PROJET GL

DOCUMENTATION DE CONCEPTION

Grenoble INP-Ensimag

Amine ANGAR
Dy EL ALEM
Mouad TARBOUI
Oussama KADDAMI
Youness LEHDILI

ARCHITECTURE:

Afin de définir l'architecture globale du compilateur, nous commonçons par la classe Decac-Main à partir de laquelle se lance la compilation. Cette dernière comporte une collection de la classe DecacCompiler(cas du lancement parallèle). La classe DecacCompiler possède une méthode deCompile qui permet le lancement effectif du processus de compilation.

De manière non exhaustive, le programme d'entrée est représenté par une classe Program, et donc possède, selon la spécification de la sémantique du langage deca, une fonction principale Main et une liste de classes. Ainsi une classe Main, représentant la fonction principale, permet par le biais de ses attributs de définir une liste de variables déclarées et une liste d'instructions. D'autre part, la classe DeclClass permet, par le biais de ses attributs, de définir les champs et les méthodes d'une classe de notre programme.

Toute classe manipulée dans la phase de compilation hérite de **Tree** permettant ainsi de fournir les champs nécessaires pour le bon déroulement de la compilation et de définir une première structure. En général, nous pouvons dire qu'il existe un premier niveau de classes; notamment les classes qui permettent de définir le programme, la fonctions main, les classes, la déclaration des variables et l'initialisation. Ensuite vient la classe AbstractInst qui représente une instruction de manière générale, et ainsi toute les instructions manipulées héritent de cette classe abstraite. Nous pouvons, encore ici distinguer deux types d'instructions. D'une part, celles qui héritent directement de la classe AbstractInst, à l'instar de Print, Return, while et les branchements conditionnels, et d'autre part celle qui constitue une expression.

concernant la partie B, Le code est organisé en différentes classes, toutes les classes sont dans le fichier src/main/java/deca/tree et représentent les différents noeuds de la grammaire abstraite, chaque classe implémente une fonction de vérification qui vérifie si le contexte est bien respecté pour la sémantique de Deca. L'architecture des classes est représentée dans la figure précédente, le code source est représenté par une instance de la classe **Program** et la vérification contextuelle consiste à propager les fonctions de vérification sur les différentes sous-classes de cette classe. Cette vérification est organisée en plusieurs passes que l'on décrira en détail dans ce qui suit.

Concernant la partie génération de code, la structure initiale définissait une fonction codeGenInst qui permet de générer le code assembleur associé à l'instruction. Dans un premier temps, nous avons changé cette structure, notamment dans le rendu intermédiaire, mais nous l'avons réadapté après durant l'implémentation du langage essentiel. Nous avons

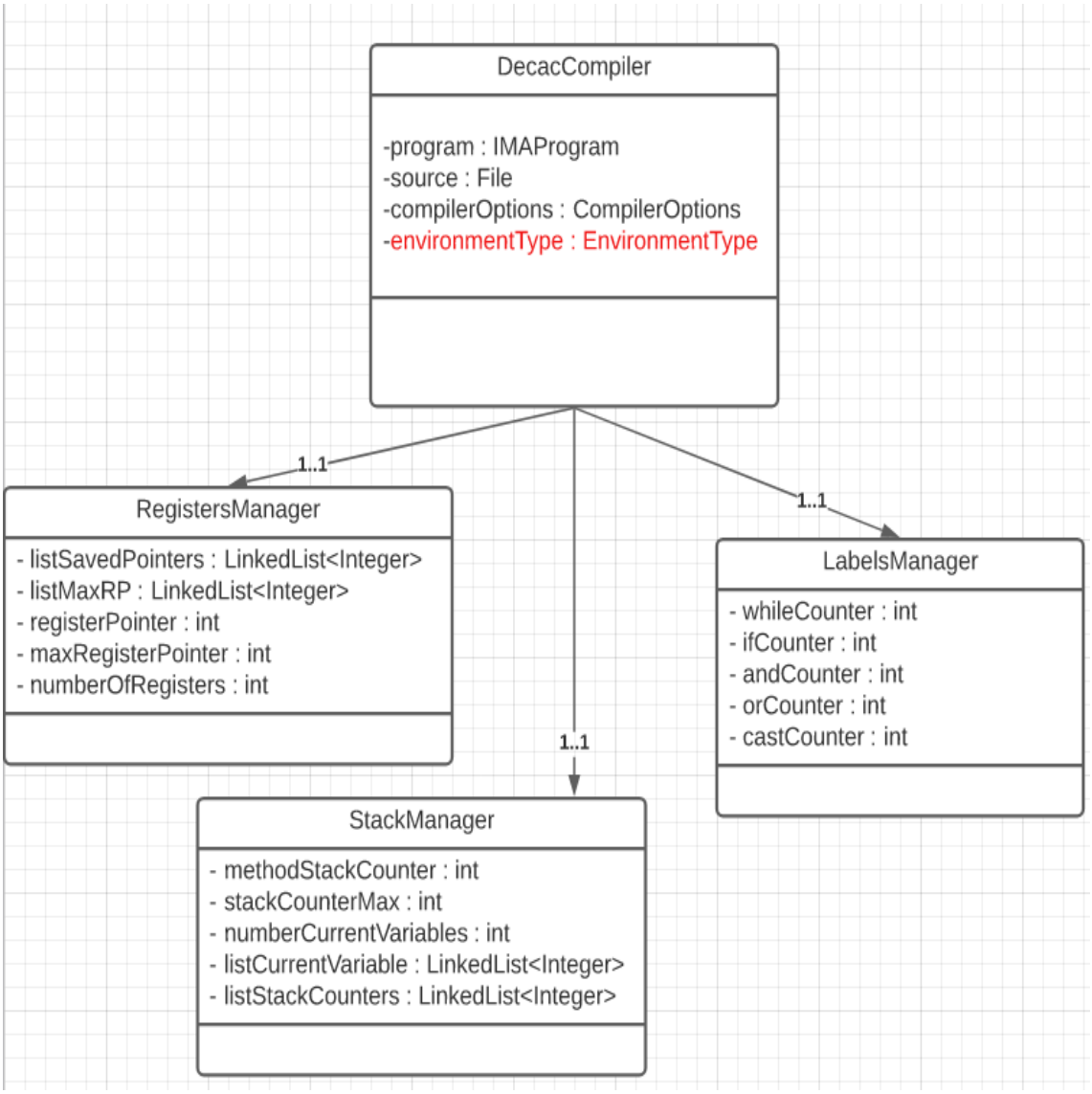
donc gardé cette fonction pour les instructions qui nécessitent de stocker un résultat dans un registre ou les instructions qui héritent directement de AbstractInst, et nous avons défini une fonction codeCond pour les instructions qui engendre des branchement conditionnel sans avoir besoin de stocker un résultat dans un registre.

Nous avons aussi défini des fonctions dans plusieurs classes afin de subvenir à un besoin propre à cette dernière.

SPÉCIFICATIONS:

Durant nos phases de conceptions tout au long du projet nous avons été mené à changer la structure du code donné et à en ajouter de nouvelle structure.

Ainsi, nous avons décidé d'ajouter des champs pour la classe DecacCompiler pour subvenir à nos besoins en termes de gestion des différents pointeurs associés à la pile, la gestion des registres ainsi que des "Labels".



La classe RegistersManger permet la gestion des registres pour des opérations qui nécessitent d'utiliser les registres pour stocker des résultats intermédiaires, L'attribut registerPointer permet donc de contrôler le comportement des opérations et gérer les cas où les opérations nécessitent pus de registres qu'il n'en existe.

La classe StackManager permet de gérer les instructions qui ont une relation avec la pile, notamment la vérification du débordement de la pile, l'allocation de place pour les variables, et la constructions de la table des méthodes. Ainsi nous avons défini les attributs : • methodStackCounter: qui permet un décompte du nombre de méthodes et ainsi gérer la construction de la table de méthodes dans la pile. • NumberCurrentVariables: qui permet le décompte du nombre de variables "globales" ou "locales" selon le cas, afin de leur allouer des cases mémoire dans la pile. • SackCounterMax: qui permet le décompte progressif du

nombre de mots mémoire utilisé dans la pile afin de tester le débordement de la pile en début de chaque bloc. La classe définit aussi des attributs pour la sauvegarde et la restauration des registres lors d'appel de méthodes et des méthodes conçues pour cette fin.

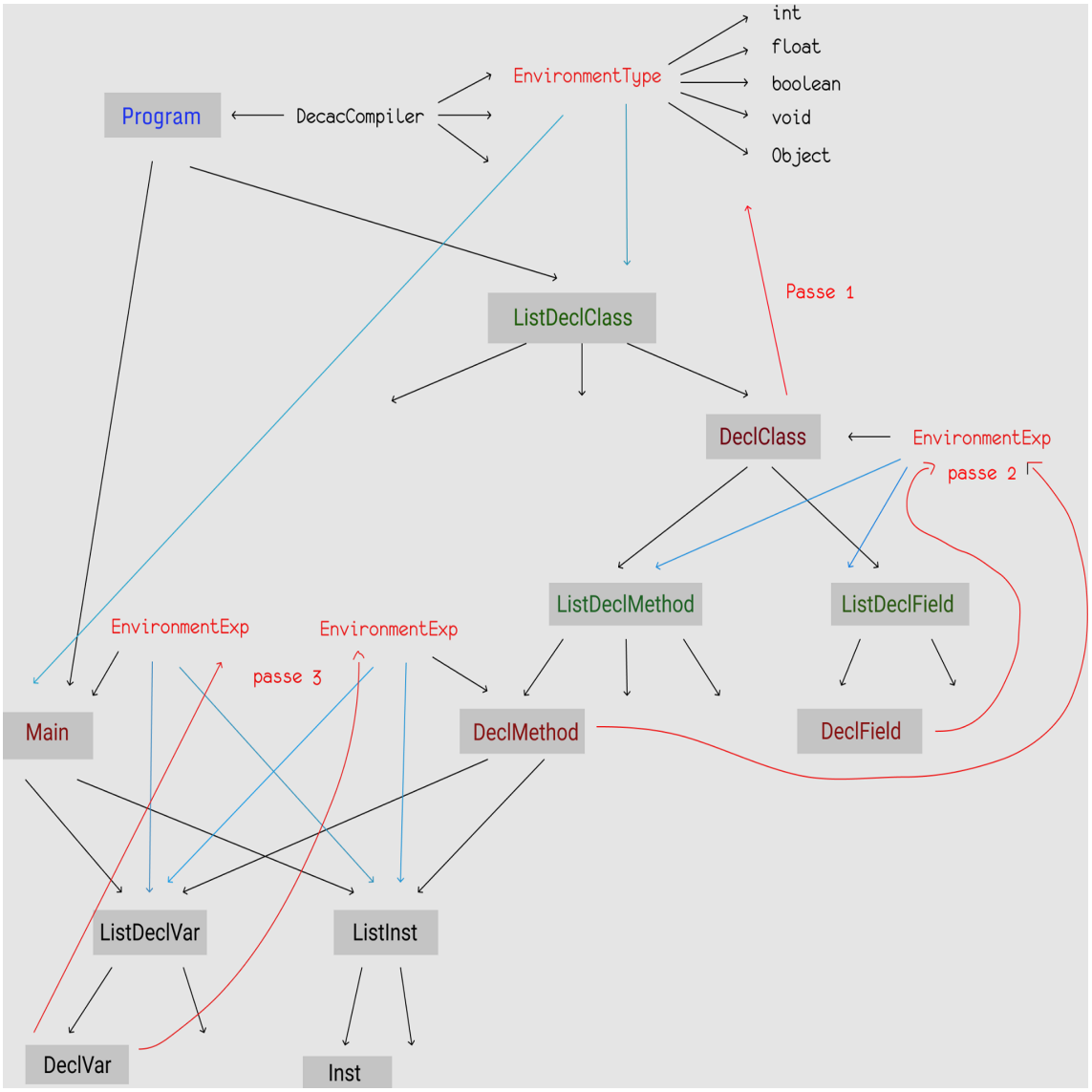
La classe LabelManager permet de gérer les labels dans notre code assembleur. Sans cette classe, l'utilisation de certaines instructions comme les boucles ou les conditions poserait un problème, puisque deux instructions différentes généreraient les mêmes Labels. Cette classe possède les attributs pour une gestion correcte des labels pour les instructions qui pourraient nous poser un problème.

En outre, nous avons ajouté des attributs à d'autres calasses, notamment l'attribut dAddr pour la classeDefinition. Cet attribut nous permet d'associer à chaque classe l'adresse dans la pile associée au début de sa table de méthodes. Nous avons aussi ajouté un champ Type à la classe NoInitialization afin d'initialiser une variable non initialisée à 0, 0.0 ou null selon le type.

DESCRIPTION DES ALGORITHMES ET STRUCTURES UTILISÉS:

Conception Partie B

La partie B implémente la vérification contextuelle du code, l'architecture globale de cette partie B est résumé dans la figure ci-dessous.


int
float
boolean
void
Object
EnvironmentType
Program
DecacCompiler
Passe 1
ListDeclClass
DeclClass
EnvironmentExp
passe 2
ListDeclMethod
ListDeclField
EnvironmentExp
EnvironmentExp
passe 3
Main
DeclMethod
DeclField
ListDeclVar
ListInst
DeclVar
Inst

Environnements

Les environnements sont de deux types, l'EnvronmentType est l'environnement des types il contient les types prédéfinis (int, float, boolean, void et la superclasse Object) auquel on ajoute les classes déclarée. L'EnvironmentExp est l'environnemet des expressions on l'utilise pour stocker les variables déclarée dans le Main, pour stocker les champs et méthodes de chaque classe, et pour la vérification des méthodes de classes en y stockant les paramètres et les déclaration dans le corps de la méthode. Ces environnements sont définis comme des HashMap et associent à chaque Symbol la définition correspondante. Il possèdent chacun une méthode get(Symbol) qui retourne la définition associée au Symbol (si elle existe et null sinon) et une méthode declare pour déclarer une nouvelle definition (si le Symbol existe déjà on lève une exception DoubleDefException). L'environmentExp est défini comme une liste chaînée d'EnvironmentExp et sa fonction get a été défini de telle façon qu'elle cherche la première occurrence dans la liste chaînée, ce qui permet de ne pas ajouter les différents champs et méthodes de la classe mère aux classes filles et de simplement aller les chercher avec la méthode get, c'est la stratégie qui a été adoptée pour les vérification contextuelles.

1.Passe 1

La passe 1 consiste à vérifier les declaration de classes, en appelant la fonction verifyListDeclClass sur la **ListDeclClass** de la classe **Program**, cette fonction appelle la fonction verifyClass sur chacune des **DeclClass** de la
ListDeclClass
La fonction verifyClass verifie que la superClass est bien définie dans l'EnvironmentType et ajoute la nouvelle classe dans l'EnvironmentType s'il elle est déjà définie cette fonction lève une erreur contextuelle ContextualError .

2.Passe 2

La passe 2 consiste à vérifier la déclaration des différents champs et méthodes des classes. La fonction verifyClassMembers est appélée sur chacune des classes, appelant ainsi les méthodes verifyDeclField et verifyDeclMethod sur chacun des champs et des méthodes.

Vérification des identificateurs :

La classe **Identifier** implémente trois fonctions de vérification selon la sémantique de l'identifiant:

- verifyType : cette fonction vérifie les identifiants de type, elle vérifie que le type utilisé est bien définie dans l'EnvironmentType.
- verifyExpr : cette fonction vérifie les identifiants des expression , elle vérifie que l'expression utilisée est bien définie dans l'environnement local (EnvironmentExp) ou dans l'environnement de la classe courante et choisi la bonne définition si elle est definie dans les deux. Elle gère aussi l'accès au champs protégés.
- verifySelection : cette fonction vérifie les sélections des champs et des méthodes , elle vérifie que le champs ou la méthode sélectionnée est bien définie et elle gère aussi l'accès au champs protégés.

Vérification de l'initialisation :

La fonction verifyInitialization sert à vérifier l'initialisation lors de la déclaration, elle utilise la fonction assignCompatible de la classe **Type** pour vérifier que les types sont compatibles pour l'initialisation.

Vérification des déclarations de champs :

La fonction verifyDeclField utilise les différentes fonctions de vérification sur les identifiants et la fonction de vérification de l'initialisation pour vérifier l'exactitude de la déclaration du field et l'ajoute à l'EnvironmentExp de la classe.

Vérification des déclarations de méthodes :

La fonction verifyDeclMethod utilise les différentes fonctions de verification sur les identifiants pour vérifier l'exactitude de la déclaration de la méthode elle vérifie aussi le respect

de la syntaxe de la rédefinition de méthodes dans le langage Deca, et ajoute la méthode à l'EnvironmentExp de la classe.

3.Passe 3

La passe 3 consiste à vérifier les corps de méthodes et le Main, elle vérifie les corps des méthodes via la fonction verifyMethodBody appélée sur chaque méthode de chaque classe, et vérifie le Main par la méthode verifyMain appélée sur la class **Main** de la classe **Program**. Ces vérifications se résument en les vérification des déclaration de valeur et des instructions.

Vérification des déclarations de valeurs :

Ces vérifications se font par la méthode verifyDeclVar qui utilise les différentes fonctions de vérification sur les identifiants et la fonction de vérification de l'initialisation pour vérifier l'exactitude de la déclaration.

Vérification des instructions :

La vérification des instructions se fait par la fonction verifyInst qui est redéfinie pour chaque instruction. Cette vérification est totalement basée sur la vérification des expression que comporte chaque instruction.

Vérification des expressions :

La vérification des expressions se fait par les méthodes suivantes:

- verifyExpr : cette fonction est redéfinie pour chaque type d'expression, selon la vérification nécessaire. Elle sert à vérifier les différentes expressions, des expression binaires, unaires , cast, instanceof ... elle se sert de différentes fonctions définie dans la classe **Type** comme assignCompatible, subType, castCompatible, typeInstanceOfop ...
- verifyRValue : cette fonction vérifie les compatibilités de types elle prend en entrée un type et vérifie si type de l'expression est compatible avec ce type par la fonction assignCompatible elle convertie aussi les type int en float si nécessaire par la classe **ConvFloat**. Elle est utilisée sans plusieurs instructions comme **Return** et **Assign**.

- verifyCondition : cette fonction vérifie que l'expression est bien une condition elle est utilisée par les instructions **Whiel** et **IfThenElse**.

CONCEPTION DE LA PARTIE C:

La partie C permet de générer le code en machine abstraite à partir de l'arbre décoré conçu pendant les parties précédentes. Ainsi, il faut associer à chaque instruction une fonction qui permet de générer le code assembleur associé.

Pendant le premier incrément, l'enjeu était de bien comprendre la structure du code fourni pour comprendre ce qu'il fallait implémenter pendant les incréments suivant.

Fonctions de génération du code:

Dans un premier temps, pour l'incrément sans objet, nous avons choisi de suivre les algorithmes présentés par les diapositives d'introduction. Nous avons défini donc pour chaque classe héritant de la classe AbstractInst, une fonction codeExp qui prend en paramètres le DecacCompiler et le registre dans lequel nous souhaitions produire le résultat de l'instruction. Comme stipulé par le cahier de charge, les expressions étaient évaluées de gauche à droite.

La gestion d'un manque de registre est faite directement dans cette fonction. Notre raisonnement était le suivant: "Tout commence dans la classe Program qui appelle la fonction codeGenMain de la classe **Main.** Cette dernière itère sur la liste des instructions pour générer le code assembleur associé à l'aide de la fonction codeGenInst . Dans le cas de la partie sans Objet, l'appel à la fonction codeGenInst revenait à appeler la fonction codeExp avec comme paramètre le premier registre non-scratch **R2**. Ensuite, la fonction codeExp évalue les valeurs de ses opérandes an appelant la fonction codeExp sur ses opérandes avec le bon registre en paramètre. Ainsi, comme nous appelons la fonction codeExp avec le registre dans lequel le résultat est censé être renvoyé, le dépassement de capacité au niveau des registres ne peut avoir lieu que pour les fonctions nécessitant plus qu'un registre pour effectuer leurs calculs, et c'est justement le cas des instructions binaires (I.e possédant deux opérandes) manipulant des valeurs non immédiates. Ainsi, pour de telles instructions, nous mettant en place un processus de gestion du dépassement du nombre de registres. Tout d'abord, nous évaluons l'opérande gauche dans le registre fourni par l'appel, et avant l'évaluation de l'opérande droite nous testons à l'aide d'un branchement conditionnel si le nombre maximal de registre est égal à l'indice du registre (paramètre de la fonction codeExp). Si c'est le cas, nous appelons une fonction qui gère le dépassement en sauvegardant le résultat du registre **Rmax** dans la

pile et en restaurant son résultat dans le registre R1 après l'évaluation de l'opérande droite pour finir le calcul. Sinon, l'algorithme poursuit son calcul de manière normale."

Ainsi, pendant l'incrément sans objet, toutes les classes héritant d'**AbstractInst** possédaient les fonctions **codeExp** et **codeGenInst.** La fonction **codeGenInst** permettait simplement d'appeller la fonction **codeExp** avec comme paramètre le compiler et le registre R2. À ce stade, une optimisation était bien sûr prévue pour une première optimisation du code.

Durant la conception de l'incrément "**Langage essentiel**", nous avons décidé de revenir à la structure initiale en éliminant les fonctions codeExp et de céder la gestion du dépassement de capacité des registres à la fonction codeGenInst. Pour ce faire, nous avons défini la classe RegistersManager dans le paquet "fr.ensimag.deca.codegen". L'attribut registerPointer de cette classe permet de réguler l'utilisation des registres. En appelant la fonction **codeGenInst**, cette dernière commence par utiliser le registre pointé par registerPointer et utilise les registres d'indices supérieurs à celui du registerPointer pour stocker les résultats intermédiaires dans le cas d'un non dépassement de la capacité des registres. Par exemple, étant donné que le registrePointer est égale à deux et nous souhaitons appeler une fonction binaire suivante $(x+y)*(x+y)$. La fonction **codeGenInst** associée à l'opération de multiplication commence par évaluer l'opérande droit (i.e x+y). Elle appelle donc la fonction **codeGenInst** associée à l'opération d'addition. Cette dernière charge la valeur de x dans le registre R2 comme le registerPointer est égal à 2. Ensuite cette fonction a besoin d'utiliser le registre **R3** pour charger la valeur de l'identificateur y avant d'effectuer l'addition. Ainsi elle incrémente la valeur du registerPointer et appelle la fonction **codeGenInst** associée à la classe Identificateur pour charger la valeur de y dans R3(RegisterPointer étant 3). Après la fin de ce "load" , la fonction **codeGEnInst** décrémente la valeur du registerPointer faire en sorte que le résultat de l'addition soit dans le registre avec le même indice qu'au début de la fonction, Ainsi pour nous assurons du bon déroulement de ce scénario et sous l'emblème de la programmation défensive nous faisons un snapShot de la valeur du registerPointer au début de la fonction et on fait une assertion d'égalité avec la valeur du registerPointer à la fin.

Concernant, les expressions(instructions) produisant un résultat binaire, il est recommandé dans l'énoncé de faire des fonctions qui permettent de générer directement un code de branchement conditionnel sans passer par le calcul et le stockage dans un registre et enfin l'évaluation du résultat pour effectuer un branchement conditionnel. Ces fonctions sont donc utilisées lors de la manipulation des boules(while) et les structures conditionnelles. Toutefois, nous avons toujours besoin des fonctions **codeGenInst** associés à de telles expressions, dans le cas où l'on doit par exemple stocker le résultat dans une variable.

Save And restore:

Concernant les structures que nous avons associées comme attributs au DecacCompiler, certains des attributs de ces structures doivent être conservés lors d'un appel à une fonction et restaurés après la fin de la génération du code associé à ce bloc comme le listStackCounters ou encore listCurrentVariable du stackManager , Ainsi, nous avons décidé d'utiliser des structures de stockage de données **FIFO** comme les linkedList qui permettent ceci.

Table des méthodes

Concernant la création des tables de méthodes, une première étape se passe dans la partie B, et permet d'associer aux méthodes d'une classe les bons indices. Le fait d'associer les mêmes indices aux fonctions redéfinies dans le cas d'un héritage permet une gestion simple des conflits qui peuvent en découler. La fonction qui permet la génération du code assembleur, permettant de mettre en place la table de méthodes des classes dans la pile, est la fonction **buildTable** de DeclClass. Notre algorithme commence par associer les labels aux fonctions propres à la classe à l'aide de la fonction **setLabels** . Ensuite, la fonction **buildTable** de la ClassDefinition associée à la classe permet de construire le tableau des méthodes. Et enfin, notre fonction **codeTable** permet de générer le code assembleur pour chaque méthode.

Méthodes et initialisations:

Concernant la génération du code assembleur des méthodes des classes, notre algorithme utilise les mêmes fonctions utilisées lors de la génération du code pour la fonction main. Il ne restait plus qu'à gérer certains comportements différents comme la déclaration des variables de la méthode en faisant un offset par rapport au registre **LB** et non pas **GB.**

Concernant l'initialisation des champs, l'algorithme utilise les mêmes fonctions utilisées pour les déclarations des variables. Dans le cas où la variable n'est pas initialisée, nous l'initialisons avec la valeur 0, 0.0 ou null selon son type.

InstanceOf:

Concernant la méthode InstanceOf nous avons décider de coder en assembleur une fonction qui prend en paramètre l'adresse de l'objet dans le tas et l'adresse de la table de méthode associée à la classe pour nous renvoyer un booléen testant si l'objet est une instance de la classe. Cela permet donc d'éviter des duplications inutiles dans le code. Ainsi, l'utilisation

de l'instruction instanceOf revient à passer les bons paramètres à la méthode qui après son appel se charge du reste.