ELEFTHERIOU Theophanis - ELIEZER Gwennan - GINDRIER Clément
PETITEAU Matthieu - POLICET Rémi

DOCUMENTATION DE CONCEPTION

Vous trouverez dans ce document les spécificités de notre code

Equipe GL60

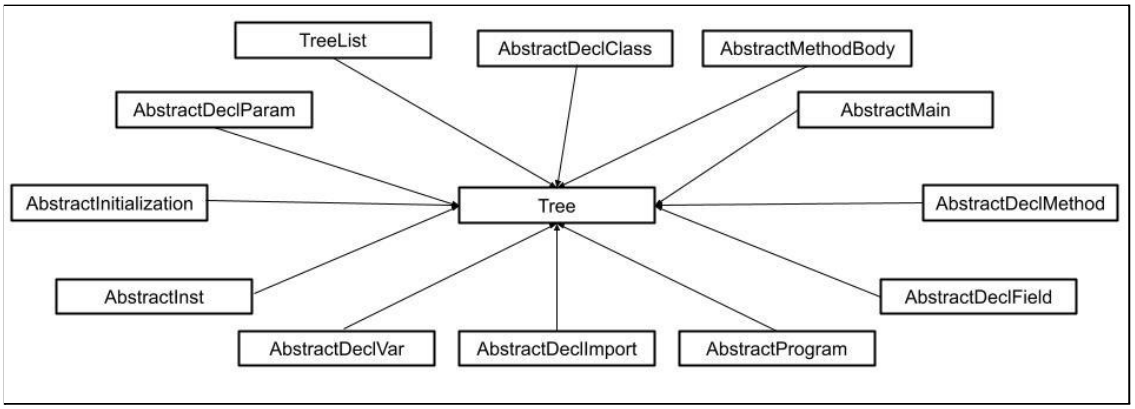
GRENOBLE INP UGA Ensimag

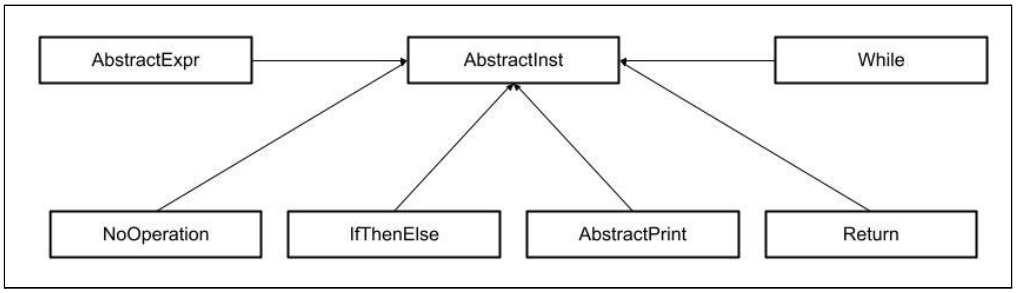
SOMMAIRE

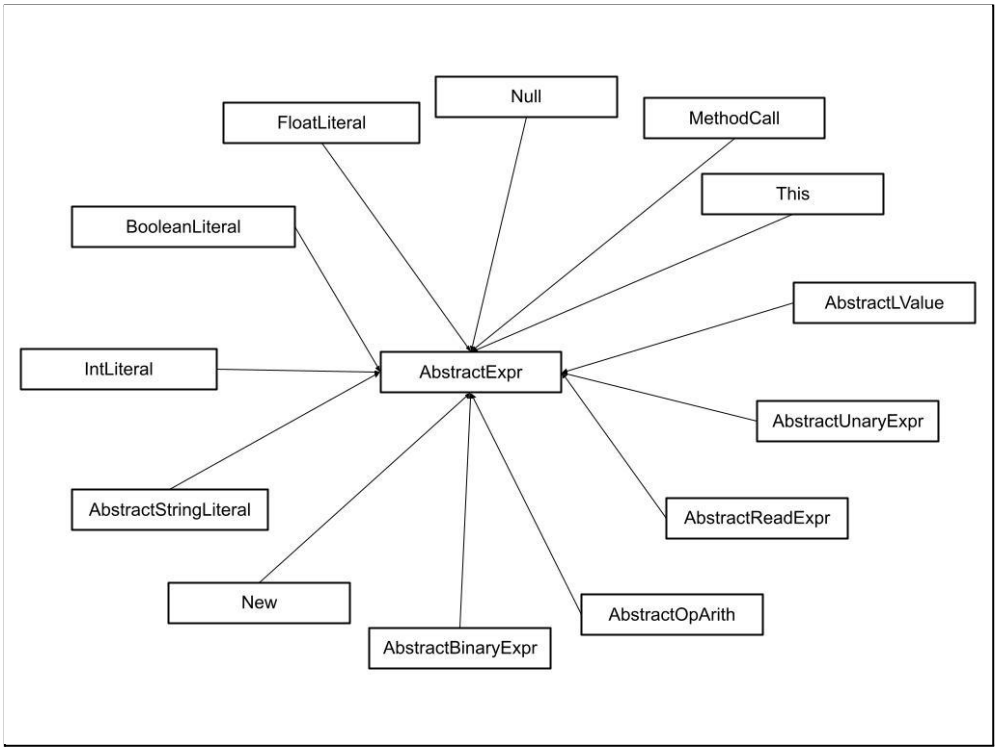
Architecture des classes

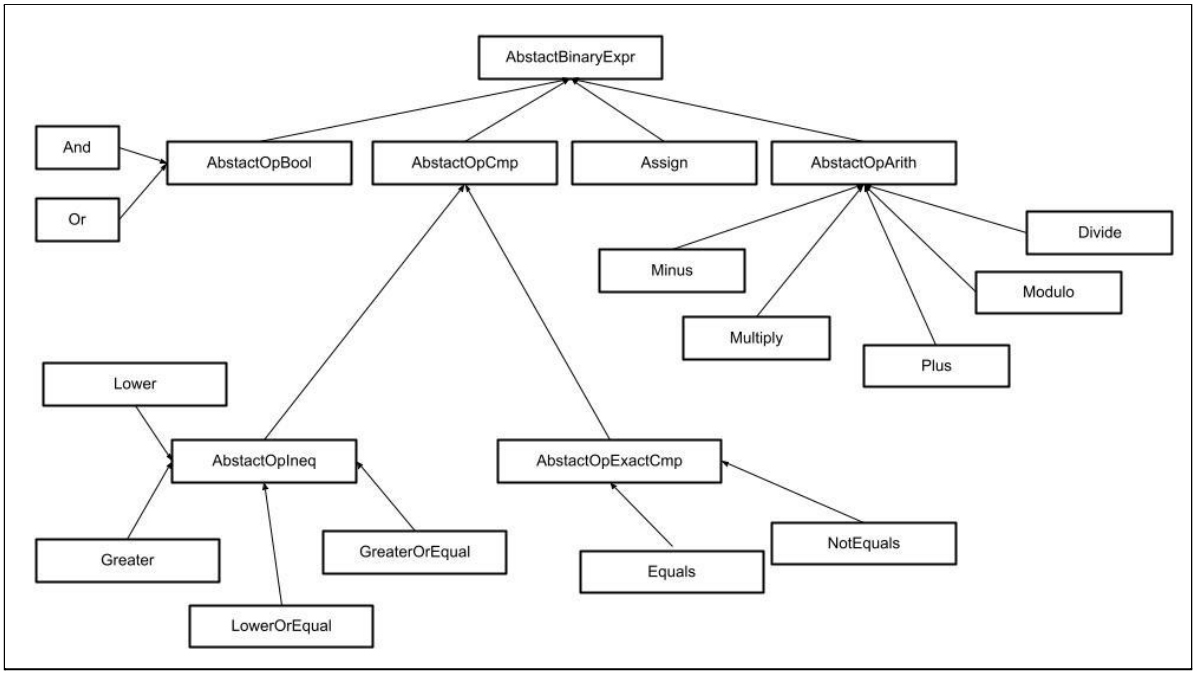
Arbre abstrait : tree

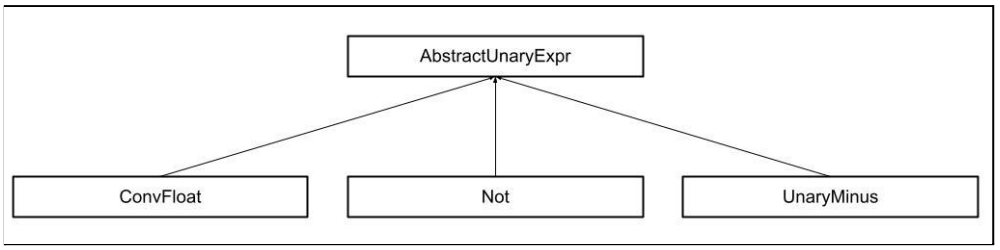
Les classes suivantes permettent de construire et de représenter l'arbre abstrait.
Toutes les classes héritent de la classe mère Tree. Elles sont ensuite regroupées par traits communs, comme les opérations booléennes ou expressions.

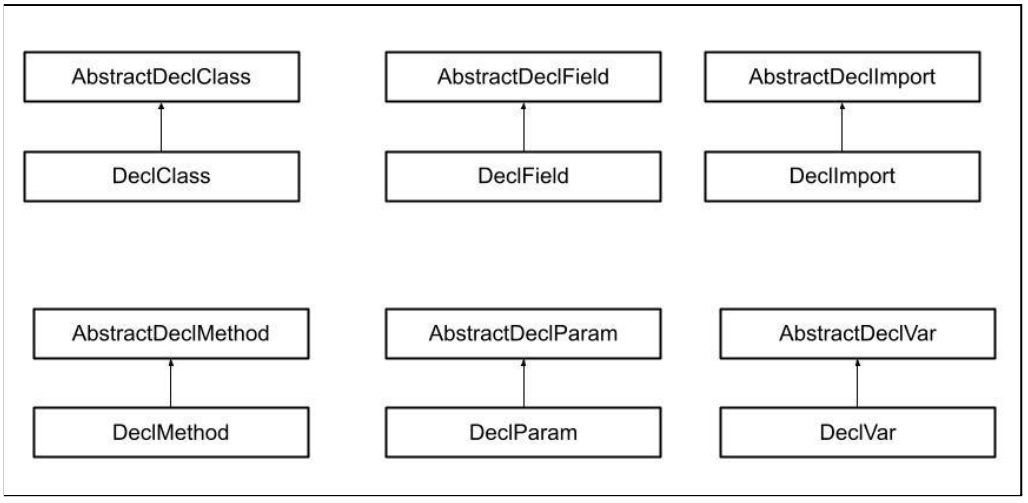


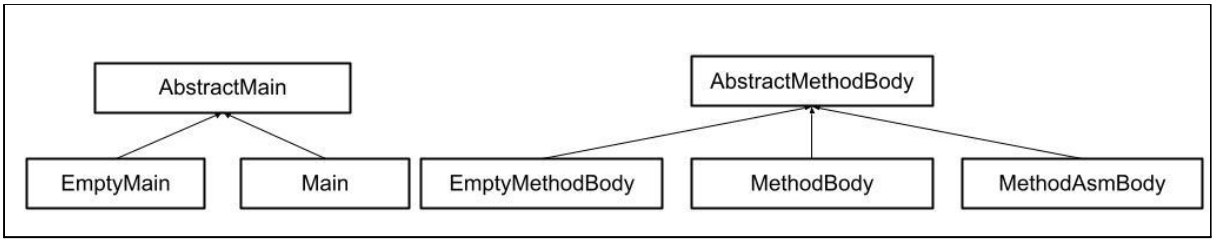


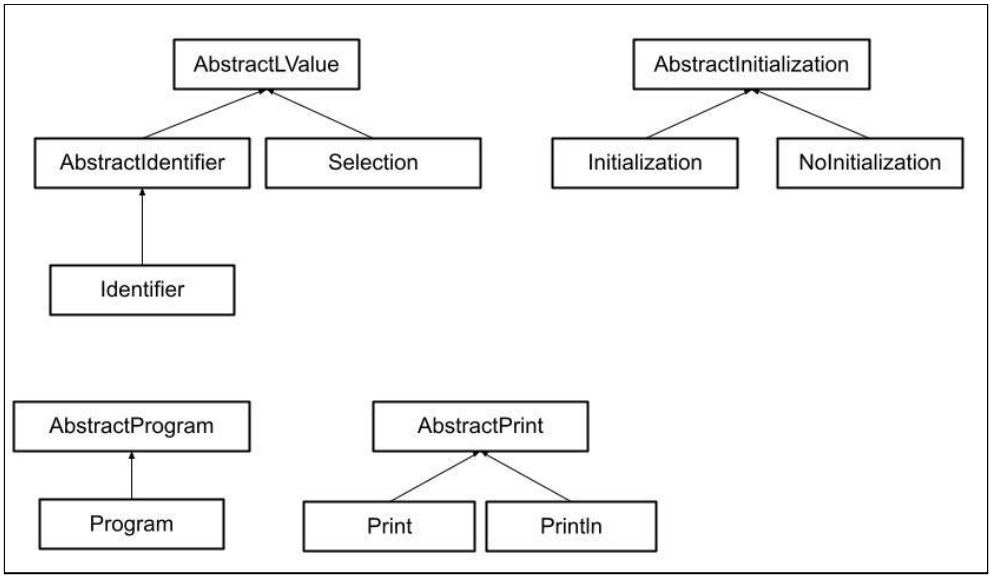


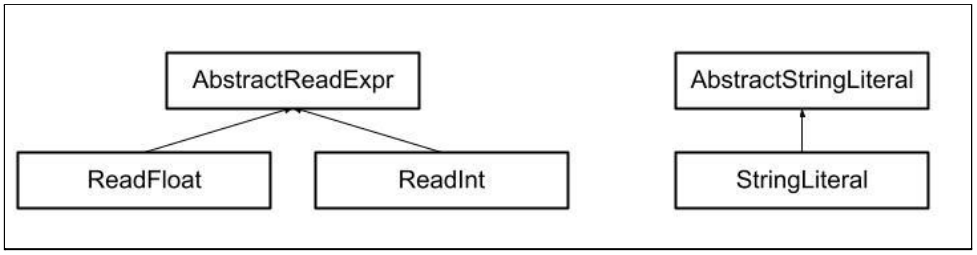

AbstactBinaryExpr
And
Or
AbstactOpBool
AbstactOpCmp
Assign
AbstactOpArith
Divide
Minus
Modulo
Multiply
Plus
Lower
AbstactOpIneq
AbstactOpExactCmp
Greater
GreaterOrEqual
Equals
NotEquals
LowerOrEqual

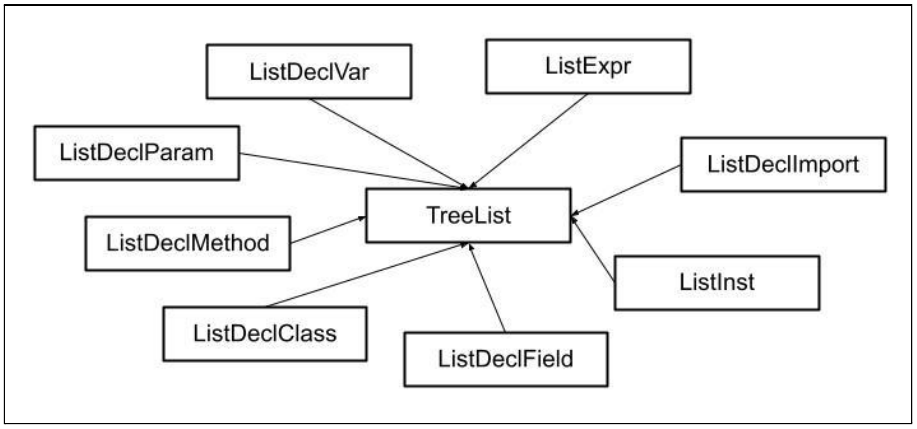

AbstractUnaryExpr
ConvFloat
Not
UnaryMinus


AbstractDeclClass
DeclClass
AbstractDeclField
DeclField
AbstractDeclImport
DeclImport
AbstractDeclMethod
DeclMethod
AbstractDeclParam
DeclParam
AbstractDeclVar
DeclVar


AbstractMain
EmptyMain
Main
AbstractMethodBody
EmptyMethodBody
MethodBody
MethodAsmBody

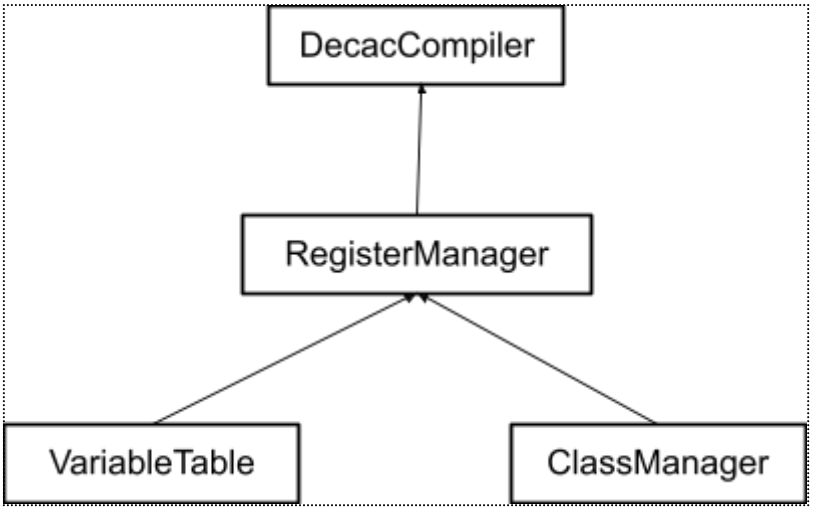






Génération de code : inclusion entre classes

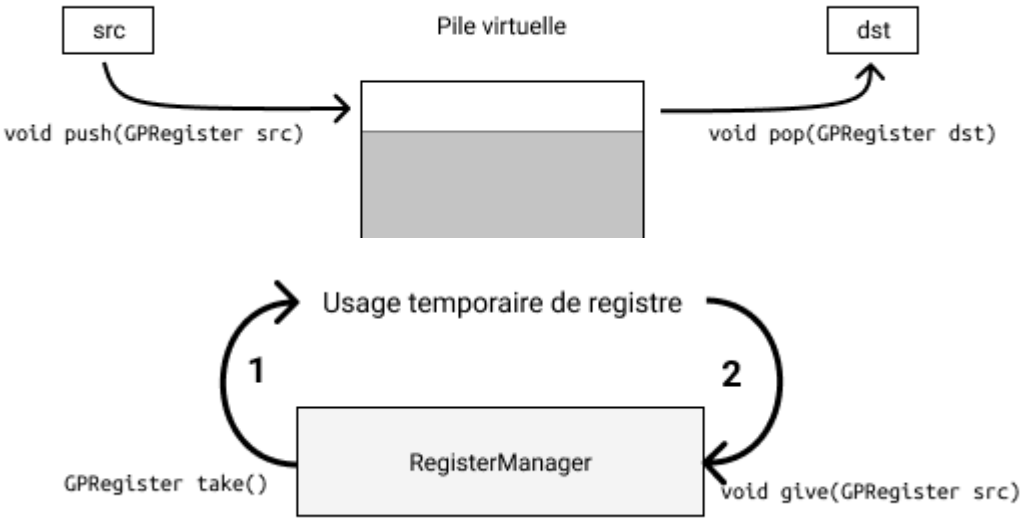
Pour la génération de code, nous avons inclus plusieurs classes utilitaires dans le compilateur. Les liens représentés ne sont pas un lien de parenté, mais un lien d'inclusion.



Spécifications sur le code

Utilisation des registres et de la pile

Afin d'optimiser le code tout en gardant le code simple, l'usage de la pile et des registres d'usage général excepté R0 et R1 doit être fait au travers de la classe RegisterManager, dans le paquetage fr.ensimag.deca.codegen. Cette classe implémente une pile virtuelle, à laquelle on a accès via les méthodes "push(reg)" et "pop(reg)". De plus, elle utilise tous les registres à partir de R2, il faut donc lui indiquer lorsque l'on souhaite utiliser l'un de ces registres. Cela se fait au travers des méthodes "reg = take()", qui demande au RegisterManager de libérer et donner un registre, et "give(reg)", qui rend au RegisterManager le registre précédemment pris reg.



Ainsi, à la place d'utiliser directement la pile et les registres, il faut utiliser les méthodes de RegisterManager, afin de ne pas corrompre l'état que RegisterManager met en place.

Utilisation des variables et paramètres

Pour utiliser les variables et paramètres, une classe adaptée a été créée, la VariableTable. Celle-ci gère l'association entre nom de variable (ou de paramètre) et la position de celle-ci dans la pile. Elle donne accès à deux méthodes, "boolean load(Symbol s, GPRegister dst)" et "boolean store(Symbol s, GPRegister src)", capables de générer le code pour charger le contenu de la variable dans un registre ou pour enregistrer le contenu de la variable depuis registre. Si tout se passe bien, ces méthodes renvoient la valeur "true", mais si la variable est introuvable dans le contexte, aucun code n'est généré et la méthode renvoie "false". Cette classe est intégrée au RegisterManager, qui gère le cas où la méthode renvoie false (en regardant ensuite dans les attributs de la classe courante) et ajoute une méthode "GPRegister load(Symbol s)", Pareil qu'une succession d'appels à "take" et "load".
Les paramètres s'utilisent de la même manière.

Utilisation des attributs et méthodes

Pour utiliser un attribut ou une méthode, la classe ClassManager a été créée. Elle donne accès à deux méthodes pour les attributs, "void getField(GPRegister addr, Symbol fieldName, Definition objDef, GPRegister dst)", capable de charger le contenu de l'attribut "fieldName" de l'objet d'adresse "addr" et de définition de classe "objDef" dans "dst".
Elle donne également une méthode pour les méthodes, "void getMethod(GPRegister addr, Symbol methName, Definition objDef, GPRegister dst)" chargeant l'adresse de la méthode "methName" de l'objet d'adresse "addr" de la classe "objDef" dans le registre "dst".

Afin d'assurer le bon déroulement d'un appel de méthode, RegisterManager donne également une méthode "void prepareMethodCall(int nbParams)" lui indiquand que les nbParams derniers éléments ajoutés sur la pile virtuelle sont arguments d'un appel de méthode. Il va donc s'assurer qu'ils sont tous sur la pile, en déplaçant l'élément parmi eux le plus bas dans la pile tout en haut (supposé être le "this"). Avant de réutiliser le RegisterManager, il faut diminuer SP de nbParams + 1, car l'implémentation actuelle laisse un espace vide là où le "this" se trouvait avant d'être déplacé (pour ne pas perdre de performance à déplacer chaque case).

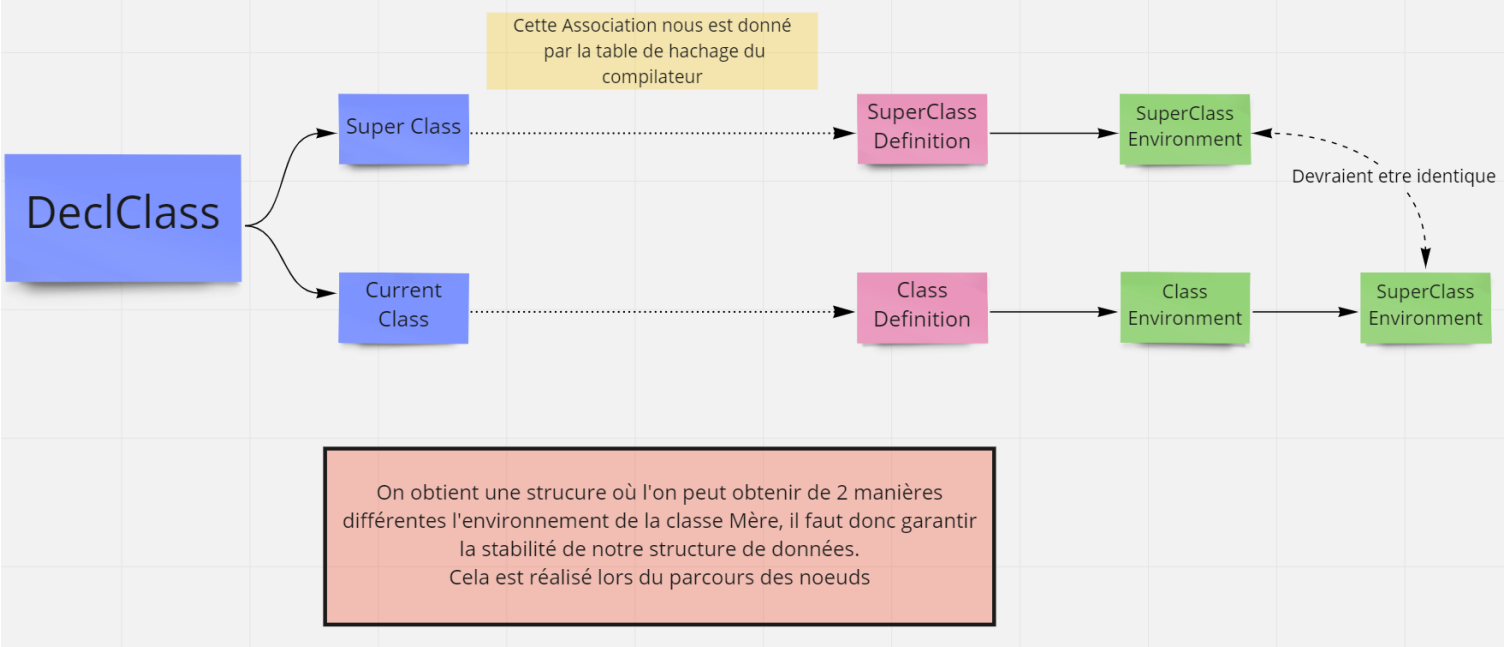
Description des algorithmes et structures employées

Choix de structure pour les environnements de Type et d'expressions :

Nous avons modifié et apporté des méthodes supplémentaires à notre compilateur durant tout le processus de développement.
En premier lieu, nous avons utilisé uniquement la Table de symbole fourni dans les outils possibles pour faire marcher notre compilateur pour la partie sans-objet.
C'est en attaquant la partie avec Objet qu'un important besoin de généralisation et concaténation s'est fait ressentir.
En effet, il était nécessaire de pouvoir garder une trace de tous les types déclarés à l'intérieur du compilateur, de chaque méthode créer à l'intérieur de chaque classe pour que l'on puisse les appeler efficacement, il fallait également avoir une liste chainée nous permettant d'explorer l'arbre d'héritage de classes.
Nous avons donc choisi d'utiliser la structure d'environnement proposée.
Ainsi, chaque environnement contient un environnement parental, et une table de hachage réalisant l'association clé-valeur des données stockées dans l'environnement. Les clés correspondent aux symboles des objets stockés dans l'environnement et les valeurs aux définitions de ces objets.
De cette manière, nous disposions d'une structure abstraite pouvant contenir différents types d'objet dès lors que leurs définitions étaient encadrées proprement.
Ainsi, notre compilateur contient deux environnements, un environnement de type qui référence une association entre les noms de tous les types pouvant être utilisés et leurs définitions, on y stocke donc les définitions des types de bases comme 'int' ou 'float' mais aussi de tous les autres types construits, comme les classes, et la classe parente de toutes les classe, à savoir la classe 'Objet'.
Le deuxième environnement présent dans le compilateur est en réalité extrêmement similaire à celui contenu dans les Classe et correspond au 'Main' du programme principal.

C'est un environnement d'expression, c'est-à-dire qu'il contient toutes les méthodes et les variables définie à l'intérieur du Main. Les variables sont sensiblement différentes des champs utilisés dans les classes car elles n'ont pas de visibilité. Elles ne pourront donc être appelées qu'à l'intérieur du bloc où elles ont été instanciées.
Nous disposons également d'autres environnements qui sont internes aux classes. En effet, chaque classe contient son propre environnement qui regroupe toutes les méthodes de la classe ainsi que ses champs, avec leur visibilité à l'intérieur de la définition. Celle-ci garantissant qu'un champ 'protected' ne puisse pas être appelé à l'extérieur de cette classe ou de ses descendants.

Nous avons donc une structure particulière, où chaque classe possède un nom mais également le nom de son parent. On peut associer à chaque classe une définition qui contient un environnement.
Et chaque environnement possède un environnement parent.

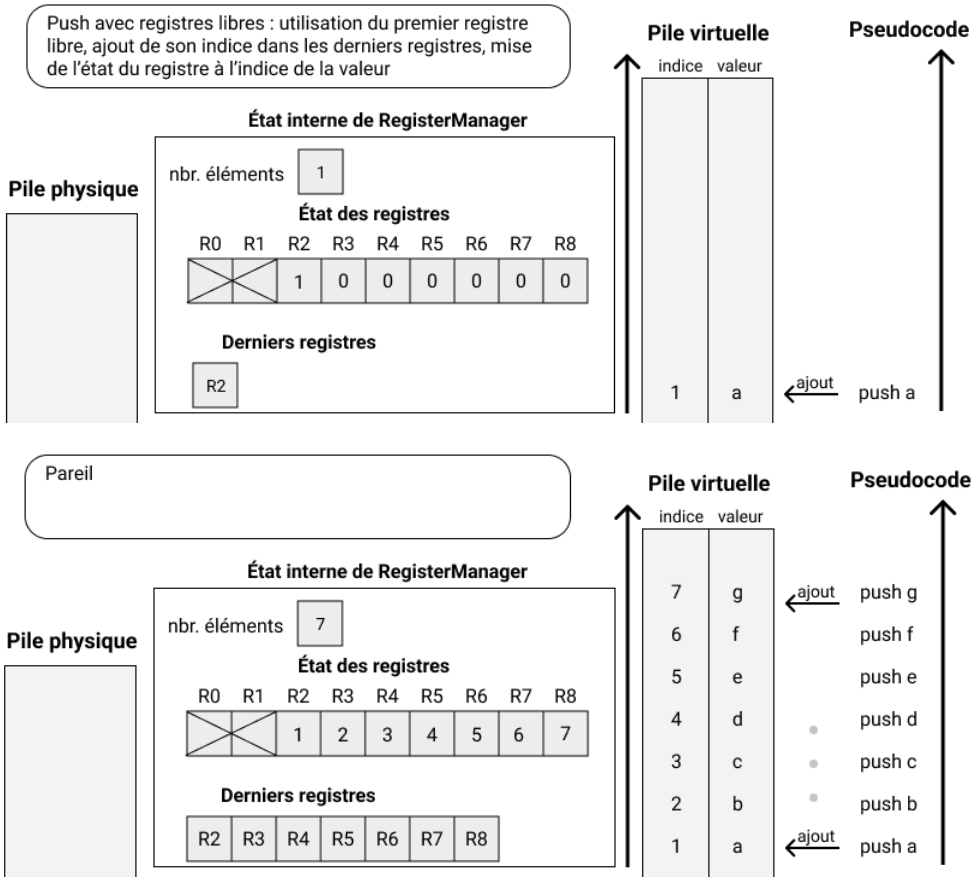


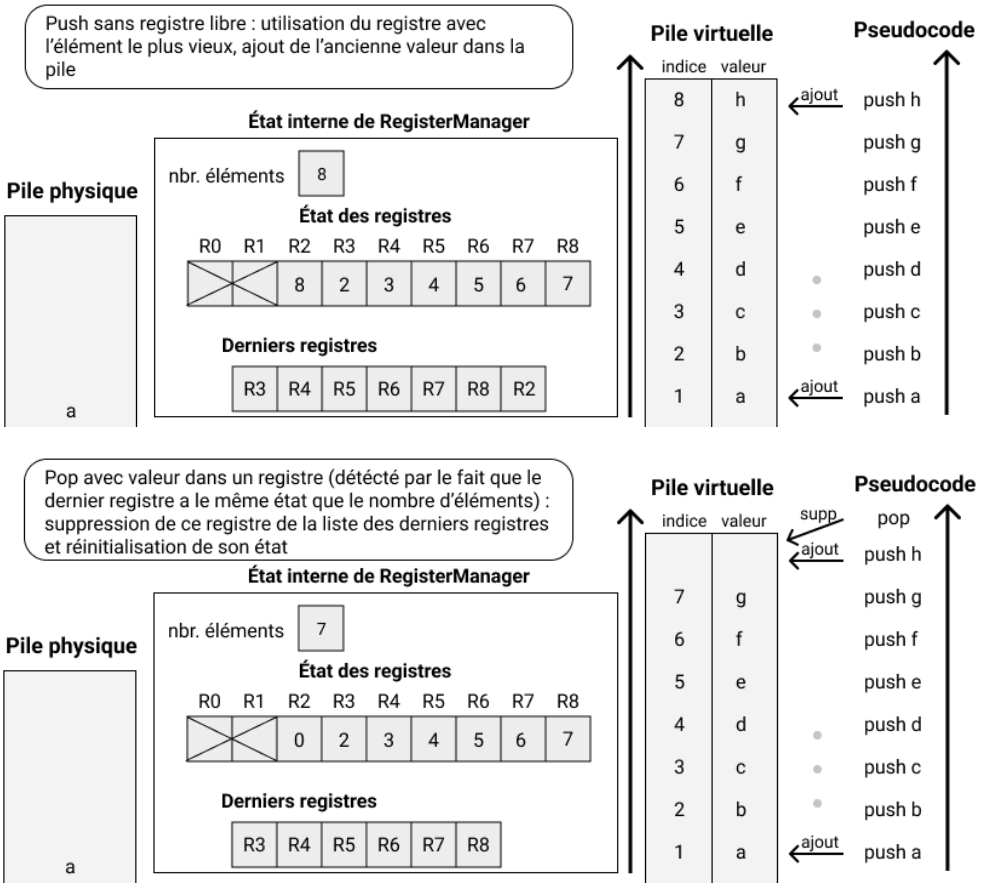
Pour l'optimisation de l'opérateur Not, nous avons utilisé le théorème de De Morgan qui permet d'utiliser l'opérateur Not sans cycle supplémentaire. Dans les cas où aucune optimisation n'était possible ('!methode()' par exemple), nous n'avons pas Override la méthode codeGenNotExpr et nous avons implémenté un cas par défaut : "1 - <expression>", qui coûte deux cycles supplémentaires.

Gestionnaire des registres

Le gestionnaire de registres sauvegarde 3 informations importantes: le nombre d'éléments sur la pile virtuelle (attribut "nbVarsStack"), l'état de chaque registre (attribut "registers") et

la liste des derniers registres utilisés (attribut “lastRegisters”). Ces informations permettent d’implémenter la pile virtuelle.
Lorsque l’on ajoute une valeur sur la pile virtuelle, si un registre est libre, elle est mise dans ce registre. Sinon, le registre dont la valeur a été ajoutée il y a le plus longtemps est libéré en mettant sa valeur sur la vraie pile, et la valeur est mise dans le registre.
Pour placer une valeur dans un registre, on ajoute l’instruction “LOAD <valeur>, <registre>”, on augmente le nombre d’éléments sur la pile virtuelle, on établi l’état du registre à cette nouvelle valeur du nombre d’éléments sur la pile, et on ajoute l’indice du registre en tête de la liste des derniers registres utilisés. Pour l’enlever, on fait exactement l’inverse: on enlève le dernier élément de la liste des derniers registres utilisés, on met l’état du registre correspondant à 0, et on diminue le nombre d’éléments dans la pile virtuelle.




Push sans registre libre : utilisation du registre avec l'élément le plus vieux, ajout de l'ancienne valeur dans la pile
Pile virtuelle
Pseudocode
indice valeur
8 h ajout push h
7 g push g
6 f push f
5 e push e
4 d push d
3 c push c
2 b push b
1 a ajout push a
État interne de RegisterManager
nbr. éléments 8
Pile physique
État des registres
R0 R1 R2 R3 R4 R5 R6 R7 R8
8 2 3 4 5 6 7
Derniers registres
R3 R4 R5 R6 R7 R8 R2
a
Pop avec valeur dans un registre (détécté par le fait que le dernier registre a le même état que le nombre d'éléments) : suppression de ce registre de la liste des derniers registres et réinitialisation de son état
Pile virtuelle
Pseudocode
indice valeur
supp pop
ajout push h
7 g push g
6 f push f
5 e push e
4 d push d
3 c push c
2 b push b
1 a ajout push a
État interne de RegisterManager
nbr. éléments 7
Pile physique
État des registres
R0 R1 R2 R3 R4 R5 R6 R7 R8
0 2 3 4 5 6 7
Derniers registres
R3 R4 R5 R6 R7 R8
a