

Documentation de Conception

Valentin Laclautre, Anthony Dard, Damien Trouche, Martin Gangand,
Basel Darwish Jzaerly

Table des matières

1	Architecture logicielle	2
1.1	Conception Architectural Etape C	2
1.1.1	Politique de gestion de pile et de registre	2
1.1.2	RegisterManager	2
1.1.3	Stack	2
1.1.4	LabelManager	2
1.1.5	Utils	3
1.1.6	Propagation du code - Les fonctions codeGen	3
1.1.7	codeGenPrint	3
2	Chemin de l'exécution du compilateur	3
2.1	Etape A	3
2.1.1	DecacMain	3
2.1.2	DecacCompiler	3
2.1.3	DecaLexer	4
2.1.4	DecaParser	4
2.1.5	AbstractProgram	4
2.1.6	Tree	4
2.2	Etape B	4
2.3	Etape C	5
2.3.1	JavaCompiler	5
3	La bibliothèque ASM	5

1 Architecture logicielle

1.1 Conception Architectural Etape C

Tout le code spécifique se situe dans le package `codegen`. Il est constitué de plusieurs fichiers permettant de gérer la génération de code. Toutes les classes nécessaires sont instanciées dans `DecacCompiler` pour pouvoir être appelées lors de l'exécution dans l'arbre.

1.1.1 Politique de gestion de pile et de registre

A chaque variable créée, on la place dans la pile et dans sa définition, on lui donne son adresse dans la pile. Lors de l'initialisation ou de l'affectation d'une variable, ou de n'importe quelle instruction nécessitant le calcul d'une expression, celui-ci est enregistré sur le registre `R1`. Bien sûr lors du calcul, si c'est nécessaire, d'autres registres sont utilisés mais le résultat final est sur `R1`.

1.1.2 RegisterManager

Cette classe permet de gérer les registres. Elle prends en attributs le nombre de registres utilisés (ceux données en paramètres par la commande `-r` ou 16 sinon). Elle possède aussi un tableau de boolean en attributs. Chaque indice de ce tableau correspond à la valeur d'un registre. La valeur du tableau à cette indice est à `vrai` si le registre est utilisé et `faux` sinon. Cette classe possède aussi des méthodes permettant de renvoyer un registre inutilisé ou d'en libérer un.

1.1.3 Stack

Cette classe possède un attributs donnant la hauteur de la pile (par rapport à `GB`). Elle possède aussi de nombreuses méthodes permettant de mettre la valeur d'un registre au sommet de la pile, ou à un endroit précis de la pile. Elle possède aussi d'autres méthodes permettant de récupérer une variable se situant à une adresse précise dans la pile.

1.1.4 LabelManager

Cette classe permet de créer et de renvoyer des labels uniques à partir d'un nom. Elle utilise pour cela un `HashMap` qui a un nom de label associe un compteur correspondant au nombre de fois que ce nom de label est utilisé. Cela permet de s'assurer que tous les labels sont uniques

1.1.5 Utils

Cette classe regroupe des méthodes statiques utilisées à de nombreux endroits permettant la génération de code. Elle permet entre autre de renvoyer un `Immediat` d'après son type. Elle permet aussi de renvoyer tout le code correspondant à la gestion d'erreur qui appelé à la fin du `codeGen` du programme.

1.1.6 Propagation du code - Les fonctions `codeGen`

A chaque action devant être réalisé, il existe une fonction `codeGen` spécifique appelant récursivement dans l'arbre d'autres fonction `codeGen`. La première fonction appelé est `codeGenProgram` qui se propage à la génération de classe et du programme principale. Il est intéressant de revenir sur quelque fonction `codeGen` importantes qui sont réutilisés de nombreuses fois.

1.1.7 `codeGenPrint`

2 Chemin de l'exécution du compilateur

2.1 Etape A

2.1.1 `DecacMain`

Le point d'entrée du compilateur est la méthode `main` de la classe **`DecacMain`**. Le `main` commence par récupérer les options de la commande `decac`. La classe responsable de ce traitement est **`CompilerOptions`**. Les arguments de la commande `decac` sont parsés et les variables booléennes correspondantes aux options seront à mises à `true`. Nous pouvons avoir les états de ces Booléens en appelant la méthode `getX` de la classe `CompilerOptions` avec `X` est le nom du booléen.

Si l'option de compilation **en parallèle est activée**, plusieurs instances de **`DecacCompiler`** seront initialisées en parallèle et chaque instance traite un **fichier deca source**. Sinon, une instance du compilateur **`DecacCompiler`** sera initialisée de façon séquentielle pour chaque fichier deca avec en paramètres les options et un fichier source.

2.1.2 `DecacCompiler`

C'est la classe coeur du compilateur. Elle contient la méthode **`doCompile`** qui réalise la compilation. Dans cette méthode, la racine de l'arbre abstrait du programme **`AbstractProgram`** est initialisée. L'initialisation est

fait grâce à la méthode **doLexingAndParsing** qui retourne un **AbstractProgram**. Cette méthode utilise une instance de la classe **DecaLexer** qui est présentée par la suite et qui a pour rôle de faire l'analyseur lexicale qui consiste à reconnaître les mots du langage (dits aussi Jetons ou Tokens) . Cette méthode utilise aussi une instance de **CommonTokenStream** pour stocker les jetons. Cette classe appartient au framework **ANTLR**. Ensuite une instance de la classe **DecaParser** est déclarée pour faire l'analyse syntaxique. A la fin de cette méthode, l'**Etap A** est terminé et l'arbre abstrait (une instance de **AbstractProgram**) est obtenue.

2.1.3 DecaLexer

Cette classe est générée automatiquement par le framework **ANTLR**. Elle a pour role de transformer le programme deca en mots (jetons). Pour configurer **ANTLER** à accepter le code lexicalement valid, on spécifie les règles lexicales dans `./src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`.

2.1.4 DecaParser

Cette classe est générée automatiquement par le framework **ANTLR**. Elle a pour role de déterminer si une suite de mots est une phrase du langage, c'est à dire correspondant à un programme deca syntaxiquement correct. Pour **ANTLER** il faut définir les grammaires du langage dans `./src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`.

2.1.5 AbstractProgram

Cette classe est dérivée de la classe **Tree**.

2.1.6 Tree

represente le point d'entrée pour la vérification contextuelle et la génération du code.

2.2 Etape B

Le but de l'étape B est de réaliser des vérifications contextuelles et modifier et décorer l'arbre abstrait du programme pour préparer l'étape C. La vérification déclenchée par la méthode `verifyProgram` se propage dans tout l'arbre en passant par la classe **Main**. Dans cette classe les vérifications se passent pour une instance de la classe **EnvironmentExp** et les listes de

ListDeclVar (les déclarations de variables) et **ListInst** (les déclarations des instructions)

2.3 Etape C

Cette étape consiste à générer le code exécutable et le stocker dans un fichier de sortie. Selon les options de la compilation, le code généré est .ass pour la machine virtuelle IMA ou bien .class pour la machine virtuelle Java JVM.

Nous avons utilisé une instance de la classe **FileOutputStream** pour stocker le resultat de la compilation dans un fichier de sortie.

Dans le cas où l'option **JavaCompilation** est activée. Nous utilisons la classe **JavaCompiler** pour préparer le fichier .class en sortie.

2.3.1 JavaCompiler

Cette une classe qui a pour rôle de gérer la structure du fichier .class généré. Elle encapsule des attributs de l'API **ASM** que l'on utilise pour manipuler les fichier .class de façon haut niveau.

3 La bibliothèque ASM