Thomas CASSAGNES
Angelica FIGUEROA
Antoine FLICHY
Mathieu VALLICCIONI
Joël ZHU

Groupe 48

30/01/20

# Documentation de conception

## Commande decac

Le compilateur est composé d'une classe principale DecacMain, cette classe contient un attribut de la classe CompilerOptions qui est utilisée pour identifier et gérer les options que l'utilisateur entre avec la commande decac. La classe CompilerOptions a des attributs booléens pour chacunes des options decac qui indiquent si l'option correspondante est utilisée pour la compilation ou pas.

Le fonctionnement général de la classe CompilerOptions est décrite ci-dessous :

1. Tout d'abord, le vecteur args est itéré, pour identifier quelles options ont été saisies par l'utilisateur, une fois qu'un élément args correspond à une option, la variable booléenne correspondante est définie à true. Quand un élément d'args ne correspond pas à l'une des options, nous voyons si l'élément est une valeur numérique qui peut être prise comme la valeur de X pour l'option register, si ce n'est pas le cas, l'élément dans args est pris comme le path source pour le fichier deca.

2. Une fois l'itération du vecteur args terminée, il y a trois conditions *if* pour les cas suivants :
    - L'utilisateur n'a pas entré au moins un chemin source.
    - L'option parallel a été saisie mais un seul chemin source a été entré.
    - Ni l'option parse ni verify ont été entrée, auquel cas la variable allCompilation est définie à true.

3. Ensuite, on défini le niveau de trace du logger. Pour ce faire, nous utilisons la valeur de la variable *debug*, qui est incrémentée à chaque fois que l'option -d apparaît dans le vecteur args. Nous décrivons ici la valeur attribuée à chaque niveau:
    - QUIET = 0
    - INFO = 1
    - DEBUG = 2
    - TRACE = 3

    Ainsi, pour activer le niveau de trace 3, il faut appeler l'option -d autant de fois, exemple : decac -d -d -d fichier.deca.

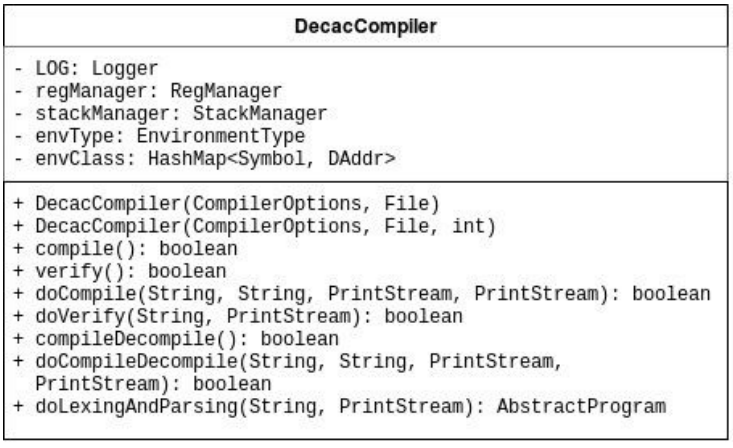
La classe DecacCompiler a un attribut de type CompilerOptions, donc chaque fois que DecacCompiler est instancié, l'instance de CompilerOptions est également initialisée.

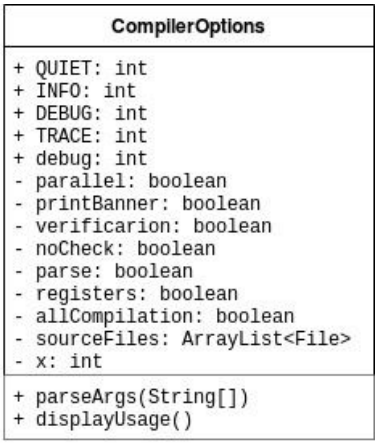
Dans la classe DecacCompiler, un second constructeur a été ajouté, la différence entre les deux constructeurs est le nombre d'arguments que chacun reçoit, le second constructeur prend les mêmes arguments que le premier mais un argument int est ajouté, l'objectif de cet argument est de définir le nombre de registres disponibles dans l'attribut regManager lorsque l'option register est saisie par l'utilisateur.

D'autres méthodes ont également été ajoutées à la classe DecacCompiler, qui sont appelées de la classe DecacMain.
Chaque méthode est brièvement décrite ci-dessous:

- *compileDecompile et doCompileDecompile* sont utilisés lorsque l'option parse est entrée par l'utilisateur et leur objectif principal est de décompiler et afficher le programme obtenu à partir de l'arbre abstrait.
- *verify et doVerify* sont utilisés lorsque l'option verify est saisie par l'utilisateur. La tâche principale de cette méthode est de faire le processus de vérification, les messages d'erreur ne s'affichent qu'en cas de détection d'erreurs.

Enfin, la classe DecacMain est celle qui vérifie les options identifiées par CompilerOptions et selon elles, elle procède à effectuer une tâche spécifique, par exemple afficher le nom de l'équipe lorsque l'option banner est entrée par l'utilisateur ou faire la compilation en parallèle.



## Conception du compilateur

Pour compiler un programme source deca et générer le code assembleur lié, plusieurs étapes préliminaires sont nécessaires.
Voici les étapes successives de la compilation, une description de chaque étape est fourni dans la suite du document :

1. Analyse lexicale
2. Analyse syntaxique
3. Vérification des locations des lexèmes
4. Affichage de l'arbre abstrait
5. Décompilation
6. Analyse contextuelle
7. Génération du code assembleur

## Analyse lexicale

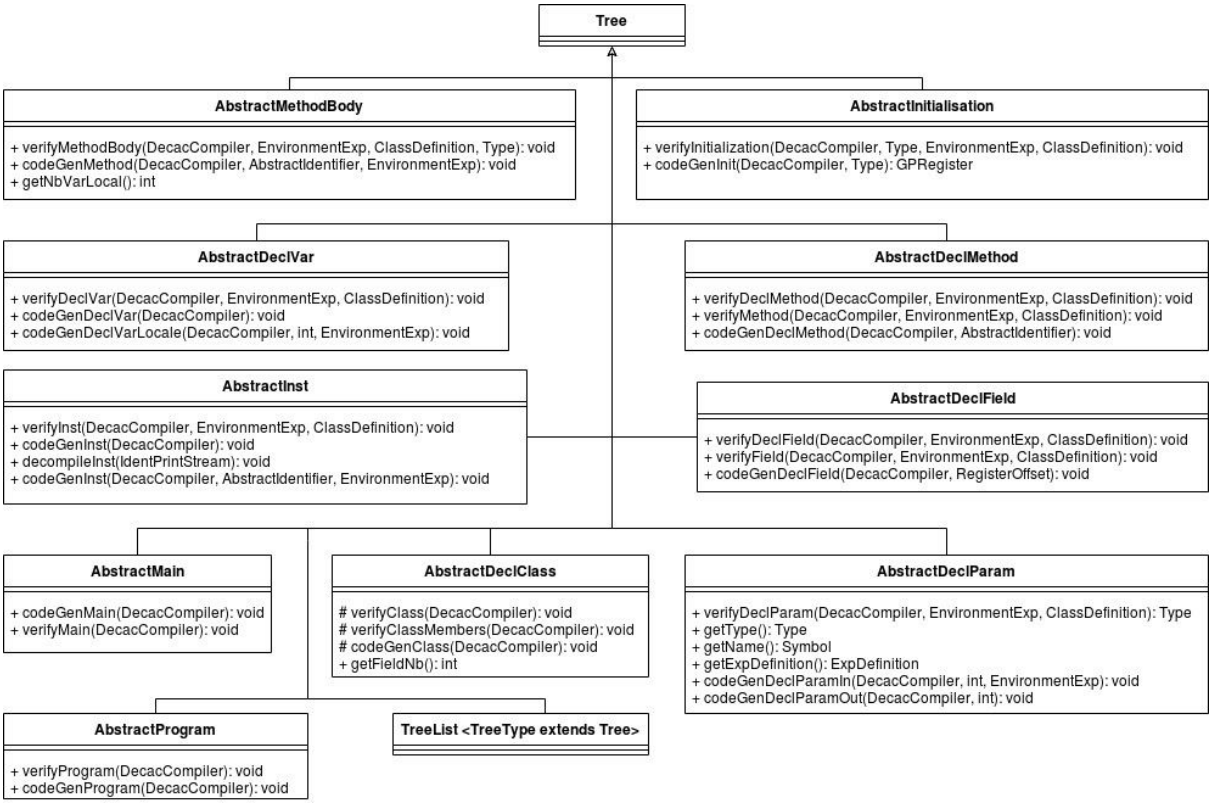
La lexique des programmes deca est décrite dans le fichier src/main/antlr4/ensimag/deca/syntax/DecaLexer.g4.

Ce fichier permet de définir tous les caractères, symboles et mot-clés reconnus par le compilateur et les traduit en différents lexèmes, lisibles par l'analyseur syntaxique.
Les spécifications des règles lexicales sont décrites dans le poly enseignant.
L'inclusion de fichier est un cas exceptionnel, l'analyse syntaxique de celle-ci est directement géré dans le lexer par l'appel à la fonction *doInclude()*.

## Analyse syntaxique

La syntaxe des programmes deca est décrite dans le fichier src/main/antlr4/ensimag/syntax/DecaParser.g4.
L'analyse syntaxique permet d'associer à chaque lexème un comportement compilateur correspondant.
Les spécifications des règles syntaxiques sont décrites dans le poly enseignant.
En l'occurrence, pour un groupement spécifique de lexèmes, on instancie un objet qui permet de définir le comportement de l'instruction quant aux étapes suivantes.

Classes abstraites représentants les instructions possibles :




## Vérification des locations des lexèmes

A chaque instruction deca aperçu par le parser est associé sa position dans le fichier. Bien que cela n'est pas nécessaire, c'est fortement recommandé à des fins de débogage.

## Affichage de l'arbre abstrait

Chaque instruction correspond à un noeud dans l'arbre abstrait qui sera ensuite lu dans le bon ordre.

## Analyse contextuelle

L'analyse contextuelle consiste en 3 parcours de l'arbre abstrait généré précédemment et la décoration de cet arbre. La grammaire de la vérification contextuelle est décrite dans le polycopié du projet.

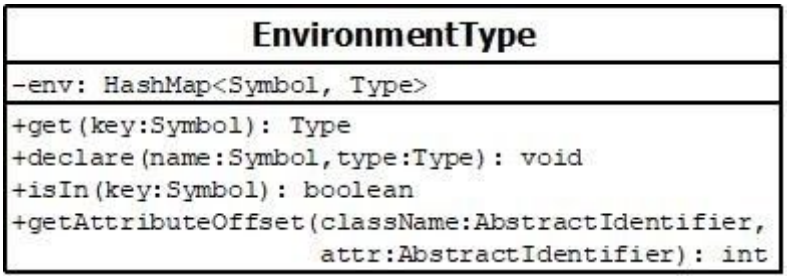
## Décompilation

La décompilation permet d'obtenir un programme deca issus de l'arbre abstrait sans ambiguïté.

## Analyse contextuelle

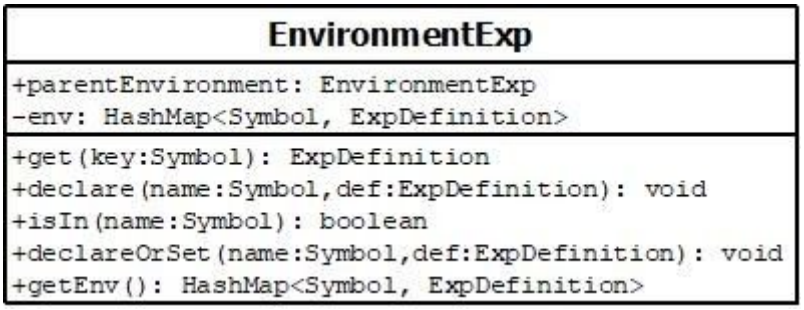
### Les Environnements

L'analyse contextuelle du programme se fait sur l'arbre construit lors de l'étape A (lexer et parser). Dans cette partie du compilateur, l'arbre est parcouru 3 fois et des instances d'objets sont créées pour les noeuds en fonction de leurs natures et de leurs besoins.

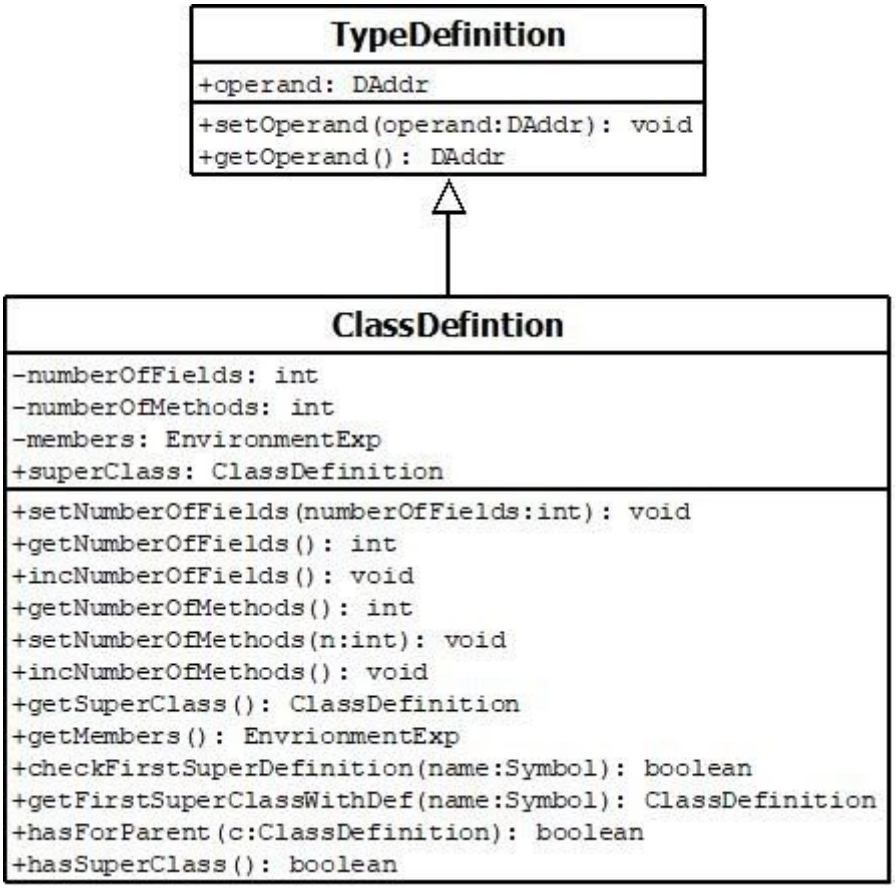



Dans cette classe le dictionnaire HashMap<Symbol, Type> a été ajouté, il correspond à l'environnement des types du programme, c'est à dire qu'il contient tous les types de variables et les classes. L'environnement des types est instancié une seule fois par programme : dans l'objet DecacCompiler que l'on nomme *compiler* dans nos fonctions. A noter que l'initialisation des types de base de Deca (int, float, void...) se fait dans le constructeur de EnvironmentType. Le choix de la HashMap nous a semblé intéressant car l'environnement des types est une structure de données qui est voué à être régulièrement sollicité par la recherche et l'insertion d'éléments.

Par exemple pour la déclaration d'une classe A : on va regarder si cette classe n'est pas déjà définie (recherche d'un élément dans la HashMap), puis on va la déclarer (insertion d'un élément dans la HashMap). La HashMap a justement des coût de l'ordre de O(1) en ce qui concerne l'accès et l'insertion d'éléments. Cette structure semblait donc toute indiquée pour nos besoins. Par la suite des fonctions comme *declare* et *isIn* ont été ajouté pour les fonctionnalités décrites précédemment.




A l'instar de la classe EnvironmentType, EnvironmentExp est une classe que nous avons enrichie avec une HashMap. L'environnement des expressions contient les différentes variables, objets, champs, paramètres et méthodes qui sont déclarés dans les programmes Deca. Son fonctionnement est donc similaire à celui de l'environnement de types : il faut régulièrement contrôler la présence d'un élément déjà déclarer et insérer de nouveaux éléments.




Lors de la passe 3 dans la partie avec objet, il est régulièrement nécessaire de chercher si une classe a un parent qui contient un champ ou une fonction. C'est le cas par exemple lorsqu'il faut déclarer une méthode. En effet, lorsque l'on veut déclarer une méthode avec un certaine signature, il faut rechercher si cette méthode

n'a pas déjà été définie dans une classe parente et récupérer sa signature ainsi que son type de retour pour contrôler que la redéfinition se fait correctement.

Afin de récupérer la signature et le type de retour, il faut récupérer la définition de cette méthode situé dans la première classe parente qui la contient en partant de la classe actuelle. C'est le rôle des fonction *getFirstSuperClassWithDef* et *checkFirstSuperClassDefinition*. Ces méthodes de recherche sont récursives car la programmation récursive se prête bien au parcours d'une chaîne d'objet (ici chaine de class imbriquées).

Le contenu principal de l'étape B réside dans la complétion (parfois leur création) des méthodes *verify*. Pour toutes les classes qui peuvent représenter un noeud de l'arbre il existe une fonction *verify* associée, et les différentes fonction *verify* s'appellent entrent elles pour réaliser la vérification totale de l'arbre abstrait. Le contenu des fonction vérifie a été créé en se basant sur la grammaire de la syntaxe contextuelle du polycopié du projet.

## Génération du code assembleur

Pour chaque instruction du programme deca, un code assembleur est généré. La génération assembleur est appelé par *codeGenProgram()*.

Celle-ci itère récursivement sur l'arbre pour générer le code de toutes les instructions.

Les déclarations de variables sont d'abord générés avant les instructions. Ainsi, il est impossible de déclarer de nouvelles variables après la première instruction du programme deca.

Lors de la déclaration d'une variable, on vérifie que son type existe dans notre environnement. Le type peut être natif comme un entier, un flottant, un booléen, ou cela peut être un nouvel objet d'une classe. Ces derniers doivent alors être d'abord créer pour pouvoir être utilisées.

Une fois la variable créée, on indique que celle-ci existe désormais en l'ajoutant dans l'environnement du compilateur. On indique également l'emplacement où la variable a été stockée dans la pile en modifiant son attribut *operand*.

Pour des variables globales, on utilisera la pile GB. Pour des variables locales, on utilisera LB.

Chaque noeud de l'arbre possède une ou plusieurs fonctions CodeGen. Plusieurs fonctions CodeGen différentes sont nécessaires car les noeuds produisent du code assembleur différent en fonction de leur noeud parent.

Ainsi, il est logique de penser qu'un noeud IntLiteral doit produire un code assembleur différent en fonction de si ce dernier correspond au paramètre d'une instruction print ou à un immédiat par exemple. Le noeud va alors produire respectivement WINT ou LOAD.

Pour une gestion des registres efficace, une fonction codeGenReg existe lorsque nécessaire. Cette fonction renvoie le registre source utilisé par l'instruction générée par le noeud. Ce registre est ensuite utilisé par le noeud parent puis libéré.

Par exemple : Pour réaliser une opération binaire entre deux immédiats, il est nécessaire de d'abord charger la valeur des immédiats dans des registres. Ces registres sont directement utilisés par le noeud père pour réaliser son instruction.

**Implémentation des classes en assembleur**

Il y a deux passes pour la génération des classes en assembleur. La première consiste à générer la table des méthodes.

Pour un programme deca, la classe Object sera toujours implicitement défini, sans que l'utilisateur ait à le déclarer. Ainsi, le compilateur générera toujours un code assembleur associé à une classe Objet contenant la fonction *equals*.

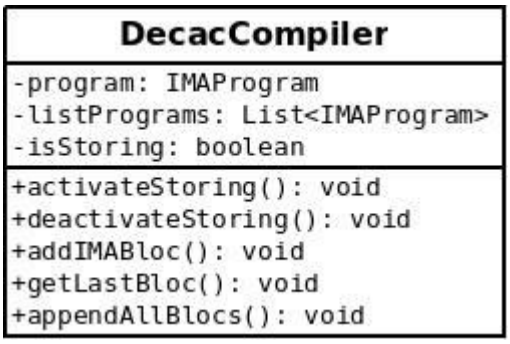
Chaque classe possède un environnement local qui connaît également l'environnement de sa classe parente si elle étend une classe. Elle parcourt récursivement ses parents pour générer un label pour chacune de ses fonctions.

La génération du code assembleur pour le corps d'une méthode n'a été que partiellement implémentée. En effet, l'architecture de l'étape C a d'abord pour faire fonctionner la partie sans objet.

Lorsque le temps était venu d'attaquer la partie objet, nous nous sommes rendu compte trop tard que l'architecture était maladroite. En effet, plusieurs raccourcis ont été employés mais qui ne sont pas réutilisables pour la partie objet. Par exemple : la génération de code pour les instructions manipule directement la base globale. Hors, lorsqu'on rentre dans le corps d'une méthode d'une classe, il est nécessaire de changer de base pour manipuler la base locale ou même les registres. Par

faute de temps, nous n'avions pas pu repenser celle-ci pour implémenter le corps des méthodes.

Cependant, certaines fonctionnalités ont quand même pu être implémentées comme la sélection par le mot-clé *this* qui ira effectivement chercher le registre de la base locale. Une logique similaire serait à implémenter pour les opérandes, les comparaisons, etc.




**Gestion des blocs assembleur**

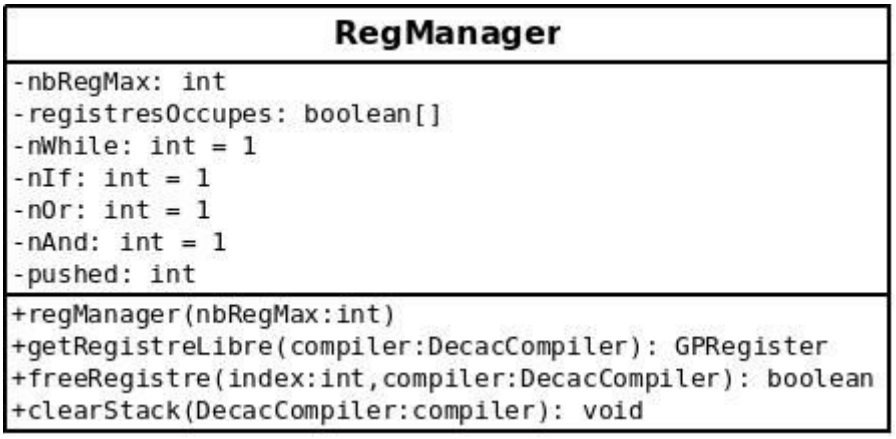
Comme mentionné précédemment, le code de l'étape C a été principalement conçu pour la partie sans objet. Ainsi, la gestion de différents blocs de code assembleur n'était pas implémenté à la sortie de la partie sans objet car étant donné qu'il n'y avait qu'un bloc main, le code assembleur pouvait également être traduit en un bloc monolithe. Afin de pouvoir manipuler plus finement les lignes de code, ce qui est nécessaire pour insérer les TSTO par exemple en début de bloc, il était nécessaire de repenser la structure du programme tout en maintenant la compatibilité.

Ainsi, la classe *DecacCompiler* est composé d'un attribut *IMAProgram* qui représente le bloc assembleur principal et d'une liste dynamique d'*IMAProgram* qui permettent d'insérer avant ou après le bloc principale d'autres blocs assembleur.

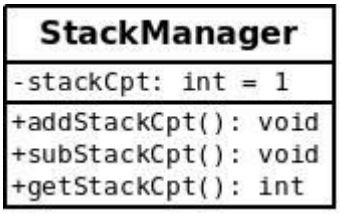
Cette liste *IMAProgram* est utilisé pour insérer par exemple les labels des méthodes avec leur corps de méthode après le bloc main.

Un flag *isStoring* est utilisé pour indiquer à *DecacCompiler* si une nouvelle instruction assembleur ajouté au bloc IMA devrait être ajouté au bloc principale ou à une des listes. Si ce flag est actif, on l'ajoute à la liste, sinon au bloc principal. Une fois

les différents blocs crées, la fonction *AppendAllBlocs()* permet de joindre les programmes ensembles.

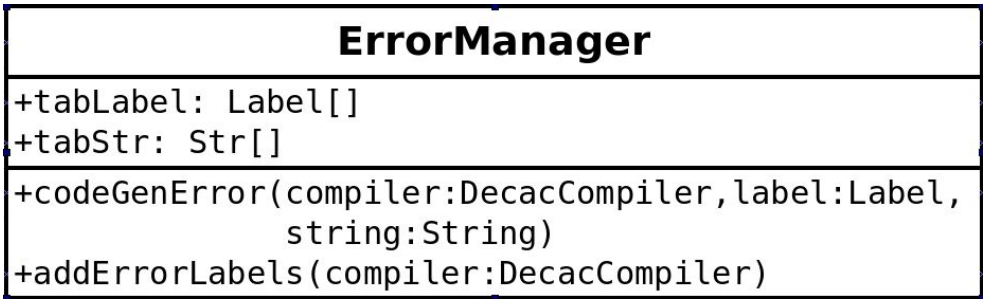



- L'attribut *nbRegMax* correspond au nombre de registres disponibles au total, modifiable en utilisant l'option -r.
- L'attribut *registresOccupes* est un tableau qui associe à chaque numéro de registre un booléen indiquant si le registre est utilisé ou pas. Ce tableau est modifié par la fonction *getRegistreLibre()*, qui renvoie le premier registre libre et modifie sa valeur booléenne correspondante à true pour indiquer que celle-ci est actuellement utilisée. Le registre doit ensuite être libéré par la fonction *freeRegistre()* qui, grâce au numéro du registre, va remettre la valeur dans le tableau à false..
- Les attributs nWhile,nIf, nAnd, et nOr sont des entiers qui permettent d'associer des indices aux labels lorsqu'une instruction While, If, Or, ou And est appelée. Ceci est nécessaire pour différencier les différentes instructions lorsqu'il y en a plusieurs. Cette variable est incrémentée à chaque fois qu'une instruction correspondante et appelée.




Le stack manager est une classe simple qui permet de retenir la position du pointeur de la pile globale. Ainsi, à chaque nouvelle déclaration de variable (c'est-à-dire lorsqu'une instruction STORE est réalisée), on peut positionner

l'emplacement de cette variable dans la pile en récupérant la valeur du compteur. Une fois la déclaration faite, il est important d'incrémenter ce compteur pour permettre à la déclaration suivante d'avoir une position correcte.




Nous avons de plus créé la classe ErrorManager permettant de générer des labels. A partir d'un tableau de label tabLabel, et d'un tableau de message d'erreur tabStr, la méthode addErrorLabels permet d'ajouter les labels d'erreur en fin de programme assembleur en utilisant la méthode codeGenError pour tout couple (Label, String).