Compilateur Decac
Document de conception

TEIMUR ABU ZAKI, ADRIEN BOUCHET, TROY FAU, PAUL MARTHELOT, HUGO ROBERT

Table des matières

Introduction

A partir de l'architecture de base du projet, ce document détaille les modifications et rajouts que nous avons effectué lors du développement du compilateur Decac. Par conséquent, hormis ce qui est mentionné dans ce document, l'ensemble de la description de l'architecture disponible dans le polycopié reste valide.

1 Classes Modifiées

1.1 Étape C

La génération de code se fait en 3 passes visibles dans `Program.java`. La première concerne la génération de la table des méthodes, la seconde la génération du code du main, et enfin la 3ème passe pour la génération de code des méthodes. Dans un souci de réduire le nombre d'instructions coûteuses en PUSH et POP, nous avons modifié les classes `Register` et `GPRegister`. Dans cette dernière classe nous avons ajouté des attributs permettant de savoir si les registres sont utilisés ou non pour la sauvegarde d'un résultat, mais également de savoir le nombre de PUSH réalisé (en attente d'un POP futur). D'autre part, les méthodes `getRegister()` et `freeRegister(Register)` permettent de récupérer et de libérer les registres sans avoir à le faire manuellement.

Actuellement, les registres sont alloués de façon à répartir au mieux les PUSH sur les différents registres : on cherche le registre le moins occupé (en termes de nombres de variables en attente de POP). De plus, d'autres méthodes telles que le remplissage de tous les registres peuvent être utiles pour calculer au mieux les PUSH et les POP nécessaires à la sauvegarde des registres avant l'appel d'une fonction Deca.

Toujours dans la partie génération de code, l'ensemble des méthodes dédiés à la génération de code dans les fichiers .ass sont décrits dans des méthodes CodeGen avec le suffixe Inst lorsqu'un élément est appelé en vue d'un calcul, ou Print pour un appel en vue d'un affichage.

Concernant la gestion des registres, nous avons pris la décision d'utiliser le Registre R0 comme Registre contenant la valeur de retour lors d'un appel à un élément. Par exemple la fonction d'addition appellera par un codeGenInst l'élément de gauche, récupérera le résultat depuis R0, le stockera dans un registre entre R2 et R15, appellera par un codeGenInst l'élément de droite puis réalise l'addition entre R0 et le Registre contenant l'élément de gauche. Ce choix permet de faciliter la gestion des registres en sachant à chaque fois ou le résultat sera stocké. Néanmoins ce choix pourra être remis en question lors d'une possible future optimisation de la génération de code car il nous pousse à devoir toujours déplacer les résultats obtenus dans R0.

1.2 Options du compilateur

En ce qui concerne les options du compilateur, nous avons modifié la fonction d'analyse de la ligne de commande (`CompilerOptions.parseArgs`) de telle manière à pouvoir récupérer uniquement les options et les fichiers qui nous intéressent, de façon linéaire. Il serait facile d'ajouter des options en rajoutant des champs dans `CompilerOptions` et en activant ces champs lors du traitement des arguments. Cette technique a d'ailleurs été réalisée pour pouvoir ajouter l'option `-a` pour la compilation ARM.

2 Nouvelles classes de l'arbre abstrait

Comme expliqué à la section 6 du chapitre [ConventionsCodage] du polycopié, l'arbre abstrait du programme est représenté selon le patron de conception « interprète ». Le squelette de code qui nous été fourni implémentait donc l'arbre pour la partie « Sans Objet » du projet. De façon systématique, nous avons complété cette hiérarchie de classe en suivant scrupuleusement le même patron « interprète » pour l'implémentation de la partie « Objet ». Ainsi toutes les classes abstraites et concrètes reflétant la syntaxe abstraite du langage Deca, telle qu'elle est définie au chapitre [SyntaxeAbstraite] du polycopié, sont implémentées dans le répertoire `src/main/java/fr/ensimag/deca/tree`.

3 Nouvelles classes pour l'analyse contextuelle

Les nouvelles classes de l'étape B se trouvent toutes dans le répertoire `src/main/java/fr/ensimag/deca/context`.

3.1 EnvironmentType

La classe `EnvironmentType` permet de stocker les *TypeDefinition* définis à la section 2.1 du chapitre [SyntaxeContextuelle] du polycopié, de la même manière que la classe `EnvironmentExp` stocke les *ExpDefinition*. Elle représente donc l'ensemble des types disponibles dans le programme.

Dans l'instance `EnvironmentType` on initialise les types prédéfinis du langage (int, float, boolean et void) dès le lancement du compilateur, puis on y rajoute, en tant que type, chaque classe défini dans le programme Deca en cours de compilation.

Alors que plusieurs `EnvironmentExp` sont instanciées lors de la compilation, ce qui permet d'établir la hiérarchie des environnements de classe, il n'y a besoin que d'une instance `EnvironmentType`. Pour garantir cela, la classe a été conçue selon le patron de conception « Singleton [1] ». Son constructeur est privé, et elle possède un champ statique qui est une instance d'elle-même. Cette instance est donc unique,

1. Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*, pages 127-134. Addison-Wesley 1995.

et elle le reste puisqu'on ne peut appeler le constructeur de la classe à l'extérieur de celle-ci.

Une fois stockés les différents types existants dans le programme, l'instance est alors appelée pour valider les identificateurs de type présents dans les déclarations de variables, de champs ou les signatures de méthodes.

3.2 ContextTools

Pour vérifier que les instructions d'un programme Deca se conforment bien aux spécifications de la section 2.2 du chapitre [SyntaxeContextuelle] du polycopié, nous avons créé la classe `ContextTools`, dont les méthodes sont toutes statiques. Un groupe de méthodes vérifient la compatibilité d'opérandes pour l'affectation ou le transtypage ; un autre ensemble de méthodes vérifient le bon typage des opérandes pour les opérations arithmétiques, de comparaison, booléennes, etc.

Les méthodes de cette classe sont appelées lors du parcours de l'arbre abstrait effectué pendant la vérification contextuelle : les appels se font donc dans les méthodes `verify`, tel qu'on peut le voir à la figure 1.

Classe	**Méthode appelante**	**Méthode ContextTools**
`ContextTools`	`assignCompatible`	`subtype`
`ContextTools`	`castCompatible`	`assignCompatible`
`DeclMethod`	`verifyMethod`	`subtype`
`AbstractExpr`	`verifyRValue`	`assignCompatible`
`Cast`	`verifyExpr`	`castCompatible`
`AbstractOpArith`	`verifyExpr`	`typeArithOp`
`Modulo`	`verifyExpr`	`typeArithModulo`
`AbstractOpBool`	`verifyExpr`	`typeBoolOp`
`AbstractOpCmp`	`verifyEpr`	`typeCmpOp`
`UnaryMinus`	`verifyExpr`	`typeUnaryMinus`
`Not`	`verifyExpr`	`typeUnaryNot`
`InstanceOf`	`verifyExpr`	`typeInstanceOf`

FIGURE 1 – Les appels aux méthodes statiques de la classe `ContextTools`.

3.3 Warning

Nous avons également défini une classe `Warning` qui gère le formatage des messages d'avertissement (option `-w` du compilateur) et leur affichage. La classe est instanciée à la détection d'une anomalie que nous avons associée à un avertissement (cf. manuel utilisateur), avec comme arguments la chaîne de caractère du message et la position du problème dans le fichier.

Une extension de Decac qui voudrait différentes catégories d'avertissement avec différents formatages des messages pourrait étendre cette classe et surcharger sa mé-

thode `emit`.

Les avertissements concernent des problèmes pouvant déclencher des erreurs à l'exécution mais qui ne doivent pas arrêter la compilation, d'après la spécification du langage Deca. Un problème qui n'est pas couvert par nos avertissements et l'accès à des variables non initialisées.

Émettre un avertissement pertinent pour ce cas de figure impliquerait de maintenir une hiérarchie d'`EnvironmentExp` parallèle qui ne stockerait que les variables ayant été initialisées, et déclencherait un avertissement dès qu'une variable déclarée mais non initialisée serait détectée en *rvalue*, dans une condition ou dans un `print`. Nous avons décidé qu'il s'agissait d'un système assez lourd qui impliquerait pas mal de tests pour valider sa pertinence, et ne l'avons pas implémenté, mais c'est un avertissement qu'une extension pourrait tout à fait rajouter.

4 Nouvelles classes pour la génération de code

Afin de faciliter l'implémentation du code de l'étape C de la compilation, nous avons décidé d'ajouter des classes gérant des fonctionnalités nécessaires à l'étape C, qui sont toutes situés dans le répertoire `src/main/java/fr/ensimag/deca/tools`.

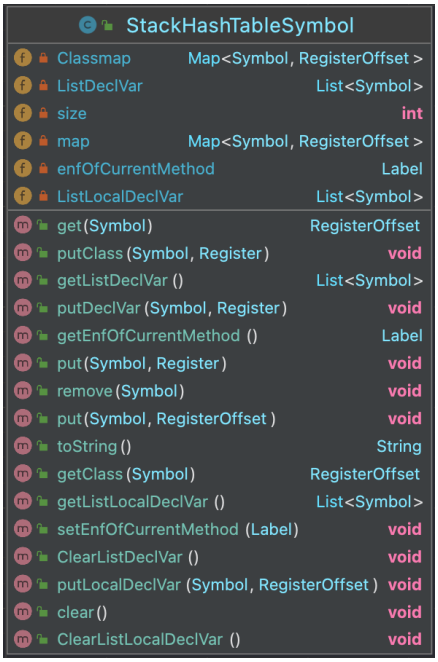
4.1 StackHashTable

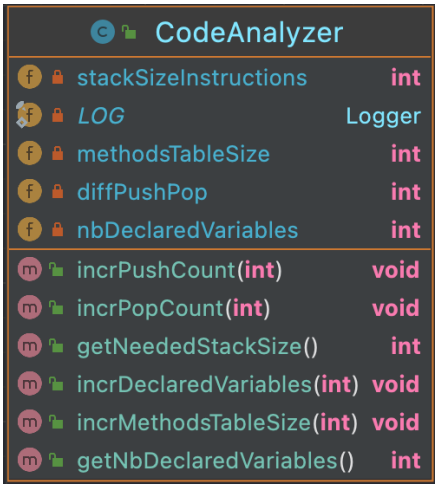
`StackHashTable` permet, à la manière de `set/getOperand` d'associer à un symbole son registre *offset* dans la pile. La structure utilisée est une *HashMap*, on peut donc récupérer facilement une adresse à partir du symbole et l'ajout et la suppression se font rapidement. On utilise également une autre *HashMap* dédié à associer le symbole d'une classer à son registre *offset* dans la pile.

Nous avons également ajouté 2 listes qui permettent de garder en mémoire quels sont les symboles qui correspondent à des variables déclarées dans le main ou dans une méthode de classe. Ces listes permettent notamment de ne garder dans la *HashMap* que les variables qui sont déclarées dans la partie dont on génère le code. Enfin la structure possède un Label `endCurrentMethod` qui est mis a jour a chaque fois qu'on génère le code d'une nouvelle méthode et qui est utilisé par le return pour savoir vers quel label renvoyer.

4.2 CodeAnalyzer

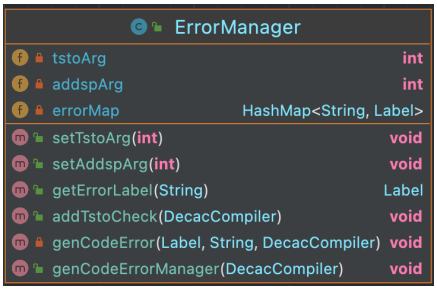
`CodeAnalyzer` fournit des compteurs permettant de compter le nombre de variables déclarées, la place nécessaire dans la pile pour un programme en particulier. Elle est très utile pour l'analyse du code généré dans un bloc d'instructions et permet de compter les arguments que prendront `TSTO` et `ADDSP` en début de chaque bloc. Cependant, l'appel de leur différents compteurs est manuel et il faut réinitialiser la classe à chaque nouveau bloc et récupérer les compteurs en fin de génération du code.





4.3 ErrorManager

`ErrorManager` fournit une *HashMap* permettant d'associer à un string faisant référence à un type d'erreur (par exemple "stack overflow, a real one") à un label qui sera récupéré par chacune des opérations donnant lieu à une telle erreur. En rajoutant simplement l'erreur dans cette *HashMap*, son label et le code débouchant sur l'instruction `ERROR` seront automatiquement ajoutés à la fin du programme.



5 Utilisation des nouvelles classes

La classe `DecacCompiler` orchestre les différentes étapes de la compilation : récupération des options de compilation, initialisation des types de base, lancement de l'analyse lexicale, syntaxique, contextuelle, puis génération du code assembleur pour la machine virtuelle IMA. Cela à été un choix logique d'utiliser les nouvelles classes (hormis `ContextTools`, qui n'est jamais instanciée) comme champs supplémentaires dans `DecacCompiler`, comme l'illustre la figure 2. Lorsque l'instance de `DecacCompiler` est passée en argument au *parser*, aux méthodes de vérification contextuelles ou aux méthodes de génération du code, on peut alors accéder à ces champs.

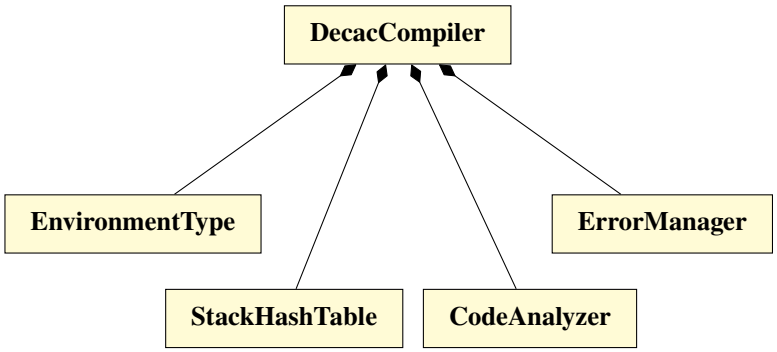


FIGURE 2 – Diagramme UML simplifié des nouvelles classes en relation avec la classe `DecacCompiler`. Traduit en Java, on a une instance de chaque classe comme champ de `DecacCompiler`.

6 Gestion des erreurs numériques

La gestion des erreurs concernant les nombres entiers et flottants a été faite au niveau du *parser* dans le cas de nombres écrits directement dans le code. Les nombres trop grands (entiers ne pouvant être stocké sur 32 bits, ou flottants trop grands pour être représentable selon la norme IEEE 754 32 bits) déclenchent une

exception Java de type `NumberFormatException`. Toute extension qui rajouterait un type numérique qui existe en Java et qui produit une exception similaire en cas d'anomalie peut gérer les erreurs de la même façon triviale.

Les cas d'erreur qui n'existent pas en Java doivent faire l'objet d'un traitement spécial. C'est le cas du soupassement en virgule flottante en Deca, qui d'après la spécification doit déclencher une erreur à la compilation, contrairement au Java.

Dans le *parser*, pour gérer le soupassement, la chaîne de caractères représentant un nombre flottant est analysée pour détecter tout caractère numérique différent de zéro avant l'exposant (si le nombre est en notation scientifique). Cette chaîne est ensuite converti en nombre flottant par une fonction Java, qui ne détecte donc pas le soupassement. Si ce nombre est égal à zéro mais qu'on a trouvé un caractère non nul dans la chaîne alors il y a soupassement, et on lève une exception.

Pour le cas de données saisies par l'utilisateur (fonctions `readInt` et `readFloat`), les erreurs sont déclenchées à l'exécution, avec affichage d'un message d'erreur généré à la compilation (cf. le manuel utilisateur).

7 Conception de l'extension ARM

Au-delà des classes précédemment décrites, l'implémentation de la génération de code pour une machine ARM demande de créer de créer une architecture assez similaires à celle des paquetages `fr.ensimag.ima.pseudocode` et `fr.ensimag.ima.pseudocode.instructions` utilisés pour générer le code exécuté par la machine virtuelle IMA.

Le code génère des instructions exécutables sur ARM-v6-fp. Les instructions manipulent des registres de 32 bits uniquement sauf lors des appels à printf qui nécessitent de placer les données dans des registres de 64 bits.

Par exemple, le nombre de registres à usage général pour ARM reste fixé à 12 mais il faut également ajouter quelques registres "s" et "d" en modifiant la classe Registre. De même, il fallait recréer un nouvel ensemble d'instructions et notamment gérer de nouveaux types d'instructions n'existant pas sur ARM : les instructions ternaires (à trois opérandes).

Si un développeur voulait continuer l'extension pour l'étendre sur le langage Deca complet, il pourrait rajouter des classes correspondant aux sections du code (.text ou .data par exemple). Actuellement, le code généré déclare toutes les variables en début de fichiers dans une section .data. Cependant, si l'utilisateur souhaite afficher une chaîne de caractères, une nouvelle section .data sera créée dans le code principal, avant de revenir à une section de code .text.

Certaines fonctions ne sont pas disponibles directement, notamment la division entre entiers. C'est pourquoi nous avons créé une classe `ARMFunctionManager` qui permet d'ajouter ces fonctions en fin de fichier. Cette classe permet aussi l'initialisation de données nécessaires au bon fonctionnement de certaines fonctions comme

les chaînes de caractères pour afficher les nombres.