Software Engineering Project

Deca compiler

BYTECODE Extension

Authors:
M. Benjamin ARGENSON
M. Jean-Baptiste LEFOUL
M. Thomas POUGET-ABADIE
M. Robinson PRADA MEJIA
M. Alexandre PROY

Supervisors:
Pr. Philippe BODIGLIO
Pr. Catherine ORIAT

Version of January 26, 2017

Contents

Abbreviations and acronyms

ima	*interpréteur de la machine abstraite*
Java VM	*Java Virtual Machine*
AST	*Abstract Syntax Tree*

Introduction

We have decided to implement the Bytecode extension for our Deca compiler. This document's purpose is to provide an explanation of how we have implemented it. We are hoping that after reading it you will fully understand how the extension works and how to use it correctly. Therefore we will walk you step by step through the thinking process that brought us here.

First we will define the specifications of the Bytecode extension along with its limitations. Then you will be able to find a literature review including an explanation of the Java Bytecode. In this section we will introduce the ASM library [Wikf] and discuss how we can apply it to our extension. The next section will deal with our implementation and architecture choices. The final two sections will be dedicated to the validation method and its results.

Chapter 1

Specifications

The extension must respect a certain number of characteristics :

- it allows the user of the Deca compiler to compile a *.deca* file and generate the corresponding Java bytecode *(.class file)*;
- it is possible to invoke the Java VM on the generated bytecode and it produces the same behavior and results as the ones produced by the *ima* program;
- the extension must cover the generation of bytecode for every Deca program that does not use objects;
- in order to validate the proper functioning of the extension its implementation must be accompanied by a validation method;
- the implementation of such an extension must be simple enough so that it does not disrupt the architecture of the whole project. More specifically it does not cause other parts of the program to malfunction;
- the extension must be scalable, that is it should be sufficiently documented and the implementation choices should allow anyone to carry on the implementation of new features;
- the extension must remain an option and the Deca compiler should only produce the bytecode resulting from the compilation when instructed so;

Chapter 2

Literature review

2.1 Java Bytecode

2.1.1 Definitions

When the Java compiler *(javac)* compiles a Java program it produces a Java class file *(.class file)*. This file contains a platform-neutral special type of bytecode called Java bytecode.

Bytecode [Wika] or portable code is a form of instruction set which was designed in order to allow efficient execution by a software interpreter. The name bytecode comes from the fact that the instructions have a one-byte opcode [Wikg]*(operation code that specifies the operation to be performed)* eventually followed by optional parameters. Bytecodes offer a compact representation of programs, they therefore allow better performance than direct interpretation of source code.

In our case, the Java bytecode [Wikb] is the instruction set of the JVM. Out of the 255 different byte-long opcodes, 198 are in use, 54 are reserved for future use and 3 are set aside and will never be implemented. We can easily see that the number of instructions is much larger than the number of instructions offered by the abstract machine we were using.

Once our initial Java program has been translated into its corresponding bytecode we can invoke the JVM [Wike]. What is important to remember here is that the JVM is an abstract virtual machine that enables a computer to run a Java program. With the JVM one can run Java bytecode regardless of one's computer's specificities.

2.1.2 Instruction set

In this part we will introduce the Java bytecode instruction set [Wikc] and its specificities.

Instructions fall under 7 different categories:

- Load and store;
- Arithmetic and logic;
- Type conversion;
- Object creation and manipulation;
- Operand stack management;
- Control transfer;
- Method invocation and return.

It has to be noted that most instructions within these categories have a prefix or a suffix referring to the type of operand they operate on.

Prefix/Suffix	Operand type
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
z	boolean
a	reference

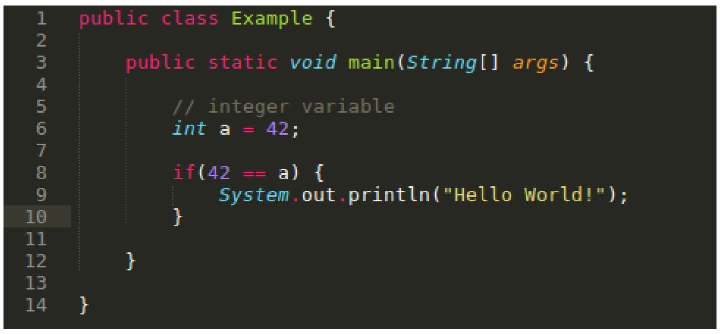
For instance, if you take the instruction *isub* it will subtract two integers whereas the instruction *fsub* will be used to subtract two floats. Other instructions will have a number as a suffix, this is the case for the instruction *iconst_4*. It will push onto the stack the constant integer value 4.

It is also important to note that most instructions will directly push the result of the operation they perform onto the stack. Similarly if they take two parameters they'll directly use the top two values on the stack. You do not have to push or pop values like you would have to do with *ima*. For example the *isub* instruction will take the top two values on the stack, perform the subtraction and then push the result onto the stack.

2.1.3 Examples

Example 1:

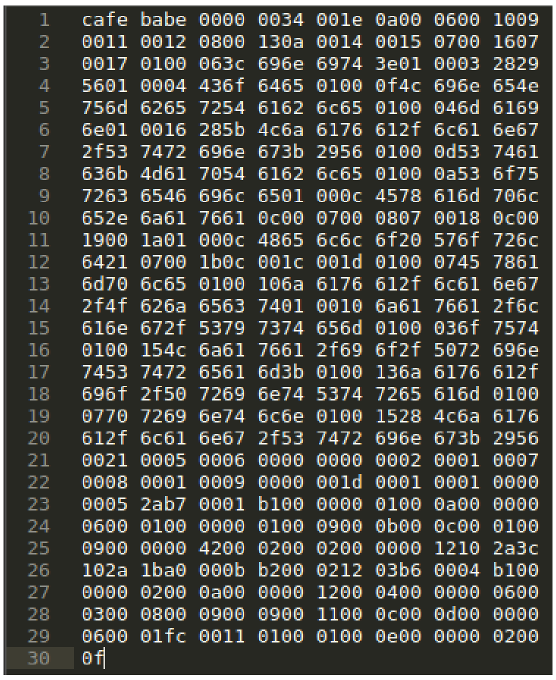
Here is an example of a simple Java program to demonstrate what has been said so far about the Java bytecode. The said program declares an integer variable *a*, initializes it with the value 42. Then it uses an *if* control structure. It evaluates whether *a* equals 42. If so, then it prints *Hello World!*.



```java
public class Example {

    public static void main(String[] args) {

        // integer variable
        int a = 42;

        if(42 == a) {
            System.out.println("Hello World!");
        }

    }

}
```

Figure 2.1: Our simple program Example.java

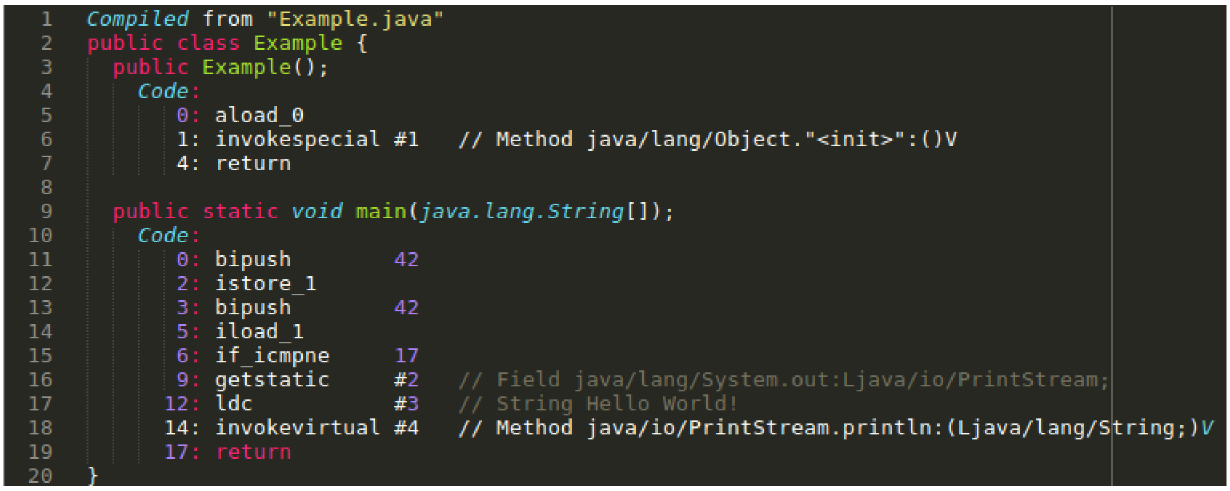
Now, if we compile this program with the following command line: *"javac ./Example.java"*, we obtain the following *Example.class* file.



```
cafe babe 0000 0034 001e 0a00 0600 1009
0011 0012 0800 130a 0014 0015 0700 1607
0017 0100 063c 696e 6974 3e01 0003 2829
5601 0004 436f 6465 0100 0f4c 696e 654e
756d 6265 7254 6162 6c65 0100 046d 6169
6e01 0016 285b 4c6a 6176 612f 6c61 6e67
2f53 7472 696e 673b 2956 0100 0d53 7461
636b 4d61 7054 6162 6c65 0100 0a53 6f75
7263 6546 696c 6501 000c 4578 616d 706c
652e 6a61 7661 0c00 0700 0807 0018 0c00
1900 1a01 000c 4865 6c6c 6f20 576f 726c
6421 0700 1b0c 001c 001d 0100 0745 7861
6d70 6c65 0100 106a 6176 612f 6c61 6e67
2f4f 626a 6563 7401 0010 6a61 7661 2f6c
616e 672f 5379 7374 656d 0100 036f 7574
0100 154c 6a61 7661 2f69 6f2f 5072 696e
7453 7472 6561 6d3b 0100 136a 6176 612f
696f 2f50 7269 6e74 5374 7265 616d 0100
0770 7269 6e74 6c6e 0100 1528 4c6a 6176
612f 6c61 6e67 2f53 7472 696e 673b 2956
0021 0005 0006 0000 0000 0002 0001 0007
0008 0001 0009 0000 001d 0001 0001 0000
0005 2ab7 0001 b100 0000 0100 0a00 0000
0600 0100 0000 0100 0900 0b00 0c00 0100
0900 0000 4200 0200 0200 0000 1210 2a3c
102a 1ba0 000b b200 0212 03b6 0004 b100
0000 0200 0a00 0000 1200 0400 0000 0600
0300 0800 0900 0900 1100 0c00 0d00 0000
0600 01fc 0011 0100 0100 0e00 0000 0200
0f
```

Figure 2.2: The corresponding Example.class

This is what the Java bytecode looks like. Every class file start with the same 4 bytes [Wikd], *0xCA 0xFE 0xBA 0xBE*, they serve as an identifier. Here you can see all the instructions needed to realize the program. Since it is in hexadecimal it is not very readable. Fortunately a special command line can be used to make it easier to read and to understand. Indeed using this command: *"javap -c ./Example"*, will get you the following result.

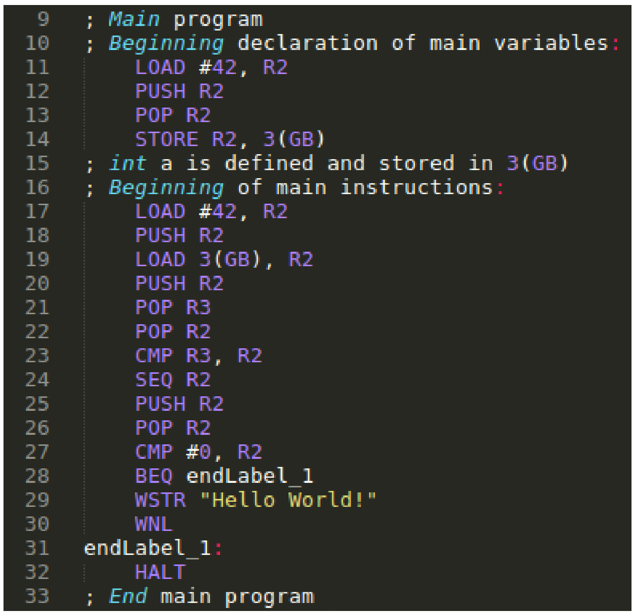


```
Compiled from "Example.java"
public class Example {
  public Example();
    Code:
       0: aload_0
       1: invokespecial #1   // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: bipush        42
       2: istore_1
       3: bipush        42
       5: iload_1
       6: if_icmpne     17
       9: getstatic     #2   // Field java/lang/System.out:Ljava/io/PrintStream;
      12: ldc           #3   // String Hello World!
      14: invokevirtual #4   // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      17: return
}
```

Figure 2.3: The revealed instructions

Here we can see all the instructions it took realize the previous program in the Java bytecode. An interesting thing to notice here is that even though we had not coded a constructor for our class *Example* one was automatically generated. Then there is the code associated to the *main* method. The value 42 is assigned to the first integer variable, instructions: *0: bipush* and *2: istore_1*. Then there is the comparison between the variable and 42, instructions: *3: bipush 42*, *5: iload_1* to get the value stored in the variable and *if_cmpne*. Finaly the next three instructions are used to print the string *Hello World!*. There is another version of the previous command line, *"javap -v ./Example"*, that gives more information. But for the moment the first command line is enough to understand how the Java bytecode works.

As we can see the instruction set used by the Java bytecode is a bit different from the one we were used to work with. But if we create a similar Deca program and we compile it, here is what we will find in the corresponding *.ass* file.



```
; Main program
; Beginning declaration of main variables:
    LOAD #42, R2
    PUSH R2
    POP R2
    STORE R2, 3(GB)
; int a is defined and stored in 3(GB)
; Beginning of main instructions:
    LOAD #42, R2
    PUSH R2
    LOAD 3(GB), R2
    PUSH R2
    POP R3
    POP R2
    CMP R3, R2
    SEQ R2
    PUSH R2
    POP R2
    CMP #0, R2
    BEQ endLabel_1
    WSTR "Hello World!"
    WNL
endLabel_1:
    HALT
; End main program
```

Figure 2.4: The corresponding Deca program compiled

The two versions produce exactly the same result and have the same behavior. An integer variable is declared and initialized with the value 42. Then there is the comparison. Even though the number of instructions used is not the same and their names are different, we cannot help but notice similarities. Values are pushed on the stack, variables are stored and comparison instructions exist.

The goal of this extension is to be able to generate a Java class file that looks a bit similar to the one found in figure 2.2 so that it can be run by the JVM.

2.2 ASM library

2.2.1 Definition

> The goal of the ASM library is to generate, transform and analyze compiled Java classes, represented as byte arrays (as they are stored on disk and loaded in the Java Virtual Machine). For this purpose ASM provides tools to read, write and transform such byte arrays by using higher level concepts than bytes, such as numeric constants, strings, Java identifiers, Java types, Java class structure elements, etc. Note that the scope of the ASM library is strictly limited to reading, writing, transforming and analyzing classes. In particular the class loading process is out of scope. (E. Bruneton 2007)

2.2.2 Motivations

For this project extension, the main task was to generate the bytecode starting from an AST Tree structure previously generated. For this purpose the OW2 Consortium project, ASM, was chosen between many other tools such as: BCEL, SERP or JOEI. This Java class manipulation tool is designed to dynamically generate and manipulate Java classes and whose features fit quite well with our need. Compared to the mentioned existing tools, ASM works with a different approach consisting in using the "VISITOR" design pattern avoiding the user the need of an explicit representation of the the visited tree with objects.

For our practical needs, features like the followings represent much better performances than those of other existing tools:

- A simple, user friendly modular API.
- A complete documentation and Internet examples.
- Support for the latest Java Version.
- A small and fast addition to the project files.
- Its open source license allows a variety of uses.

2.2.3 API

In this section the main use the core ASM API in the project is explained. Starting with a brief introduction and then presenting the corresponding ASM interfaces, components and tools to generate them, with an illustrative example.

Structures

As shown in the bytecode section, in the inside compiled classes (.class) files the code of methods is stored as a sequence of bytecode instructions starting by the predefined 4 bytes CA FE BA BE. This section gives the overview of these instructions which allow the user to start coding simple instructions. For a better global understanding it is recommended to read the Java Virtual Machine Specification.

Bytecode Instruction

For each bytecode instruction in the class file, there is an Opcode which identifies it and a fixed number of arguments.

- The **bytecode** is an unsigned byte value identified by a mnemonic symbol. The most simple example is the opcode value 0, whose mnemonic symbol is NOP and represents "perform no operation".
- The **arguments** are static values placed right after the opcode indicating each instruction behavior. A practical example is given: "For instance the GOTO label instruction, whose opcode value is 167, takes as argument label, a label that designates the next instruction to be executed" (Java bytecode engineering library 33).

Chapter 3

Design analysis

3.1 Implementation choices

Having that understanding of how the Java bytecode works and the ability to use the ASM library two options were possible to generate Java bytecode. The first one was to reuse the *.ass* file generated from the compiler to generate the corresponding *.class* file. The second option was to reuse only the A and B parts of our compiler, that is the lexing and parsing part and the contextual check part, and to create a second version of part C (code generation) dedicated to generating Java bytecode.

The first option was however quickly discarded. We deemed it useless to make our compiler do twice the part C, once to generate the assembly language and another time to generate the Java bytecode. Moreover it is an option of our compiler to choose the target language of the compilation, therefore there is no point to generate both results when the user is only interested in getting the Java bytecode.

We thus retained the second option. Indeed this option seemed more reasonable and more feasible. First it can be seen as another version of part C, which is exactly what it is, instead of creating a target file in assembly language we create it in Java bytecode. Then it appeared easier to us to reuse the architecture used during the code generation in part C. This way we were not disrupting the general behavior of the compiler and the implementation of the extension is not too heavy.

In order for you to understand our implementation we think it is better if we introduce the data structures and tools offered by the ASM library first therefore this is what you will discover in the next section.

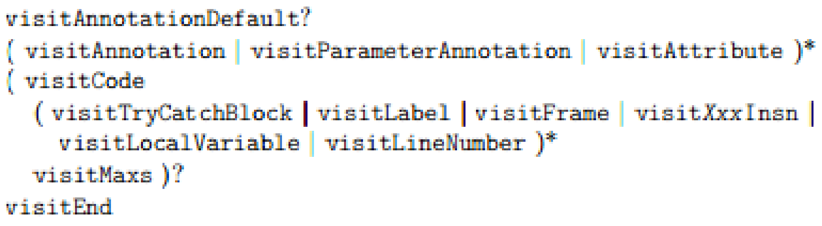
3.2 Data structures

The API offered by ASM for the generation and transformation of methods already compiled is directly related to the MethodVisitor class mentioned in the previous section. About this class, it is important to understand how each method belongs to a specific category depending on its bytecode instruction, the number and type of arguments.

The specific order to be followed when generating methods is the following:

- Annotations and attributes.
- A call to visitCode.
- Bytecode for the method.
- A call to visitMaxs.

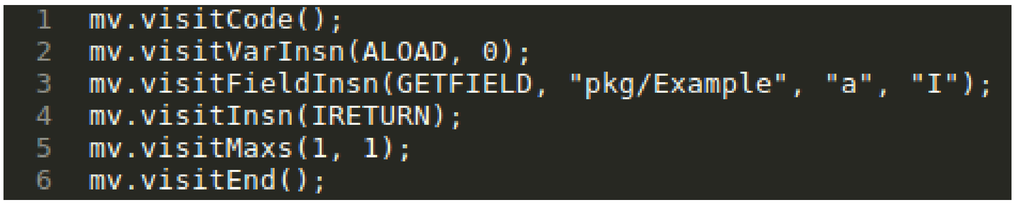
In this Bytecode extension, the visitCode method is the starting point for every bytecode generation followed by a sequence of instructions and finishing with a call to the visitMaxs method.



```
visitAnnotationDefault?
( visitAnnotation | visitParameterAnnotation | visitAttribute )*
( visitCode
  ( visitTryCatchBlock | visitLabel | visitFrame | visitXxxInsn |
    visitLocalVariable | visitLineNumber )*
  visitMaxs )?
visitEnd
```

Figure 3.1: Basic structure example (Taken from Java bytecode engineering library 41)

Following the Example 2 in section 2.1.3, the bytecode generation for the method getA() is composed of the following method calls from the MethodVisitor class:



```
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitFieldInsn(GETFIELD, "pkg/Example", "a", "I");
mv.visitInsn(IRETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();
```

Figure 3.2: Basic method structure example.

For the specific purpose of this project, the main common structures as the calls to visitCode, visitMaxs and visitEnd are predefined in one class so that it was not necessary to define them every time that a method was going to be implemented. This giving as a result a general data structure similar to the assembly one.

In order to transform methods, the ASM library provides three different main data structure components which are based on the MethodVisitor API:

- The **MethodVisitor** class is in charge of taking every method call and handing it to another instance of its same class.
- The **ClassWriter's visitMethod** method implements and returns the MethodVisitor interface in order to turn compiled methods into binary form.
- The **ClassReader** class parses the content of compiled methods and calls the corresponding methods on the MethodVisitor objects returned by the ClassVisitor passed as arguments to its accept method (Java bytecode engineering library 47).

3.2.1 ClassWriter

The ClassWriter [Brua] is a ClassVisitor which generates classes directly in byte arrays (bytecode form) which we are able to be read with the JVM. "More precisely this visitor generates a byte array conforming to the Java class file format". Even though the ClassWriter can be used alone to generate a very simple java class, for the bytecode generation of this project, classes are built including the ClassReader class and an adapter class visitor. Summarizing this idea, the ClassWriter class could be seen as an event consumer.

The methods included in this class allow the user to work directly with classes. Some of the most important ClassWriter's methods for this project are:

- **toByteArray**(), This method is used once a class has been built, and returns the bytecode corresponding to the class from the same instance of its ClassWriter.
- **visit**(int version, int access, String name, String signature, String superName, String[] interfaces). This method is used in the project to create classes.
- **visitField**(int access, String name, String desc, String signature, Object value). This method is used in the project to create fields, for instance for the creation of attributes.
- **visitMethod**(int access, String name, String desc, String signature, String[] exceptions). This method allows the user to visit/create specific methods of a class.

3.2.2 MethodVisitor

As the object returned by the ClassVisitor's visitMethod method, the MethodVisitor [Brub] abstract class can be used for the definition of attributes and methods' code. This important class also provides access to a variety of methods which allow the use of instructions similar to the assembly ones and generate the respective bytecode.

The methods included in this class allow the user to work directly with the stack and perform a variety of useful operations. Some of the most important MethodVisitor's methods for this project are:

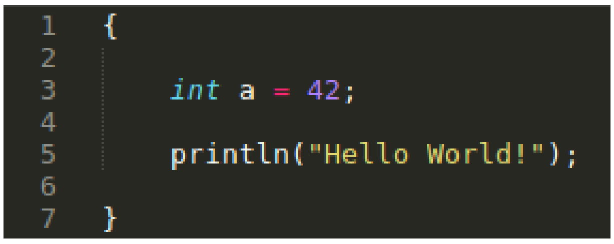
- **visitInsn**(Opcodes.X), this method allows the user to perform the most common operations directly with elements in the stack, pushing them back once the operation is done.

- **visitVarInsn**(Opcodes.X , Variable Id), this method is used in this project for the declaration and initialization of variables. Each variable keeps its own Id number (integer) which allows the user to STORE and LOAD values in and from the variable.
- **visitMethodInsn**(Opcodes.X, method's owner class, the method's name, the method's descriptor, true if interface). This method is used to visit java methods such as print and scan among others.
- **visitJumpInsn**(Opcodes.X(condition), Desired label for condition). This method allows the implementation of control structures as ifThenElse, and the while loop as well as comparison operations.
- **visitLabel**(Label label). This method is used to add a label to a specific needed point in the code, it also makes possible the implementation of control structures and comparison operations.

3.3 Architecture

You will find that the architecture is the same as the one used in the regular code generation. However there are some specificities to the generation of Java bytecode. And in order for you to understand fully the implementation we deemed it more efficient to use an example. We will take as an example a simple *Deca* program and walk through the compiler's process to see every step of the Java bytecode generation.

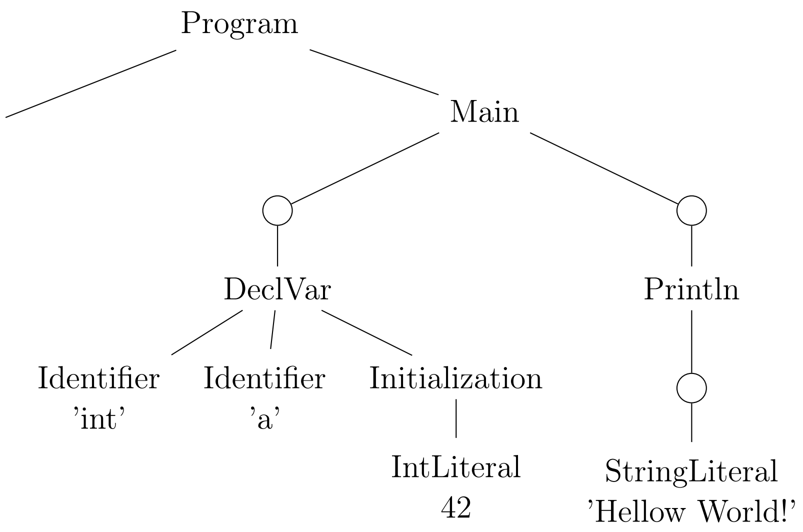
Here is our example:



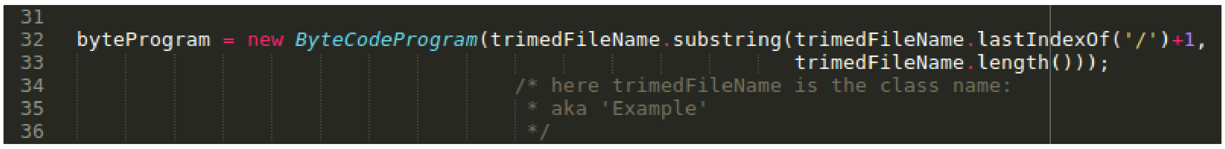
```
{

    int a = 42;

    println("Hello World!");

}
```

Figure 3.3: Example.deca

This simple program only has two instructions but it is enough to understand how the Java bytecode is generated. The last step before diving in the code is to draw the AST corresponding to this program.



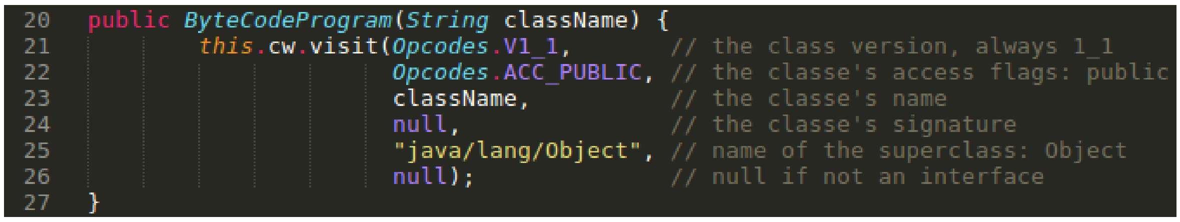
When we compile Example.deca with the command line *"decac -B Example.deca"* we start with the file DecacMain.java. It creates an instance of DecacCompiler and calls its *compile()* method. This is the first function that interests us. This is where the Java bytecode generation starts.



```
byteProgram = new ByteCodeProgram(trimedFileName.substring(trimedFileName.lastIndexOf('/')+1,
                                                          trimedFileName.length()));
                                       /* here trimedFileName is the class name:
                                        * aka 'Example'
                                        */
```

Figure 3.4: Where the Java bytecode generation starts

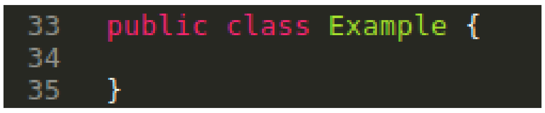
We create an instance of the **ByteCodeProgram** class we have implemented. In this class we can find an attribute whose type is **ClassWriter**. As explained in the previous section we will use it to generat Java bytecode. Calling the constructor of the **ByteCodeProgram** class calls the first very important function of the ASM library, the function *visit()*.



```
public ByteCodeProgram(String className) {
        this.cw.visit(Opcodes.V1_1,        // the class version, always 1_1
                      Opcodes.ACC_PUBLIC, // the classe's access flags: public
                      className,          // the classe's name
                      null,               // the classe's signature
                      "java/lang/Object", // name of the superclass: Object
                      null);              // null if not an interface
}
```

Figure 3.5: Generating the header of the class

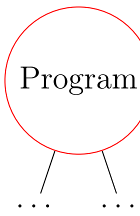
This function allows us to generate the Java bytecode for the header of the class, that is the bytecode for this:



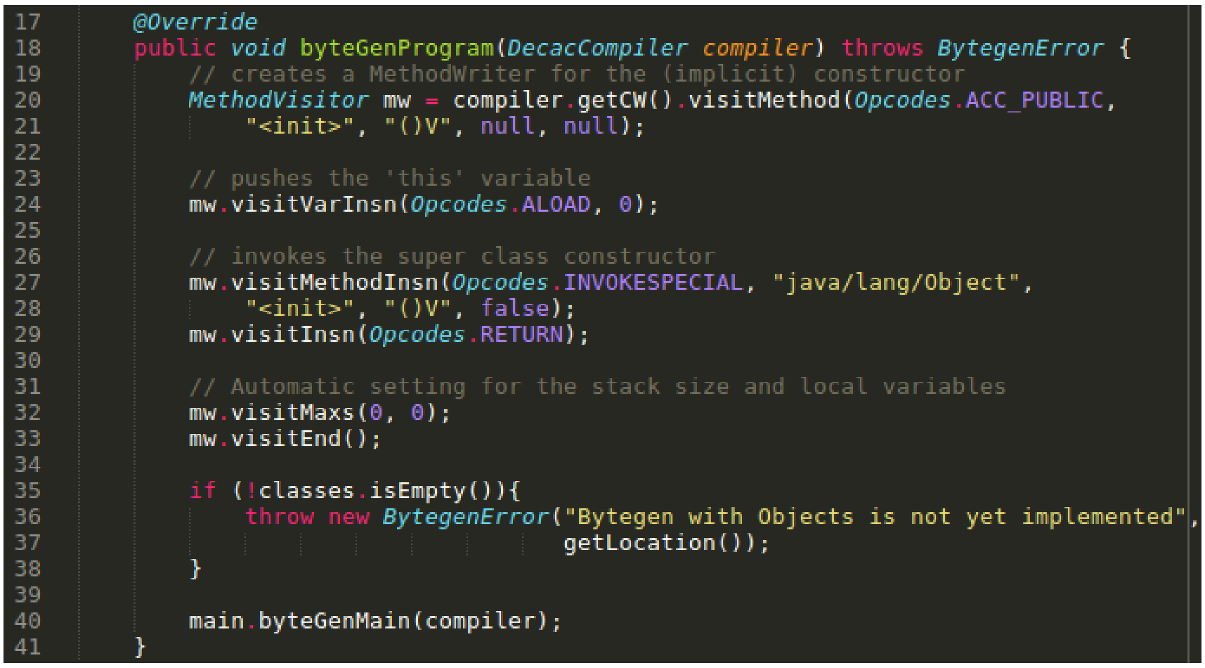
```
public class Example {

}
```

Figure 3.6: Header of the class

We then go back to the function *compile()* in the file DecacCompiler.java and execute the function *doCompile()*. In this function we will call the function *byteGenProgram()* on prog, the instance of **AbstractProgram** created during the lexing and parsing part. We are now on this node of the tree:



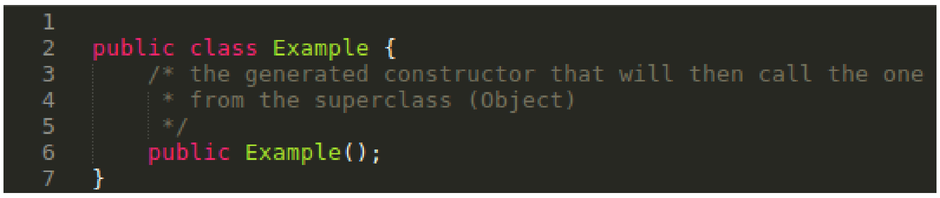
In *byteGenProg()* we continue to generate some Java bytecode:



```
@Override
public void byteGenProgram(DecacCompiler compiler) throws BytegenError {
    // creates a MethodWriter for the (implicit) constructor
    MethodVisitor mw = compiler.getCW().visitMethod(Opcodes.ACC_PUBLIC,
        "<init>", "()V", null, null);

    // pushes the 'this' variable
    mw.visitVarInsn(Opcodes.ALOAD, 0);

    // invokes the super class constructor
    mw.visitMethodInsn(Opcodes.INVOKESPECIAL, "java/lang/Object",
        "<init>", "()V", false);
    mw.visitInsn(Opcodes.RETURN);

    // Automatic setting for the stack size and local variables
    mw.visitMaxs(0, 0);
    mw.visitEnd();

    if (!classes.isEmpty()){
        throw new BytegenError("Bytegen with Objects is not yet implemented",
                              getLocation());
    }

    main.byteGenMain(compiler);
}
```

Figure 3.7: byteGenProg()

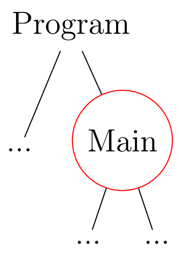
Indeed we need to explicitly generate bytecode for the constructor of the class. If the class doesn't have a constructor, which is our case, we create one, and call the constructor of the superclass **Object** on *"this"* class. On lines 20 and 21 we create a **MethodVisitor** instance for the definition of the constructor's method. In order to get it we need to call *visitMethod()* on the instance of the **ClassWriter** we had created in the DecacCompiler.java file. This method defines that the fact that it is a public method. *"<init>"* indicates that it is a constructor. *"()V"* is the method's descriptor, here it means it doesn't take any parameters and that it does not return anything. The next two fields are null. We then push onto the stack the instance of the class **Example** referred to by *"this"*, this is done on line 24. The next thing to do is to call the constructor of the superclass. This is what we do with the instruction *visitMethodInsn()* on lines 27 and 28. The opcode INVOKESPECIAL invokes the constructor of the superclass on *"this"* that we put onto the stack. Then we need to specify the method's owner class, this is why put *"java/lang/Object*. We now have the bytecode corresponding to this Java program:



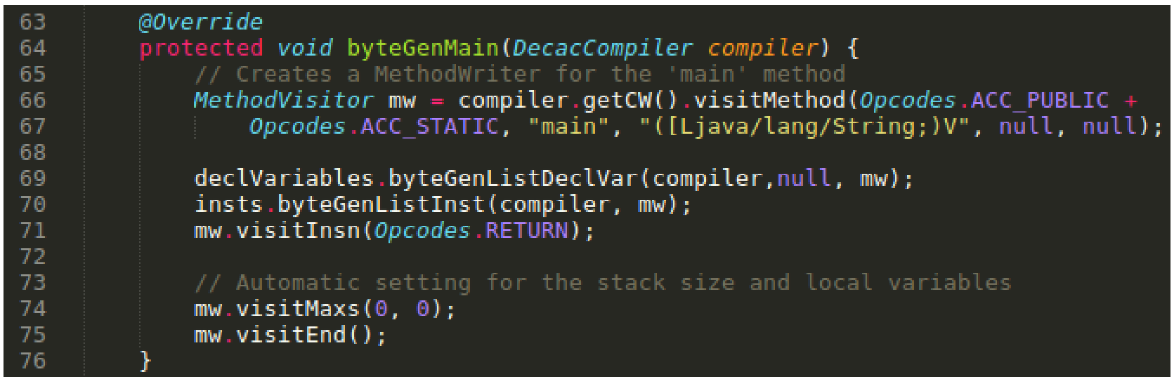
```

public class Example {
    /* the generated constructor that will then call the one
     * from the superclass (Object)
     */
    public Example();
}
```

Figure 3.8: Example.java

We can now move on to the main of our program, which is really what we are interested in. This is what we do with the call to the function *byteGenMain()* of the class **Main** on line 48. This takes us to the next node of the tree:



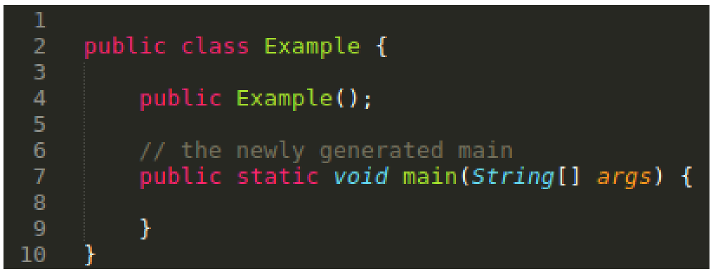
In this function we will generate the bytecode for the main method and generate the bytecode for the list of declared variables as well as for the list of instructions. Here is the code for *byteGenMain()*:



```java
63      @Override
64      protected void byteGenMain(DecacCompiler compiler) {
65          // Creates a MethodWriter for the 'main' method
66          MethodVisitor mw = compiler.getCW().visitMethod(Opcodes.ACC_PUBLIC +
67              Opcodes.ACC_STATIC, "main", "([Ljava/lang/String;)V", null, null);
68
69          declVariables.byteGenListDeclVar(compiler,null, mw);
70          insts.byteGenListInst(compiler, mw);
71          mw.visitInsn(Opcodes.RETURN);
72
73          // Automatic setting for the stack size and local variables
74          mw.visitMaxs(0, 0);
75          mw.visitEnd();
76      }
```

Figure 3.9: byteGenMain()

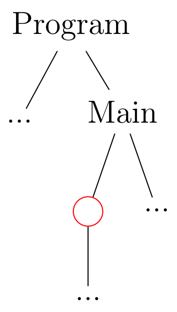
In order to generate the bytecode for the main method we proceed the same way we did to create the constructor. We create an instance of **MethodVisitor** and initialize it by calling the function *visitMethod()*. On lines 66 and 67 you will notice that on top of using the opcode ACC_PUBLIC to set the access we combine it with ACC_STATIC because this is the signature of the main method. Since it requires a string array as a parameter we need to add it within the brackets: *Ljava/lang/String;*. Now we have the bytecode corresponding to this Java program:



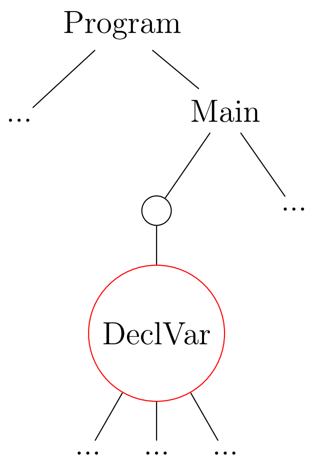
```java

public class Example {

    public Example();

    // the newly generated main
    public static void main(String[] args) {

    }
}
```

Figure 3.10: Example.java

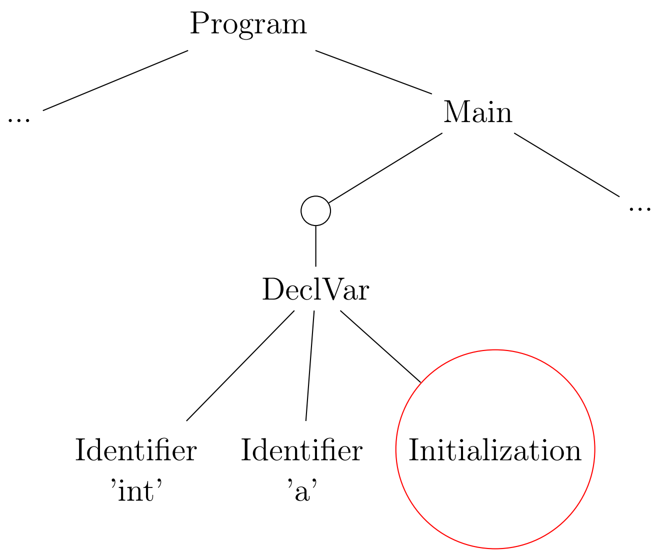
Then we call the function *byteGenListDeclVar()* of the class **ListDeclVar** to generate the bytecode for the declaration of the variables. This takes us to the next node:



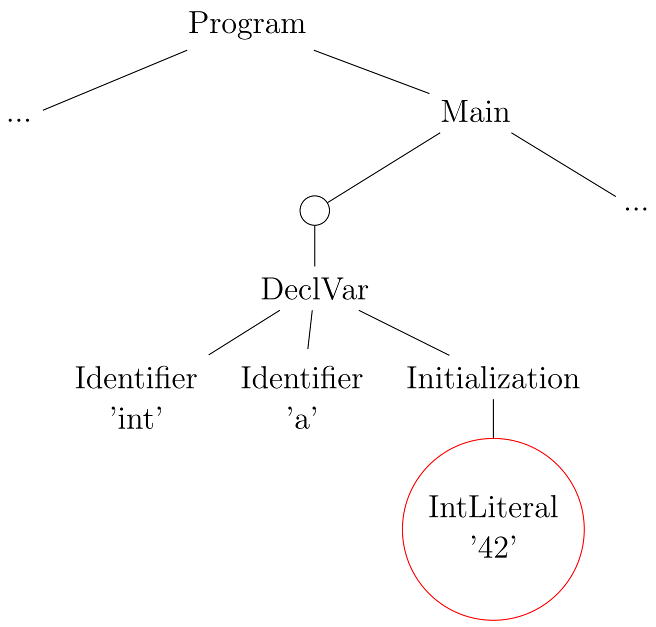
This function loops on the list of **AbstractDeclVar**, that is on all the variable declarations, and calls on each the function *byteGenDeclVar()*. In our case we only have one variable, so this is the next node that we visit:



Since it is an initialization there is no need to generate bytecode for the identifiers *'int'* and *'a'*. So we call the function *byteGenInitialization()* on the attribute *initialization* of the class. And we move on to this node:



We are now in the file *Initialization.java*. The first thing the called function does is to call *byteGenExp()* on its attribute *expression*. *expression* is an abstract expression, in our case it is an instance of the class **IntLiteral**. Therefore it is the function *byteGenExp()* of this class that is called and we move on to the next node:



This function generates bytecode by pushing onto the stack the integer value. The instruction: *"mw.visitLdcInsn(this.value);"* does just that. The value 42 is now on the stack. Now we go back to where the function was originally called, here it is *byteGenInitialization()* and generate more bytecode. Since *'a'* is of type integer we do this instruction: *"mv.visitVarInsn(Opcodes.ISTORE,id);"* that stores an integer value at the address of the variable *'a'*. Now we have generated the bytecode equivalent to this Java program:

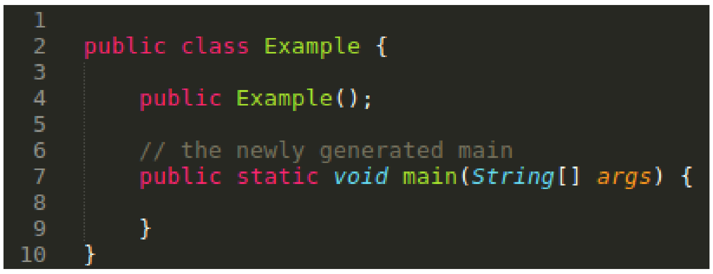
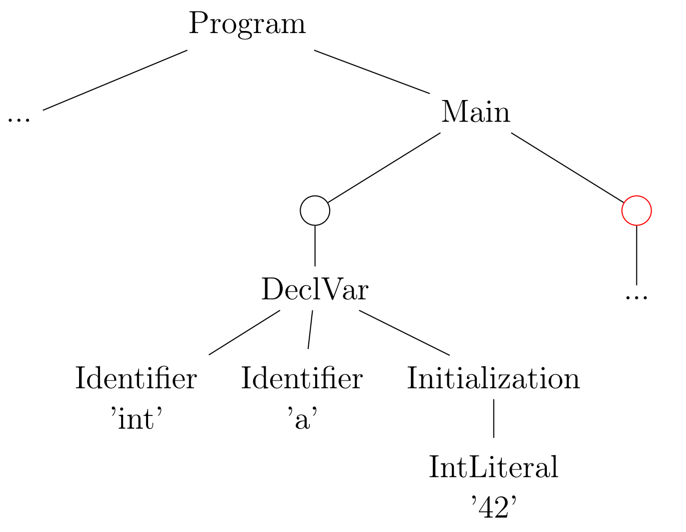

```java

public class Example {

    public Example();

    // the newly generated main
    public static void main(String[] args) {

    }
}
```

Figure 3.11: Example.java

As it was the only variable declaration we now go back to *byteGenMain()* and call *byteGenListInst()* on the list of instructions. And we are now on this node of the tree:

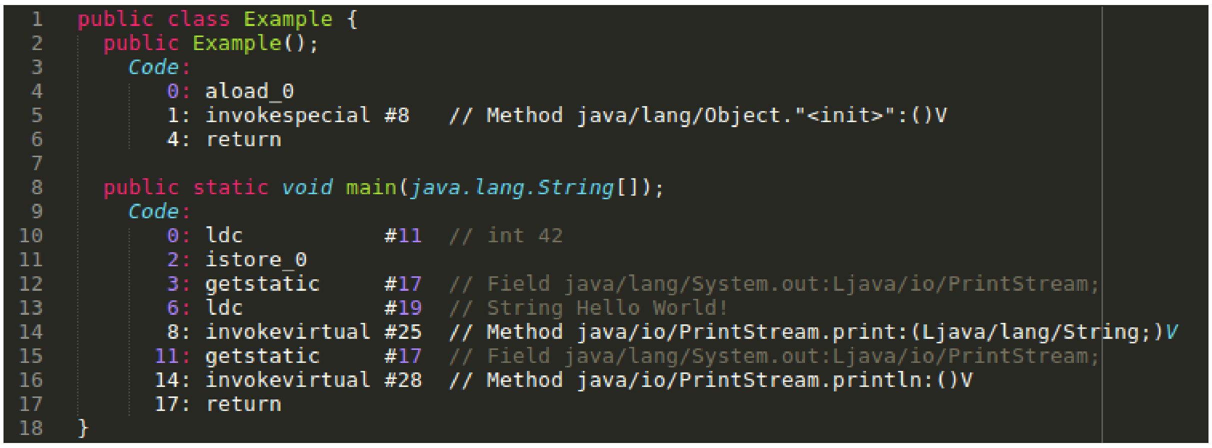


This function, similarly to its counterpart *byteGenListDeclVar()*, will loop through all the instructions of the list and call on each of them *byteGenInst()*. In our example, we call the function *println()* which corresponds to *byteGenInst()* of the class **Println**.

If we apply the same principle we applied to the variable declaration part, we will reach the end of the tree and have generated the bytecode for the whole program. Since you now have a better understanding of how it works we can directly move on to the last part of the bytecode generation that is to get a byte array with all the bytecodes generated and put it in a *.class file*.

We are back in the function *doCompile()* from the beginning. The instruction that realizes that is this one: *"byte[] code = getCW().toByteArray();"*. We can find the function *toByteArray()* in the class **ClassWriter**. This function returns the bytecode of the class that was built with the class writer in the form of a byte array. Now all we have to do is write in a class file and the bytecode generation is done.

If we run the command *"javap -c"* on the class file *Example.class* that we have just generated, we obtain this:



```
public class Example {
  public Example();
    Code:
       0: aload_0
       1: invokespecial #8   // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: ldc           #11  // int 42
       2: istore_0
       3: getstatic     #17  // Field java/lang/System.out:Ljava/io/PrintStream;
       6: ldc           #19  // String Hello World!
       8: invokevirtual #25  // Method java/io/PrintStream.print:(Ljava/lang/String;)V
      11: getstatic     #17  // Field java/lang/System.out:Ljava/io/PrintStream;
      14: invokevirtual #28  // Method java/io/PrintStream.println:()V
      17: return
}
```

Figure 3.12: javap -c Example.class

On this figure 3.12 you will find all of the elements we have just seen step by step.

Chapter 4

Validation method

4.1 Tests

4.1.1 Type

We have essentially tested our implementation of the extension with two different methods, the first using manual tests and and the second one using automated tests.

First we would test manually each small feature added by writing a Deca file specially designed to target the upgrade. This would allow us to make adjustments if need be before testing it more thoroughly with the automated tests.

Once we were satisfied with what we had implemented we reused the test basis used to test the without object part of our compiler for the assembly code generation. Since the JVM does not print floats exactly the same way and raises execution exceptions that are slightly different from the ones the traditional code generation does we had to check every result manually the first time. It allowed us to check that every single test was showing a correct behavior for our compiler and that we were getting the expected results.

4.1.2 Organization

In order to differentiate the results files of the two different kinds of code generation, we created in the deca folder a bytegen folder. This folder is divided into two sub-folders, valid and invalid. In these sub folders we store the expected results for our tests. These text files will then be compared to the ones generated when we run the different tests scripts.

4.1.3 Objectives

The objectives of these tests are simple. However they are slightly different for each method.

For the manual tests the goal is see whether what we are doing is correct and meets our expectations in terms of functionalities. Their purpose is to validate our implementation before moving to the next feature to implement.

The purpose of the automated tests is to make sure that the implementation as a whole is doing exactly what we want. It also helps us verify whether all the features implemented interact with each other correctly and that by being implemented together they do not break anything that was working correctly when tested separately.

4.2 Tests scripts

4.2.1 Motivation

Test scripts are genuinely convenient when one wants to test efficiently one's code. First it allows one to execute all the tests in a much faster way. Second it helps one make sure that the tests that were passing before are still passing now and that the feature implemented is not breaking anything somewhere else.

In our case we also have a python script called *test-manager.py* that allows us to create test results and save them in the correct folders automatically. It was already being used before we started to work on the extension so we modified it a bit so that it could also generate the test results for the bytecode generation.

4.2.2 How to run them

If you want to run all the tests that directly concern the execution you should use the command line *"python ./basic-exe.py"*. If you want to run all the tests you should use the command line *"./<path>/our-tests.sh"*.

If you want to create a test you should use the command line *"python ./test-manager.py"* and let it guide you through the creation of the test.

Chapter 5

Validation results

In order to keep the same efficiency achieved for the deca compiler in its first part (basic Assembly Instructions), the validation test process for the bytecode extension was designed, implemented and based on the same parameters. In this way we could ensure the quality of the bytecode generation and its compatibility with *javac* generated bytecode files. Taking into account the high Cobertura Level (aprox. 97%) of the compiler during its basic stage, the same type and amount of tests were performed on the extension. Valid tests as well as invalid ones were passed while generating bytecode, interactive *.deca* files were also compiled into byte arrays and run through the JVM obtaining the expected results.

As a general review of the results obtained testing the project under the requirements demanded, the behavior of the compiler is rather satisfying. This allowed us to check that the product meets the requirements and specifications as well as the acceptance and suitability for a possible costumer, fulfilling its final intended purpose.

Conclusion

We finished the implementation of the Java bytecode generation for all Deca files without objects. We have tested our implementation and have not found any issue. All the specifications have been respected. Therefore we are proudly announcing that the extension meets our expectations.

Moreover, even though we were not sure we would be able to deliver it on time our compiler is also able to generate bytecode for all Deca files. The reliability of this bytecode extension for Deca files with classes remains unsure. We did not have time to test it thoroughly. However, on all the manual test we ran on it we did not find any issues. Furthermore we tried to create a Java file instantiating an object of a class file compiled with our compiler. The results are positive and we get the expected behaviour.

Bibliography

[Brua] Eric Bruneton. *Class ClassWriter*. URL: http://asm.ow2.org/asm50/javadoc/user/org/objectweb/asm/ClassWriter.html#visitMethod-int-java.lang.String-java.lang.String-java.lang.String-java.lang.String:A-.

[Brub] Eric Bruneton. *Class MethodVisitor*. URL: http://asm.ow2.org/asm50/javadoc/user/org/objectweb/asm/MethodVisitor.html#visitMethodInsn-int-java.lang.String-java.lang.String-java.lang.String-boolean-.

[Wika] Wikipedia. *Bytecode*. URL: https://en.wikipedia.org/wiki/Bytecode.

[Wikb] Wikipedia. *Java bytecode*. URL: https://en.wikipedia.org/wiki/Java_bytecode.

[Wikc] Wikipedia. *Java bytecode instruction listings*. URL: https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings.

[Wikd] Wikipedia. *Java class file*. URL: https://en.wikipedia.org/wiki/Java_class_file.

[Wike] Wikipedia. *Java virtual machine*. URL: https://en.wikipedia.org/wiki/Java_virtual_machine.

[Wikf] Wikipedia. *ObjectWeb ASM*. URL: https://en.wikipedia.org/wiki/ObjectWeb_ASM.

[Wikg] Wikipedia. *Opcode*. URL: https://en.wikipedia.org/wiki/Opcode.

ENSIMAG
681, rue de la Passerelle
38400 Saint-Martin-d'Hères