UGA – Grenoble INP

Projet Génie Logiciel

Documentation du Bytecode

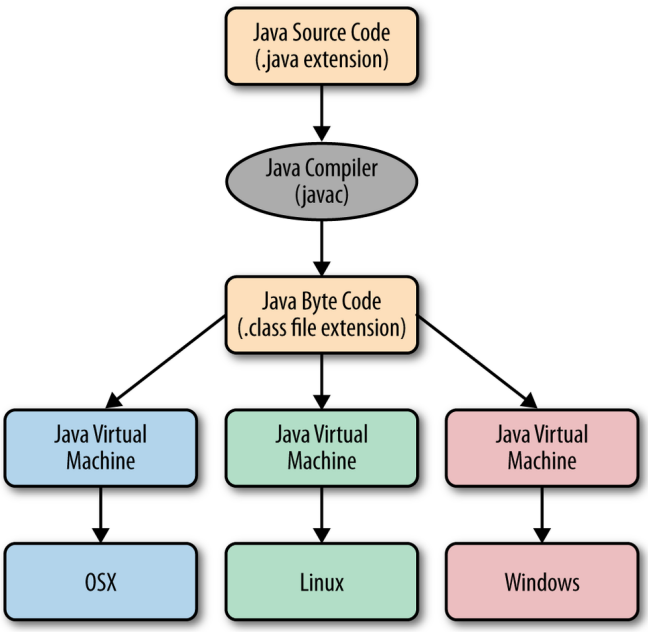
Grenoble, 26 janvier 2022.

Table des matières

1 Présentation de l'extension

1.1 Présentation du Byte Code

Un code Java est toujours compilé en premier lieu dans des fichiers *.class*. Ils sont "pré-compilés" dans ce type de fichier afin d'être interprétés par la machine virtuelle Java, qui exécutera éventuellement le programme. Les fichiers *.class* sont écrits en **Byte Code**, un langage que la machine virtuelle Java peut réellement analyser et exécuter. Ceci est illustré dans la figure suivante :



Le but de notre extension est de générer des fichiers Byte Code (*.class*) à partir de fichiers Deca (*.deca*), afin de les rendre exécutables par une machine virtuelle Java.

Cependant, pourquoi est-il intéressant de traduire un fichier Deca en Byte Code ? Ne suffit-il pas de générer les fichiers en assembleur (*.ass*) ?

1.2 Avantages du Byte Code

L'une des raisons les plus importantes de travailler avec du Byte Code est la portabilité du programme, car il utilise une machine virtuelle Java, et Java est en fait disponible sur tous les systèmes d'exploitation et appareils. La compilation des fichiers *.deca* en fichiers *.class* permettrait à tout utilisateur de notre compilateur de générer des programmes exécutables sur sa machine et en outre de les exécuter, les modifier et les partager sans prendre le risque de voir ses fichiers non pris en charge par une machine.

De plus, Java est en fait beaucoup plus utilisé que l'assembleur, et beaucoup de gens maîtrisent Java et ses spécificités plutôt que l'assemblage qui est beaucoup moins courant.

Enfin, il y a une raison environnementale pour laquelle la compilation en Byte Code serait une excellente solution, plutôt qu'en assembleur. Java étant un langage "international", ce qui signifie qu'un programme Java est totalement indépendant de la plate-forme qui exécute le programme, le code généré en Byte Code est en fait exécutable sur chaque machine dans laquelle Java est installé, ce qui signifie par exemple que le code généré peut être exécuté sur une machine Linux 64 bits, une machine Windows 32 bits et une machine Windows 64 bits, alors que l'assembleur dépend de la machine utilisée. Cela signifie que, dans notre exemple, un seul code doit être généré en Byte Code au lieu de trois codes assembleurs différents.

1.3 Notre choix pour le Byte Code

Comme il existe plusieurs bibliothèques et outils pour générer et manipuler le Byte Code, nous avons finalement choisi d'utiliser **Jasmin**, qui peut être téléchargé à partir du lien suivant :

http://jasmin.sourceforge.net

Jasmin est un assembleur pour la machine virtuelle Java. Il prend des descriptions ASCII de classes Java, écrites dans une syntaxe simple de type assembleur utilisant le jeu d'instructions de la machine virtuelle Java. Il les convertit en fichiers binaires de classes Java, qui peuvent être chargés par un système d'exécution Java. (MEYER, 2004)

Fondamentalement, notre compilateur génère des fichiers *.j*, ce qui signifie qu'ils sont écrits en Jasmin, puis les fichiers en question peuvent être compilés à l'aide de Jasmin en fichiers .class, en utilisant la ligne de commande suivante :

```
java -jar jasmin.jar myfile.j
```

1.4 Installer Jasmin

Pour compiler des programmes en fichiers *.class*, il faut d'abord installer Jasmin à partir du site web (http://jasmin.sourceforge.net), puis suivre le lien de téléchargement présent sur la page.

Ensuite, il suffit de générer les fichiers *.class* au niveau du répertoire Jasmin en tapant la ligne de commande ci-dessus.

2 Implémentation de l'extension

2.1 Les commandes Jasmin

Les commandes Jasmin sont très ressemblantes à la syntaxe Byte Code, à quelques détails près. L'ensemble des instructions de Jasmin est spécifié dans le site web de l'outil dans la section *Jasmin Instructions*, acccessible directement depuis le lien suivant :

http://jasmin.sourceforge.net/instructions.html

On remarque alors que les instructions ressemblent fortement à ce qui se fait en terme de langage assembleur, ce qui est plutôt intéressant au niveau de l'implémentation car cela nous permettrait de reprendre la même arborescence qu'avec le compilateur en assembleur ; ceci sera élaboré en détails dans la suite de ce document.

2.2 Exemple de code Jasmin

Voici un exemple de code Jasmin, celui du bien célèbre "HelloWorld". Nous nous basons sur ce programme pour implémenter les classes et méthodes nécessaires pour générer le fichier .class, puis nous étendrons le résultat à d'autres programmes comprenant d'autres types d'objets :

```
; --- Copyright Jonathan Meyer 1996. All rights reserved. -----------------
; File:      jasmin/examples/HelloWorld.j
; Author:    Jonathan Meyer, 10 July 1996
; Purpose:   Prints out "Hello World!"
; ------------------------------------------------------------------------

.class public HelloWorld
.super java/lang/Object

;
; standard initializer
.method public <init>()V
   aload_0

   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

.method public static main([Ljava/lang/String;)V
   .limit stack 2

   ; push System.out onto the stack
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; push a string onto the stack
   ldc "Hello World!"

   ; call the PrintStream.println() method.
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; done
   return
.end method
```

2.3 Implémentation des classes

Comme spécifié ci-avant dans la section 2.1, nous avons exploité l'arborescence des classes définies dans les fichiers **deca/tree** ainsi que les fichiers **deca/ima.pseudocode**. Cela nous a permis de nous affranchir d'une duplication de ces classes pour générer nos fichiers Jasmin. Les fichiers *DecacCompiler* et *CompilerOptions* ont aussi été modifiés pour prendre en compte l'option de compilation en Jasmin.

En effet, pour compiler un fichier .deca en Jasmin, il suffit de taper la commande suivante dans un terminal :

decac -j /chemin/vers/le/fichier.deca

Ainsi, au niveau du fichier *CompilerOptions*, l'option *-j* a été ajoutée pour traiter le cas de la compilation en Bytecode.

Au niveau de l'arborescence *tree*, nous avons ajouté à plusieurs endroits des fonctions qui génèrent le code Jasmin. Elles sont donc notées **codeGenXYZJasmin** comme dans l'exemple ci-dessous. Cela correspond à la génération du code assembleur qui renvoie à l'étape C du projet :

```
@Override
public void codeGenProgram(DecacCompiler compiler) {
    compiler.addComment("Main program");
    main.codeGenMain(compiler);
    compiler.addInstruction(new HALT());
}

@Override
public void codeGenProgramJasmin(DecacCompiler compiler) {
    compiler.addComment("Main program");
    main.codeGenMainJasmin(compiler);
}
```

Enfin, au niveau de la classe DecacCompiler, un nouveau programme a été développé : il s'agit de *doGenerateBytecode*. Il permet de générer les fichiers Jasmin puis Bytecode. Il reprend la même architecture du programme *doGenerate* qui permet lui de générer les fichiers en assembleurs (extension .ass).

Deux fonctions auxiliaires ont été introduites au niveau de DecacCompiler : elles permettent respectivement de générer l'en-tête du fichier Jasmin et la fin de ce fichier. Il s'agit des fonctions **writeJasminStart** et **writeJasminEnd**. En effet, les en-têtes des fichiers Jasmin sont similaires, et finissent tous par les mêmes deux lignes. *(le détail de ces deux fonctions est donné dans la page suivante)*

2.3.1 La fonction writeJasminStart

Elle permet, comme spécifié ci-avant, de générer les 10 premières lignes communes de tout fichier Jasmin ; seul le nom de la classe à la première ligne varie. C'est pour cela que la fonction prend en argument un objet **className** de type *String*, et affiche la ligne suivante sur le fichier Jasmin généré :

.class public className

Voici la fonction **writeJasminStart** (qui se trouve dans la classe DecacCompiler) :

```
private static void writeJasminStart(PrintStream pS, String className) {
    pS.println(".class public " + className);
    pS.println(".super java/lang/Object");
    pS.println(".method public <init>()V");
    pS.println("aload_0");
    pS.println("invokenonvirtual java/lang/Object/<init>()V");
    pS.println("return");
    pS.println(".end method");
    pS.println(".method public static main([Ljava/lang/String;)V");
    pS.println(".limit stack 256"); // feel the magic in the air
    pS.println(".limit locals 256"); // allez allez allez allez
}
```

Comme spécifié auparavant, les 8 premières lignes du programme ci-dessus sont communes à tout fichier Jasmin, les deux lignes suivantes sont quant à elles spécifiques à chaque programme :

.limit stack n
.limit locals p

Quand nous nous penchons sur la définition de ces deux lignes, nous trouvons cela comme définition : la directive .limit stack n dans la méthode main fixe la taille de la pile d'opérandes de la méthode "main" à n. Cela signifie que nous pouvons pousser jusqu'à cinq éléments sur la pile d'opérandes de cette méthode pendant l'exécution de la méthode. Chaque méthode possède sa propre pile d'opérandes dans la JVM. ("CENG477", 2018)

La directive .limit locals p fixe à 100 le nombre de variables locales dans la méthode "main". Il est fixé à 1 par défaut. Il faut utiliser cette directive dans une méthode qui possède plus d'une variable locale. ("CENG477", 2018)

Pour être totalement francs, ces deux instructions nous semblent très difficiles à cerner et à manipuler ; c'est pour cela que nous avons opté pour une solution intermédiaire et un peu radicale. Nous avons établi une limite très large pour la pile et les variables en y associant la valeur 256, qui est, d'après nos recherches, une valeur très grande et quasiment inatteignable. Nous associons donc ces valeurs par défaut pour tous les programmes que notre compilateur génère en Jasmin. Il se peut toutefois qu'une erreur de type *Stack Overflow* soit générée causée par une pile trop grande ou un trop grand nombre de variables. Il est alors nécessaire d'ajuster manuellement ces valeurs dans la classe DecacCompiler, en augmentant ces valeurs. Mais cela nous semble trop peu plausible compte tenu du grand nombre de variables prévu (dans beaucoup de programmes .j que nous avons rencontrés, la limite de la pile ne dépasse pas la dizaine).

2.3.2 La fonction writeJasminEnd

Quant à la fonction **writeJasminEnd**, elle ne pose strictement aucun problème au niveau des fichiers générés. Elle permet simplement d'afficher les deux lignes suivantes à la fin de chaque fichier Jasmin généré (ces deux lignes sont communes à tous les fichiers .j) :

return
.end method

Cette fonction est écrite dans la classe DecacCompiler, la voici ci-dessous :

```
private static void writeJasminEnd(PrintStream pS) {
    pS.println("return");
    pS.println(".end method");
}
```

Les deux fonctions décrites dans ces deux sous-sections sont appelées respectivement **avant** et **après** l'appel de la fonction *codeGenProgramJasmin*, qui elle permet de générer l'ensemble des instructions Jasmin qui seront reproduites sur le fichier .j généré.

2.3.3 Classes introduites : instructions

Pour générer des fichiers Jasmin, il faut naturellement mettre en place des instructions Jasmin, tout comme c'est le cas avec la partie du projet en langage assembleur. Voici la liste exhaustive des classes introduites, qui sont présentes dans le dossier *ima.pseudocodes/instructions/jasmin* :

— **aload**, qui permet de charger une référence dans la pile depuis une variable locale ;

— **astore**, qui permet de stocker une référence dans une variable locale ;

— **bipush**, qui permet de pousser un octet dans la pile sous forme d'entier ;

— **f2l**, qui permet de convertir un nombre flottant de type *float* en un nombre flottant de type *long* ;

— **fadd**, qui permet de réaliser une addition de flottants ;

— **fdiv**, qui permet de réaliser une division de flottants ;

— **fload**, qui permet de charger un objet de type *float* dans la pile depuis une variable locale ;

— **fmul**, qui permet de réaliser une multiplication de flottants ;

— **fneg**, qui permet de renvoyer l'opposé d'un nombre passé en paramètre, de type *float* ;

— **fstore**, qui permet de stocker un objet de type *float* dans une variable locale ;

— **fsub**, qui permet de réaliser une soustraction de flottants ;

— **getstatic**, qui permet d'obtenir la valeur d'un champ statique d'une classe, où le champ est identifié par une référence de champ dans le *pool* de constantes ;

— **i2f**, qui permet de convertir un nombre entier de type *int* en un nombre flottant de type *float* ;

— **i2l**, qui permet de convertir un nombre entier de type *int* en un nombre flottant de type *long* ;

— **iadd**, qui permet de réaliser une addition d'entiers ;

— **idiv**, qui permet de réaliser une division d'entiers ;

— **ifeq**, qui permet de tester si l'opérande passée en paramètre est égale à 0, et se branche sur le registre associé le cas échéant ;

— **ifge**, qui permet de tester si l'opérande passée en paramètre est supérieure ou égale à 0, et se branche sur le registre associé le cas échéant ;

— **ifgt**, qui permet de tester si l'opérande passée en paramètre est supérieure strictement à 0, et se branche sur le registre associé le cas échéant ;

— **ifle**, qui permet de tester si l'opérande passée en paramètre est inférieure ou égale à 0, et se branche sur le registre associé le cas échéant ;

— **iflt**, qui permet de tester si l'opérande passée en paramètre est inférieure strictement à 0, et se branche sur le registre associé le cas échéant ;

— **ifne**, qui permet de tester si l'opérande passée en paramètre est différente de 0, et se branche sur le registre associé le cas échéant ;

— **iload**, qui permet de charger un objet de type *int* dans la pile depuis une variable locale ;

— **imul**, qui permet de réaliser une multiplication d'entiers ;

— **ineg**, qui permet de renvoyer l'opposé d'un entier ;

— **invokespecial**, qui permet d'invoquer une méthode d'instance sur l'objet *objectref* et place le résultat sur la pile (peut être void) ; la méthode est identifiée par l'index de la référence de la méthode dans le *pool* de constantes ;

— **invokestatic**, qui permet d'invoquer une méthode statique et placer le résultat sur la pile (peut être void) ; la méthode est identifiée par l'index de référence de la méthode dans le *pool* de constantes ;

— **invokevirtual**, qui permet d'invoquer une méthode virtuelle sur l'objet *objectref* et met le résultat sur la pile (peut être void) ; la méthode est identifiée par l'index de référence de la méthode dans le *pool* de constantes ;

— **irem**, qui permet de donner le reste entier d'une division d'entiers ;

— **istore**, qui permet de stocker un objet de type *int* dans une variable locale ;

— **isub**, qui permet de réaliser une soustraction d'entiers ;

— **lcmp**, qui permet de renvoyer 0 si les deux opérandes sont de type *long* sont égales, 1 si la première opérande est supérieure à la seconde, et -1 sinon ;

— **ldc**, qui permet de pousser une constante d'un *pool* de constantes (String, int, float, Class, java.lang.invoke.MethodType, java.lang.invoke.MethodHandle, ou une constante calculée dynamiquement) sur la pile ;

— **newI**, qui permet créer un nouvel objet de type identifié par la référence de la classe dans l'index du pool constant. Il est à souligner que l'instruction s'appelle initialement **new**, mais ce nom est interdit comme nom de classe Java, c'est pour cela que nous en avons modifié le nom ;

— **spaghetti**, qui permet d'aller à une autre instruction. Cette instruction permet d'afficher *"goto"* en ligne de commande. Cette classe a un nom assez atypique, premièrement pour la même raison que l'instruction **newI**, *goto* étant un nom interdit pour une classe Java, mais aussi en référence à un article de Dijkstra ("Go to instruction considered harmful", 1968).

2.3.4 Classes introduites : opérandes particulières

En plus des instructions Jasmin, nous avons introduit un certain nombre de classes au niveau de *ima.pseudocodes/jasmin*. Ce sont des opérandes particulières qui permettent de renvoyer des commandes spécifiques au Jasmin. Voici la liste exhaustive des classes introduites :

— **BinaryInstructionJasmin**, qui reprend exactement le même schéma que sa fonction homonyme en assembleur, et qui permet d'afficher l'opération suivie de ses deux opérandes. La seule différence entre les deux fonctions est que celle en assembleur affiche l'opération suivie des deux opérandes séparées par une virgule, et en Jamsin les deux opérandes sont simplement séparées par un espace.

— **Constant**, qui permet d'introduire un objet de type *constante*, pour les renvoyer dans les lignes de commande.

— **PrintInvoked**, cette classe permet de renvoyer la ligne de commande suivante :

"java/io/PrintStream/print" + suffix + "(Ljava/lang/String;)V"

Elle permet donc de générer l'instruction Jasmin associée aux fonctions *print* et *println*. Le paramètre suffixe dans la ligne de commande renvoie au *-ln*. Ainsi, s'il n'y a pas de suffixe, c'est la fonction *print* qui est appelée, et si le suffixe est -ln, c'est *println* qui l'est.

— **PrintStreamOp**, cette classe permet de renvoyer la ligne de commande suivante :

"Ljava/io/PrintStream;"

Elle permet donc de générer l'instruction Jasmin associée aux fonctions *PrintStream*.

— **StringValueOf**, cette classe permet de renvoyer la valeur d'un objet sous forme de chaîne de caractères. Elle renvoie ainsi, en fonction du type d'objet rencontré, la ligne de commande suivante :

"java/lang/String/valueOf(" + type + ")Ljava/lang/String;"

avec la variable locale *type* qui vaut "I" s'il s'agit d'un entier, "F" s'il s'agit d'un flottant. Seuls ces deux types d'objets ont été implémentés. Les autres types passés comme paramètre renvoient alors l'erreur suivante : "Type not supported".

— **SystemOut**, cette classe permet de renvoyer la ligne de commande suivante :

"java/lang/System/out"

Elle permet de générer cette instruction qui permet d'afficher des valeurs en sortie du programme.

— **VarID**, cette classe permet tout simplement d'afficher la valeur d'entiers sur la ligne de commande en sortie du programme.

3 Utiliser notre compilateur Jasmin

3.1 Ce que réalise notre compilateur

Notre compilateur Jasmin réalise une compilation de tous les programmes **sans objets**. En effet la partie avec objet est assez conséquente et ne pourra être réalisée dans le délai imparti. En particulier, notre compilateur permet de générer des fichiers .class pour l'ensemble des tests fournis, sauf ceux qui font intervenir des booléens (cf partie 3.2).

Ainsi, il est possible de réaliser les opérations suivantes avec notre compilateur :

— Les programmes Deca faisant intervenir des affichages sur le terminal (fonctions print et println) et des saisies de nombres par l'utilisateur (readInt et readFloat) ;

— Les programmes Deca avec des instanciations de nombres et chaînes de caractères ;

— Les programmes Deca avec des opérations arithmétiques sur des entiers ou des opérations mathématiques (division, multiplication, soustraction, addition, congruences) ;

— Les programmes Deca avec des opérations logiques (comparaisons entre nombres, vérifications de condition d'égalité, de supériorité ou d'infériorité) ;

— Les programmes Deca avec des opérations mathématiques sur des nombres de types différents, auquel cas les entiers sont convertis en flottants et les opérations sont effectués de la façon la plus naturelle possible ;

— Les programmes Deca faisant intervenir tous les points précédents. Il est possible de compiler un programme faisant intervenir des instanciations de variables, des opérations sur celles-ci, des affichages divers, des comparaisons ...

En résumé, le compilateur Jasmin permet de compiler quasiment tous les programmes de la partie sans objet. Il réalise à peu près tout ce que peut réaliser notre compilateur en langage assembleur pour la partie sans objet, à une exception près, examinées en sous-section 3.2.2.

3.2 Limites de notre compilateur

Notre compilateur, comme spécifié ci-haut, ne permet pas de générer du code Jasmin pour tout programme Deca, contrairement au compilateur en assembleur développé en parallèle. Il y a quelques limites bien définies pour notre compilateur. Nous examinerons dans la section 4 de ce polycopié quelques pistes pour réaliser les parties manquantes, et des pistes d'amélioration pour le code que nous avons rendu.

Il y a deux points importants au niveau des limites de notre compilateur, explorés dans les deux sous-sections suivantes :

3.2.1 Les programmes avec objets

Il a été spécifié que notre compilateur n'est pas complet. Cela est essentiellement dû au fait que notre programme ne peut pas compiler des programmes Deca faisant intervenir des objets.

En effet, pour réaliser la partie avec objet, il aurait fallu s'inspirer de la partie avec objet du compilateur en assembleur, puis l'adapter au compilateur Jasmin en introduisant les classes et instructions nécessaires.

Ainsi, tout programme avec objet ne pourra pas être compilé en Jasmin avec notre produit, et renverra une erreur.

3.2.2 Les booléens

Les booléens sont définis de manière très complexe et très fastidieuse. En assembleur par exemple, un booléen *false* correpond à 0, et *true* correspond à 1, mais ici la définition des booléens peut être faite de plusieurs manières :

— soit reprendre la notation de booléens utilisée en langage assembleur ;

— soit partir sur une nouvelle définition : 0 pour le booléen *false*, et tout autre entier correspondrait au booléen *true*.

Devant la difficulté et la complexité de la gestion des booléens, nous avons laissé de côté cet aspect du compilateur. Ainsi, il n'est pas possible de compiler le programme suivant en Jasmin :

```
int a = 5 ;
int b = 6 ;
if (a==5 && b==6)
```

Mais il est entièrement possible de compiler cette version là du programme :

```
int a = 5 ;
int b = 6 ;
if (a==5){ if (b==6) }
```

En effet, les instructions de comparaison de nombres sont totalement indépendantes des booléens. Il s'agit des fonctions ayant pour préfixe *if-* (la liste exhaustive est en section 2.3.3), et aucune d'elles ne manipule ni renvoie des booléens.

Il est donc possible de s'affranchir de l'usage des booléens dans notre compilateur, et il est théoriquement possible de compiler tout type de programme sans objet en code Jasmin. Mais il faut toutefois revoir la structure des programmes pour les adapter à notre gestion des booléens.

3.3 Quelques exemples de compilation de programmes

3.3.1 Retour du HelloWorld

Comme spécifié en section 2.2, le traditionnel programme HelloWorld est notre point de départ pour générer du Bytecode depuis une batterie de tests en Deca. Nous allons donc tester notre compilateur sur le programme hello.deca, qui se trouve dans src/test/deca/syntax/valid/provided. On lance dans le terminal la commande suivante :

decac -j src/test/deca/syntax/valid/provided/hello.deca

Voici le résultat de cette commande :

```
(base) MBP-de-Ayoub:Projet GL ayoubmiguil$ source env_settup.sh
(base) MBP-de-Ayoub:Projet GL ayoubmiguil$ decac -j src/test/deca/syntax/valid/provided/hello.deca
Generated: Hello.class
(base) MBP-de-Ayoub:Projet GL ayoubmiguil$
```

Le fichier .class a donc bien été généré. Il se trouve à côté du fichier source .deca.
Ensuite, pour exécuter le programme, il suffit de lancer la commande suivante dans le terminal :

java Hello

Ce qui donne :

```
(base) MBP-de-Ayoub:Projet GL ayoubmiguil$ java Hello
Hello
(base) MBP-de-Ayoub:Projet GL ayoubmiguil$
```

On peut donc voir que notre compilateur passe avec brio le tout premier test de tout langage de programmation qui se respecte. Il permet, en passant par du code Jasmin, de générer un fichier .class qui lui est exécutable sans problème par la Machine Virtuelle Java.

Cependant, on peut se demander où est passé le fichier *.j* . La bonne nouvelle c'est qu'il a bien été généré, car autrement le *.class* ne serait jamais généré par Jasmin. Nous allons donc chercher à quoi ressemble notre fichier *.j*. Pour cela, il suffit de debugger le programme en lançant la commande suivante dans un terminal :

```
decac -j -d -d chemin/vers/le/fichier.deca
```

En lançant cette instruction avec le fichier *hello.deca*, voici ce que renvoie le terminal :

```
DEBUG fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:335) - Generated jasmin assembly code:
; start main program
; Main program
        getstatic java/lang/System/out Ljava/io/PrintStream;
        astore 1
        new java/util/Scanner
        dup
        getstatic java/lang/System/in Ljava/io/InputStream;
        invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
        astore 2
; Beginning of main instructions
        aload 1
        ldc "Hello"
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; end main program

INFO  fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:340) - Output file assembly file is: /Users/a
id/provided/Hello.j
INFO  fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:348) - Writing jasmin assembler file ...
INFO  fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:353) - Compilation of /Users/ayoubmiguil/Des
lo.deca successful.
Generated: Hello.class
INFO  fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:357) - Conversion in class file successful.
INFO  fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:361) - .j file deleted successfully
```

On remarque alors que la syntaxe de Jasmin est bien omniprésente. Le programme *.j* ressemble alors fortement à celui présenté en section 2.2, la syntaxe est exactement la bonne, et on remarque qu'à la suite de la compilation en *.class*, le fichier *.j* est supprimé.

Nous allons explorer alors les résultats de nos compilations Jasmin pour d'autres programmes, différents du HelloWorld, et faisant intervenir d'autres types et fonctions.

3.3.2 Petit programme de calcul

Voici un petit programme de calcul, que vous pouvez retrouver dans l'emplacement suivant :

`src/test/deca/bytecode/valid/self/calcul.deca`

Voici le programme original en Deca :

```
// Description:
//    Programme vérifiant le bon fonctionnement d'un calcul
//
// Resultats:
//    22
//
// Historique:
//    cree le 06/01/2022

{
    int x = 10;
    x = x + 12;
    println(x);
}
```

Et voici le programme correspondant en Jasmin :

```
DEBUG fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:335) - Generated jasmin assembly code:
; start main program
; Main program
        getstatic java/lang/System/out Ljava/io/PrintStream;
        astore 1
        new java/util/Scanner
        dup
        getstatic java/lang/System/in Ljava/io/InputStream;
        invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
        astore 2
        ldc 10
        istore 3
; Beginning of main instructions
        ldc 12
        istore 4
        iload 3
        iload 4
        iadd
        istore 3
        iload 3
        istore 5
        aload 1
        iload 5
        invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; end main program
```

Il est possible de faire des mélanges d'opérations arithmétiques tout en les affichant sur la sortie standard. Cela renvoie au test *addition.deca*, mais par souci de commodité, nous n'afficherons pas le résultat (qui tient sur des dizaines de lignes). Cependant le lecteur est vivement encouragé à lancer cette commande sur le test en question pour voir ce que cela donne en Jasmin.

3.3.3 Boucle while

Voici le code Jasmin associé au test suivant :

`src/test/deca/bytecode/valid/self/while.deca`

```
; Main program
        getstatic java/lang/System/out Ljava/io/PrintStream;
        astore 1
        new java/util/Scanner
        dup
        getstatic java/lang/System/in Ljava/io/InputStream;
        invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
        astore 2
        ldc 1
        istore 3
; Beginning of main instructions
        goto condWhile.2
while.1:
        iload 3
        istore 4
        aload 1
        iload 4
        invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
        ldc 1
        istore 5
        iload 3
        iload 5
        iadd
        istore 3
condWhile.2:
        ldc 3
        istore 6
        iload 3
        i2l
        iload 6
        i2l
        lcmp
        ifle while.1
; end main program
```

3.3.4 Modulo

Voici le résultat en Jasmin du test modulo.deca, qui se trouve à cet emplacement :

`src/test/deca/bytecode/valid/self/modulo.deca`

```
DEBUG fr.ensimag.deca.DecacCompiler.doGenerateBytecode(DecacCompiler.java:335) - Generated jasmin assembly code:
; start main program
; Main program
        getstatic java/lang/System/out Ljava/io/PrintStream;
        astore 1
        new java/util/Scanner
        dup
        getstatic java/lang/System/in Ljava/io/InputStream;
        invokespecial java/util/Scanner/<init>(Ljava/io/InputStream;)V
        astore 2
; Beginning of main instructions
        aload 1
        ldc "modulo de 5%2 : "
        invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
        ldc 2
        istore 3
        ldc 5
        iload 3
        irem
        istore 4
        aload 1
        iload 4
        invokestatic java/lang/String/valueOf(I)Ljava/lang/String;
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
; end main program
```

3.4 Autres tests intéressants

Quoique tous les tests soient vraiment intéressants, certains méritent en particulier d'être lancés pour visualiser le code Jasmin et vérifier que la syntaxe a bien été mise en place, et que tous les programmes sans objets (et sans booléens) fonctionnent très bien. Nous sommes toujours dans l'emplacement suivant :

`src/test/deca/bytecode/valid/self`

Nous recommandons toutefois de lancer les deux tests suivants qui sont particulièrement intéressants : *sansObjet* et *ifThenElse*.

4 Pistes d'amélioration

4.1 Amélioration des classes

Notre compilateur n'étant pas parfait ni optimal, il est possible d'y apporter plusieurs améliorations qui le rendront certainement meilleur quant à ses performances.

En ce qui concerne les classes, il est possible d'y apporter des améliorations notamment au niveau des instructions. En Jasmin et en Bytecode en général, les fonctions arithmétiques sont préfixées par le type de l'opération. Par exemple l'instruction de l'addition d'entiers est *iadd*, l'instruction d'addition de flottants est *fadd* (en Bytecode pour sommer deux flottants à précision double il faut utiliser *dadd* et pour des flottants de type long il faut utiliser *ladd*). Ainsi, il est possible d'introduire une seule classe *add*, et préfixer l'instruction en fonction de l'objet manipulé.

Il en est de même pour plusieurs autres instructions, par exemple *store*, *load* ... Ainsi, mettre en place une seule classe "mère" réduirait considérablement le nombre de classes au niveau des instructions.

4.2 Amélioration du compilateur

Au niveau de l'amélioration du compilateur, il n'y a rien de spécial à revoir, à part les deux points capitaux qui ont été examinés en sections 3.2.1 et 3.2.2. De ce fait, notre compilateur serait complet en y introduisant les variables booléennes ainsi que la reconnaissance et la compilation de programmes avec objets.

4.3 Amélioration des packages

Il est possible d'améliorer la gestion des packages de notre compilateur Jasmin en rassemblant toutes les fonctionnalités de celui-ci dans un seul fichier source.

Il est donc possible de séparer les fonctions liées à la compilation en Jasmin des autres fonctions qui compilent en assembleur, et de toutes les rassembler dans de nouvelles classes propres au Jasmin, au côté des instructions Jasmin que nous avons introduites. Ainsi, le produit final serait tout simplement un compilateur en Jasmin et en Jasmin seul.

4.4 Pistes pour réaliser ce qui nous manque

Pour réaliser ce qui nous manque, d'après la section 3.2, il faudrait introduire la compilation de fichiers avec objet et surtout il faudrait introduire la gestion des booléens.

Nous proposons d'introduire les objets de type booléen en utilisant le formalisme suivant :

```
boolean false <-> int 0
boolean true <-> int i != 0
```

En utilisant ce formalisme, il est possible de réaliser les opérations de comparaison de booléens, qui sont omniprésentes dans les programmes que nous testons en général.

Enfin, pour mettre en place la compilation des programmes avec objet, il suffirait, tout comme pour la partie sans objets, de s'inspirer de toute l'arborescence du compilateur en langage assembleur, et de l'adapter (en adaptant notamment les fonctions de vérification *verifyXYZ* et les fonctions de génération de code *codeGenXYZ*).

Il est aussi nécessaire d'introduire les instructions spécifiques au code Jasmin concernant la programmation des objets, et de les rajouter à la batterie d'instructions déjà codées pour la partie sans objet.

Bibliographie

"CENG477". *JVM and Jasmin tutorial*. 2018. <https://saksagan.ceng.metu.edu.tr/courses/ceng444/link/jvm-cpm.html>. [Online ; accessed 23-January-2022].

MEYER, J. *Jasmin Home Page*. 2004. <http://jasmin.sourceforge.net/>. [Online ; accessed 19-January-2022].