Conception Compilateur

Projet Génie Logiciel

Fabien Galzi, Abdelmlek Mrad, Lisa Cornu,

Matéo Bedel, Eliot Clerc

Groupe GL 31

22/01/20025

Sommaire

Introduction

Ce document a pour objectif de présenter la conception du compilateur et de fournir une vue d'ensemble de son organisation et des choix techniques effectués. Il s'adresse aux développeurs qui devront maintenir ou améliorer ce projet, en leur permettant de comprendre l'implémentation et le fonctionnement du compilateur. Nous allons voir l'organisation du code, notamment comment les trois différentes parties fonctionnent. Nous expliquons aussi les choix qui ont été faits, que ce soit au niveau algorithmique, aussi bien qu'au niveau des structures de données. Enfin, ce document se veut être un guide pour les futurs développeurs, afin qu'ils puissent appréhender facilement les bases du projet et intervenir efficacement sur le code.

Architecture du projet

Vérification de la syntaxe contextuelle et décoration

Concernant cette partie, vous devez vous concentrer sur deux dossiers particulièrement, ils se trouvent tous deux dans le *src/main/java/fr/ensimag/deca/*. Les deux dossiers concernées sont celui du *context* et le *tree.*

- Dans le ***Context*** :

Dans cette partie, les types sont définis ainsi que leur définitions. Tout ce qui concerne la partie sans objet est relativement simple en ne donnant que la définition et des vérifications du type. Concernant la partie avec objet, c'est plus compliqué, les définitions sont plus complexes avec plus de paramètres concernant les classes, champs et méthodes. Les tests peuvent être un peu plus complexes par exemple pour vérifier si une classe est une sous classe. C'est une partie qui est très utile dans la suite pour les vérifications, il faut donc bien savoir ce qui est faisable dans cette partie pour bien faire fonctionner les vérifications et les décorations de la partie B.

Tout ce qui concerne les *environnements*, une partie un peu plus complexe, est défini dans le *context*. Tout d'abord, pour implémenter l'environnement des expressions, le choix s'est porté sur une HashMap car cette structure permet un accès facile et rapide aux informations via le nom de l'expression, en plus de garantir l'absence de doublons et l'ajout facile de nouveaux éléments. L'environnement des types fonctionne de la même manière, en utilisant cependant des types et non des définitions. Il faut donc bien faire attention à utiliser le bon environnement au vu de leurs similarités. Au départ, l'environnement du compilateur n'était pas modifiable car nous n'avions pas de classes à ajouter car elle n'était pas utilisées et nous avions donc toujours les mêmes types. Nous avons aussi choisi d'introduire le type *Object* dans l'environnement dès sa définition pour être certain que celui-ci s'y trouve toujours.

- Dans le ***Tree*** :

Il existe un certain nombre de classes abstraites dont le rôle est de permettre la compatibilité entre les expressions, ce qui entraîne un code succinct au niveau de la vérification et décoration de l'arbre ainsi qu'à la génération du code. En effet, simplement en vérifiant le type et les définitions on peut très rapidement relever les erreurs ou non.

Il existe principalement deux types de classes abstraites : celles permettant la déclaration de variables, de classes, etc… et celles concernant les instructions comme les opérations binaires, les instructions d'affichage et les expressions de tous types.

Dans chacun des fichiers se trouvent les vérifications et éventuellement les décorations de l'arbre se référant à la syntaxe contextuelle de Déca. Ces vérifications se font principalement sur le type des variables. Pour les terminaux en revanche, on ne vérifie rien dans la majeure partie des cas (déjà fait avant si on arrive là) mais on décore l'arbre.

Il y a quelques fichiers un peu plus particuliers, notamment le code de *Program* et de *Main* (l'initialisation est aussi un cas un peu différent des autres).

Au niveau de *Program* on va lancer la vérification de tous le programme, pour cela on commence avec les déclarations de classes : voir passe 1. Puis les déclarations de champs et de méthodes dans les classes : voir passe 2. Ensuite, on vérifie tout le corps des classes, de plus on va vérifier le Main : voir passe 3.

Dans le *Main* on va vérifier les listes de variables et d'instructions. Toutes les vérifications fonctionnent avec des listes sur lesquelles on vérifie chaque valeur de la liste, que ça soit pour les classes, les champs, les méthodes, les variables ou bien les instructions. Ça permet de rendre le code plus clair et lisible.

Dans ces fichiers, il existe des parties liées à l'affichage ou bien la décompilation du programme (prendre l'arbre et refaire le code deca).

Génération de code

Tout le code concernant la génération du code assembleur pour la machine IMA se trouve dans le répertoire *src/main/java/fr.ensimag/deca,* à sa racine ainsi que dans les sous-répertoires *codegen* et *tree*.

La structure du répertoire *tree* ayant déjà été détaillée dans la partie précédente, nous n'allons pas revenir dessus. Dans cette partie de génération de code, la séparation en de nombreuses classes abstraites et concrètes est très utile car elle permet de rendre le code de la génération d'assembleur très clair et succinct. En effet, chaque classe concrète s'occupe d'implémenter la petite chose qui la rend spécifique par rapport à ses classes "soeur", le reste étant implémenté dans la classe mère, et ainsi de suite.

Le répertoire *codegen* quant à lui contient les structures que nous avons ajoutées au code déjà fourni afin d'améliorer le code produit, ou d'implémenter des fonctionnalités manquantes. Son contenu est détaillé dans la partie structure de données ajoutées au projet.

Enfin, à la racine du répertoire *deca* se trouve les trois fichiers à la base du compilateur : les fichiers *CompilerOptions*, *DecacCompiler* et *DecacMain*. Le premier gère les options du compilateur et est utile à la création de ce dernier. *DecacCompiler* contient le code du compilateur en lui-même, qui appelle les fonctions implémentées dans *tree* au cours de son exécution. Enfin, le fichier *DecacMain* est le fichier Java a compiler pour générer l'éxecutable *decac*.

Changements apportés à la spécification originelle

La génération d'expression

Afin d'optimiser l'utilisation des registres, nous avons changé la spécification de la fonction *codeGen Expr* définie dans le fichier *AbstractExpr.java* et héritée par toutes les classes représentant des expressions (opérations, littéraux, variables, etc). A l'origine, cette fonction ne renvoyait rien. Nous l'avons modifié de sorte à ce qu'elle retourne une *DVal* représentant la valeur de l'expression si c'est un littéral, l'endroit où le résultat du calcul de l'expression est stocké sinon. Cela peut-être en registre, en pile (si la fonction retourne *null*), ou à une adresse spécifique dans la pile. Une DVal peut aussi être un label ou une opérande nulle, mais ces valeurs ne sont jamais retournées par *codeGenExpr* en pratique.

Ce changement dans le prototype de *codeGenExpr* optimise l'utilisation des registres car il est est utilisé en parallèle de la classe *RegisterHandler* à chaque interaction avec la mémoire (pile, registre ou adresse dans la pile). C'est cette classe qui permet l'optimisation à proprement parler, vous en trouverez les détails sous ce lien.

La génération d'instructions

Nous avons très légèrement modifié le prototype de la fonction *codeGen Inst* afin que celle-ci prenne également en paramètre le nom de la méthode à laquelle cette instruction appartient. Ce paramètre vaut *null* si l'instruction appartient au *main*. Cette modification permet de générer l'étiquette de saut à la fin d'une méthode en assembleur directement dans la fonction de génération de code de la classe *Return*. Comme la portion de code

assembleur de la génération d'erreur dans le cas où une méthode devant retourner quelque chose ne retourne rien est généré directement après le code de la méthode, cette modification (associée à une condition plus haut dans la hiérarchie pour le type void) permet de lever l'erreur proprement.

En effet, si une fonction dont le type de retour n'est pas void n'a pas d'instruction *return*, alors la fonction *codeGenExpr* de la classe *Return* ne sera jamais appelé pour cette méthode, et donc l'étiquette disant de sauter à la fin de la méthode (après la portion générant l'erreur) n'est jamais ajoutée au code. Cela entraîne qu'à la fin de la méthode, l'erreur du manque d'instruction *return* est générée.

Structures de données ajoutées

Deux nouvelles structures de données nous ont été nécessaires afin d'optimiser le code produit. Vous pourrez trouver leur code dans le répertoire *codegen*, comme mentionné ici.

- ***StackUsageWatcher :***

Cette classe assez simple garde le compte de l'usage de la pile au fil des instructions d'un bloc de programme. Elle compte le nombre de registres sauvegardés, le nombre de variables et le nombre de paramètres s'il y en a, car tous utilisent la pile. Si nous avions eu le temps de l'implémenter, nous aurions également mis en place un algorithme permettant de suivre le maximum de registre temporaires utilisés à un même moment, car cela influe également sur la taille de la pile.

Cette structure de données nous a paru nécessaire à la génération de l'instruction *TSTO* en début de bloc. Cette instruction indique l'utilisation prévue de la pile au cours de la méthode, et déclenche un overflow si cette prévision est dépassée. Nous lançons alors l'erreur au niveau de l'étiquette pile_pleine. Malheureusement, par manque de temps nous avons abandonné l'idée de calculer précisément la valeur à donner à l'instruction TSTO, et nous avons à la place donné un nombre grand a priori suffisant pour l'exécution de la majorité des programmes. Nous sommes bien sûr conscients que ce n'est pas vraiment optimal, mais via la classe *StackUsageWatcher*, vous avez un aperçu de la méthode que nous aurions utilisé pour l'implémentation si nous avions eu plus de temps.

- ***RegisterHandler :***

Le rôle de la classe *RegisterHandler* est de centraliser toutes les opérations impliquant des interactions entre les registres et la mémoire. Elle est principalement utilisée pour traiter la DVal retournée par les fonctions de génération de code des expressions.

Par exemple, pour stocker le résultat d'une expression en mémoire, l'instruction assembleur à utiliser n'est pas la même en fonction du type de la DVal : registre, haut de pile, adresse en pile. Au lieu de tester le type de la DVal à chaque retour de *codeGenExpr*, ce qui aurait entraîné énormément de code dupliqué étant donné la hiérarchie de classe, chaque appel à *codeGenExpr* est en général suivi d'un appel à une des fonctions statiques de la classe RegisterHandler, permettant de manipuler ce résultat proprement.

Les méthodes appelées du RegisterHandler sont :

```
DVal popIntoDVal(DVal addr, GPRegister tempRegister)
```

- Cette méthode vérifie que addr (renvoyé par un codegen) n'est pas dans la pile sinon elle pop le résultat dans un registre temporaire et renvoie l'adresse/le registre concerné.

```
GPRegister popIntoRegister(DVal addr, GPRegister tempRegister)
```

- Cette méthode vérifie que **`addr`** n'est pas dans la pile et le place dans un registre si nécessaire, sinon elle pop le résultat dans un registre temporaire et renvoie le registre concerné.

```
GPRegister pushFromDVal(DVal addr, GPRegister tempRegister)
```

- Cette méthode cherche un registre pour stocker **`addr`** si aucun registre n'est disponible, elle stock **`addr`** dans **`tempRegister`**, et push le résultat dans la pile. La méthode renvoie le registre concerné (ou null si le résultat est push dans la pile).

```
GPRegister pushFromRegister(GPRegister reg)
```

- Cette méthode vérifie si **`reg`** n'est pas temporaire, sinon elle le place dans un registre libre (non-temporaire) ou le push dans la pile. Elle Renvoi le registre concerné (ou null si le résultat est push dans la pile)