Génération de Bytecode Java avec ASM

Roboam Guillaume Raballand Cyprien Talbi El Mehdi
Charles-Mennier Matéo Altieri Aubin

GL1, GL01, Équipe 1
22 janvier 2025

Contents

1 Spécification de l'extension

L'objectif de cette extension est de permettre la génération automatique du bytecode Java à partir d'un programme écrit en langage Deca. Elle vise à produire des fichiers `.class` directement exécutables par la JVM (Java Virtual Machine), sans nécessiter une étape intermédiaire de compilation en code source Java. L'extension assure ainsi une traduction fidèle du code Deca en instructions bytecode JVM, tout en respectant les contraintes du modèle de données et les règles de gestion mémoire du langage Deca.

1.1 Objectifs principaux

L'extension vise à fournir les fonctionnalités suivantes :

- **Traduction automatique de Deca vers bytecode JVM**
  - Génération de fichiers `.class` valides et exécutables par la JVM.
  - Support des structures de contrôle (boucles, conditions, etc.).
  - Prise en charge des types de données primitifs et objets.
- **Interopérabilité avec la JVM**
  - Respect des conventions de la JVM en termes de pile d'opérandes, d'appels de méthodes et de gestion de la mémoire.
  - Compatibilité avec les outils d'analyse et de désassemblage de bytecode (ex. `javap`).

1.2 Fonctionnalités prises en charge

L'extension est spécifiée pour prendre en charge les éléments suivants du langage Deca :

- **Deca de base (sans objets)**
  - Prise en charge des types de données primitifs (`int`, `float`, `boolean`, `String`).
  - Opérations arithmétiques et logiques sur les types primitifs.
  - Structures de contrôle de base (`if-else`, `while`).
- **Deca essentiel**
  - Déclaration et utilisation d'objets avec gestion des constructeurs.
  - Gestion correcte du flux d'exécution avec les opcodes de branchement (`GOTO`, `IFNE`, etc.).
  - Prise en charge des appels de méthodes et de la surcharge.
  - Gestion de readFloat, readInt à partir de la classe Scanner
- **Deca complet (cast et instanceof)**
  - Implémentation des opérations de cast explicites et des tests d'instance (`instanceof`).

1.3 Technologies et outils utilisés

L'extension repose sur les outils et bibliothèques suivants :

- **ASM (ObjectWeb ASM)** :
  - Utilisé pour la génération et la manipulation du bytecode Java de manière programmatique.
- **Java Development Kit (JDK)** :
  - Pour l'exécution et la validation des fichiers `.class` générés.
- **Scripts d'automatisation (Bash)** :
  - Permettant l'exécution et la comparaison des résultats d'exécution.

2 Utilisation de la librairie `ASM`

`ClassWriter` est la classe centrale d'ASM pour générer une classe complète en bytecode. Lors du parcours de l'AST, lorsqu'un nœud correspondant à une classe est rencontré, le `ClassWriter` est utilisé pour commencer à écrire la définition de cette classe.

2.1 Exemple de code

Voici un exemple d'utilisation de `ClassWriter` en Java :

```
1 ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES |
2                                  ClassWriter.COMPUTE_MAXS);
```

Listing 1: ClassWriter

2.2 Options de `ClassWriter`

L'option `ClassWriter.COMPUTE_FRAMES` permet à ASM de calculer automatiquement les stack frames nécessaires pour les instructions conditionnelles telles que les sauts, les boucles, etc. Cela simplifie le processus de génération de bytecode en évitant à l'utilisateur de fournir manuellement ces informations complexes.

L'option `ClassWriter.COMPUTE_MAXS` permet à ASM de calculer automatiquement :

- La profondeur maximale de la pile utilisée par la méthode.
- Le nombre maximal de variables locales utilisées.

Ces options garantissent une génération correcte et optimisée du bytecode sans risque d'erreurs liées à la gestion manuelle des ressources.

2.3 Utilisation de `TraceClassVisitor`

`TraceClassVisitor` est un outil de traçage qui intercepte les appels à `ClassWriter` et écrit une version lisible par un humain des instructions bytecode, également appelée pseudo-ASM. Cela permet de générer un fichier contenant le bytecode sous une forme plus compréhensible.

Principe de fonctionnement Lors du parcours de l'AST, chaque instruction émise par `MethodVisitor` est interceptée par `TraceClassVisitor`, qui la traduit en pseudo-assembleur lisible.

Exemple d'utilisation avec TraceClassVisitor

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES |
    ClassWriter.COMPUTE_MAXS);
PrintWriter pw = new PrintWriter(System.out);
TraceClassVisitor tcv = new TraceClassVisitor(cw,
                                              new ASMifier(),
                                              pw);
```

Listing 2: Exemple d'utilisation de TraceClassVisitor

2.4 Utilisation de `MethodVisitor`

`MethodVisitor` est utilisé pour définir le contenu d'une méthode, c'est-à-dire ajouter des instructions au corps de la méthode.

Création d'une méthode Voici un exemple de création de la méthode `main` en utilisant `MethodVisitor` :

```
MethodVisitor mv = tcv.visitMethod(
                Opcodes.ACC_PUBLIC + Opcodes.ACC_STATIC,
                "main",
                "([Ljava/lang/String;)V",
                null,
                null
            );
```

Listing 3: Exemple d'utilisation de MethodVisitor

2.5 Flux de travail général

1. Création d'une instance de `ClassWriter` pour générer le bytecode.
2. Ajout de la classe et de ses méthodes via `MethodVisitor`.
3. Optionnellement, traçage du bytecode généré à l'aide de `TraceClassVisitor`.
4. Génération finale et conversion en tableau de bytes.

2.6 Les Opcodes

Les opcodes sont les instructions du bytecode Java. Ils représentent les commandes que la machine virtuelle Java (JVM) peut exécuter. Chaque opcode est une instruction bas-niveau qui effectue une tâche précise, comme :

- Charger une valeur sur la pile.
- Effectuer une opération arithmétique.
- Appeler une méthode.

Dans notre extension, nous utilisons les opcodes pour générer dynamiquement le bytecode correspondant aux instructions du langage source. Ces opcodes permettent, par exemple, de :

- **IADD** : effectuer une addition d'entiers.
- **ISUB** : effectuer une soustraction d'entiers.
- **ILOAD** : charger un entier depuis une variable locale sur la pile.
- **ISTORE** : stocker une valeur dans une variable locale.
- **IF_ICMPGE** : effectuer un saut conditionnel si le entier est supérieur ou égal à l'entier 2.
- **INVOKEVIRTUAL** : appeler une méthode d'instance.

2.7 LocalIndex : Gestion des variables locales

Pour bien comprendre le rôle de la pile d'exécution, il est essentiel de parler des indices locaux (`LocalIndex`), qui sont des identifiants numériques faisant référence aux variables locales utilisées par une méthode dans la JVM.

2.7.1 Pourquoi les indices locaux sont-ils nécessaires ?

La JVM utilise une pile d'exécution pour évaluer les instructions, mais cette pile est temporaire. Les variables locales permettent de :

- **Sauvegarder des résultats intermédiaires** : Lors des opérations arithmétiques, les résultats doivent être stockés pour des calculs ultérieurs.
- **Référencer les paramètres de méthode** : Les paramètres d'une méthode sont automatiquement stockés dans la table des variables locales, à partir de l'index 0.
- **Préserver les données entre appels de méthodes** : Contrairement à la pile, la mémoire locale permet de conserver des valeurs accessibles plusieurs fois.

2.7.2 Exemple d'utilisation

```
MethodVisitor mv = cw.visitMethod(Opcodes.ACC_PUBLIC + Opcodes
    .ACC_STATIC,
                                  "add", "(II)I", null, null);

mv.visitVarInsn(Opcodes.ILOAD, 0);  // Charger le premier
    param tre
mv.visitVarInsn(Opcodes.ILOAD, 1);  // Charger le deuxi me
    param tre
mv.visitInsn(Opcodes.IADD);         // Additionner les deux
    valeurs
mv.visitVarInsn(Opcodes.ISTORE, 2);  // Stocker le r sultat
    dans l'index 2
mv.visitInsn(Opcodes.IRETURN);      // Retourner le r sultat
mv.visitMaxs(3, 3);
mv.visitEnd();
```

Listing 4: Exemple d'utilisation des variables locales

2.8 Comment tout cela fonctionne ensemble ?

Le processus de génération du bytecode suit les étapes suivantes :

2.8.1 1. Initialisation de `ClassWriter`

Le `ClassWriter` est utilisé pour créer une structure de classe binaire afin de générer un fichier `.class`.

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES |
    ClassWriter.COMPUTE_MAXS);
```

Listing 5: Initialisation de ClassWriter

2.8.2 2. Ajout du traçage avec `TraceClassVisitor`

Le `TraceClassVisitor` enregistre une version lisible du bytecode, qui peut être affichée dans la console ou enregistrée dans un fichier texte.

```
PrintWriter pw = new PrintWriter(new FileOutputStream("Main.
    txt"));
TraceClassVisitor tcv = new TraceClassVisitor(cw, new ASMifier
    (), pw);
```

Listing 6: Ajout du traçage

2.8.3 3. Création des méthodes avec `MethodVisitor`

Les `MethodVisitor` permettent d'ajouter des instructions dans chaque méthode.

```
MethodVisitor mv = tcv.visitMethod(Opcodes.ACC_PUBLIC +
    Opcodes.ACC_STATIC,
                                    "main", "([Ljava/lang/
                                        String;)V", null, null)
                                        ;

mv.visitCode();
mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "out"
    , "Ljava/io/PrintStream;");
mv.visitLdcInsn("Hello, ASM!");
mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL,
                   "java/io/PrintStream",
                   "println",
                   "(Ljava/lang/String;)V",
                   false);
mv.visitInsn(Opcodes.RETURN);
mv.visitMaxs(0, 0);
mv.visitEnd();
```

Listing 7: Création de la méthode main

2.8.4 4. Finalisation et écriture du fichier

Une fois toutes les méthodes et champs ajoutés, le bytecode est écrit dans un fichier `.class`.

```
byte[] bytecode = cw.toByteArray();

try (FileOutputStream fos = new FileOutputStream("Main.class")
    ) {
    fos.write(bytecode);
    System.out.println("Wrote Main.class successfully.");
}
```

Listing 8: Écriture du fichier .class

Le résultat génère deux fichiers:

- `XXX.class` : Un fichier binaire exécutable contenant le bytecode Java, pouvant être exécuté via la commande suivante :

  ```
  java -cp path/de/XXX.class XXX
  ```

- `XXX.txt` : Une version lisible du bytecode généré grâce à `TraceClassVisitor`, affichant le code en pseudo-assembleur lisible.

3 Extension pour Deca sans Objet

3.1 Opérations Arithmétiques

Cette section décrit le processus de génération du bytecode pour les opérations arithmétiques. Étant donné leur similitude, il est suffisant d'expliquer en détail le code d'une seule opération, les autres suivant un schéma comparable.

```
@Override
protected void codeGenByteInst(MethodVisitor mv, DecacCompiler
     compiler) {
    getLeftOperand().codeByteExp(mv, compiler);
    int leftVarIndex = compiler.allocateLocalIndex();

    if (getType().isInt()) {
        mv.visitVarInsn(Opcodes.ISTORE, leftVarIndex);
    } else if (getType().isFloat()) {
        mv.visitVarInsn(Opcodes.FSTORE, leftVarIndex);
    } else {
        throw new UnsupportedOperationException("Unsupported
            type: " + getType());
    }

    getRightOperand().codeByteExp(mv, compiler);

    if (getType().isInt()) {
        mv.visitVarInsn(Opcodes.ILOAD, leftVarIndex);
        mv.visitInsn(Opcodes.IADD);
    } else {
        mv.visitVarInsn(Opcodes.FLOAD, leftVarIndex);
        mv.visitInsn(Opcodes.FADD);
    }
}
```

Listing 9: Génération du bytecode pour les opérations arithmétiques (plus dans l'example)

3.1.1 Explication du processus

Évaluation du premier opérande La méthode `codeByteExp` est appelée pour générer le bytecode du premier opérande, laissant sa valeur au sommet de la pile. Ensuite, un index local est alloué dans la table des variables locales pour stocker cette valeur.

Stockage selon le type Selon le type de l'opération (déterminé par `getType()`), la valeur est stockée comme suit :

- `ISTORE` : si le type est un entier (`int`), la valeur au sommet de la pile est stockée dans la variable locale à l'index alloué.

- `FSTORE` : si le type est un flottant (`float`), la valeur est stockée de la même manière.
- Si aucun des deux types n'est reconnu, une exception est levée.

Évaluation du second opérande Le second opérande est ensuite évalué de la même manière, laissant sa valeur au sommet de la pile.

Récupération et addition La valeur du premier opérande est récupérée à partir de la table des variables locales et l'opération d'addition est effectuée en utilisant :

- `IADD` : pour les entiers (`int`).
- `FADD` : pour les flottants (`float`).

Importance des indices locaux Les indices locaux sont utilisés pour stocker temporairement le premier opérande, ce qui permet de :

- Libérer la pile entre les évaluations des deux opérandes.
- Réutiliser la valeur du premier opérande sans la recalculer.

Sans la table des variables locales, les deux opérandes devraient être conservés simultanément sur la pile, ce qui compliquerait la gestion des calculs complexes.

Autres opérations arithmétiques La même logique est appliquée à d'autres opérations arithmétiques, en remplaçant simplement `IADD/FADD` par les opcodes appropriés :

- `IMUL/FMUL` : multiplication.
- `ISUB/FSUB` : soustraction.
- `IDIV/FDIV` : division.

3.2 Gestion des Structures Conditionnelles

Pour gérer les structures conditionnelles dans le bytecode, nous utilisons les labels d'ASM pour marquer les points clés du bloc `if-else`. Le processus repose sur la création de labels pour structurer les sauts conditionnels.

3.2.1 Code Généré pour une Structure `if-else`

Le code suivant illustre la génération du bytecode pour une instruction conditionnelle en utilisant la bibliothèque ASM.

```
@Override
protected void codeGenByteInst(MethodVisitor mv, DecacCompiler
     compiler) {
    Label elseLabel = new Label();
    Label endLabel  = new Label();

    condition.codeGenByteBool(mv, false, elseLabel, compiler);

    thenBranch.codeGenListInstByte(mv, compiler);
    mv.visitJumpInsn(Opcodes.GOTO, endLabel);

    mv.visitLabel(elseLabel);
    elseBranch.codeGenListInstByte(mv, compiler);

    mv.visitLabel(endLabel);
}
```

Listing 10: Génération du bytecode pour if-else

Explication du Processus

L'approche repose sur les étapes suivantes :

- **Création de labels :**
  - `elseLabel` : Détermine où commence la branche `else`.
  - `endLabel` : Marque la fin de la structure conditionnelle.
- **Génération de la condition :** L'instruction `condition.codeGenByteBool(mv, false, elseLabel, compiler)` génère le bytecode pour évaluer la condition. Si la condition est fausse (`false`), le programme saute directement à `elseLabel`, ignorant la branche `then`.
- **Branche then :**
  - `thenBranch.codeGenListInstByte` génère le bytecode pour les instructions de la branche `then`.
  - `mv.visitJumpInsn(Opcodes.GOTO, endLabel)` permet de sauter directement à la fin pour éviter d'exécuter accidentellement la branche `else`.
- **Branche else :**
  - `mv.visitLabel(elseLabel)` définit le point de départ de la branche `else`.
  - `elseBranch.codeGenListInstByte` génère le bytecode pour les instructions de la branche `else`.
- **Fin de la structure :** `mv.visitLabel(endLabel)` indique la fin du bloc `if-else`, garantissant la poursuite correcte de l'exécution.

3.3 Gestion des Boucles `while`

Pour implémenter les structures de boucle `while` en bytecode Java à l'aide d'ASM, nous utilisons des labels pour marquer le début et la fin de la boucle.

3.3.1 Code Généré pour la Boucle `while`

```
@Override
protected void codeGenByteInst(MethodVisitor mv, DecacCompiler
     compiler) {
    Label whileStart = new Label();
    Label whileEnd = new Label();

    mv.visitLabel(whileStart);

    condition.codeGenByteBool(mv, false, whileEnd, compiler);
    body.codeGenListInstByte(mv, compiler);

    mv.visitJumpInsn(Opcodes.GOTO, whileStart);
    mv.visitLabel(whileEnd);
}
```

Listing 11: Génération du bytecode pour une boucle while

Explication du Processus

La génération du bytecode pour une boucle `while` suit les étapes suivantes :

- **Création des labels :**
  - `whileStart` : Marque le début de la boucle.
  - `whileEnd` : Indique la fin de la boucle.
- **Évaluation de la condition :**
  - `condition.codeGenByteBool(mv, false, whileEnd, compiler)` génère le bytecode pour vérifier la condition de la boucle.
  - Si la condition est fausse, l'exécution saute à `whileEnd`, sortant de la boucle.
- **Exécution du corps de la boucle :**
  - `body.codeGenListInstByte(mv, compiler)` génère les instructions du corps de la boucle.
- **Retour au début de la boucle :**
  - `mv.visitJumpInsn(Opcodes.GOTO, whileStart)` fait revenir le contrôle à l'évaluation de la condition.
- **Fin de la boucle :** Le label `whileEnd` marque le point de sortie de la boucle.

3.4 Génération du Bytecode pour les Instructions d'Affichage

```
1
2  @Override
3  protected void codeGenByteInst(MethodVisitor mv, DecacCompiler
       compiler) {
4      super.codeGenByteInst(mv, compiler);
5
6      mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "
           out",
7                        "Ljava/io/PrintStream;");
8      mv.visitLdcInsn("\n");
9      mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/
           PrintStream",
10                       "print", "(Ljava/lang/String;)V", false)
                             ;
11 }
12
13 protected void codeGenBytePrint(MethodVisitor mv,
       DecacCompiler compiler) {
14     mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "
           out",
15                       "Ljava/io/PrintStream;");
16
17     codeByteExp(mv, compiler);
18
19     Type type = getType();
20     if (type.isInt()) {
21         mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/
               PrintStream",
22                            "print", "(I)V", false);
23     } else if (type.isFloat()) {
24         mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/
               PrintStream",
25                            "print", "(F)V", false);
26     } else {
27         throw new UnsupportedOperationException("Unsupported
               print type: " + type);
28     }
29 }
```

Listing 12: Génération du bytecode pour println et print

L'implémentation des instructions d'affichage repose sur deux méthodes principales : `codeGenByteInst` et `codeGenBytePrint`. La méthode `codeGenByteInst` assure que chaque élément pouvant être imprimé est traité en appelant d'abord la méthode de la classe parente via `super.codeGenByteInst`. Ensuite, l'objet `PrintStream` est récupéré grâce à l'instruction `GETSTATIC`, suivie d'un appel à la méthode `print` de Java avec la chaîne de caractères correspondant à un retour à la ligne. Cela garantit que chaque impression est suivie d'un saut de ligne, assurant une sortie claire et lisible.

La méthode `codeGenBytePrint` est plus spécifique et prend en compte le type des éléments à afficher. Après avoir récupéré l'objet `PrintStream`, la valeur à imprimer est placée sur la pile via `codeByteExp`. Le type

est ensuite vérifié pour déterminer la méthode `print` appropriée : pour les entiers, la signature utilisée est `(I)V`, tandis que pour les flottants, c'est `(F)V`. Si le type n'est pas pris en charge, une exception est levée. Cette gestion fine des types assure une impression correcte en fonction des données à traiter.

L'implémentation des instructions d'affichage repose sur deux méthodes principales : `codeGenByteInst` et `codeGenBytePrint`. La méthode `codeGenByteInst` assure que chaque élément pouvant être imprimé est traité en appelant d'abord la méthode de la classe parente via `super.codeGenByteInst`. Ensuite, l'objet `PrintStream` est récupéré grâce à l'instruction `GETSTATIC`, suivie d'un appel à la méthode `print` de Java avec la chaîne de caractères correspondant à un retour à la ligne. Cela garantit que chaque impression est suivie d'un saut de ligne, assurant une sortie claire et lisible.

La méthode `codeGenBytePrint` est plus spécifique et prend en compte le type des éléments à afficher. Après avoir récupéré l'objet `PrintStream`, la valeur à imprimer est placée sur la pile via `codeByteExp`. Le type est ensuite vérifié pour déterminer la méthode `print` appropriée : pour les entiers, la signature utilisée est `(I)V`, tandis que pour les flottants, c'est `(F)V`. Si le type n'est pas pris en charge, une exception est levée. Cette gestion fine des types assure une impression correcte en fonction des données à traiter.

3.5 Génération du Bytecode pour la Lecture d'Entrée

La génération du bytecode pour les méthodes `ReadInt` et `ReadFloat` suit une structure similaire, avec quelques différences (remplacer float par int). Le processus commence par la création d'un nouvel objet `Scanner` via l'instruction `NEW`, suivi d'une duplication de la référence sur la pile avec `DUP`, afin de préparer son initialisation. Le champ statique `System.in` est récupéré à l'aide de l'instruction `GETSTATIC`, puis le constructeur de la classe `Scanner` est invoqué via `INVOKESPECIAL`, ce qui permet d'associer le flux d'entrée standard à l'objet nouvellement créé. Une fois l'objet `Scanner` prêt, une seconde duplication est effectuée afin d'appeler la méthode `hasNextInt`, qui vérifie si la prochaine valeur en entrée est un entier valide.

Si l'entrée est correcte, un saut conditionnel est effectué vers le label `validInput` à l'aide de l'instruction `IFNE`, permettant de poursuivre normalement la lecture. En cas d'entrée invalide, un bloc de gestion d'erreurs est exécuté. Un nouvel objet `RuntimeException` est instancié via les instructions `NEW` et `DUP`, un message d'erreur est chargé sur la pile à l'aide de `LDC`, puis le constructeur de l'exception est invoqué via `INVOKESPECIAL`. L'exception est levée avec l'instruction `ATHROW`, interrompant l'exécution si l'entrée n'est pas valide. Si la valeur est correcte, la méthode `nextInt` est appelée sur l'objet `Scanner` à l'aide de `INVOKEVIRTUAL`, ce qui permet

de récupérer l'entier saisi et de l'empiler sur la pile pour une utilisation ultérieure.

```
@Override
protected void codeByteExp(MethodVisitor mv, DecacCompiler
    compiler) {
    mv.visitTypeInsn(Opcodes.NEW, "java/util/Scanner");
    mv.visitInsn(Opcodes.DUP);
    mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System", "
        in", "Ljava/io/InputStream;");
    mv.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/util/
        Scanner", "<init>", "(Ljava/io/InputStream;)V", false)
        ;

    mv.visitInsn(Opcodes.DUP);
    mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/util/
        Scanner", "hasNextInt", "()Z", false);
    org.objectweb.asm.Label validInput = new org.objectweb.asm
        .Label();
    mv.visitJumpInsn(Opcodes.IFNE, validInput);

    mv.visitTypeInsn(Opcodes.NEW, "java/lang/RuntimeException"
        );
    mv.visitInsn(Opcodes.DUP);
    mv.visitLdcInsn("Invalid integer input.");
    mv.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/
        RuntimeException", "<init>", "(Ljava/lang/String;)V",
        false);
    mv.visitInsn(Opcodes.ATHROW);

    mv.visitLabel(validInput);
    mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/util/
        Scanner", "nextInt", "()I", false);
}
```

Listing 13: Génération du bytecode pour la lecture d'un entier

4 Extension pour Deca Objet

4.1 Génération du Bytecode pour les Classes Deca Objet

La méthode `codeGenByteClass(DecacCompiler compiler)` est responsable de la génération du bytecode d'une classe en utilisant la bibliothèque ASM. Un objet `ClassWriter` est d'abord créé avec les options `COMPUTE_FRAMES` et `COMPUTE_MAXS`, permettant à ASM de calculer automatiquement les frames de la pile ainsi que les tailles maximales des piles locales et de l'operand stack, simplifiant ainsi le processus de génération.

Ensuite, les noms internes de la classe et de sa super-classe sont définis. Ces noms suivent la convention de la JVM, en remplaçant les points par

des barres obliques (/). Un traitement spécifique est appliqué pour la classe `Object`, dont le nom est converti en `"java/lang/Object"`.

4.2 Génération des Champs, du Constructeur et des Méthodes

Après l'initialisation de la classe, les champs sont générés via :

```
declFields.codeGenByteFields(cw, compiler, classInternalName);
```

Listing 14: Génération des champs

Cette méthode génère le bytecode nécessaire à leur déclaration.

Le constructeur par défaut est généré grâce à la méthode `generateDefaultConstructor`, qui appelle explicitement celui de la super-classe et initialise les champs de la classe avec :

```
declFields.codeGenByteFieldsInit(mv, compiler,
    classInternalName);
```

Listing 15: Initialisation des champs

Cela garantit que l'héritage est respecté et que l'ordre d'initialisation est conforme aux règles de la JVM.

Les méthodes de la classe sont ajoutées via :

```
declMethods.codeGenByteDeclMethods(cw, compiler, name.
    getClassDefinition());
```

Listing 16: Ajout des méthodes

Une fois la classe générée, la méthode suivante finalise l'écriture du bytecode et le récupère sous forme de tableau d'octets :

```
cw.visitEnd();
byte[] bytecode = cw.toByteArray();
```

Listing 17: Finalisation de la classe

4.3 Génération du Constructeur par Défaut

La méthode privée `generateDefaultConstructor` est dédiée à la création d'un constructeur par défaut. Un objet `MethodVisitor` est utilisé pour générer le code du constructeur en suivant ces étapes :

```
MethodVisitor mv = cw.visitMethod(Opcodes.ACC_PUBLIC, "<init>"
    , "()V", null, null);

mv.visitCode();
mv.visitVarInsn(Opcodes.ALOAD, 0);
```

```
mv.visitMethodInsn(Opcodes.INVOKESPECIAL, superInternalName, "
    <init>", "()V", false);
declFields.codeGenByteFieldsInit(mv, compiler,
    classInternalName);
mv.visitInsn(Opcodes.RETURN);
mv.visitMaxs(2, 1);
mv.visitEnd();
```

Listing 18: Génération du constructeur par défaut

Le constructeur commence par charger l'instance actuelle (`this`) sur la pile à l'aide de l'instruction `ALOAD`. Ensuite, il appelle le constructeur de la super-classe avec `INVOKESPECIAL`, puis procède à l'initialisation des champs de la classe. Enfin, la méthode retourne avec l'instruction `RETURN`, et les limites maximales de la pile et des variables locales sont définies avant la finalisation.

Génération du Bytecode pour l'Initialisation des Champs de Classe

La méthode `codeGenByteFieldInit` est responsable de l'initialisation d'un champ d'une classe. Elle commence par charger l'instance actuelle de l'objet avec l'instruction `ALOAD 0`, correspondant au mot-clé `this`. Ensuite, elle détermine le type du champ ainsi que son descripteur JVM à partir des méthodes `type.getType()` et `toJVMDescriptor()`.

4.3.1 Code de Génération de l'Initialisation des Champs

```
public void codeGenByteFieldInit(MethodVisitor mv,
    DecacCompiler compiler, String classInternalName) {
    mv.visitVarInsn(Opcodes.ALOAD, 0);

    Type trueType = type.getType();
    String fieldName = name.getName().toString();
    String fieldDesc = trueType.toJVMDescriptor();

    if (init.isNoInitialization()) {
        if (trueType.isClass()) {
            mv.visitInsn(Opcodes.ACONST_NULL);
        } else if (trueType.isInt() || trueType.isBoolean()) {
            mv.visitInsn(Opcodes.ICONST_0);
        } else if (trueType.isFloat()) {
            mv.visitInsn(Opcodes.FCONST_0);
        } else {
            throw new UnsupportedOperationException("
                Unsupported field type: " + trueType);
        }
    } else {
        int localIndex = compiler.allocateLocalIndex();
        init.codeGenByteInitialization(mv, localIndex,
            compiler);

        if (trueType.isInt() || trueType.isBoolean()) {
            mv.visitVarInsn(Opcodes.ILOAD, localIndex);
        } else if (trueType.isFloat()) {
```

```
25              mv.visitVarInsn(Opcodes.FLOAD, localIndex);
26          } else if (trueType.isClass()) {
27              mv.visitVarInsn(Opcodes.ALOAD, localIndex);
28          } else {
29              throw new UnsupportedOperationException("
                    Unsupported field type: " + trueType);
30          }
31      }
32
33      mv.visitFieldInsn(Opcodes.PUTFIELD, classInternalName,
            fieldName, fieldDesc);
34  }
```

Listing 19: Initialisation des champs de classe

Si aucune initialisation explicite n'est fournie, la méthode attribue une valeur par défaut en fonction du type du champ :

- **null** pour les objets (`ACONST_NULL`),
- **0** pour les entiers et les booléens (`ICONST_0`),
- **0.0** pour les flottants (`FCONST_0`).

Si une initialisation est spécifiée, celle-ci est générée et stockée dans une variable locale, puis rechargée à l'aide d'instructions adaptées (`ILOAD` pour les entiers et booléens, `FLOAD` pour les flottants, et `ALOAD` pour les objets). Enfin, l'instruction `PUTFIELD` est utilisée pour stocker la valeur dans le champ de la classe en utilisant le nom interne de la classe, le nom du champ et son descripteur JVM.

4.3.2 Conversion des Types en Descripteurs JVM

La méthode `typeToJVMDescriptor` est une fonction utilitaire permettant de convertir un type de données abstrait en son équivalent sous forme de descripteur JVM. Elle analyse le type de la donnée et retourne une chaîne de caractères correspondant à sa représentation dans la JVM.

```
1   public static String typeToJVMDescriptor(Type t) {
2       if (t.isInt()) {
3           return "I";
4       } else if (t.isFloat()) {
5           return "F";
6       } else if (t.isBoolean()) {
7           return "Z";
8       } else if (t.isClass()) {
9           String internalName = t.getName().toString().replace('
                .', '/');
10          return "L" + internalName + ";";
11      } else if (t.isVoid()) {
12          return "V";
13      }
14      throw new UnsupportedOperationException("Unsupported Deca
            Type for JVM descriptor: " + t);
```

```
15 }
```

Listing 20: Conversion des types en descripteurs JVM

Cette méthode retourne les descripteurs JVM suivants selon le type de la donnée :

- **I** pour un entier (`int`),
- **F** pour un flottant (`float`),
- **Z** pour un booléen (`boolean`),
- **V** pour le type `void`,
- Pour les objets, le nom de la classe est transformé en notation interne de la JVM, remplaçant les points par des barres obliques et encadré par les caractères `L` et `;`.

4.4 Génération du Bytecode pour la Déclaration des Méthodes

La méthode `codeGenByteDeclMethod` est responsable de la génération du bytecode d'une méthode d'une classe en utilisant ASM. Elle commence par récupérer le nom de la méthode et génère sa signature JVM à l'aide de la méthode `buildMethodDescriptor`, qui construit la description des paramètres et du type de retour. Ensuite, les modificateurs d'accès sont définis en utilisant l'opcode `ACC_PUBLIC`, indiquant que la méthode est publique.

Un objet `MethodVisitor` est ensuite créé via l'appel à `visitMethod`, ce qui permet d'ajouter les instructions de bytecode dans la méthode. Après la création de l'objet `MethodVisitor`, les paramètres sont initialisés à l'aide de la méthode `params.codeGenByteParamsInit(mv, compiler)`. Une fois les paramètres gérés, la méthode génère le corps de la méthode en appelant `body.codeGenByteMethodBody(mv, compiler, type.getType())`.

Une gestion explicite du retour est mise en place en fonction du type de retour de la méthode. Si la méthode ne retourne pas `void`, une exception est levée avec les instructions suivantes :

```
1 mv.visitInsn(Opcodes.ACONST_NULL);
2 mv.visitInsn(Opcodes.ATHROW);
```

Si la méthode retourne `void`, une instruction standard de retour est insérée avec :

```
1 mv.visitInsn(Opcodes.RETURN);
```

Enfin, les limites maximales de la pile et des variables locales sont définies via `mv.visitMaxs(0, 0)`, et la méthode est finalisée par `mv.visitEnd()`.

```java
@Override
protected void codeGenByteDeclMethod(ClassWriter cw,
    DecacCompiler compiler, ClassDefinition currentClass) {
    String methodName = name.getName().toString();
    String desc = buildMethodDescriptor(this.params, this.type
        .getType());
    int access = Opcodes.ACC_PUBLIC;

    MethodVisitor mv = cw.visitMethod(
        access,
        methodName,
        desc,
        null,
        null
    );
    mv.visitCode();

    params.codeGenByteParamsInit(mv, compiler);
    body.codeGenByteMethodBody(mv, compiler, type.getType());

    if (!type.getType().isVoid()) {
        mv.visitInsn(Opcodes.ACONST_NULL);
        mv.visitInsn(Opcodes.ATHROW);
    } else {
        mv.visitInsn(Opcodes.RETURN);
    }

    mv.visitMaxs(0, 0);
    mv.visitEnd();
}
```

Listing 21: Déclaration d'une méthode

4.5 Génération du Bytecode pour l'Initialisation des Paramètres

La méthode `codeGenByteParamsInit` initialise les paramètres en commençant par définir l'index du premier paramètre à 1, car l'index 0 est réservé pour l'instance de l'objet (`this`). Deux labels, `startLabel` et `endLabel`, sont créés pour marquer le début et la fin de la portée des variables locales, insérés dans le bytecode via l'instruction `visitLabel`. La méthode parcourt ensuite la liste des paramètres, vérifie leur validité à l'aide de `verifyDeclParam`, et enregistre le type du paramètre dans une variable temporaire ; en cas d'erreur contextuelle, une exception est levée. Après validation du type, le nom et le descripteur JVM du paramètre sont extraits, et le paramètre est associé à son index dans la table des variables locales via `setIndexInLocalTable`. Le paramètre est déclaré dans le bytecode grâce à `visitLocalVariable`, en précisant son nom, son descripteur, ainsi que les labels délimitant sa portée. L'index est ensuite incrémenté à chaque itération pour traiter le paramètre suivant. Enfin, le label de fin est inséré pour compléter la portée des variables locales.

```java
public void codeGenByteParamsInit(MethodVisitor mv,
    DecacCompiler compiler) {
    int paramIndex = 1;

    org.objectweb.asm.Label startLabel = new org.objectweb.asm
        .Label();
    org.objectweb.asm.Label endLabel   = new org.objectweb.asm
        .Label();

    mv.visitLabel(startLabel);

    for (AbstractDeclParam param : getList()) {
        Type paramType;
        try {
            paramType = param.verifyDeclParam(compiler);
        } catch (ContextualError e) {
            throw new RuntimeException(e);
        }
        String paramName = param.getName().getName().toString
            ();
        String paramDescriptor = paramType.toJVMDescriptor();

        ParamDefinition pd = (ParamDefinition) param.getName()
            .getDefinition();
        pd.setIndexInLocalTable(paramIndex);

        mv.visitLocalVariable(
            paramName,
            paramDescriptor,
            null,
            startLabel,
            endLabel,
            paramIndex
        );

        paramIndex++;
    }

    mv.visitLabel(endLabel);
}
```

Listing 22: Initialisation des paramètres

4.6 Génération du Bytecode pour le Corps des Méthodes

La méthode `codeGenByteMethodBody` commence par appeler les méthodes `codeGenListDeclVarByte` et `codeGenListInstByte`, qui génèrent respectivement le bytecode pour la déclaration des variables locales et l'exécution des instructions de la méthode. Une fois ces instructions générées, la méthode gère la valeur de retour en fonction du type de retour. Si la méthode ne retourne pas `void`, une valeur de retour par défaut est insérée dans la pile selon le type de retour : pour un type entier ou booléen, la valeur `0` est placée sur la pile à l'aide de l'instruction `ICONST_0`, suivie

de l'instruction de retour `IRETURN`; pour un type flottant, l'instruction `FCONST_0` est suivie de `FRETURN`; et pour un type objet, l'instruction `ACONST_NULL` est suivie de `ARETURN`. Si la méthode est de type `void`, une simple instruction de retour `RETURN` est insérée. Cette approche assure que la méthode se termine correctement, évitant d'éventuelles erreurs d'exécution dues à l'absence d'une valeur de retour explicite.

```
@Override
public void codeGenByteMethodBody(MethodVisitor mv,
    DecacCompiler compiler, Type returnType) {
    variables.codeGenListDeclVarByte(mv, compiler);
    insts.codeGenListInstByte(mv, compiler);

    if (!returnType.isVoid()) {
        if (returnType.isInt() || returnType.isBoolean()) {
            mv.visitInsn(Opcodes.ICONST_0);
            mv.visitInsn(Opcodes.IRETURN);
        } else if (returnType.isFloat()) {
            mv.visitInsn(Opcodes.FCONST_0);
            mv.visitInsn(Opcodes.FRETURN);
        } else if (returnType.isClass()) {
            mv.visitInsn(Opcodes.ACONST_NULL);
            mv.visitInsn(Opcodes.ARETURN);
        }
    } else {
        mv.visitInsn(Opcodes.RETURN);
    }
}
```

Listing 23: Génération du bytecode pour le corps de la méthode

4.7 Les fonctions codeGenByteListXXX

Les méthodes `codeGenByteListXXX` servent principalement à parcourir une liste d'éléments et à appliquer la méthode `codeGenByteXXX` sur chacun d'entre eux. Elles ne réalisent pas de traitement spécifique, mais permettent de déléguer l'exécution de la génération du bytecode à chaque élément de la liste de manière systématique.

```
public void codeGenByteDeclMethods(ClassWriter cw,
    DecacCompiler compiler, ClassDefinition currentClass) {
    for (AbstractDeclMethod m : getList()) {
        m.codeGenByteDeclMethod(cw, compiler, currentClass);
    }
}
```

Listing 24: Génération du bytecode pour la liste des méthodes

4.8 Génération du Bytecode pour les Appels de Méthodes

La méthode `codeByteExp` génère le bytecode nécessaire pour un appel de méthode en plusieurs étapes. Elle commence par générer le bytecode de

l'opérande gauche, représentant l'objet sur lequel la méthode est appelée. Ensuite, elle parcourt la liste des arguments passés à la méthode et génère le bytecode correspondant à leur évaluation. Chaque argument est empilé dans l'ordre approprié afin qu'ils soient disponibles pour l'appel de la méthode conformément aux conventions d'appel de la JVM.

Une fois l'objet et les arguments empilés, la méthode récupère la définition de la méthode cible ainsi que la classe propriétaire à laquelle elle appartient. Le nom interne de cette classe est obtenu en utilisant la convention de la JVM qui remplace les points par des barres obliques afin de respecter la notation interne. La signature complète de la méthode est ensuite construite à l'aide de la méthode `buildMethodDescriptor` de la classe `DeclMethod`, en combinant les types des paramètres et le type de retour.

Enfin, l'instruction `INVOKEVIRTUAL` est insérée dans le bytecode. Cette instruction permet d'appeler la méthode sur l'objet cible en utilisant son nom interne, le nom de la méthode et sa signature, garantissant un appel correct dans la JVM.

```
1  @Override
2  protected void codeByteExp(MethodVisitor mv, DecacCompiler
       compiler) {
3      leftOperand.codeByteExp(mv, compiler);
4
5      for (AbstractExpr arg : rightOperand.getList()) {
6          arg.codeByteExp(mv, compiler);
7      }
8
9      MethodDefinition methodDef = methodName.
           getMethodDefinition();
10     String ownerInternalName = ((ClassType) leftOperand.
           getType()).getDefinition().getInternalName();
11     String descriptor = DeclMethod.buildMethodDescriptor(
           methodDef.getSignature(), methodDef.getType());
12
13     mv.visitMethodInsn(
14         Opcodes.INVOKEVIRTUAL,
15         ownerInternalName,
16         methodName.getName().toString(),
17         descriptor,
18         false
19     );
20 }
```

Listing 25: Génération du bytecode pour un appel de méthode

5 Extension pour Deca Complet

5.1 Génération du Bytecode pour les Opérations de Cast

La méthode `codeByteExp` suit un schéma standard en commençant par appeler `codeByteExp` sur l'expression source, ce qui permet de l'empiler sur la pile de la JVM. Ensuite, en fonction du type cible, différentes instructions de conversion sont appliquées. Par exemple, si l'on souhaite convertir un flottant en entier, l'instruction `F2I` est utilisée, et inversement, l'instruction `I2F` est insérée pour convertir un entier en flottant. Ces opérations garantissent une conversion correcte entre les types primitifs.

Pour les conversions de types objets, si le type cible est une classe et que l'expression source n'est pas de type `null`, un processus de cast standard est suivi. La valeur est d'abord dupliquée sur la pile à l'aide de l'instruction `DUP`, ce qui permet d'effectuer la vérification sans retirer l'élément. Ensuite, deux labels sont définis : `castSuccess` pour un cast réussi et `castFail` en cas d'échec. Si l'objet n'est pas `null`, l'exécution continue jusqu'au label de succès où l'instruction `CHECKCAST` est appliquée pour effectuer la conversion vers le type cible. Sinon, si l'objet est `null`, l'exécution passe au label d'échec où un message d'erreur est affiché via `System.out.println`. Pour éviter toute erreur d'exécution, la valeur en sommet de pile est supprimée à l'aide de l'instruction `POP`, et une valeur `null` est empilée à la place.

```
1  @Override
2  protected void codeByteExp(MethodVisitor mv, DecacCompiler
       compiler) {
3      expr.codeByteExp(mv, compiler);
4
5      if (type.getType().isInt()) {
6          if (expr.getType().isFloat()) {
7              mv.visitInsn(Opcodes.F2I);
8          }
9      } else if (type.getType().isFloat()) {
10         if (expr.getType().isInt()) {
11             mv.visitInsn(Opcodes.I2F);
12         }
13     } else if (type.getType().isClass() && !expr.getType().
           isNull()) {
14         mv.visitInsn(Opcodes.DUP);
15
16         org.objectweb.asm.Label castSuccess = new org.
               objectweb.asm.Label();
17         org.objectweb.asm.Label castFail = new org.objectweb.
               asm.Label();
18
19         mv.visitJumpInsn(Opcodes.IFNONNULL, castSuccess);
20
21         mv.visitLabel(castFail);
```

```
22          mv.visitFieldInsn(Opcodes.GETSTATIC, "java/lang/System
                ", "out", "Ljava/io/PrintStream;");
23          mv.visitLdcInsn("Error: Incompatible cast");
24          mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "java/io/
                PrintStream", "println", "(Ljava/lang/String;)V",
                false);
25          mv.visitInsn(Opcodes.POP);
26          mv.visitInsn(Opcodes.ACONST_NULL);
27
28          mv.visitLabel(castSuccess);
29          mv.visitTypeInsn(Opcodes.CHECKCAST, type.getName().
                getName().replace('.', '/'));
30      }
31  }
```

Listing 26: Génération du bytecode pour le cast

5.2 Génération du Bytecode pour l'Opération `instanceof`

La méthode `codeByteExp` génère le bytecode nécessaire pour vérifier si une expression est une instance d'un type spécifique en utilisant l'instruction `INSTANCEOF`. Deux labels, `isInstance` et `endLabel`, sont définis pour gérer le contrôle de flux. Comme pour toute génération d'expression, la première étape consiste à évaluer l'expression cible en appelant `codeByteExp` sur l'objet concerné. Ensuite, l'instruction `INSTANCEOF` est utilisée pour vérifier si l'expression est une instance d'une classe spécifique. Le nom interne de la classe cible est obtenu en remplaçant les points par des barres obliques afin de respecter la convention de la JVM.

L'instruction `INSTANCEOF` pousse sur la pile la valeur `1` si l'objet est une instance de la classe spécifiée, et `0` sinon. Une instruction conditionnelle `IFNE` est ensuite insérée pour vérifier si le résultat de `INSTANCEOF` est différent de zéro, indiquant que l'objet appartient au type attendu. Si c'est le cas, l'exécution saute au label `isInstance`, où la valeur `1` (vrai) est empilée sur la pile. Sinon, l'instruction `ICONST_0` est insérée, suivie d'un saut inconditionnel vers le label `endLabel`, garantissant que la valeur `0` (faux) est empilée sur la pile.

```
1   @Override
2   protected void codeByteExp(MethodVisitor mv, DecacCompiler
        compiler) {
3       org.objectweb.asm.Label isInstance = new org.objectweb.asm
            .Label();
4       org.objectweb.asm.Label endLabel = new org.objectweb.asm.
            Label();
5
6       expr.codeByteExp(mv, compiler);
7
8       String classInternalName = compType.getName().toString().
            replace('.', '/');
9       mv.visitTypeInsn(Opcodes.INSTANCEOF, classInternalName);
10
```

```
11      mv.visitJumpInsn(Opcodes.IFNE, isInstance);
12
13      mv.visitInsn(Opcodes.ICONST_0);
14      mv.visitJumpInsn(Opcodes.GOTO, endLabel);
15
16      mv.visitLabel(isInstance);
17      mv.visitInsn(Opcodes.ICONST_1);
18
19      mv.visitLabel(endLabel);
20  }
```

Listing 27: Génération du bytecode pour l'opération instanceof

6 Validation de la génération du ByteCode

Au début de notre processus de validation, nous avons envisagé de comparer directement le bytecode généré par notre extension ASM avec celui produit par `javac` à partir d'un fichier Java équivalent. Cette approche, bien que rigoureuse, s'est révélée très exhaustive et difficile à automatiser en raison de légères différences d'optimisation entre les outils de génération de bytecode.

Nous avons ensuite réalisé qu'une validation plus simple et efficace consiste à vérifier si le fichier `.class` généré peut être exécuté correctement par la JVM. En effet, si le bytecode est valide et exécutable sans erreur, nous pouvons être certains de sa conformité. Toutefois, pour garantir une couverture maximale, nous continuons d'exécuter des tests qui comparent le résultat attendu avec le résultat obtenu via l'exécution de la commande suivante :

```
java -cp path/vers/XXX.class XXX
```

Processus de validation

1. Génération du bytecode à partir du code Deca via notre extension ASM.
2. Exécution du fichier `.class` généré avec la JVM.
3. Comparaison du résultat obtenu avec le résultat attendu.
4. Tests supplémentaires pour des cas limites et des fonctionnalités avancées.

Exemple de test

```
1  java -jar my_compiler.jar example.deca
2  java example.class > actual_output.txt
3  diff expected_output.txt actual_output.txt
```

Listing 28: Exécution et validation du bytecode généré

Cette méthode permet d'assurer que le code généré produit bien le comportement attendu et est conforme aux spécifications de la JVM.

Pour automatiser cette validation, nous avons mis en place un script de test qui vérifie l'exécution correcte des fichiers générés :

```
test_decac_e() {
    prompt_strong "[decac -e]"
    find "src/test/deca/codegen/extension/" -type f -name '*.deca' | while read -r file; do
        if [ "$(basename "$file")" = "printCarriageReturn.deca" ]; then
            continue
        fi
        prompt "- decac -e $file"
        decac_moins_p=$(decac -e "$file")
        check_zero_status "$?" "ERREUR: decac -e a termine avec un status different de zero pour le fichier $file."
        check_no_output "$decac_moins_p" "ERREUR: decac -e a produit une sortie pour le fichier $file."
        check_no_error "$decac_moins_p" "ERREUR: decac -e a produit une erreur pour le fichier $file."
        check_java_execution "$file" "ERREUR: decac -e a produit une d compilation .class qui donne un r sultat diff rent sur le fichier $file."
        clean_temp_test_files "src/test/deca/codegen/"
    done

    success "SUCCESS: test_decac_e"
}CCESS: test_decac_e"
}
```

Listing 29: Script de test pour `decac -e`

Ce script permet d'effectuer une validation automatique en comparant les résultats attendus et obtenus tout en détectant les erreurs éventuelles au cours de l'exécution.

7 Résultats de la Validation de l'Extension

Les tests de validation montrent que l'extension développée est capable de générer du bytecode fonctionnel pour un large éventail d'opérations, y compris les opérations arithmétiques et logiques sans objets, ainsi que celles impliquant des objets. Les fonctionnalités avancées du langage Deca, telles que les conversions de type explicites (casts) et l'opérateur `instanceof`, sont également correctement prises en charge. Ces résultats attestent de la robustesse de l'extension et de sa conformité aux spécifications du langage Deca, offrant une gestion efficace des différents cas d'utilisation.

Afin de démontrer la capacité de l'extension à gérer des cas d'utilisation complexes et réels, nous avons pu développer un jeu de **morpion** entièrement

en Deca. Ce projet fait appel à plusieurs fonctionnalités du langage, telles que :

- La gestion des objets et des classes,
- L'utilisation des structures de contrôle (`if-else`, `while`),
- La manipulation variables, d'expressions booléenes, d'appels de méthodes...,
- L'affichage et la lecture d'entrée utilisateur via la console,

L'extension a permis de générer un fichier `.class` fonctionnel, exécutable sans erreur par la JVM, démontrant ainsi sa capacité à produire du bytecode valide et performant. Ce jeu du morpion a permis d'évaluer à travers un jeu ludique les capacités du compilateur à gérer des fonctionnalités avancées et à garantir un bon comportement lors de l'exécution.

Cependant, certains problèmes ont été identifiés et nécessitent des corrections pour assurer un fonctionnement complet et sans erreur. L'un d'entre eux réside dans la génération de bytecode pour la création d'une instance de la classe `Objet`. Lors de l'exécution, une erreur survient, indiquant un problème de génération du nom de fichier, où l'extension produit `ObjetClass.class` au lieu de `Objet.class`. Ce dysfonctionnement pourrait être lié à une mauvaise configuration des conventions de nommage lors de la création de l'instance.

Ces observations nous permettent d'affiner encore davantage l'extension afin de garantir une compatibilité et une conformité totales avec les spécifications du langage Deca et de la JVM.