Documentation de Conception

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

Table des matières

1 Architecture logicielle

1.1 Conception Architectural Etape A

Tout le code specifique se situe dans le package deca. Il est constitué de plusieurs fichiers permettant de gérer l'analyse lexicale, l'analyse syntaxique et la construction de l'arbre abstrait. Toutes les classes nécessaires sont instanciées dans DecacMain pour pouvoir être appelées lors de l'execution dans du programme avec ses options.

1.1.1 DecacCompiler

On instancie un DecacCompiler pour chaque fichier .deca. La classe contient un attribut **SymbolTable**, **EnvironmentType** et **CompilerOptions**. On spécifie dans cette classe les types primitifs autorisés de façon statique et les nouveaux types seront ajoutés dynamiquement pendant l'exécution.

1.1.2 SymbolTable

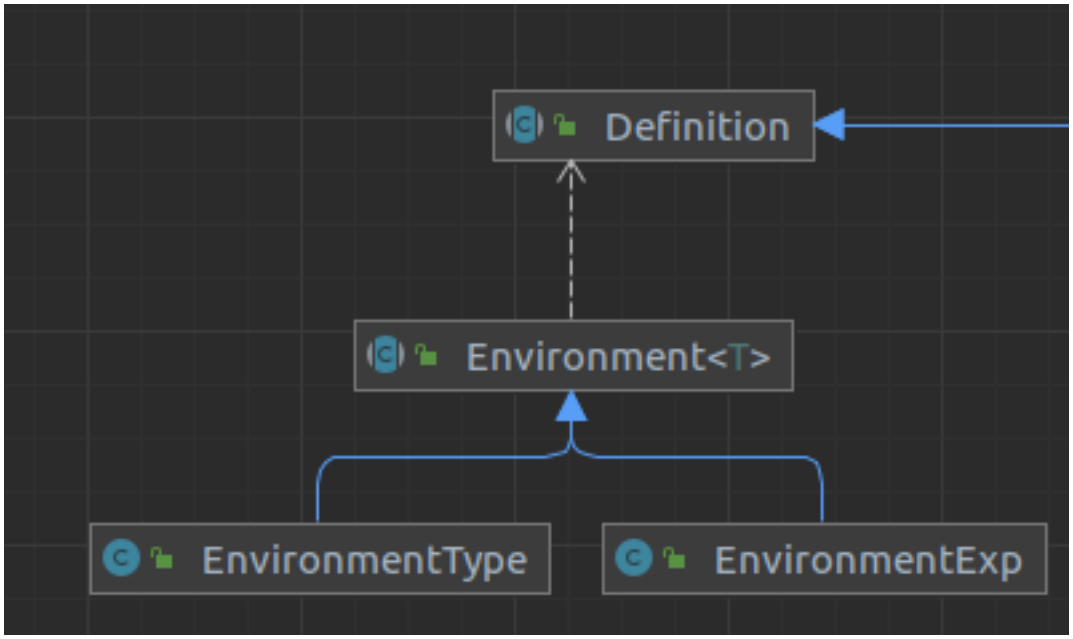
Permet d'assurer l'unicité d'un symbol dans son environnement. Elle est héritée de la phase de lexing/parsing.

1.2 Conception Architectural Étape B

1.2.1 Environment

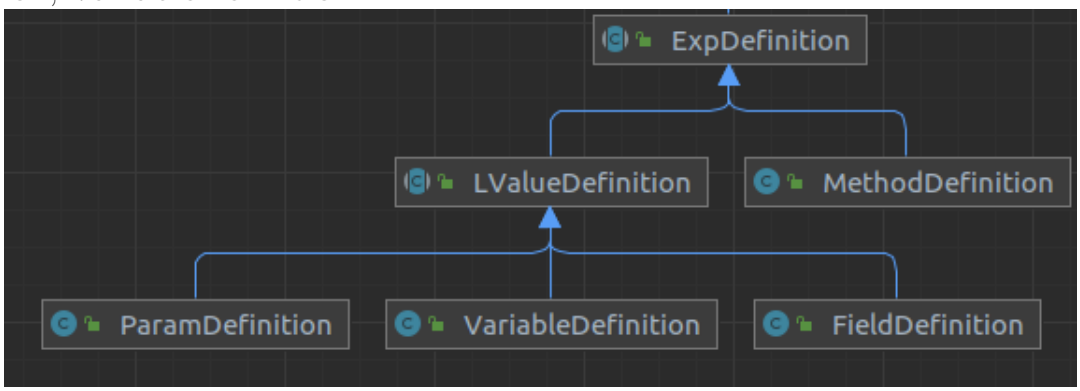
Il s'agit d'un dictionnaire associatif permettant de stocker les **Definition** dont la clé est le symbole qui lui correspond dans la**SymbolTable**. Elle implémente les opérations **stack** et **declare** correspondant respectivement aux opérations d'empilement et d'union disjointe (au sein de l'environnement local).

Cette classe est abstraite, il y a deux classes qui en dérivent : **EnvironmentExp** et **EnvironmentType**



EnvironmentExp

Cette classe n'autorise que les sous-type de **ExpDefinition**, elle sert à déclarer les **MethodDefinition**, **LValueDefinition** (classe abstraite dont l'objectif est de permettre de vérifier si des identifiant sont des constantes pour de futur optimisations) dont hérite les **FieldDefinition**, **ParamDefinition**, **VariableDefinition**.



Ainsi, cette classe est utilisé afin de représenter l'environnement des identifiant qui est construit lors de l'étape B.

EnvironementType

De manière analogue, cette classe change uniquement dans le fait qu'elle stocke des **TypeDefinition** dont la classe concrète est **ClassDefinition** qui en particulier stocke les membres de la classes (champs et méthodes) dans un **EnvironmentExp**.

1.2.2 Vérification contextuelles

L'étape de vérification contextuelle s'effectue en 3 passes. Chacune des passes fait appelle aux méthodes du type **verify*** qui renvoient une erreur en cas de problèmes contextuels, déclarent/complètent les définitions associées aux noeud de l'arbre issue de la phase de lexing/parsing.

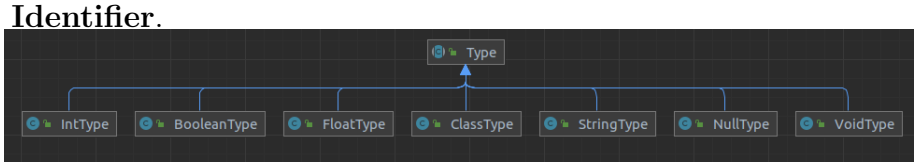
Passe 1 La passe 1 s'assure que les classes sont correctement déclarées et les ajoute dans l'envType présent dans la classe **DecacCompiler** afin de pouvoir les utiliser comme type par la suite.

La méthode principale réalisant cette passe est donc **verifyClass**

Passe 2 La passe 2 est également réservé à la partie orienté objet, en faisant les vérifications contextuelles des corps des classes.

On se charge alors de déclarer les champs, les méthodes dans **ClassDefinition** dans l'environnement **members** via les méthodes **verifyClassMembers**, **verifyField** et **verifyMethod** (c'est en particulier dans cette méthode que nous gérons les redéfinitions de méthodes et de champs)

Passe 3 Enfin, la passe 3 s'occupe des dernier traitement généraux sur l'ensemble de l'arbre comme l'initialisation des champs, les vérifications des corps des méthodes mais également des complétions des **Definition** et **Type** des **Identifier**.



Les méthodes misent en jeu lors de cette étape sont nombreuses, nous invitons donc le lecteur à regarder les méthodes du type **verify¡nom de règle de la grammaire¿**

A la fin de cette étape, l'ensemble des vérifications contextuelles sont réalisées ce qui signifie que le programme est évalué correct par le compilateur et peut donc être envoyé à la phase de génération de code.

1.3 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.

1.3.1 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 sur lors du calcul, si c'est nécessaire, d'autres registres sont utilisé mais le résultat final est sur R1.

1.3.2 RegisterManager

Cette classe permet de gérer les registres. Elle prends en attributs le nombres 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.

1.3.3 RegisterAllocator

Cette classe permet d'allouer des registres créés de manière virtuelle. Cette classe est utilisé lors des opérations arithmétiques et peut être très utiles et améliorables pour les développeurs souhaitant travailler sur ce projet. En effet dans le programme, lorsque l'on veut utiliser des registres, il suffira de créer un registre virtuel avec la classe **VirtualRegister**, puis de l'ajouter dans le Set de registres virtuelles de **RegisterAllocator**. Ensuite à l'aide d'une méthode de cette classe, tout les registres virtuelles créés vont être alloué dans de vrai registres. S'il n'y a plus de place, les registres déjà utilisés sont sauvegardés dans la pile. Lors de la désallocation, les valeurs étant éventuellement stockées dans la pile suite à l'allocation sont remis à leurs registres d'origine.

1.3.4 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 à partir de son **Identifier**. Elle possède aussi d'autres méthodes permettant de récupérer une variable se situant à une adresse précise dans la pile. La classe **Stack** est aussi utilisée pour pour initialiser les champs des classes deca dans le tas. Elle permet de plus de mettre à jour des champs lorsque l'on se situe dans une méthode, en vérifiant dans sa définition que c'est un champ et en utilisant son index afin de savoir où mettre à jour le champ à partir de l'adresse dans le tas.

1.3.5 LabelManager

Cette classe permet de créer et de renvoyer des label 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 tout les labels sont uniques. Elle possède également des méthodes permettant de donner le nom d'étiquette pour pour la création de méthodes.

1.3.6 VTable

Cette classe est la classe centrale de deca avec objet. C'est elle qui s'occupe de créer la table des méthodes au début du fichier assembleur, et qui permet de créer le constructeur (étiquette init qui permet d'initialiser les champs de la classe) ainsi que toutes les méthodes à la fin du fichier assembleur. Lors de la déclaration de variables, les premières méthodes de **VTable** appelées dans **ListDeclClass** correspondent à la création de la table des méthode pour la classe Object et à la création de la méthode equals. Ensuite on crée la table des méthodes, les constructeurs et les méthode de manière générales pour chaque classes. A la création de chaque nouvelle classe, on ajoute dans un HashMap la table des méthodes. Ce HashMap associe à un nom de classe une liste de **Label** correspondant à toutes les méthodes de cette classe. Cela permet lors de la création d'une nouvelle classe dans la table des méthodes de pouvoir récupérer et éventuellement modifier s'il y a redéfinition toutes les méthodes de la superclass. Les méthodes de création des méthodes et de constructeur font simplement appels récursivement aux classes dans l'arbre. Les constructeurs et méthodes étant définis à la fin du programme, un nouveau **IMAProgramm** a été ajouté dans **DecacCompiler** afin d'ajouter les intructions concernant cette partie. Ce **IMAProgram** est par la suite ajoutée au **IMAProgram** courant à la fin des instructions du programme principale.

1.3.7 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.

1.3.8 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 interressant de revenir sur quelque fonction codeGen importantes qui sont réutilisés de nombreuses fois.

codeGenExrOnRegister :

Cette méthode est la plus importante de l'étape C. Elle est redéfinie dans toutes les classes correspondant à une expression. Elle permet de charger la valeur de l'expression qui l'appel dans le registre donné en paramètre. Elle est définie simplement dans les littéral ou les identifiant par exemple, puisque l'on peut charger directement la valeur dans le registre. Elle simplifie grandement le calcul d'expressions plus complexes comme les opérations arithmétiques ou les calculs booléens par exemple. Le fonctionnement des calculs de ces deux opérations seront décrites plus loin. La méthode codeGenExprOnR1 est aussi définie. Elle fait directement appel à la méthode codeGenExrOnRegister, mais avec en paramètre le registre R1. Cette méthode est définie car il est régulièrement nécessaires de charger une expression sur le registre R1.

codeGenPrint :

Cette méthode est permet l'affichage d'une expression. Elle est par exemple utilisée dans la classe **StringLitteral** afin d'appeler l'instruction d'affichage IMA pour une chaine de caractères. Elle appelé à chaque intruction de demandant l'affichage sur la sortie standart. Elle est généralisé dans **AbstractExpr** et est redéfnie dans les cas particuliers.Elle est généralisée de la manière suivante : on charge l' expression à afficher dans le registre R1 avec codeGenExprOnR1, puis on vérifie son type (int ou float) et on appel l'instruction IMA correspondante.

codeGenInst :

Cette méthode est appelé pour chaque instruction. Elle est généralisé dans **AbstractExpr** de la manière suivante : elle fait simplement appel à la méthode codeGenExprOnR1, qui est redéfinie spécifiquement pour chaque tokens, comme cela a été expliqué précédemment. Pour les cas spécifiques, elle est redéfinie.

CodeGenBool :

Cette fonction est utilisé pour effectuer des calculs booléens, dans les boucles while, et dans les conditions if. Elle permet de faire des branchements en fonctions de l'expression booléennes à calculer. En effet, l'argument **negation** précise si on fait le branchement sur le **label** lorsque l'expression est évalué à **negation** (true ou false).

1.3.9 Opération arithmétique

Les opérations artihmétiques sont effectués en calculant récursivement l'expression de l'opérande de gauche. Si il y a des registres libres (connu grâce à **RegisterManager**), on calcule de même sur le registre suivant l'opérande de droite. Sinon, on push l'opérande de gauche dans la pile, on calcul l'opérande de droite sur le registre actuel. On charge ensuite le résultat dans le registre R0, puis on on pop de la pile le résultat de l'opérande de gauche.

Des optimisations ont été implémentée afin d'évaluer les expressions constantes à la compilation via les méthodes **getDirectInt** et **getDirectFloat** renvoyant la valeur si l'expression est une constante ou null sinon. De plus, les multiplications/divisions par des puissances de 2 sont remplacées par des décalages à gauche/droite jusqu'à 2^{20}, après cette valeur, l'optimisation n'est plus efficace. Des pistes d'amélioration ont été laissé, l'utilisation de l'instruction **FMA** peut être utilisée afin d'optimiser le code. Cependant il est nécessaire de revoir les convention d'évaluation des expressions qui utilise le registre **R1** nécessaire à cette optimisation.

1.3.10 Calculs booléens

L'évaluation des expressions booléennes est paresseuse grâce à une implémentation par flot de contrôle (succession de branchement) permettant d'éviter le stockage des valeurs intermédiaires dans des registres. Le point de départ se situe dans **AbstractExpr** dont la méthode **codeGenExprOnRegister** initialise le processus en évaluant par défaut la valeur à false. L'appel à **codeGenBool** permet de générer le code d'évaluation. Ainsi soit il n'y a pas eu de branchement et l'expression arrive a **endlabel** et reste donc évaluée à 0 (false) soit il y a un saut à **label** et la valeur est mise à 1 (true). On remarque que dans le cas de clause **IfThenElse** et **While** la valeur booléenne n'est jamais chargée dans un registre.

Certaines optimisations ont également été mises en place afin d'évaluer à la compilation les expression triviallement vraie/fausse via les méthode **isTriviallyTrue** et **isTriviallyFalse** de **AbstractExpr**. Pour l'instant, elles se limitent aux expressions constantes (pour e = true false, **e.isTriviallyFalse()** est true). Ceci permet l'élimination de code mort lors de clauses **IfThenElse** et **While**.

Dans le cas de **IfThenElse**, si **isTriviallyTrue()** est vraie alors seule la branche **then** est générée. Inversement, si **isTriviallyFalse()** est vraie alors seule la branche **else** est générée.

Dans le cas de **While**, si **isTriviallyFalse()** est vraie alors le corps de la

boucle est ignoré.

1.3.11 Méthodes

La création de méthode repose sur le même principe que la création d'un programme principale puisqu'elle doit générer récursivement la déclaration de variables et les instructions. Cependant, il faut mettre à jour les adresses des paramètres. Pour cela, on met à jour un compteur correspondant à l'offset du paramètre par rapport à SP et on fait appel au codeGen de **DeclParam** avec qui met à jour l'adresse du paramètre dans sa définition. Lors de l'appel d'une méthode dans **MethodCall**, on récupère l'index de la méthode à appeler dans la table des méthodes avec le champs index définit dans la définition de la méthode

1.3.12 Champs

Lors de la déclaration d'un champ, on fait un simplement un appel à une méthode de la classe **Stack** qui va le stocker dans le tas. Pour sélectionner ce champs, on charge l'adresse dans la tas de l'instance de la classe et on récupère la valeur du champs à partir de son index.

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. Dans ce cas là, chaque fichier est lancé dans un thread qui lance la compilation avec la classe **ParallelCompile** qui implemente **Runnable**. 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 cœur 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'**Etape A** est terminé et l'arbre abstrait (une instance de **AbstractProgram**) est obtenue.

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

L'option **JavaCompilation** permet de décider le compilateur à utiliser **IMACompiler** pour IMA où **JavaCompiler** pour une execution sur la JVM.

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 valide, 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**.