Documentation de conception

Projet Gl

groupe :10, équipe :53

Sommaire

I. Etape A :

Lors de cette partie, on vérifie que le programme (*en deca*) soit lexicalement et syntaxiquement correct.

Concernant l'analyse lexical, on a implémenté un lexer en langage ANTLR sous forme d'un fichier **lexer.g4** qui contient tous les tokens possibles.

Cette partie était trop guidée sur le poly, donc pour ajouter d'autres tokens, il faut juste voir l'exemple qu'on a fait, et faire une petite adaptation.

Dans cet ordre d'idées, on a fait de même pour l'analyse syntaxique sous forme d'un fichier **parser.g4** qui gère les positions de chaque élément syntaxique du programme. Voire l'addition des fichier.java qui constituent l'architechture de notre compilateur.

Pour la partie Objet, on a fait une analogie par rapport au squelette de base tout en implémentant d'autre classes suivant la même logique de la partie Sans Objet.

NB : Le répertoire **tree/** contient les feuilles et les noeuds de l'arbre primitif sous forme des classes fille de la classe **Tree.**

II. Etape B :

Il va sans dire que la sortie de la partie A constitue bien une entrée pour la partie B,

Autrement dit, le **parser** génère un arbre primitif sans aucune définition, alors le rôle de la partie B est le fait de faire une analyse contextuelle tout en ajoutant les types et les définitions, c'est-à-dire, on essaie de faire une conversion de cet arbre primitif à un arbre bien décoré.

De surcroît, cette partie contient seulement 3 passes contrairement au langage JAVA qui contient plusieurs.L'appel à ces passes se fait à travers la méthode **verifyProgram** qui appelle à son tour des *verify* d'autre classe, tout en parcourant l'arbre primitif. Donc à chaque fois on ajoute un noeud ou bien une feuille (classe java) on doit ajouter aussi la méthode verify.

Lors de cette étape, il y'a la notion des environnements, EnvType et EnvExp.

Le premier contient les types prédéfini (int, boolean, float) et ceux des classes définie dans le programme. Par contre le deuxième est utilisé d'une manière à lier une classe à sa classe fille sous forme d'une liste chaînée.

En plus des classes trouvées dans le répertoire **tree/,** cette étape a besoin d'autre classes dans le répertoire **context/** , avec lesquelles on peut bien implémenter ces environnements tout en ajouter les bonnes définitions et les bonnes types.

L'objectif est de bien suivre l'arbre dans un ordre logique tout en respectant les règles contextuelles.

Dans la partie Objet, on a fait un travaille similaire à la partie sans objet.

On a ajouté des classes dans **tree/** tout en faisant une analogie par rapport à celle du sans objet.

Comme on a dit, les deux choses à décorer sont les types et les définitions.

L'architecture de la classe Type est la suivante: c'est la classe à partir de laquelle on hérite la définition type tout en créant les différents types possibles:

IntType, FloatType, VoidType …..et *ClassType*, ce dernier est celui qui mérite le plus d'effort, puisqu'il est directement lié à la définition d'une classe *ClassDefinition.*

La classe Definition a deux classes filles: TypeDefinition et ExpDefinition qui héritent elles mêmes les définitions d'une méthode, champ, variable, classe et paramètre.

III. Étape C:

A. Introduction

Dans cette partie on génère le code assembleur des fichiers .deca en utilisant deux types de fonctions (le package tree est celui dans lequel on a ajouté ces fonctions) : codeGenLoad qui renvoie le registre où on a stocké la valeur de l'expression et génère son code assembleur et codeGenXX(codeGenPrint, codeGenClass, codeGenOp ...) qui ne renvoie rien mais génère les portions du code assembleur en appelant la fonction codeGenLoad .

On ajoute les instruction dans les fichiers .ass en utilisant les fonction du paquetage fr.ensimag.ima.pseudocode.Instructions et la fonction addInstruction de l'objet compiler.

B. RegisterManager

les portions de code non-triviales comme la gestion des registres et de la pile sont factorisés dans le paquetage fr.ensimag.codegen dans les classes InitManager et

RegisterManager, le principe de fonctionnement de ce dernier est le suivant:

- -allocReg: vérifie s'il existe un registre disponible dans la pile "regDispo" , si c'est le cas elle l'alloue en le mettant dans "regNonDispo" (LinkedList) sinon elle push l'ancien registre
  qui se trouve dans regNonDispo et le met dans la pile "regEcrase".
- -freeReg : verifie si regEcrase contient le registre qu'on veut libérer si c'est le cas on ajoute l'instruction POP au compilateur, sinon on retire le registre de la liste des registres non disponibles et on le met dans la pile des registres disponibles .

on utilise aussi une HashSet pour stocker les registres utiliser dans une méthode pour les pusher avant le corp de la méthode dans le code assembleur.

C.Implémentation de la partie Sans Objet:

- déclaration des variables:

Le codage des déclarations consiste à associer une adresse à chaque variable, on a implémenté cette fonctionnalité à l'aide des fonctions codeGenIdent, codeGenOperand et codeGenField de la

classe Identifier, pour les variables locales aux méthodes ont des adresses de la forme 1(Lb), 2(Lb)... et pour les variables globales ont des adresses de la forme 1(GB)...et on a initialisé ces variables à l'aide de la classe Initialization et NoInitialization

- codage des expressions:

-Expression arithmétique: op(e1, e2)

pour générer une expression arithmétique on a besoin de deux méthodes .La première c'est codeGenOp qui ajoute des instructions assembleur à l'objet compiler(comme mul, add, sub …), et la deuxiéme méthodes c'est codeGenLoad qui charge dans les registres la valeur des expressions e1 et e2 et renvoie ces registres.

-Expression booléenne:

les opérateurs booléens sont évalués paresseusement de gauche à droite comme indiqué dans le cahier de charge, les classes responsable sur l'implémentation de ces expressions sont: AbstractOpBool et AbstractOpCmp.

D. Implémentation de la partie Objet:

pour réaliser cette partie nous avons implémenté des nouvelles classes permettant de réaliser le cahier de charge.

- Construction de la tables des méthodes:

elle contient des pointeurs sur la table des méthodes de la super-class et vers le code de chaque méthode de la classe, elle est créée dans le premier passe à l'aide de la méthode récursive codeGenClass de la classe DeclClasse , sauf pour la table de méthode de la classe Object qui est implémentée à l'aide de la fonction codeGenObj de la classe Identifier

- Codage des champs:

le codage des champs est implémentés dans le deuxième passe par les classes ListDeclField, DeclField et DeclClass , le principe de l'implémentation est le suivant: on passe tout d'abord par la fonction codeGenField de la classe DeclClass pour créer le label "init.NomDelaClasse " et on vérifie ensuite si la classe possède une super classe , si c'est le cas on initialise les champs à zéro et on appelle init.SuperClassName, enfin on appelle la fonction codeGenListField de la classe ListDeclField qui fait appelle aussi à la fonction codeGenField de la classe DeclField.

- Codage des méthodes:

L'implémentation des méthodes est gérée dans la classe DeclMethode avec la fonction codeGenMethod qui fait appel à plusieurs fonctions comme codeGenListparam pour générer les paramètres et codeGenMethodbody pour générer le corp de la méthode.

pour écrire les registres utiliser dans une méthode avant le corp de la méthode en assembleur , on a utilisé un nouveau IMAProgram (initialement vide) pour ajouter les instruction au compilateur et en même temps stocker dans un HashSet les registres utilisés,
et enfin on ajoute ses registres au début du programme avec l'instruction addFirst et on merge ce deuxième IMAProgram avec l'ancien.

- Classes spécifiques à l'objet:

Pour la partie Objet, on a fait une analogie par rapport au squelette de base tout en implémentant d'autre classes suivant la même logique de la partie Sans Objet, par exemple:

This:

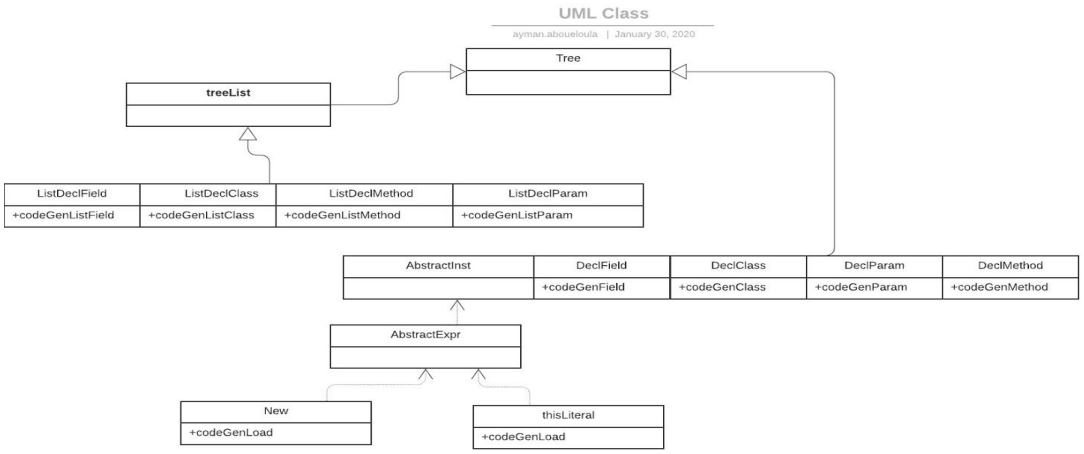
Le code qui génère cette fonctionnalité est la méthode codeGenLoad de la classe ThisLiteral, elle alloue un registre et fait le load de -2(Lb) dans ce registre puis elle le retourne.

New:

pour déclarer une nouvelle variable de l'objet et l'initialiser(A a = new A()), on génère le code de cette fonctionnalité à l'aide de la méthode codeGenLoad de la classe New.

Null:

pour initialiser les champs de classes non initialiser, on génère son code assembleur par la classe NullLiteral.



uml représentant quelques classes utilisés dans la partie objet: