[gl19]

Extension : Une bibliothèque de génération du bytecode Java

L'analyse, la génération et la transformation de programmes sont des techniques fondamentales de l'ingénierie logicielle, elles sont utilisées dans de nombreuses situations :

1. **L'analyse de programmes :** cela peut aller d'une simple analyse syntaxique à une analyse sémantique complète, peut être utilisée pour trouver des bugs potentiels dans les applications, pour détecter du code inutilisé, etc.
2. **La génération de programmes :** utilisée principalement par les compilateurs pour générer du code dans un langage cible à partir d'une certaine structure de donnée décorée pendant l'étape d'analyse.
3. **La transformation de programmes :** peut être utilisée pour optimiser ou obscurcir des programmes, pour insérer du code de débogage ou de contrôle des performances dans les applications.

L'étape qui nous intéresse plus particulièrement dans cette extension est la génération du bytecode, il s'agit de générer des fichiers .class prêtes à être exécutés directement sur la JVM .
Pour ce faire, on va utiliser un outil logiciel qui s'appelle ASM.

ASM :

ASM est un cadre polyvalent de manipulation et d'analyse du bytecode Java. Il peut être utilisé pour modifier des classes existantes ou pour générer dynamiquement des classes, directement sous forme binaire. ASM fournit quelques transformations de bytecode et algorithmes d'analyse communs à partir desquels des transformations complexes et des outils d'analyse de code personnalisés peuvent être construits. ASM offre des fonctionnalités similaires à celles des autres frameworks Java bytecode, mais se concentre sur les performances. Parce qu'il a été conçu et implémenté pour être aussi petit et aussi rapide que possible, il est bien adapté à une utilisation dans des systèmes dynamiques (mais peut bien sûr être utilisé de manière statique aussi, notamment pour des compilateurs).

Modèle :
La bibliothèque ASM fournit deux API pour générer et transformer les classes compilées. l'API core fournit une représentation des classes basée sur des événements, tandis que l'API arbre fournit une représentation basée sur les objets.

Avec le modèle basé sur les événements, une classe est représentée par une séquence d'événements, chaque événement représentant un élément de la classe, tel que son en-tête, un champ, une déclaration de méthode, une instruction, etc.
L'API basée sur les événements définit l'ensemble des événements possibles et l'ordre dans lequel ils doivent se produire, et fournit un analyseur de classe qui génère un événement par élément analysé, ainsi qu'un rédacteur de classe writer qui génère des classes compilées à partir de séquences de tels événements.

Avec le modèle basé sur les objets, une classe est représentée par un arbre d'objets représentant une partie de la classe, comme la classe elle-même, un champ, une méthode, une instruction, etc. et chaque objet a des références aux objets qui représentent ses constituants.
L'API basée sur les objets fournit un moyen de convertir une séquence d'événements représentant une classe vers l'arbre d'objets représentant la même classe et, inversement, de convertir un arbre d'objets en une séquence d'événements équivalente. En d'autres termes, l'API basée sur les objets est construite au-dessus de l'API basée sur les événements.

ASM fournit les deux API car il n'y a pas de meilleure API. En effet, chaque API a ses propres avantages et inconvénients :
➔ L'API basée sur les événements est plus rapide et nécessite moins de mémoire que l'API basée sur les objets, puisqu'il n'est pas nécessaire de créer et de stocker en mémoire un arbre d'objets représentant la classe.
➔ Cependant, l'implémentation de transformations de classes peut être plus difficile avec l'API basée sur les événements, puisqu'un seul élément de la classe est disponible à un moment donné (l'élément qui correspond à l'événement en cours), alors que la classe entière est disponible en mémoire avec l'API basée sur les objets.

L’API qu’on va utiliser dans ce projet est L’API core, ce choix se justifie par la simplicité et la lisibilité du code généré. cet API se situe dans le paquetage org.objectweb.asm

Format des fichiers class :

La structure globale d'une classe compilée est assez simple. En effet, une classe compilée conserve les informations structurelles et presque tous les symboles du code source. Elle contient :

➔ Une section décrivant les modificateurs (par ex : la visibilité ,le code du format), le nom, la super classe, les interfaces et les annotations de la classe.

➔ Une section par attribut déclaré dans cette classe. Chaque section décrit les modificateurs, le nom, le type et les annotations d'un attribut.

➔ Une section par méthode et constructeur déclarés dans cette classe. Chaque section décrit les modificateurs, le nom, les types de retour et de paramètres, et les annotations d'une méthode. Elle contient également le code compilé de la méthode, sous la forme d'une séquence d'instructions en bytecode Java.

Une classe compilée contient aussi une section appelée constant pool. Il s’agit d’un tableau contenant toutes les constantes numériques, chaînes de caractères et types qui apparaissent dans la classe. Ces constantes sont définies une seule fois dans la section constant pool, et sont référencées par leur index dans toutes les autres sections du fichier de classe.

Le tableau ci-dessous résume la structure globale d'une classe compilée :

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

L'API ASM est basée sur la classe abstraite ClassVisitor. Chaque méthode de cette classe correspond à la section de la structure du fichier de la classe du même nom. Les sections simples sont visitées par un seul appel de méthode dont les arguments décrivent leur contenu, et qui renvoie void. Les sections dont le contenu peut être de longueur et de complexité arbitraires sont visitées avec un appel de méthode initial qui renvoie une classe auxiliaire de visiteur. C'est le cas de visitAnnotation, visitField et visitMethod, qui renvoient un AnnotationVisitor, un FieldVisitor et un MethodVisitor respectivement.

```
public abstract class ClassVisitor {
  public ClassVisitor(int api);
  public ClassVisitor(int api, ClassVisitor cv);
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces);
  public void visitSource(String source, String debug);
  public void visitOuterClass(String owner, String name, String desc);
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  public void visitAttribute(Attribute attr);
  public void visitInnerClass(String name, String outerName,
      String innerName, int access);
  public FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value);
  public MethodVisitor visitMethod(int access, String name, String desc,
      String signature, String[] exceptions);
  void visitEnd();
}
```

Les méthodes de la ClassVisitor doivent être appelées dans l'ordre suivant :

```
visit visitSource? visitOuterClass? ( visitAnnotation | visitAttribute )*
( visitInnerClass | visitField | visitMethod )*
visitEnd
```

ASM fournit trois composants principaux basés sur l'API ClassVisitor pour
générer et transformer des classes :

➔ La classe ClassReader analyse une classe compilée donnée sous la forme d'un tableau d'octets, et appelle les méthodes visitXxx correspondantes sur l'instance de ClassVisitor passée en argument à sa méthode accept. Elle peut être vue comme un producteur d'événements.

➔ La classe ClassWriter est une sous-classe de la classe abstraite ClassVisitor qui construit les classes compilées directement sous forme binaire. Elle produit en tant que produit en sortie un tableau d'octets contenant la classe compilée, qui peut être récupéré avec la méthode toByteArray. Elle peut être considérée comme un consommateur d'événements.

➔ La classe ClassVisitor délègue tous les appels de méthode qu'elle reçoit à une autre instance de ClassVisitor. Elle peut être considérée comme un filtre d'événements.

Modèle d'exécution :

Avant de présenter les instructions du bytecode, il est nécessaire de présenter le modèle d'exécution de la JVM :

Le code Java est exécuté à l'intérieur de threads. Chaque thread possède sa propre pile d'exécution, qui est constituée de frames . Chaque frame représente une invocation de méthode : chaque fois qu'une méthode est invoquée, une nouvelle frame est placée sur la pile d'exécution du thread en cours. Lorsque la méthode se termine, que ce soit normalement ou à cause d'une exception, cette frame est retirée de la pile d'exécution et l'exécution continue dans la méthode appelante.

Chaque frame contient deux éléments : une table de variables locales et une pile d'opérandes (les valeurs utilisées comme opérandes par les instructions du bytecode).
Les variables locales sont accessibles immédiatement par leur index. alors que les opérandes ne peuvent être accédées que dans l'ordre imposé par la pile (Last In First Out).

Lorsqu'elle est créée, une frame est initialisée avec une pile vide, et ses variables locales sont initialisées avec l'objet cible this (sauf pour les méthodes statiques) et avec les arguments de la méthode. Par exemple, l'appel de la méthode a.equals(b) crée une frame avec une pile vide et avec les deux premières variables locales initialisées à a et b (les autres variables locales ne sont pas initialisées).

Instructions bytecode :
Une instruction bytecode est composée d'un opcode qui identifie cette instruction, et d'un nombre fixe d'arguments :
➔ L'opcode : une valeur d'octet non signée - d'où le nom de bytecode - et est identifiée par un symbole mnémonique. Par exemple, la valeur opcode 0 est désignée par le symbole mnémonique NOP, et correspond à l'instruction qui ne fait rien.
➔ Les arguments : des valeurs statiques qui définissent le comportement précis de l'instruction. Ils sont donnés juste après l'opcode. Par exemple, l'instruction GOTO label dont la valeur de l'opcode est 167, prend comme argument label, une étiquette qui désigne l'instruction suivante à exécuter.

Les instructions du bytecode peuvent être divisées en deux catégories : un petit ensemble d'instructions est conçu pour transférer des valeurs des variables locales vers la pile d'opérandes et vice versa, les autres instructions n'agissent que sur la pile des opérandes : elles extraient certaines valeurs de la pile, calculent un résultat basé sur ces valeurs, et le repoussent sur la pile.

Les instructions xLOAD lisent une variable locale et poussent sa valeur sur la pile d'opérandes. Elles prennent comme argument

l'index de la variable locale qui doit être lue. ILOAD est utilisée pour charger un booléen, octet, char, short, ou int variable locale. LLOAD, FLOAD et DLOAD sont utilisés pour charger une valeur longue, flottante ou double, respectivement Enfin, ALOAD est utilisé pour charger une référence c'est-à-dire un objet ou une variable. Symétriquement, les instructions xSTORE extraient une valeur de la pile d'opérandes et la stockent dans une variable locale désignée par le nom de la variable .

Toutes les autres instructions du bytecode travaillent uniquement sur la pile d'opérandes. Elles peuvent être regroupées dans les catégories suivantes :

Pile :
Ces instructions sont utilisées pour manipuler les valeurs sur la pile :
POP : retire la valeur au sommet de la pile
DUP : pousse une copie de la valeur au sommet de la pile
SWAP : retire deux valeurs et les pousse dans l'ordre inverse, etc.

Constantes :
Ces instructions poussent une constante sur la pile des opérandes :
ACONST_NULL : pousse null
ICONST_0 : pousse la valeur int 0
BIPUSH : pousse la valeur d'un octet donné en argument
SIPUSH : pousse la valeur d'un short donné en argument
LDC : pousse une valeur arbitraire int, float, ou référence, etc.

Arithmétique et logique :
Ces instructions extraient des valeurs numériques de la pile d'opérandes les combinent et poussent le résultat sur la pile. Elles n'ont pas d'arguments.
xADD, xSUB, xMUL, xDIV et xREM correspondent respectivement aux opérations d'addition, soustraction, multiplication, division et modulo et où x désigne le type des opérandes.

Casts :
Ces instructions extraient une valeur de la pile, la convertissent en un autre type et repoussent le résultat, elles ont la forme x2y où x désigne le type de base et y le type voulu.

Objets :
Ces instructions sont utilisées pour créer des objets,tester leur type, etc. Par exemple, l'instruction NEW type pousse un nouvel objet de type type sur la pile .

Attributs :
Ces instructions lisent ou écrivent la valeur d'un attribut.
GETFIELD owner name desc : extrait une référence d'objet, et pousse la valeur de son attribut name dans la pile
PUTFIELD owner name desc :retire une valeur et une référence d'objet, et stocke cette valeur dans son attribut name

Dans les deux cas, l'objet doit être de type owner, et son attribut doit être de type desc. GETSTATIC et PUTSTATIC sont des instructions similaires, mais pour des attributs statiques.

Méthodes :
Ces instructions invoquent une méthode ou un constructeur. Elles génèrent autant de valeurs qu'il y a d'arguments de méthode, plus une valeur pour l'objet cible. et poussent le résultat de l'invocation de la méthode.
INVOKEVIRTUAL name owner desc : invoque la méthode nom définie dans la classe propriétaire et dont le descripteur de méthode est desc.
INVOKESTATIC : est utilisé pour les méthodes statiques
INVOKESPECIAL : pour les méthodes privées et les constructeurs
Sauts :
Ces instructions sautent vers une instruction arbitraire si une certaine condition est vraie. Elles sont utilisées pour compiler les instructions if else, for, while Par exemple, l'étiquette IFEQ extrait une valeur int de la pile, et saute à l'instruction désignée par

l'étiquette si cette valeur est égale à 0 (sinon, l'exécution continue normalement à l'instruction suivante).

Return :
Enfin, les instructions xRETURN et RETURN sont utilisées pour terminer l'exécution d'une méthode et pour retourner son résultat à l'appelant. RETURN est utilisé pour les méthodes qui retournent void, et xRETURN pour les autres méthodes.

Le sous langage traité :
Dans cet extension on se restreindra au sous langage suivant :

```
PROGRAM     →
                Program[MAIN]
MAIN        →
                EmptyMain
            |   Main[DECL_VAR LIST_INST]
DECL_VAR    →
                DeclVar[IDENTIFIER IDENTIFIER INITIALIZATION]
IDENTIFIER  →
                Identifier
INITIALIZATION  →
                NoInitialization
            |   Initialization[EXPR]
EXPR        →
                BINARY_EXPR
            |   LVALUE
            |   STRING_LITERAL
            |   UNARY_EXPR
            |   BooleanLiteral
            |   FloatLiteral
            |   IntLiteral
BINARY_EXPR →
                OP_ARITH
            |   OP_BOOL
            |   OP_CMP
            |   Assign[LVALUE EXPR]
```

```
OP_ARITH      →
                Divide[EXPR EXPR]
            |   Minus[EXPR EXPR]
            |   Modulo[EXPR EXPR]
            |   Multiply[EXPR EXPR]
            |   Plus[EXPR EXPR]
OP_BOOL       →
                And[EXPR EXPR]
            |   Or[EXPR EXPR]
OP_CMP        →
                OP_EXACT_CMP
            |   OP_INEQ
OP_INEQ       →
                Greater[EXPR EXPR]
            |   GreaterOrEqual[EXPR EXPR]
            |   Lower[EXPR EXPR]
            |   LowerOrEqual[EXPR EXPR]
LVALUE        →
                Identifier
STRING_LITERAL →
                StringLiteral
UNARY_EXPR      →
                Not[EXPR]
            |   UnaryMinus[EXPR]

INST      →
                EXPR
            |   PRINT
            |   IfThenElse[ EXPR LIST_INST LIST_INST]
            |   While[ EXPR LIST_INST]
PRINT     →
                Print[LIST_EXPR]
            |   Println[LIST_EXPR]

LIST_DECL_VAR →   [DECL_VAR*]
LIST_EXPR     →   [EXPR*]
LIST_INST     →   [INST*]
```

Structure des classes générées :
Chaque fichier Deca est associé à une classe portant le même nom, la méthode main de cette classe contient exactement le bloc main du fichier Deca.
Ainsi , sauf erreur de compilation l'exécution du fichier Deca sous ima devra donner le même résultat d'exécution de la classe généré (ou plutôt de sa fonction main) sous la JVM.
Il faut bien noter que le classe générée est en classformat, et donc immédiatement exécutable sur la JVM.

Mécanisme de génération :
La génération du bytecode se fait en suivant le patron de conception Visiteur :
En effet chaque classe de la grammaire implémente une fonction genByte dont la signature et la suivante :
void genByte(MethodVisitor method ,HashMap<Symbol,Integer> indexTable);
method : représente le MethodVisitor de la fonction main de la classe généré.
indexTable : représente la table des variables locales de la frame .
cette fonction utilise l'ensemble des instructions suivantes :

```
abstract class MethodVisitor { // public accessors ommited
  MethodVisitor(int api);
  MethodVisitor(int api, MethodVisitor mv);
  AnnotationVisitor visitAnnotationDefault();
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  AnnotationVisitor visitParameterAnnotation(int parameter,
      String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitCode();
  void visitFrame(int type, int nLocal, Object[] local, int nStack,
      Object[] stack);
  void visitInsn(int opcode);
  void visitIntInsn(int opcode, int operand);
  void visitVarInsn(int opcode, int var);
  void visitTypeInsn(int opcode, String desc);
  void visitFieldInsn(int opc, String owner, String name, String desc);
  void visitMethodInsn(int opc, String owner, String name, String desc);
  void visitInvokeDynamicInsn(String name, String desc, Handle bsm,
      Object... bsmArgs);
  void visitJumpInsn(int opcode, Label label);
  void visitLabel(Label label);
  void visitLdcInsn(Object cst);
  void visitIincInsn(int var, int increment);
  void visitTableSwitchInsn(int min, int max, Label dflt, Label[] labels);
  void visitLookupSwitchInsn(Label dflt, int[] keys, Label[] labels);
  void visitMultiANewArrayInsn(String desc, int dims);
  void visitTryCatchBlock(Label start, Label end, Label handler,
      String type);
  void visitLocalVariable(String name, String desc, String signature,
      Label start, Label end, int index);
  void visitLineNumber(int line, Label start);
  void visitMaxs(int maxStack, int maxLocals);
  void visitEnd();
}
```

Vous trouverez ici plus de détails sur L'API asm

Exemple de code :

```
@Override
public void genByte(MethodVisitor method,
                    HashMap<SymbolTable.Symbol, Integer> indexTable) {
    org.objectweb.asm.Label label = new org.objectweb.asm.Label();
    method.visitLabel(label);
    condition.genByte(method,indexTable);
    org.objectweb.asm.Label end = new org.objectweb.asm.Label();
    method.visitJumpInsn(IFEQ, end);
    body.genByte(method,indexTable);
    method.visitJumpInsn(GOTO, label);
    method.visitLabel(end);
}
```

Cette fonction génère le bytecode de l'instruction while, il commence par charger la condition dans la pile d'opérandes, si cette condition est vraie i.e : si sa valeur est non nulle, on exécute le corps de la boucle et on recharge la condition à nouveau jusqu'à ce que la condition devienne fausse auquel cas on sort de la boucle.

Inclusion dans le compilateur Decac :

La présence du compilateur bytecode est traduit par l'option -j au sein de la classe CompilerOptions .

Une fois cet option est activée : DecaCompiler appelle la fonction genByte sur l'instance program et génère ainsi un fichier .class qui sera automatiquement placé dans le répertoire courant.

Démo :