Documentation de l'extension

L'extension choisie à l'unanimité dans le cadre de ce projet est: La génération de **ByteCode Java**. Il s'agit alors de générer des fichiers *.class* contenant le ByteCode à partir de fichiers source *.deca* en utilisant le travail effectué ex ante sur le compilateur *deca* et l'utilisation de librairie existante disponibles sur le réseau internet.

En premier lieu, il s'agirait peut-être d'éclaircir la notion de ByteCode Java grâce à des recherches documentées afin d'en saisir la substance. Ci-dessous est expliqué la génération de ByteCode Java à partir de programme *.java* et non *deca* comme le projet le veut, concernant le *deca* nous y reviendrons un peu plus tard.

Java est doté d'une spécification nodale puisque qu'il s'agit d'un langage avec une approche du type compilateur-interpréteur. Il y a une phase de compilation puis d'interprétation. Le fichier source *.java* est d'abord compilé par d'IDE en *.class* contenant le ByteCode. Ce dernier est alors soit exécuté par la JVM installé sur la machine (d'une certaine façon interprété) ou est à nouveau compilé en code machine pour être exécuté.

Ainsi, le ByteCode n'est pas stricto sensu du code machine (ce qu'est appelé *assembleur* dans le jargon des informaticiens) puisqu'il est indépendant de toute plateforme. Les instructions contenues dans le ByteCode ne sont spécifiques à aucun type de CPU.

De surplus, le ByteCode n'est pas du code source puisque l'homme est dans l'incapacité d'en saisir le sens.

Celà implique que le ByteCode est à la fois bas niveau et portable. Puisqu'il est plus proche du langage machine que le code source, il est exécuté plus rapidement.

Ici demeure cependant une question: Pourquoi l'existence d'une telle structure de compilation ? Nous pouvons d'abord avancer que cela facilite la distribution de programme sur internet. En effet, Java autorise les moteurs de recherche à posséder des interprètes qui exécutent des "Applets" (des petits programmes Java).

Les utilisateurs ne doivent alors pas recevoir le code source, compiler puis interpréter le code au sein du browser serait une surcharge inutile à

ce dernier ajoutée à une perte de confidentialité qui ouvrirait la porte à de possibles subversions de la part de certains utilisateurs.
D'un autre côté, les utilisateurs ne doivent pas recevoir de code exécutable car il est trop spécifique selon les CPU pris individuellement.

Ergo, le ByteCode semble être un compromis. Il n'est pas lisible par les humains ce qui garantit une confidentialité et reste tout de même proche du langage machine conduisant à une interprétation et exécution d'autant plus rapide.

L'approche de génération de ByteCode Java à partir d'un programme Deca s'appuie sur l'utilisation circonspecte des outils développés durant le projet en association avec ceux déjà existants. En effet, il semble peu judicieux de vouloir re-développer des outils déjà existants dont une reconstruction serait une grande perte de temps.
Il s'agit alors de convertir le code Deca en code Java puis de compiler le code obtenu en ByteCode. La première étape s'appuie sur les fonctions *decompile* développées durant le projet. En effet, un remaniement en fonctions *decompileToJava* permet de générer le code Java équivalent au code *deca* de programme source. Il est ici judicieux de constater une grande ressemblance dans la syntaxe des deux langages afin de simplifier le travail. Une fois le code Java obtenu, il suffit d'appeler le compilateur **javac** afin d'obtenir le ByteCode Java du fichier source Deca. Ce dernier processus est automatisé grâce à l'écriture de script .sh. Sont générés un unique *Main.class* si le fichier source *deca* ne comporte pas d'objet sinon est ajouté à ce dernier d'autre fichier .*class* constituant les class déclaré dans le fichier source *deca*.

Si l'on souhaite obtenir le ByteCode, le processus vient appeler cette méthode *decompileToJava*

```
prog.decompileToJava(new PrintStream(javaStream));
```

présente dans *DecaCompiler* qui va appeler récursivement au sein de l'arbre abstraite les fonctions *decompileToJava* des nœuds de l'arbre.

Pour un non-terminaux, par exemple *Main*, il y a appel récursif pour récupérer le code Java de la déclaration des variables et des instructions. Nous pouvons ajouter que puisque Java est légèrement

différent de *Deca*, il est nécessaire de rajouter des morceaux de code propre à Java, notamment le *public static void main(String[] args)* par exemple pour transformer le main *{ … }* de deca.

```
public void decompileToJava(IndentPrintStream s) {
      s.println("public class Main {");
      s.indent();
      .. . .
      .. . .
s     .println("public static void main(String[] args) {");
      s.indent();
      declVariables.decompileToJava(s);
      insts.decompileToJava(s);
      s.unindent();
      s.println("}");
      s.unindent();
      s.println("}");
}
```

Lorsque les appels récursif ont atteint la fin d'une branche, c'est-à-dire qu'il n'y a plus d'appels nécessaires, cette méthode renvoie les caractères en question: exemple pour la class *FloatLiteral* renvoyant la représentation Java d'un Float.

```
public void decompileToJava(IndentPrintStream s) {
   s.print(java.lang.Float.toHexString(value) + "f");
}
```

La validation de ByteCode est similaire à celle de la vérification du code Assembleur des programmes Deca. En effet, il suffit simplement d'exécuter sous Java le ByteCode obtenu et de constater si la sortie est bien celle attendue. (par exemple un print "ok").

Il demeure cependant quelques limites concernant le passage d'un programme Deca au bytecode Java associé. En effet, en Deca, il est possible d'écrire des instructions non atteignables (par exemple, "if (false)" { instruction; }), contrairement au Java. Pour régler ce problème, il faut rajouter, lors de la décompilation, à chaque classe, un attribut

BOOLEAN_VALUE, afin de transformer certains codes non atteignables ainsi :

Version initiale :

```
if (false) {
        instruction;
}
```

Version finale :

```
BOOLEAN_VALUE = false;
if (BOOLEAN_VALUE) {
        instruction;
}
```

Ceci rend la plupart des codes non atteignables compilables par Java. Cependant, cela ne concerne pas les instructions après un return dans une méthode, qui causent un problème lors de la génération du bytecode, et qu'il faut donc éviter.
Ce problème est très difficile à régler sans faire de nombreuses évaluations lors de la compilation (si le return est dans un "if", par exemple).

Par ailleurs, il est naturellement impossible d'intégrer au code à générer en bytecode une méthode écrite en assembleur ima.