et leur complexité rendent L'utilisation des méthodes *getRegister* et *getRegisterSwapIfNecessary* non triviales. Pour faciliter la chose, de la programmation défensive préviendra une mauvaise utilisation de ces fonctions. Cependant, elles doivent tout de même être utilisées avec une attention toute particulière quant à leur impact sur les différents *VirtualRegister* qui peuvent être affectés.

Lorsqu'un *VirtualRegister* n'est plus utilisé, il faut alors appeler la méthode *free*, qui va désallouer le registre réel associé ou retirer la première valeur de la pile selon où se trouve la valeur pointée.

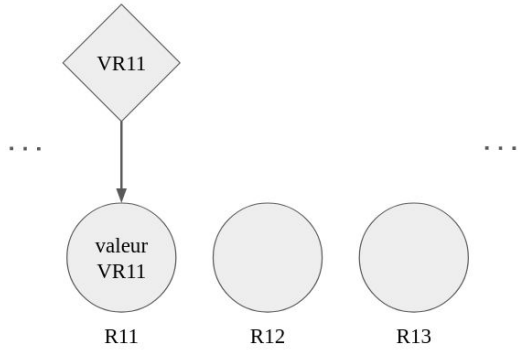
En règle générale, une bonne utilisation des *VirtualRegister* va se traduire par l'utilisation des méthodes *getRegister* et *getRegisterSwapIfNecessary* le plus tard possible, afin de ne pas allouer des registres trop tôt, et par l'utilisation de la méthode *free* le plus tôt possible, afin de désallouer les registres réels quand ils ne sont plus utilisés.

2.3.2.2. RegistersHandler

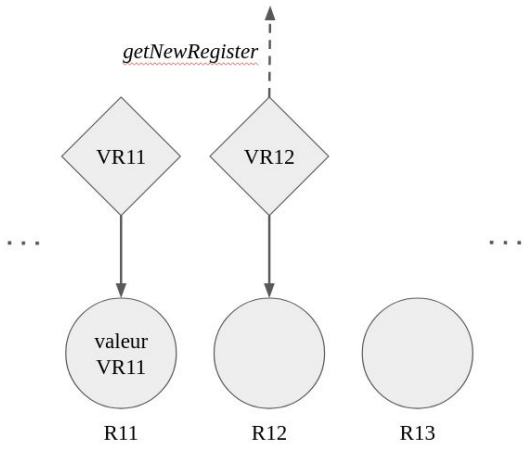
Pour savoir quels registres sont alloués et pour créer de nouveau *VirtualRegister*, on utilise la classe *RegisterHandler*. Cette classe permet de garder la trace des allocations de registres ainsi que de l'empilement des valeurs pointées par les *VirtualRegister*. Le *DecacCompiler* contient un *RegisterHandler* commun pour toute la compilation du programme. Il est accessible grâce à la méthode *getRegisterHandler*.

La méthode *getNewRegister* permet de créer un nouveau *VirtualRegister*. Celui-ci sera toujours associé à un registre réel qui lui est alloué - et qui n'est jamais R0 ni R1. Si il y a encore des registres réels non alloués, alors c'est celui d'index le plus bas qui est utilisé (fig. 6.1 et 6.2). Si il n'y a plus aucun registre réel non alloué, on va empiler le registre réel le plus haut, et c'est celui-ci qui sera alloué au nouveau registre virtuel. Ce dernier cas de figure a une

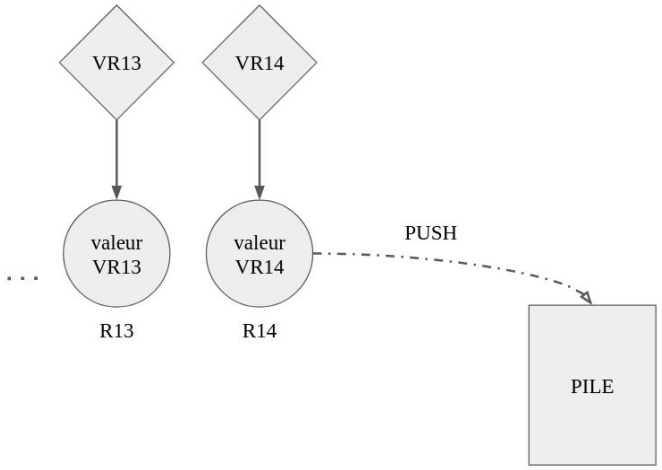
conséquence directe : le *VirutalRegister* auquel le registre le plus haut avait été alloué a donc sa valeur pointée empilée (fig 7.1 et 7.2). C'est là l'origine de tout empilement de valeurs pointées par un *VirtualRegister*. Ainsi, pour que la valeur pointée par ce dernier puisse être dépilée correctement par l'appel de la méthode *getRegister*, il faut appeler avant la méthode *free* sur le registre virtuel qui vient d'être créé. Si ce n'est pas possible, on peut toujours se rabattre sur l'argument *canBeR0* à vrai ou sur la méthode *getRegisterSwapIsNecessary*. Si la méthode *getRegister* n'est pas appelée par la suite, cela signifie que le *VirtualRegister* dont le registre a été désalloué n'est plus utilisé et la méthode *free* aurait dû être appelée avant la création du nouveau registre.



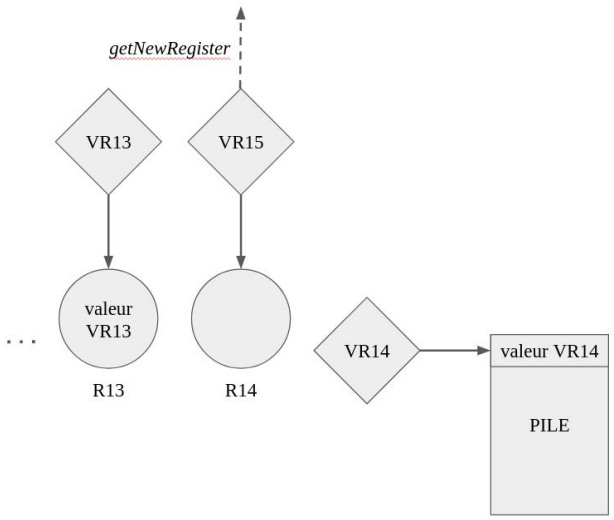
(fig. 6.1) Des registres réels ne sont pas alloués



(fig. 6.2) Allocation d'un registre et retour de la méthode *getNewRegister*



(fig 7.1) Tous les registres sont alloués : empilement de la valeur du registre le plus haut



(fig 7.2) Allocation du registre libéré et retour de la méthode *getNewRegister*

Entre les instructions, tous les registres doivent être désalloués. Pour cela, on peut utiliser la méthode *resetRegisters*. Cette méthode permet aussi de vérifier par programmation défensive qu'aucune valeur pointée par un registre virtuel n'est restée dans la pile. S'il reste des valeurs empilées, cela signifie que certains *VirtualRegister* n'ont pas été désalloués, alors qu'ils auraient dû l'être.

Une autre utilité du RegisterHandler est de garder une trace de toutes les instructions *PUSH* et *POP* ainsi que de tous les registres qui vont être utilisés au cours de l'exécution d'un bloc du programme. Cela permet de générer facilement le code de vérification de *StackOverflow* et celui de sauvegarde des registres lors d'un appel de méthode. Les méthodes *getMaxPush* et *getMaxLowerReg* permettent de récupérer respectivement la taille maximale de la pile et le nombre de registres utilisés dans un bloc. Si une instruction *PUSH* ou *POP* est appelée en dehors du *RegisterHandler*, on peut utiliser les méthodes *pushToStack* et *popFromStack* pour l'en informer et pour qu'il puisse calculer la taille maximale de la pile utilisée dans un bloc. Cela peut être utile lors de l'appel de méthode par exemple, lorsque paramètre sont empilés puis dépilés.

2.3.2.3. VirtualRegisterOffset

Un des effets de l'utilisation des registres virtuels est que nous ne pouvons plus directement utiliser l'objet *RegisterOffset* qui nous a été fourni. En effet, imaginons que nous voulions faire un offset par rapport à un registre virtuel qui a sa valeur pointée stockée dans le registre R14. Nous allons déclarer un *RegisterOffset* qui aura pour arguments la valeur de l'offset et le retour de la méthode *getRegister* sur le registre virtuel, qui va donc nous renvoyer R14. Par la suite, imaginons que nous faisions d'autres instructions qui empilent le registre R14 de sur la pile. La valeur pointée par le registre virutel n'est donc plus dans R14. Ainsi, lors de l'utilisation du *RegisterOffset*, ce dernier ne serait plus valide et l'offset ne se fait donc pas correctement. C'est pour résoudre ce genre de problèmes que nous avons créé les *VirtualRegisterOffset*, ou offset sur registre virtuel.

La méthode la plus importante de cette classe est la méthode *getDAddr*. Celle-ci renvoie un *RegisterOffset* qui se fait par rapport au registre réel alloué registre virtuel associé. Ce dernier à été récupéré grâce à la méthode *getRegister*, donc les conditions d'utilisation de celle-ci - vues plus haut - s'appliquent à la méthode *getDAddr*. Cette méthode doit être appelée lors de l'utilisation effective de l'offset renvoyé, lors d'une instruction store par exemple.

Une autre méthode utile est *getOffsetRegister*, qui renvoie le registre virtuel sur lequel se base le *VirtualOffsetRegister*. Elle peut être utilisée lorsque l'on veut manipuler la valeur pointée par le registre virtuel.

Comme certains offset se font sur des registres qui ne peuvent pas être virtuels, comme GB, LB et autres, nous avons aussi créé la classe *VirtualStaticOffsetRegister*. Cette classe est relativement équivalente aux *RegisterOffset* classiques, à la différence qu'elle

dépend de la classe abstraite *VirtualDAddr*, tout comme *VirtualRegisterOffset*. Cela permet de manipuler indifféremment ces deux types d'offset sans avoir à se soucier du type de registre sur lequel ils se basent. Le résultat de la méthode *getDAddr* est simplement un *RegisterOffset* classique qui dépend directement des attributs du *VirtualStaticOffsetRegister*.

2.3.2.4. VirtualDVal

La dernière facette à la gestion des registres est l'utilisation des DVal. En effet, de nombreuses instructions assembleur permettent d'utiliser indifféremment les immédiats, les registres et les adresses. Pour profiter de cette conception, nous avons créé une classe abstraite *VirtualDVal*. De celle-ci va dépendre les classes *VirtualRegister*, *VirtualDAddr*, et les différentes classes de type *VirtualImmediate*. Il y a une classe *Virtualmmediate* par type d'immédiat.

La méthode la plus utile est la méthode *getDVal* qui renvoie la *DVal* associée. Si la *VirtualDVal* est un registre virtuel, cette méthode va renvoyer le même résultat que la méthode *getRegister*. Si c'est un immédiat virtuel, elle va simplement renvoyer une classe *Immediate* selon le type de l'immédiat. Si c'est une *DAddr* virtuelle, elle va renvoyer le même résultat que la méthode *getDAddr*.

Si on a besoin que la valeur associée à une *VirtualDVal* soit stockée dans un registre, on peut alors appeler la méthode *getVRegister*, qui va renvoyer un registre virtuel dont la valeur pointée est la valeur associée souhaitée. Si la *VirtualDVal* est un registre virtuel, cette méthode va renvoyer l'objet lui-même, sans plus de modification. Si c'est un immédiat virtuel, la méthode va déclarer un nouveau registre virtuel dans lequel sera stocké l'immédiat, et qui sera renvoyé par la fonction. Si c'est un *VirtualStaticRegisterOffset*, alors la valeur pointée par l'adresse de l'offset est stockée dans un nouveau registre virtuel qui sera renvoyé par la fonction. Si c'est un offset sur registre virtuel, alors la valeur pointée par l'adresse de l'offset est stockée dans le registre virtuel qui sert à l'offset, qui est renvoyé par la fonction. Il est donc important de comprendre que dans ce dernier cas, le registre sur lequel se fait l'offset change alors de valeur, et que par conséquent, l'offset virtuel en lui-même n'est plus utilisable. Un point primordial à prendre en compte est que selon la nature de la *DVal* virtuelle, l'appel de cette méthode peut créer des registre virtuel, auxquels seront alloués des registres réels. Il est donc impératif d'utiliser la méthode *free* sur le retour de cette méthode lorsque celui-ci n'est plus utile. Il est donc nécessaire de stocker le résultat de cette fonction. La méthode ne peut donc pas être utilisée pour faire une sélection directement dessus.

La méthode *free* est évidemment une méthode très importante. Étant donné que l'on souhaite manipuler de manière indépendante les registres virtuels des immédiats et autres, on pourrait avoir tendance à oublier qu'une *VirtualDVal* peut potentiellement avoir un registre réel qui lui est alloué. Il est cependant très important d'appeler la méthode *free* afin de désallouer un registre réel si nécessaire. Si la *VirtualDVal* est un registre virtuel ou un offset sur registre virtuel, cette méthode va désallouer le registre réel alloué ou désempiler la valeur pointée de la pile selon le cas de figure. Sinon, cette méthode n'a aucun effet.

2.3.2.5. Conclusion sur notre gestion des registres

Notre solution pour la gestion des registre peut paraître complexe mais elle est très puissante si elle est bien utilisée. Nous n'avons plus à nous occuper de savoir s'il y a besoin de sauvegarder des registres, ni de savoir quel registre utiliser lorsqu'il y a besoin de stocker une valeur. La manipulation des *VirutalRegister* permet de s'assurer que l'on va avoir un registre contenant la bonne valeur lorsque l'on en a besoin. De plus, l'abstraction des *DVal* grâce à *VirutalDVal* permet de ne pas avoir à s'occuper de stocker les immédiats ou les adresses lorsqu'on a besoin qu'elles soient dans un registre. Cependant, cette abstraction à un coût. Certaines instructions produites pourraient être optimisées par une gestion manuelle des registres.

Malgré tout, il faut admettre que l'utilisation des registres et DVal virtuels n'est pas triviale et qu'elle nécessite de très bien comprendre le fonctionnement des différentes classes. Cela est en partie dû au fait que la conception de la gestion automatique des registre a été faite relativement tôt dans le développement et que l'on avait pas une compréhension totale du contexte d'utilisation des registres. Ainsi, certains choix de conceptions ont été faits en anticipant des cas de figure qui ne peuvent pas arriver. Nous restons tout de même satisfait de notre implémentation et le code produit semble marcher selon nos tests.

2.3.3. Calcul des expressions à opérateurs binaires

Etant donné que toutes les expressions à opérateurs binaires reposent sur les mêmes instructions pour le calcul des opérandes, nous avons décidé de créer la méthode *preCodeGenExpr* pour les classes *AbstractBinaryExpr*. Cette méthode génère les instructions pour le calcul des opérandes puis renvoie un objet de classe *BinaryExprDVal* qui contient les *VirtualDVal* correspondant aux résultats de ce calcul. De manière équivalente, nous avons créé la méthode *prepareCodeGenBool* de la classe *AbstractOpCmp* pour le calcul des opérandes d'une comparaison binaire.

2.3.4. Adresse d'une lValue

Pour générer les instructions qui chargent l'adresse d'une lValue, nous avons créé la méthode *codeGenAssign*. Cette méthode renvoie une *VirtualDAddr* qui contient l'adresse chargée de la lValue.

La méthode *codeGenDecl* des classes *DeclVar* et *DeclParam*, malgré son nom mal choisi, ne génère pas de code, mais indique à la définition d'un paramètre ou d'une variable locale son adresse dans la mémoire. C'est celle-ci qui est renvoyée par la méthode *codegenAssign*.

2.3.5. Gestion des blocs (main, méthodes, …)

Les instructions correspondant au code d'un bloc sont générées grâce à la méthode *codeGenListInst* de la classe *AbstractListInst*.

Lors de la définition d'un bloc, il est primordial de vérifier que la pile possède suffisamment de place afin de pouvoir empiler toutes les valeurs nécessaires. Cela est géré grâce au *StackOverflowHandler*. Grâce à la méthode *insertCheck*, celui-ci va calculer la taille de la pile nécessaire pour pouvoir exécuter le bloc et va insérer les instructions associées qui génèrent une erreur en cas de débordement potentiel de la pile. Cette méthode utilise le *RegisterHandler* et en particulier ses méthodes *getMaxPush* et *getMaxLowerReg* afin de calculer la taille maximale des empilements lors de l'exécution du bloc.

Lors de l'entrée dans un bloc, il faut appeler la méthode *switchNewBlock* du *DecacCompiler*, avec pour argument vrai si le bloc correspond au main, faux sinon. Celle-ci va indiquer au *StackOverflowHandler* ainsi qu'au *RegisterHandler* que l'on entre dans un nouveau bloc, et qu'il faut par conséquent réinitialiser les valeurs utilisées pour le calcul de la taille maximale de l'empilement lors de l'exécution du bloc. Il faut ensuite indiquer le nombre de variables locales qui va être déclaré grâce à la méthode *setNbVar*. La génération des instructions du bloc va garder une trace de tous les empilement et dépilements effectués. Enfin les méthodes *addBlocChecks* et *appendCurrentBlock* doivent être appelées pour indiquer la sortie du bloc.

Pour savoir quels registres il faut sauvegarder à l'entrée d'un bloc puis restaurer lors de la sortie, on peut utiliser la méthode *getMaxLowerReg* de la classe *RegisterHandler*. Celle-ci renvoie l'indice du registre le plus haut à être utilisé lors de l'exécution d'un bloc. Cette valeur est calculée automatiquement lors de la génération des instructions du bloc en question. Il faut donc, avant la sortie du bloc, itérer depuis le registre d'indice 2 jusqu'au registre d'indice le plus haut à être utilisé pour ajouter les instructions nécessaires à la sauvegarde et à la restauration de ces registres.

2.3.6. Génération des classes

La génération des instructions relatives aux classes se fait en deux passes.

La première passe permet de générer les instructions qui s'occupent de créer la table des méthodes. Cela se fait grâce à la méthode *buildMethodTable* de la classe *DeclClass*. C'est lors de son exécution que l'on génère les labels qui précèdent les instructions des méthodes ainsi que celui qui précède les instructions d'initialisation de la classe. C'est aussi à ce moment que la position de la classe dans la table des méthodes est initialisée.

La seconde passe permet quant à elle de générer le code de l'initialisation de la classe ainsi que celui des méthodes en elle-même. Cela se fait respectivement au travers des méthodes *codeGenInit* et *codeGenClass*. C'est à ce moment qu'est généré les instructions d'initialisation des champs de la classe. Celle-ci se fait en deux temps. D'abord, si la classe dépend d'une superclasse qui n'est pas *Object*, alors la méthode *codeGenFieldPreDecl* est appelée sur chaque champ afin de les initialiser à 0. Ensuite, après l'appel à l'initialisation de l'éventuelle super-classe, la méthode *codeGenFieldDecl* est appelée pour sur chaque champ afin de les initialiser à la bonne valeur.

La génération des instructions des méthodes se fait quant à elle grâce à la méthode *codeGenMethod* de la classe *MethodDecl*. Celle-ci initialise l'adresse de ses différents paramètres puis appelle la méthode *codeGenBody* afin de générer les instructions du bloc.

2.3.7. Gestion des labels

Pour s'assurer d'avoir des labels uniques lors des branchements, nous avons implémenté un système de compteurs, qui prend la forme des classes *Counter* et *CounterManager*. Le *DecacCompiler* possède un *CounterManager* qui va avoir un *Counter* différent pour chaque type de label. Ce *CounterManager* peut être accédé grâce à la méthode *getCounterManager*. Pour ajouter un compteur pour un type de label, il faut utiliser la méthode *addCounter*. La méthode *addOne* permet d'incrémenter le compteur du type de label passé en argument et de renvoyer sa valeur. La méthode *getLast* permet de récupérer la valeur du compteur du type de label passé en argument sans incrémenter sa valeur.

Un exemple simple de l'utilisation de ces compteurs est la génération des labels pour une instruction *IfThenElse*. Comme il peut y avoir plusieurs instructions *IfThenElse* dans un code deca, il est important qu'elles branchent sur des labels qui soient tous différents, d'où l'utilisation de compteurs. Il faut avant la génération de code informer le *CounterManager* de l'existence de ce compteur avec l'utilisation de la méthode *addCounter*. On a besoin ensuite de trois labels différents : un qui précède les instructions du Then, un qui précède les instructions du Else, et un qui est situé à la fin du code du *IfThenElse*. Pour le premier label déclaré, on va donc utiliser la méthode *addOne* afin de différencier ce label de tous les autres du même type. On peut ensuite utiliser la méthode *getLast* afin d'utiliser le même compteur sur les autres labels, en supposant que ceux-ci ne vont pas être complètement identiques. On aura donc par exemple trois label de la forme *"IfThenElse.n.Then"*, *"IfThenElse.n.Else"* et *"IfThenElse.n.End"* avec *n* la valeur de retour des méthode *addOne* et *getLast*.

2.3.8. Gestion des erreurs

Pour ne pas avoir à gérer le code de gestion des erreurs à la main, nous avons mis en place les classes *ErrorManager* et *AsmError*. Le *DecacCompiler* possède un *ErrorManager* qui va avoir une *AsmError* pour chaque type d'erreur. Cet *ErrorManager* peut être accédé grâce à la méthode *getErrorManager*. Pour ajouter un type d'erreur, il faut utiliser la méthode *addError* qui prend en argument l'identifiant du type de l'erreur ainsi que le message affiché lors de l'exécution de l'erreur. Pour brancher vers une erreur dans la génération de code, il faut utiliser la méthode *getErrorLabel* en mettant en paramètre l'identifiant précédemment utilisé pour la création de l'erreur, qui renvoie le label vers lequel brancher. La génération du code de l'erreur se fait par la méthode *codeGenError*.