# Documentation of Lab 3

**1. An introduction where the problem is described. For example, what should the program do? Which classes do you have to define? Which methods do you have to implement for these classes?**

The goal of Lab 3 is to define some classes that could inherit attributes and methods from the superclass Instruction. These subclasses are instances of an Instruction object with extra functionality. To be able to create subclasses of the Instruction class, all its attributes should be initialized as protected in order to allow their access. The program should start the execution when a method named "run" of the Interpreter class is called. Therefore, the implementation of Lab 3 consists of three extra classes (Function, TurtleInstruction and Interpreter) plus the modification of the Instruction and Logo classes.

To start with, the **FUNCTION** class has been designed as following:

Since Function is a subclass of Instruction we only define one attribute:

- - associatedProgram: a Function object is indeed a Program object associated with a name.

In order to initialize the class, we call its constructor Function:

- + Function(initword: String, p: Program): it called the superclass constructor to set its name. Given that Function objects do not have parameters, it is the Instructor constructor without bounds that is called.

In addition, we have implemented a method that returns the Program:

- + getProgram(): returns the associatedProgram attribute.

In addition, the **TURTLEINSTRUCTION** class has been designed as following:

A class whose objects are made of a single attribute:

- - turtle: an instance of a Turtle object.

In order to initialize the class, we call its constructor TurtleInstruction:

- + TurtleInstruction(initWord: String, initTurtle: Turtle, initMinRange: int, initMaxRange: int): since the TurtleInstruction is a subclass of the Instruction superclass we must call its constructor to declare the name and bounds attributes. Given that all these instructions that

operate on the turtle have parameters, we only have designed this constructor that takes the range as arguments.

We have designed a method to print the Instruction name and parameter in the terminal:

- + apply(parameter: int): by taking a parameter as argument, it prints to the screen name and parameter.

Moreover, to use a Function object as a **LOGO** instruction in other programs, we need to modify this class:

The modification only consists of implement a new method:

- + addFunction(name: String, program: Program): it creates a new instance of a Function with the parameters given. Then, add it to the dictionary. Its key is the name itself.

Finally, we have implemented the **INTERPRETER** class as it follows:

Its attributes let the Interpreter execute a Logo program and keep track of the loops inside it:

- - logo: a Logo object.
- - program: the Program to be executed.
- - line: variable to keep track of the current line

Then, we have its constructor:

- + Interpreter(logo: Logo, program: Program):

Its unique method allows us to start execution:

- + run(): this method is well-explained in the next question.

**2. A description of possible alternative solutions that were discussed, and a description of the chosen solution and the reason for choosing this solution rather than others. It is also a good idea to mention the related theoretical concepts of object-oriented programming that were applied as part of the solution.**

We have followed almost all the references from the given Lab 2 diagram, although we have made some changes in the Interpreter's attributes.

The attributes of the Interpreter class referenced were:

- logo: Logo
- program: Program
- line: int
- nloops: int

However, we have just declared three of them:

- logo: Logo
- program: Program
- line: int

Therefore, we have decided not to declare the nloops attributes since our implementation for letting the run() function execute nested loops, using a stack and a 2D ArrayList of integers, leads us to avoid declaring this attribute. However, despite not declaring it, we have used another type of local variables that have a similar utility.

In order to iterate through statements in the program we have used the variable “line” to keep track of which line the statements are. By implementing a while loop we can iterate until the value of the line is bigger than the total number of lines, i.e. the number of statements in the program. Besides, once this variable line indicates the beginning of a loop with the word “REP”, we push it into the stack. In this way, if there are nested loops, we will always be able to know the line of the current one since the last will be pushed into the stack. Finally, once a loop has finished, we execute the pop() function.

Additionally, instead of using the nloops attribute, we have used a 2D ArrayList of integers to keep track of the number of executed loops and in what line that loop starts. Thus, the ArrayList has as many rows as loops are and each row will have two integers. The first element will store the number of the beginning line of the loop, and the second element will keep track of the number of times that loop has been executed. In this way, using a 2D ArrayList instead of just a variable to keep track of the number of executed loops, allows us to handle nested loops since with just an integer value, we could just handle one. Finally, once a loop has finished, we remove its row from the ArrayList using the remove() function.

**3. A conclusion that describes how well the solution worked in practice, i.e. did the tests show that the classes were correctly implemented? You can also mention any difficulties during the implementation as well as any doubts you might have had.**

To conclude, we need to test if the classes have been correctly implemented. In order to do so, we have designed an algorithm test in the main method located at TestLogo class following the suggested test seen in class.

To start with, we have created a Program object whose purpose is to advance 100 units and rotate 50 degrees three times. Then, we have initialized an instance of a Logo. In addition, we have added to the Logo the previous Program object as a Function with name "ABC". Moreover, we have implemented another Program object. This Program called "p2" is the one which will be run later. To this object, we have added a list of statements in which there is one called "ABC", the same name as before. Given that, we have ensured that the implementation for a Function object in the run method has been well-implemented. Finally, we have called the method run of a new Interpreter object created with the Logo and "p2" Program as arguments.

Therefore, the following statements will be printed. The "ABC" Statement, which is indeed the Program described above, and two statements extra. Moreover, we have added the "PEN" Instruction at the bounds of "p2" to activate the pen and then deactivate it to ensure that the turtle is printing while it is moving. As it can be seen by the following image, we can show that the solution works properly.

Output of the program:

```
Executing PEN with parameter 1 to the turtle.
Executing FWD with parameter 100 to the turtle.
Executing ROT with parameter 50 to the turtle.
Executing FWD with parameter 100 to the turtle.
Executing ROT with parameter 50 to the turtle.
Executing FWD with parameter 100 to the turtle.
Executing ROT with parameter 50 to the turtle.
Executing ROT with parameter 40 to the turtle.
Executing FWD with parameter 20 to the turtle.
Executing PEN with parameter 0 to the turtle.
```