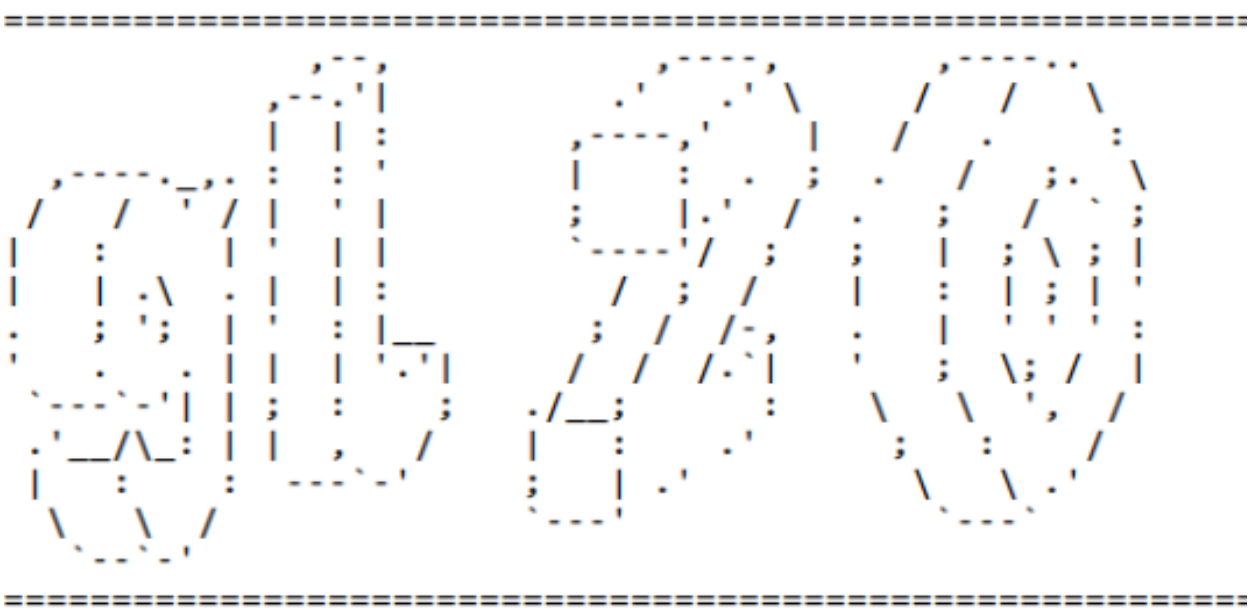
Documentation utilisateur du compilateur Deca
- Groupe GL20



Melvin Bonamour
Steve Lening
Mathieu Meunier
Paul Grassi
Emmanuel Loroux

Spécifications de l'Extension Bytecode pour le Compilateur deca vers Java bytecode

Objectif de l'Extension :

L'objectif final de l'extension aurait été de pouvoir compiler et générer du Bytecode Java pour des programmes du langage complet de Deca. Au vu des contraintes de temps, cet objectif à été restreint pour compiler et générer du Bytecode Java pour des programmes du langage Deca sans objets.
Utilisation simple de l'extension : decac -B
Compiler les programmes sur lesquels est utilisée la commande sous forme de bytecode Java permettant ainsi l'exécution sur n'importe quelle machine équipée de la machine virtuelle Java (JVM).

Langage de Programmation :

Pour l'implémentation de cette extension, le langage de programmation utilisé est le Java. Plus précisément pour la partie modification et génération de Bytecode Java, la bibliothèque ASM est utilisée (ASM ne signifie rien).

Structure du Bytecode :

L'extension bytecode doit suivre la structure standard des fichiers .class Java, incluant les en-têtes, les constantes, les méthodes, les attributs, etc.

Instructions Bytecode Gérées :

Les instructions prises en charge sont celles du langage Deca sans la partie objet. On se limite donc à ces instructions :

- Assignations
- Opérateurs de comparaison : {>,<,>=,<=, ==, !=}
- Print
- PrintLn
- And (&&)
- Or (||)
- DeclVar
- Equals
- NotEquals
- IfThenElse
- While
- Not
- ReadInt

Les instructions gérées incluent les opérations arithmétiques, les assignations, les comparaisons, les structures conditionnelles (if-then-else), les boucles (while), et les instructions de sortie (print).

Format de Fichier :

Le format du fichier doit être conforme aux spécifications des fichiers .class Java. Les sections du bytecode doivent être organisées de manière à permettre une interprétation correcte par la JVM.

Gestion des Erreurs :

Les erreurs de compilation doivent suivre la spécification de notre compilateur. Elles doivent être claires et précises concernant la nature de l'erreur et doivent indiquer leur localisation dans le programme Deca.
Les erreurs d'exécution seront gérées par la JVM pendant l'exécution du fichier **main.class** généré.

Interopérabilité :

L'extension bytecode doit garantir une compatibilité avec la JVM, permettant ainsi l'exécution sur différentes plates-formes sans modification du bytecode généré.

Optimisations :

Des optimisations peuvent être implémentées pour améliorer l'efficacité du bytecode généré, par exemple en éliminant les opérations redondantes ou en appliquant des techniques de compilation à la volée.

Sécurité :

Les mesures de sécurité standard pour les programmes Java doivent être respectées, et le bytecode généré doit être conforme aux restrictions de sécurité imposées par la JVM.

Exemple d'Utilisation :

Lors de la compilation d'un programme deca, le résultat devrait être un fichier .class en Bytecode Java utilisable sur n'importe quelle machine avec la JVM installée.

Implémentation :

L'implémentation de l'extension se base sur la librairie ASM (ASM ne signifie rien de particulier). D'autres librairies (notamment Javassist) auraient plus être utilisées mais la librairie ASM a été choisie en raison de sa vitesse. Cette librairie propose deux API : la Core API et la Tree API. L'implémentation de l'extension se base sur la Core API qui a l'avantage

d'être beaucoup plus rapide que la Tree API mais ne permet de ne générer qu'une ligne de bytecode à la fois et ne peut pas en supprimer alors que la Tree API présente une représentation sous forme d'arbre de chaque classes et méthodes générées par la librairie.
Le processus utilisé est le suivant : On instancie un objet de type **ClassWriter** lors de la génération de byteCode dans la classe Program. On instancie un autre objet, de type **TraceClassVisitor** qui servira à l'affichage de la classe générée sous une forme plus lisible (On génère une suite d'instruction de JVM (la liste des instructions peut être trouvée sur à cette adresse https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html). Le **TraceClassVisitor** produit une sortie qui n'est pas du bytecode à proprement parler, la sortie est semblable à ce que donnerait l'utilisation de la commande **javap -c** sur le fichier **.class** généré par le compilateur (Voir images suivantes).

```
{
    int x = 2024;
    int i = 10;
    if (x > 2023 && x<2025) {
        println("C'est bientôt la nouvelle Année");
    }
    else {
        print("KO");
    }
    while (i != 0) {
        println(i);
        i = i - 1;
    }
    println("Bonne Année");
}
```

Fig 1 : programme test.deca

```
// class version 59.0 (59)
// access flags 0x1
public class Main {


  // access flags 0x9
  public static I x

  // access flags 0x9
  public static I i

  // access flags 0x9
  public static main([Ljava/lang/String;)V
    LDC 2024
    PUTSTATIC Main.x : I
    LDC 10
    PUTSTATIC Main.i : I
    GETSTATIC Main.x : I
    LDC 2023
    IF_ICMPGT L0
    LDC 0
    GOTO L1
   L0
    ICONST_0
    POP
    LDC 1
    GOTO L1
   L1
```

Fig 2 : fichier “bytecode lisible” test_class

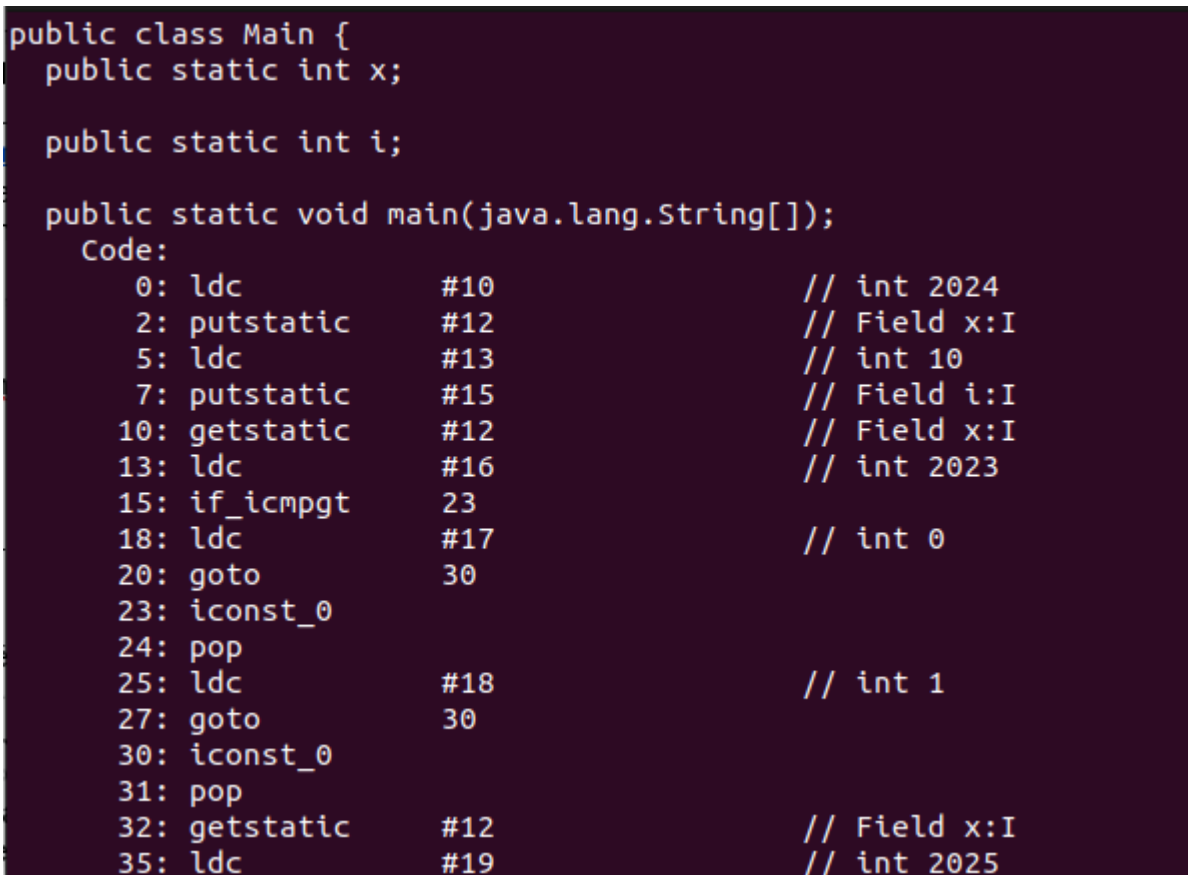


*Fig 3 : résultat de la commande **javap -c Main.class***

Après la création de ces deux fichiers on crée dans la classe **Main** (du compilateur Deca) un objet de type **MethodVisitor** pour parcourir la méthode **main** (**public static void main(String[] args)**) qui sera générée dans le fichier **.class** à la fin de l'exécution. Pour simplifier notre exécution, toutes les variables dans le programme Deca sont déclarées comme des variables statiques et globales de la classe **Main**. Des détails sur l'implémentation de ces objets (**ClassWriter, MethodVisitor** et **TraceClassVisitor**) peuvent être trouvés dans la documentation officielle de la librairie ASM.

Voici comment nos différentes classes ont été implémentées dans le cadre de l'extension Bytecode Java.

- **BooleanLiteral :** si la valeur est true on place 1 sur la pile avec "visitLdcInsn" sinon on place 0 sur la pile.
- **IntLiteral :** on place la valeur sur la pile avec "visitLdcInsn".
- **FloatLiteral :** on place la valeur sur la pile avec "visitLdcIsnsn"
- Identifier : on récupère la valeur d'un identifier et place sa valeur sur la pile avec "visitFieldInsn" et le code "GETSTATIC".
- **NoOperation :** on utilise "visitInsn(NOP)"
- **Program :** on instancie un "PrintWriter" et un "TraceClassVisitor" pour générer une classe on lance ensuite la génération de code du Main. On utilise ensuite "visitEnd" pour terminer la classe. On récupère ensuite la liste d'instruction Bytecode Java et on l'enregistre dans un fichier .class dans le même dossier que le fichier source.
- **Assignations :** instruction du type "x = 4", on place d'abord la valeur 4 dans la pile généralement en appelant la méthode "byteCodeGenInst" sur l'opérande de droite puis on la stocke dans l'identifier avec "visitFieldInSn" avec le code d'opération "PUTSTATIC".

- **Comparaison :** on instancie deux labels : “tru” et “end” , on utilise “byteCodeGenInst” pour récupérer la valeur de la condition sur la pile, puis on utilise “visitJumpInsn” avec le code IFGT et le label “tru” pour aller vers celui-ci si la condition est remplie.
  si non ensuite on place 0 sur la pile avec “ visitLdcInsn(0)” puis on va au label “end”.
  On place le label “tru” ici avec “visitLabel(tru)”. On place ensuite 1 sur la pile, puis on place le label end. On utilise enfin “visitInsn(F_SAME)” pour placer le résultat de la comparaison sur la pile.
  En résumé si la condition est remplie on place 1 sur la pile sinon on place 0.
- **Print :** On va chercher le paramètre out du système avec “visitFieldInsn” et le code “GETSTATIC”, on appelle “byteCodeGenInst” qui place la valeur du print sur la pile. Enfin on utilise “visitMethodInsn(INVOKEVIRTUAL, ..)”
- **And :** (byteCodeGenInst) on utilise “byteCodeGEnInst” sur les deux opérandes pour placer leur valeur sur la pile. On utilise ensuite “visitInsn(IAND)” pour placer le résultat du and sur la pile.
- **Or :** (byteCodeGenInst) on utilise “byteCodeGEnInst” sur les deux opérandes pour placer leur valeur sur la pile. On utilise ensuite “visitInsn(IOR)” pour placer le résultat du and sur la pile.
- **DeclVar :**à la première passe on instancie le champ avec “visitField(ACC_PUBLIC + ACC_STATIC, varName, …, null, null).visitEnd()”, puis à la passe 2 on place sur la pile la valeur de la variable “visistLdcInsn(var)” puis on on la stocke dans un champ “visitFieldInsn(PUTSATIC, …, varName, “I/F”)” I ou F si la variable est un flottant ou un entier.
- **Equals :** on instancie deux labels : “tru” et “end” , on utilise “visitJumpInsn” avec le code IF_ICMPEQ et le label “tru” pour aller vers celui-ci si les deux dernieres valeurs de la pile sont égales si l'opération concerne des entiers, sinon on utilise d’abord “visitInsn(FCMPL)” . si non ensuite on place 0 sur la pile avec “ visitLdcInsn(0)” puis on va au label “end”. On place le label “tru” ici avec “visitLabel(tru)”. On place ensuite 1 sur la pile, puis on place le label end. On utilise enfin “visitInsn(F_SAME)” pour placer le résultat de la comparaison sur la pile.
  En résumé si la condition est remplie, on place 1 sur la pile sinon on place 0.
- **NotEquals :** on instancie deux labels : “tru” et “end” , on utilise “visitJumpInsn” avec le code IF_ICMPNE et le label “tru” pour aller vers celui-ci si les deux dernieres valeurs de la pile ne sont pas égales si l'opération concerne des entiers, sinon on utilise d’abord “visitInsn(FCMPL)” . si non ensuite on place 0 sur la pile avec “ visitLdcInsn(0)” puis on va au label “end”. On place le label “tru” ici avec “visitLabel(tru)”. On place ensuite 1 sur la pile, puis on place le label end. On utilise enfin “visitInsn(F_SAME)” pour placer le résultat de la comparaison sur la pile.
  En résumé si la condition est remplie, on place 0 sur la pile sinon on place 1.
- **IfThenElse :** on instancie deux labels : “tru” et “end” , on utilise “visitJumpInsn” avec le code IFGT et le label “tru” pour aller vers celui-ci si la condition est remplie . Si non ensuite on appelle “byteCodeGenInst” sur la branche else, puis on va au label “end”. On place le label “tru” ici avec “visitLabel(tru)”. On appelle “byteCodeGenInst” sur la branche then, puis on place le label end. On utilise enfin “visitInsn(F_SAME)” pour placer le résultat de la comparaison sur la pile. On utilise ensuite “visitInsn(POP)” et “visitInsn(NOP)”.
  En résumé si la condition est remplie on va dans le code de la branche then sinon dans celui de la branche else.

- **While :** on instancie deux labels : “tru” et “end”. On place le label “tru” ici avec “visitLabel(tru)”. On appelle “byteCodeGenCond” pour récupérer le résultat de la condition sur la pile , on utilise “visitJumpInsn” avec le code IFGT et le label “end” pour aller vers celui-ci si la condition est remplie . Si non ensuite on appelle “byteCodeGenListInst” sur le corps du while, puis on va au label “tru”. Puis on place le label end. On utilise enfin “visitInsn(F_SAME)” pour placer le résultat de la comparaison sur la pile. On utilise ensuite “visitInsn(POP)” .
  En résumé si la condition est remplie on va directement a la fin de cette structure, sinon on exécute les instruction du corps du while jusqu’à ce que la condition soit remplie.
- **Main :** On instancie les variables à la première passe avec “byteCodeGenListDeclVar”. On instancie le MethodVisitor puis on rentre dans le code avec “visitCode” on fait ensuite la 2e passe sur les déclarations de variables ainsi que les instructions du main avec “byteCodeGenListInst”. Enfin on utilise “visitInsn(RETURN)”, “visitMaxs(13, 4)” et “visitEnd” pour finir le main.
- **Not :** si la valeur est true on place 0 sur la pile avec “visitLdcIsns” sinon on place 1.
- **ReadInt :** on récupère le champ in du système grâce à “visitFieldInsn(GETSTATIC, “java/lang/system”, “in”, …)” puis on récupère l’input stream avec “visitMethodInsn(INVOKEVIRTUAL, “.../InputStream”, “read”, …, …)”

Validation

Concernant la validation de l’extension, la majorité des tests ont été effectués à la main et un par un. Cependant, l’ensemble des tests créés qui concernent la partie sans objet du langage Deca ont été testés et passés avec succès.

Les programmes tests Deca implémentés :

Les programmes tests Deca implémentés sont les suivants:

- **entier2.deca :** on déclare 2 variable x et y qu’on initialise à 1 et 2. On déclare une 3e variable z. On assigne ensuite le résultat à z de 2*x-3*y. Enfin on affiche le résultat. Ce test permet de vérifier la génération de Bytecode Java pour les assignations.
- **maths.deca :** on teste différents aspect de la génération de code. L’affichage de flottants, de valeurs flottantes hexadécimales et le débordement arithmétique.
- **print-hexa.deca :** On teste l’affichage de flottant hexadécimaux.
- **structure.deca :** on teste la structure if avec une comparaison simple (x > 2023 && x < 2025 avec x initialisé à 2024) puis on teste les boucles while avec un indice i initialisé à 10 qu’on décrémente à chaque passe du while. On affiche cet indice à chaque passe.
- **jeu_plus_moins.deca :** test d’un programme plus complet qui teste plus en profondeur les structures if et while.
- **shifoumi.deca :** implémentation du célèbre jeu shifumi en deca pour tester les structures if et while ainsi que les comparaisons et les égalités.

Limitations :

L'option **-B** ne fonctionne que pour la partie sans objet du langage Deca.
L'option **-B** est incompatible avec l'utilisation de decac sur plusieurs fichiers. Il faut la lancer obligatoirement avec un seul argument sous peine de créer un comportement inattendu.

Liens utilisés :

Durant l'ensemble du développement deux liens ont quasiment été exclusivement utilisé :

- Le site officiel de la documentation ASM.
- La liste des instructions de la JVM.