JAVA FOR THE STUDY OF EVOLUTION

VOLUME XVII version 2

EASY JAVA GENETIC PROGRAMMING

José del Carmen Rodríguez Santamaría
http://evoljava.com

August 2017

Contents

Preface

The series *Java for the Study of Evolution* is directed to scientists that want to manage a serious but not excessively expensive tool to study evolution by direct experimentation under perfectly controlled conditions. These requirements cannot be met in nature but only in simulations and mathematical models. In consequence, the series has three main purposes:

1. To endow the community of **researchers in biology and evolution** with *high level programming*, enabling an accurate study of models and simulations of the most diverse nature.

2. To clearly show how this tool is used to study the fundamental questions of evolution.

3. To suggest that the study of Java could be *very fruitful* for **undergraduates** in biological sciences even more than calculus alone.

This is the 17th volume: we test with Java a mandatory prediction of the Evolutionary Theory: it must be possible to use evolution to synthesize computer programs for the most diverse functions. This is the generating idea of GP (Genetic Programming). We use grammars to show with a simple example that this program is viable. Version 2 adds more options from the literature.

Introduction

We have used evolution to solve various problems including the development of software for very specific purposes, say, to enable the coordinate movement of some rectangles to simulate a worm that climbs over the screen. To pass from specific examples to general software is to arrive into the world of **universal computing**. It is similar to passing from arithmetic calculators to desktop computers. How to do that by means of evolution is the field of `GP (Genetic Programming)`. This is widely known since the 1980's thanks to the efforts of John Koza ([41] 1980, [42] 2007) who in his turn made profit of the works of other pioneers. Our aim is to explore here this thematic from the perspective of Java.

After some Theoretical Considerations expressed in Chapter I we pass to chapter II *From strings to code*, where we propose a solution to the problem of universal computing by showing a Java program that can generate and run all possible Java programs. The trouble is that needed time is cosmological. So, a challenge arises: can we pass from eons to, say, minutes? In this regard, we use genetic algorithms to solve a very simple problem whose solution is automatically packed in a Java class that is ready to be used. This achievement shows that Easy Java Genetic Programming is feasible.

To graduate from genetic algorithms in order to step into *Universal computing* is the challenge formulated in chapter III. Over an extremely simple demo we show how to run evolution over the space of Java classes to select one to fulfill a specific function.

We invite the Reader in Chapter IV to give a look at the entrails of Java for the sake of GP. It happens that Java runs over a virtual, simulated processor called the *JVM (Java Virtual Machine)*. The corresponding machine language is bytecode and we argue that it is better for GP than Java itself. That is why the JVM is important for GP practitioners.

In chapter V we commend ourselves to *Parsing .class files* which convey the bytecode of java files. It happens that this aim needs heavy duty machinery and so, we prefer to make a revision of the literature and to run some free software. We consider that parsing is the correct beginning for GP on the JVM and that available

software seems to be very useful in this regard. Experts in the field have gotten impressive achievements and our training along the present material has allowed us to enjoy their publications.

Chapter VI deals with a simple yet a very powerful idea: if the JVM is a (virtual) processor, it must support other languages apart from Java much as (real) computers do. This is how we have today many languages, more than 100, that boil down to the same bytecode language machine. The high variety is sustained by a simple fact: there is no good weapon for every battle. So, one language claims to be good for one thing and another for a complement and so on. Now, a question is in order: Which is the best JVM language for GP? We already know that GP is very difficult and that Java is not a panacea. So, we must research other languages or maybe invent one in our own. It happens that the JVM language *Clojure* (Bendersky [8] 2017) has created an environment in which these and more technical questions have a natural place where to live and reproduce giving abundant and fine fruit. It is a pleasure to share a guide for the study of this wonderful language whose application to GP is immediate. The Author feels that Clojure is a wonderful way of making easy-rich-and-productive-JVM-GP. Did I say easy? Not too much, indeed. Nevertheless, there is an additional plus: the experts in the area make their best to release the code of their state of the art articles as soon as possible. Thanks a lot to them.

Now, let us express in regrd with IDE's (Integrated Development Environments).

We already know how to work with `NetBeans`. We will continue to use it but our main and default IDE for this volume is `Eclipse`. It was created at IBM in 2001 and became a foundation by 2011. By 2017 the last version is `Neon`. It is as friendly as `NetBeans`. Installation instructions are system dependent. For Ubuntu Linux and Java 8:

1. Open a dedicated folder for Eclipse.
2. Follow the instructions by `Ji m (2014)` from

   http://ubuntuhandbook.org/index.php/2014/02/install-oracle-java-6-7-or-8-ubuntu-14-04/

   to download and install Java 8 form Oracle.
3. Follow the instructions by `Ji m (2014)` from

   http://ubuntuhandbook.org/index.php/2016/01/how-to-install-the-latest-eclipse-in-ubuntu-16-04-15-10/

   to download, install and lunch `Eclipse` IDE. Choose `Eclipse IDE for Java Developers`.

4. To work with Eclipse follow your instinct: Eclipse is quite similar to NetBeans. Anyway, keep at hand a tutorial like this by tutorialpoint (2017) : https://www.tutorialspoint.com/eclipse/

5. Choose a suitable folder as workbench.

6. To explore Eclipse, click the welcome page here and there and use the forward and back arrows on the top bar to navigate.

7. To go to the editor window click the orange play-icon marked as Workbench. Then you will find an ambient for work very similar to that of NetBeans.

To import a NetBeans project into Eclipse (Moinul Islam, [57] 2015):

1. Unzip you NetBeans project and localize your src folder. Open it and select all of its elements.

2. In Eclipse, Go to File $->$ New $->$ Java Project

3. Give a name to your project and click finish to create your project

4. When the project is created find the source folder in NetBeans project, drag and drop all the source files from the NetBeans project to 'src' folder of your new created project in eclipse.

5. Move the Java source files to respective package (if required)

6. Now you should be able to run your NetBeans project in Eclipse.

All our Java programs used for this volume come packed into a single zipped Eclipse project and can be downloaded from our site. Once downloaded and unzipped, it can be imported into Eclipse (Williams and Smith, [100] 2007) :

1. Click File... Import... from the Eclipse main menu.

2. Expand General, select Projects from folder or archive, and click Next.

3. In the opened dialog browse for the target folder.

4. Click Finish

5. You should see now the new project in the Package Explorer.

Somewhere along the text you will be invited to work with `Clojure` -a language of the Java clan. We will use `Eclipse` but if a different IDE is chosen, keep in mind that each IDE must have its separate copy of Java.

All procedures in this introduction are guaranteed to function. In contrary case, write to me `jose@evoljava.com` to negotiate a truly functional methodology.

About Software License: we will use all along abundant code that has been developed by third parties. As a rule, that software is licensed, say, with the ECLIPSE PUBLIC LICENSE `https://www.eclipse.org/legal/epl-v10.html`. In this regard, let us follow the following behavior: reprint not original code apart form short snippets; instead, link citations to its url at Github or wherever else. Modifications (commented code with the original material included + the url to the original code) made by the Author together with all his productions come not only with unrestricted license but with a blessing for good so, every one shall proceed as he or she desires -most probably by laziness.

Bogotá, Colombia, *José Rodríguez*
August 2017

Chapter 1

Theoretical considerations

```
Clear concepts beget experiments that give rise to durable
truths.
```

1 ***Introduction and purpose.*** *The genome is software and natural evolution is believed by modern science to be the developer responsible for its existence. So the synthesis of software by means of evolution must be possible. This is a mandatory prediction. Some other details are specified.*

1.1 A mandatory prediction

GP (Genetic Programming) is the synthesis of software by means of evolution. Let us see why it is a mandatory prediction of the Evolutionary Theory.

2 ***Species and software.***

The bird, the frog, the dog, the cat, the horse, the cow, the lizard. These 7 species are too many and too different to be so exceedingly perfect. And there are more (Locey [51] 2016). Where do they come from?

According to the Evolutionary Theory all of them appeared by evolution of descent. No one has put this belief in words better than Darwin ([19] 1859):

```
It is interesting to contemplate an entangled bank,
clothed with many plants of many kinds,
with birds singing on the bushes,
with various insects flitting about,
and with worms crawling through the damp earth,
and to reflect that these elaborately constructed forms,
```

```
so different from each other,
and dependent on each other in so complex a manner,
have all been produced by laws acting around us ...
Thus, from the war of nature,
from famine and death,
the most exalted object which we are capable of conceiving,
namely, the production of the higher animals,
directly follows.
There is grandeur in this view of life,
with its several powers,
having been originally breathed into a few forms or into one;
and that,
whilst this planet has gone cycling on
according to the fixed law of gravity,
from so simple a beginning endless forms most beautiful
and most wonderful have been, and are being, evolved.
```

While this was written in 1859, the further development of the theory leaded to the postulate that there is no gap in within ordinary and living matters. So, we officially became sons and daughters of the dust of the stars.

This theory is universally accepted. To understand why, it is enough to recognize that we are incredible similar to pigs, hens, fishes, ...

Thus far, so good.

The problem is that the Darwinian proposal has turned into a plainly scientific theory and in the realm of science it has various mandatory predictions. The one that interests us at this moment reads as follows:

3 ***Mandatory prediction.***

According to materialism, living beings result from the execution of genetic programs written in the DNA that is run over extant living cells which constitute a tightly constrained environment. These programs classify as verbal instructions for a computer: they are verbal because they need a code of interpretation given by the Genetic Code. The interpreter is the set of aminoacyl tRNA synthetases and the computer that executes them is the ribosome. So, the genome is software. Now, the Evolutionary Theory reads: evolution was the developer responsible for the software of all genomes. By evolution we mean variability created by mutation

and recombination plus differential death and/or reproduction in response to the effects of the environment + sexual selection.

Now, the great complexity of life, its exceeding perfection and its huge variability imply a mandatory prediction of the Evolutionary Theory: evolution in the space of Java classes is predicted to produce every kind of needed software of the most high quality, variability and complexity.

In short, our mandatory prediction reads: GP must be feasible.

4 ***Exercise .*** *Solve the following objection: evolution is not teleonomic, it has no purpose at all and species arouse without a purpose just by a tinkering game that was filtered by differential surviving and reproduction. By contrast, GP is assumed to be important because it promises to synthesize the software that we need, so GP is by its very nature teleonomic. Therefore, the aforementioned prediction is no prediction at all because they refer to different settings.* *Answer*

As we see, the objection deals with natural and directed evolution, both very important for us. That is why we must be ready to distinguish the two flavors one from another. Since in both flavors GP is a mandatory prediction, we can peacefully continue.

1.2 Conclusion

GP is a mandatory prediction of the Evolutionary Theory. In first place, the abundance, complexity and perfection of living beings force the prediction that evolution shall be useful to produce virtual living beings -software included- of the most high quality for our most varied needs. On the other hand, the Evolutionary Theory argues that adaptation to the environment is the fuel of evolution. So, it predicts that it shall be possible to use evolution to produce programs that are adapted to Java -which is the environment in which our programs live. Besides, other constraints can be added to filter programs for a specific function. This new environments also are expected under the Evolutionary Theory to be fruitful although with slower evolutionary rates i.e., with more computational cost in time and memory.

Chapter 2

From strings to code

`Evolution of strings that encode for Java programs`

5 ***Purpose.*** *We show here how to use grammars to evolve strings that encode for viable Java programs which are ready to be used.*

2.1 The runtime compiler

Java offers the possibility to understand a string as the code for a Java class and, if it is viable, to compile and run it. Hence, we are done: it shall be possible to use evolution over strings to synthesize Java programs with a predefined function. So, to begin with let us show how to pass from strings to code forth and back.

6 ***Code as a string.***

This is the most renown Java class:

```
//Program R6 MyHelloClass

package Programs;


public class MyHelloClass {

    public static void main(String args[]) {
    System.out.println("Hello World 1357");
  }

}//End of main class R6 MyHelloClass
```

We can copy the whole code from the NetBeans IDE to the clipboard and paste it into a string. To do that declare a variable as a void string

```
String s = "";
```

and then insert the content of the clipboard inside the double quotes.
Next, we can command to print that string:

7 The following programs contains a string that encodes for another Java class:

```
//Program R7 HelloClassAsString

package Programs;
//A Java class was encoded as a string
//that next was printed to the console.

public class HelloClassAsString {

    public static void main(String args[]) {
        //This  string  encodes for the class
        //MyHelloClass.
        //It was produced by copy-paste directly.
        String s =
"//Program R6 MyHelloClass\n" +
"\n" +
"package Programs;\n" +
"\n" +
"\n" +
"public class MyHelloClass {\n" +
"    \n" +
"    public static void main(String args[]) {\n" +
"    System.out.println(\"Hello World 1357\");\n" +
"  }\n" +
"    \n" +
"}//End of main class R6 MyHelloClass";
    System.out.println(s);
  }

}//End of Program R7 HelloClassAsString
```

8 ***Exercise.*** *Run the previous program and play with the code.*

Next, we need a tool to test, compile and run a string that possibly encodes for a Java class. Thus, let us learn how to make use of the **runtime compiler**.

9 ***Definition.*** *Program* `R9 RuntimeCompilerExample` *shows how a string can be interpreted as code, compiled and run. A string that encodes for a viable Java class is called* ***virtual code****.*

10 ***Exercise.*** *Run previous program and play with the code.*

11 ***Exercise.*** *Verify that a minor encoding error of the string will cause the program to crash. Hence, the previous program is not compatible with evolution because no future is allowed. So, add an appropriate exception management procedure to develop a program that must be capable to survive ill strings. That type of programs is called* ***fault tolerant****, which is a must not only in modern standards of the software industry but in other branches of technology, say, electronics. Our answer can be seen in program* `R11 RuntimeCompilerExample2`

We know how to test a string to decide whether or not it encodes for a Java class. So, we can generate all possible Java strings and if we test them with the previous program, we would have a program that eventually might generate all possible Java classes.

12 ***How to generate all Java strings.*** *Java can generate all numbers in base 10 "0123456789". So, we need an encoding converter from base 10 to an arbitrary base having in mind the alphabet that Java uses. If we couple this generator to our previous test-program, we are receiving a string as input so, we can decide whether or not it encodes for a Java program and, if that is the case, we can compile and run it. So, the first step is to generate all numbers on a given arbitrary alphabet. This is done in* `Program R12a ConverterToBaseN` *with variations and tests. A short version prepared to be shared with* `Stackoverflow` *can be seen in* `Program R12b FromBase10ToBaseN` *(joser, [38] 2016).*

13 ***Exercise.*** *Compose a program to generate all Java strings. Beware: you need a loop that never stops. Such type of programs is called* ***never halting****. You can see our answer in program* `R13 AllJavaStrings`

14 ***Exercise and challenge.*** *Compose a program to generate* ***all*** *Java programs -as much as Java numerical types allow to do that. You can see our answer in program* `R14 AllJavaClasses`*.*

2.2 Bugs and Computer Science

On its face value, the program composed for the previous exercise is pretty useless:

15 ***Challenge.*** *Prove that you need to run the previous program during cosmological time to reach the first Java program with a visible output, something like the Hello class.*

Now, this failure is a characteristic of Java that is not shared with all languages. For instance, in Python ([47], 2016) the shortest program with a visible output might be

```
print("Hi!")
```

16 ***Challenge.*** *Modify program* `R14 AllJavaClasses` *that it could produce all possible Java programs but the shortest one immediately.*

Anyway, in Python and in whatever language a program 10 lines long needs cosmological time to be synthesized by our previous program `R14`. Nevertheless, for a human developer most programs that long are extremely simple ones. So, a question immediately arises: what do human developers have that can make such marvel things like breaking cosmological time into some few minutes? Our answer is this: Human beings have a powerful intelligence that is animated by rudimentary implementations of an evolutionary process. So, it is not insane to think that evolution could help us to synthesize software. Besides, one might keep in mind that most possible the easiest way to synthesize software is to implement an artificial intelligence thousands of times more powerful than Watson ([35], 2016) and to endow it with evolution. This is not a fantasy because Watson is already worth of respect.

So, we put our hopes in the marvelous fact that intelligence exists. Nevertheless, it is prudent to ask: how sure is a realizable intelligence as a panacea against complexity? At the dawn of Computer Science the incredible power of the human intelligence propelled serious and respected researchers to belief that everything is possible for us without barriers or limitations. Nevertheless, in our modern culture, pervaded by the scientific method, a proof of that belief is demanded. This enterprise was begun around 1930 and rather soon produced terrific fruits: very simple questions about programs cannot be answered automatically if we use integer numbers. A renown example is the **Halting problem** (Wikipedia [95] 2017a):

```
In computability theory,
```

```
the Halting problem is the problem of determining,
from a description of an arbitrary computer program
and an input, whether the program will finish running
or continue to run forever.
```

The point is that no computer can exist that can solve this problem for all instances ([90], 2016).

But, what is a computer? This is a troublesome question because the industry has produced too many types of these machines. Troubles fade away thanks to two principles. First: all possible computers are equivalent in the sense that they can compute the very same problems although an insignificant (polynomial) variation exists as to which arrives first to the solution on which inputs. This is the **Church thesis** that is certain for all built types of computers that work on integers. Second: because all computers are equivalent, we can take the simplest one, the **Turing Machine** . This special machine was proposed by allan Turing in the 1930-s both to fit mathematical research and to be built by smart carpenters ([20], 2016).

When one needs to precise concepts, two definitions can help. First, a **Universal Turing Machine** is the equivalent of an ordinary computer that can run whatever program. Second: a **Turing Machine** is equivalent to an ordinary computer than runs a specific program. Most of times, the context must be examined to determine what case is the discourse about. A Turing machine also can be simulated in Java:

17 ***Research.*** *Scan the web for Java simulations of Turing machines, choose one, and mount it. One example is the code by Tzimoulis ([88] 2015)*

Returning to the halting problem, one might imagine that a program whose halting has not been determined anyhow could be something esoteric. Seemingly this is not the case. In fact, there is such a mathematical conjecture know as the **Goldbach's conjecture** Udiprod ([89], 2013) that says:

```
Every even integer greater than 4 can be expressed as
the sum of two primes.  Examples 5 = 3 + 2; 28 = 23 + 5.
```

Using supercomputers this conjecture has been verified up to very long integers but what will be further is not known because mathematicians have been unable to decide the conjecture. This failure induces us to think that complexity has a core that is tremendously difficult to dismantle even by smart humans (some people have suggested that human brains can use quantum computing on molecular items

that are effectively disconnected from its warm environment and, in any case, the human mind uses too much advanced trickery, say, topological data analysis).

18 ***Exercise.*** *Implement and run a program to test the Goldbach's conjecture. Our answer recalls an ancient idea due to Eratosthenes to find prime numbers. See* ***program R18a Eratosthenes*** *that is used to test the conjecture in program* `R18b GoldbachTest`*.*

19 ***Challenge.*** *Express your opinion about the following conjecture: with enough time, program* `R18b GoldbachTest` *will appear as output of the program* `R14 AllJavaClasses` *and, by the same token, every undecidable program will be followed by another undecidable program of brand new type. Say, a program that test* `the multiples of 16 can be expressed as the sum of two primes` *does not count. A decidable proposition about integers is one about which one can say that it true else false. An undecidable proposition about integers is one for which there is no way of predicting that it is certain for every integer nor that it is false for certain integer and so it must run until eternity.*

So, it seems that in regard with some propositions about programs, all we can do is to run humble simulations during extremely short periods of time. In other words, experimentation is a fundamental part of computer science. And, the most immediate reality for every human experimentalist in the development of software is that there is no final software product without bugs whose correction creates more bugs. So, Are bugs a necessity?

In short, we have the following trend of ideas: sophistication is tied to specific and tremendously complex tasks. These are solved by programs whose correctness must be tested maybe with computer experimentation or maybe with mathematical tools. The result is always the same: the developed program does not fit `spec` (a document with the specificities of the task to be implemented). When differences are traced to specific errors, these are called `bugs`. The point is that we cannot avoid testing because, apart from minor simplifications that could be very fruitful for the Software Industry, `the fastest form to know what a program does is to run it on every possible input`. Or, in terms of the ancient culture that educated me:

Ye shall know them by their fruits.

In fact, the Author has verified once and again that to make a program write the relevant calculations is the fastest way to know where a bug is and what it consists in. But, why spec (or a mental plan) and the first versions of the program shall be

so different? Because we, human beings, are incapable of correctly running in our minds the program that we have in mind or maybe on paper. More to the point, for the Author and in regard with the mental running of programs it is difficult for him to see beyond his nostrils.

20 ***Example: Java quizzes.*** *Java has such a poison in itself that one always can assign a function to any program but in general in the wrong way. For instance, the Reader is gently asked to correctly predict the output of the following program presented by Maroof ([52] 2016). To check your answer you can run the program and/or go to the corresponding link. The program is:*

```
public class MyClass{

    int x;

    MyClass(int y){

        this.x = y ++ ;

    }

    int method(MyClass mc){

        if(mc.x == 5){

            mc.x += 6;

            method(mc);

        }

        if(mc.x == 6){

            mc.x += 9;

            method(mc);

        }

        return ++ mc.x;
```

```
    }

    public static void main(String[] args){

        MyClass mc = new MyClass(5);

        System.out.println(mc.method(mc));

    }

}
```

What is the result of our incapability of instantly run in our mind a given program? Tons of bugs. So, there is no software development without a fierce battling against bugs. The hope that some experts have is that with appropriate tools and hard work the battle can be won and software can be released in terminated, perfectly correct form. That is seemingly not the case: design always come with bugs and there is no testing without introducing other heavy bugs. And there is no more inexpensive way to verify a program than testing. A very good introduction to the technicalities of this thematic was presented by Pressler ([60] 2016)

This implies that the Evolutionary Theory of the Origin of the Species is obviously false. Why? It is because every one knows that his or her body is a marvel and that marvels are difficult to make. Therefore, it is highly improbable that it could be made perfect on the very first attempt. So, a history of the improvement of perfection must exist. This must be certain as for evolution as for aliens as for future molecular biologists all alike. Since such evidence is not found in nature, it implies that life on Earth does not appeared by evolution. Now, if you are a perfectionist, there is one more point for you:

21 ***Exercise.*** *Solve the following objection: a bug is an error against a purpose of design and a trend in a given implementation. But evolution has no purpose at all so, the word bug does not belong in the Evolutionary Theory and the proposed falsification of that theory is devoid of meaning.* *Answer*

Now, there are simple problems that are solved by evolution quite easily so, one cannot demand to see a fossil record of the evolutionary activity showing the evolution of perfection towards the corresponding perfected trait, organ or function under explanation. Example: Hawaii hosts some 1000 species of flies that are endemic (Wylie, [102] 2015). Is not this a case of an easy problem for evolution? We

can answer with a lazy yes, as everybody does, but the EvolJava Community must see here a challenge: to bring together specialists in Ecology, Population Genetics, Molecular Biology, Developmental Biology, Morphology, Genetic Programming and Genetic Engineering working shoulder to shoulder to give a resilient consensus, one that could be improved after every attack.

Besides simple problems, we also know that there are problems that could be hard, very hard, even impossible for evolution (this was explained in the previous volume of this series). For instance, everything relating to function is very hard, a conclusion that we achieved by studying SAT. This predicts that were evolution the explanation of our beings, we shall find our physiology filled in malfunctions. Wrong prediction. On the other hand, our studies of trees have shown us that when a tree is tied to pleitropy, evolution over the space of trees is impotent. Now, since morphogenesis depends on cellular reproduction which is described by a tree while pleitropy might come from the bauplan to fit a reliably organism, we consider that everything that is related to the form, as bones and carcases, is a candidate to be considered as an extremely hard problem for evolution. This picture enables two important predictions: First: living fossils must exist (true) and evolution of the form must be accompanied with myriads of malformations, of cripple fossils (false). That is why the Evolutionary Theory is false.

2.3 Grammars

We know how to output all possible Java classes were time and memory unlimited. Our purpose to break this hell of situation is to experiment with grammars, sets of replacement rules, and to choose problems one step afar from the solution because, as we learned in our previous volume, evolution is very useful in a case like that. As usually, we start with simply exercises with the hope of gaining experience.

22 ***Generating*** **`a + b`.** *We want to generate a class that adds two integers like that of program* `R22a AddtwoIntegers`*. A generator that outputs this class and nothing else can be seen in* `program R22b AddTwoIntegersGP`*.*

23 ***Exercise.*** *Run the previous programs and play with their code.*

24 ***Exercise.*** *Reuse the code of the previous program to compose a generator for the subtraction of two integers. Our answer can be seen in* `program R24 SubtractTwoIntegersGP`*.*

The same changes but with different arguments serve to generate a class that multiplies two integers or a class that divides an integer by another different than zero. That is why the changes can be done automatically with a grammar.

25 ***Exercise.*** *Compose a generator of Java classes that matches an arithmetic operation randomly chosen from* ***+, -, * or /*** *on two integers. Our answer can be seen in* `program R25 ArithmeticOperationsGP`.

26 ***How difficult is debugging!.*** *The Author finds the debugging of this type of programs to be extremely difficult. So, a trick is proposed to see how it can help: to totally virtualize the program under generation that it could be debugged apart in an exclusive file. A first example can be found in program* `R26a TotalVirtualization`. *A second example is program* `R26b ArithmeticOperationsGP2` *and the virtual code can be run on its own file as shown by program* `R26c ArithmeticOperationsGP2Virtual`. *The Author felt that this trick was helpful.*

27 ***What is this good for?*** *Our technology allows us to generate classes and to run them with output to the console. So, our machinery can be used for interactive tasks say, in art, music or painting, where a human being is in charge of the selection procedure and of the fitness function.*

But, we as yet do not know how to pass the output of virtual classes to a list to next be automatically evaluated as a fitness function that evolution could be implemented. So, let us make a try:

28 ***Exercise: Can we get and reuse the output of a method of a virtual class?*** *We already know that a method can be called from inside a virtual class if it belongs in that class. But a virtual class seemed to be totally isolated from its mother application: it listens to no voice and outputs nothing to it. What does happen if communication is tried out from inside a virtual universe? The Author found a negative answer.*

2.4 Using the hard disk

According to our experience, a virtual class is isolated from its outer space with the exception of the possibility of printing information from a method to the console. Does this mean that GP turns into a vain dream? No, by no means.

29 ***Idea: redirect the output to a file that must be saved to disk.*** *If we write the output of a virtual class to a file, we can read it and analyze it automatically to*

compute the fitness function associated by an evolutionary process. How to redirect the output to a file can be seen in program `R29a RedirectSystemOut`. *This program creates the file* `out.txt` *that in NetBeans can be seen immediately and be opened directly. However, over Eclipse the file is invisible but can be automatically opened. Every time that this program is run, the output file is overwritten and previous content erased. How to rescue the content of output file* `out.txt` *is shown by program* `R29b ReadAFile`.

30 ***Exercise.*** *Test whether or not the standard output can be redirected to a file from a virtual class. Use a virtual class that adds two Integers. Our answer in program* `R30 AddTwoIntegersGP2`. *Possible trouble: to save a file takes time so, one must insert a delay in within the saving and reading procedures. Else, one must use* `runnable`.

31 ***Exercise.*** *Compose a program that runs a virtual class, redirects its output to a file, reads the output and prints it to the console. Our answer is in program* `R31 AddTwoIntegersGP3`.

We can solve now the easiest task of evolution: to match an operation that has been taken from some few options.

32 ***Exercise.*** *Reuse previous programs to automatically generate a class that matches the outputs of a given arithmetic operation on two integers. Our answer can be seen in the program* `R33 ArithmeticOperationsGP3`.

Let us pass to a more serious task: we will use evolution to work out an approximation to the cumulative standard normal distribution and to report the output as a Java class that shall be ready to be used. The first step is to provide exact values of the cumulative function. So, we need to recall from `Vol V The Scientific Method numeral E127` the Simpson's rule to do that:

The area under a curve $f(x)$ in within $a = x_i$ and $b = x_{i+2}$ can be approximated by

$$\int_{x_i}^{x_{i+2}} p(x)dx = \frac{h}{3}[f(x_i) + 4f(x_{i+1}) + f(x_{i+2})]$$

This approximation will define what we call the true values of the distribution.

33 ***Tunning the Simpson's rule.*** *Program* `R32 TrapezoidalVsSimpson` *compares the outputs of the Trapezoidal and Simpson's rules in regard with the values of the cumulative function of the standard bell.*

34 ***Exercise.*** *Run the previous program and play with the code. Play enough to agree else disagree with the Author: Simpson's rule has been programmed perfectly.*

We can use now the Simpson's rule to define the fitness of algebraic expressions to approximate the cumulative function of the standard bell.

35 ***Evolution for software development.*** *Program* `R35a CumulativeNormalDistribution` *uses evolution to fit the parameters of an algebraic function to approximate the cumulative function of the standard normal distribution. Our next intention is to pack the found approximation into a Java class that must be ready to be used. So, Program* `R35b ApproxCumStandardBell` *shows how that class shall look like. To end, we glue everything: Program* `R35c CumulativeNormalDistribution2` *shows how evolution is used to generate a ready to be used class that computes the demanded approximation.*

2.5 Javac, the primary compiler

Let us notice that we have used two strategies to use evolution to synthesize software.

36 ***Two strategies.***

The first strategy runs evolution in the space of parameters as an ordinary genetic algorithm and the output is used to synthesize a Java class with the demanded function. The second runs evolution in the space of programs, of Java classes. In the first approach, genetic programming is an addendum, an improvement. In the second, genetic programming is something constitutive, essential. What approach is better? The first is easier but the second sounds more powerful. Nevertheless, the second looks also more complicated. Thus, we have a reason to explore other technologies to see whether or not the second methodology can be made simpler. In this regard let us explore how to use `Javac`, the primary Java complier, to see how it can help us to run evolution in the space of programs.

37 ***Two compilers.*** *We have used the* `RuntimeCompiler` *and plan to use* `javac`*. What is the the difference? The answer is understandable if we review the process of software creation and execution: the User creates a program* `myJavaprogram.java` *using Java, which is a high level language that cannot be understood by microprocessors. Another program,* `javac`*, translates* `myJavaprogram.java` *into myJavaprogram.class which is a program in* ***bytecode****, the Java machine code. So,* `javac` *is called a* ***compiler****. The* `bytecode`

is executed by the `JVM`, a simulated microprocessor that is platform independent, the Java Virtual Machine. To execute or run a piece of bytecode, it must be translated to machine language, that is the language that is understood by the hardware, the CPU (Central Processor Unit). This translation is platform and hardware dependent. The time when a program is run is called ***runtime****. The User has the possibility to program a call to an instance of a compiler that is called `RuntimeCompiler` and that operates at runtime and that is totally independent and with separated rules from `javac`, the primary compiler. Thus, a string that encodes for a program is for `javac` just a string but for the `RuntimeCompiler` it can be a Java program. According to our experience, the `RuntimeCompiler` is very restrictive. By contrast, we imagine that `javac` is more powerful. So, we hope that it could be of great help for clean cut GP, in which Java classes evolve directly.*

The first task is to run a Java class from another Java program.

38 ***Javac in action.*** *The code in `Program R37 JavacTest` shows how to run a program from another one thanks to `javac`.*

39 ***Exercise.*** *Run the previous program and play with the code.*

40 ***Exercise.*** *Use the file manager of your System to verify the following points that are IDE dependent. Suppose we have a class in the package `Programs` of the `/src/` folder. Then*

For Eclipse:

First: Eclipse creates a folder for each project and inside it there are other folders with specific functions. So, Java programs, with `.java` suffix, are located in a subfolder of `src: src/Programs`. When Eclipse runs a `.java` program, the compiled class `.class` is put in the `bin/Programs` folder. Second: when a `.java` program is run with `javac` from inside another master `.java` program, the corresponding `.class` file is put by demand in the same folder as the master `.java` program. To make experiments, you must delete files: the `.class` files in `build/classes/Programs` can be deleted at anytime that they will be restored on new compilation. The files in the working folder of the Eclipse project can be deleted using right clicking on the tree of the project that appears under the tab `Projects` which can be made visible under the menu `Window`. All files created from inside a program can be deleted from the working folder at any time that they will be restored on compilation of the master program.

For Netbeans:

First: NetBeans creates a folder for each project and inside it there are other folders with specific functions. So, Java programs, with `.java` *suffix, are located in a subfolder of* `src`*:* `src/Programs`*. When NetBeans runs a* `.java` *program, the compiled class* `.class` *is put in a subfolder of* `build` *as follows:* `build/classes/Programs`*. Second: when a* `.java` *program is run with* `javac` *from inside another master* `.java` *program, the corresponding* `.class` *file is put by demand in the same folder as the master* `.java` *program. To make experiments, you must delete files: the* `.class` *files in* `build/classes/Programs` *can be deleted at anytime that they will be restored on new compilation. The files in the working folder of the NetBeans project can be deleted using right clicking on the tree of the project that appears under the tab* `Projects` *which can be made visible under the menu* `Window`*. All files created from inside a program can be deleted from the working folder at any time that they will be restored on compilation of the master program.*

41 ***In favor of pragmatism.*** *In the previous section we found an algebraic approximation to the cumulative distribution of the Gauss Bell. We ran evolution in the space of parameters of the problem to find an optimal solution to next use the output to generate a Java class. GP resulted to be an addendum to the genetic algorithm. Can we aspire now that we have* `javac` *to make clean cut GP, full colored evolution in the space of Java classes and not in the space of parameters? No, we can not: if we hand over to evolution the string that encodes for a Java class, we will need cosmological time to get results, because such a string has too many places to be mutated and recombined. So, our strategy always will be to find suitable restrictions that solve the problem in seconds (our problems are chosen with that aim in mind). For instance, our algebraic fitting hangs on parameters and so it is useless to involve in the evolutionary process other things apart from them. That is why we expect our work with* `javac` *to produce disguised genetic algorithms.*

42 ***Exercise.*** *Make your best to reuse program* `R35c CumulativeNormalDistribution2 + javac` *to compose a class that solves the same problem but runs evolution in the space of Java classes. Our answer consists of three programs. The first program* `R42a ApproxCumStandardBell2` *presents the solution that we want to see and that contains the template that will be given to evolve. Its output is a vector with approximate values of the cumulative function. The second program* `R42b JavacTest2` *runs the first one, reads its output and prints it. Thus, we know how to run a program and how to recover its output. The third is the master program* `R42c CumulativeNormalDistribution3` *that runs evolution in a space that is a mixture of parameters and Java classes.*

43 ***Example.*** *Let us translate the program developed for the previous exercise to Functional Java. This can be seen in program* `R43 CumulativeNormalDistribution4`*.*

44 ***Exercise.*** *Improve the translation of our previous program* `R43 CumulativeNormalDistribution4` *to functional Java which was not complete. Our answer can be seen in program* `R44 CumulativeNormalDistribution5`*. It reformulates the density function of the Z distribution and some other functions into lambda expressions.*

45 ***Challenge.*** *The previous program was mounted on top of* `javac` + `RuntimeComplier`*, the primary Java compiler. Adapt it to run on top of the* `RuntimeComplier` *alone. Express your opinion about the advantages of each method.*

46 ***Challenge.*** *Work out the GP of regression, linear, polynomial or ad hoc: for a data set, you must develop a program that outputs another program that computes a prediction given by a regression model.*

2.6 Conclusion

We have learned how to use two Java compilers, `javac` and `RuntimeComplier` that can be used to go from strings to Java classes that can be compiled and run. The bridge that joins the gap is the wonderful world of grammars, sets of substitution rules of strings. We have shown how to make the GP of problems that can be solved by evolution on a family of Java Classes that differ by the values of some parameters.

Chapter 3

Universal computing

`Every kind of problems`

47 ***Introduction and purpose.*** *We know how to make GP when a problem can be solved by evolution in a space of Java classes that differ by the value of some parameters. But this parametric type of evolution is rather a tiny part of the problems that one must face. So, the generality of problems that a developer must face to is known as* `Universal Computing`*. Now, our aim is to show along a simple example how to invent a grammar to attack a given problem that belongs in that immense source of problems. In our opinion, all the rest is more of the same game. Nevertheless, the difficulty of problems, the degree of generality and the ensuing need for creativity may amount to thousands of PhD dissertations and beyond.*

3.1 Our task

Let us battle with the GP of the mean.

48 ***The full desired code shall be equivalent to the following implementation of the mean:***

```
//Program  R48 MeanVarOfData

package Programs;

public class MeanVarOfData {

    public static void main(String args[]) {
        //Declaration and assignment of the data array
```

```
        int Data[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};
        // Report  data
        System.out.println("Data are");
        for (int i = 0; i < Data.length; i++) {
            System.out.print(Data[i]);
            System.out.println();
        }
        //   Let us calculate the mean
        double sum = 0;
        int n = Data.length;
        for (int i = 0; i < n; i++) {
            sum = sum + Data[i];
        }
        double mean = sum / n;
        System.out.println("Sum = " + sum);
        System.out.println("Mean = " + mean);
        //    Let us calculate the variance
        double sum2 = 0;
        for (int i = 0; i < n; i++) {
            sum2 = sum2 + (Data[i] - mean) * (Data[i] - mean);
        }
        double var = sum2 / (n - 1);
        System.out.println("Variance = " + var);
    }

}//End of class R48 MeanVarOfData
```

The great task is naturally divided in subtasks: to report data, to calculate the mean and to calculate the variance. Once a subtask is done, it might be frozen and used to go further. So, a divide and conquer strategy is what we must follow.

3.2 Evolving strings

We verified in the previous chapter that all Java programs can be produced by combining chars but that a price in cosmological time must be paid. Let us see how looks evolution on appropriate strings, that must include the reserved words of the language.

In regard with the posed task, the simplest subtask is to report the number of entries in the given array of data. So, we must synthesize in first place a program

that must be equivalent, isofunctional, to the following one:

49 ***Reporting data.***

```
//Program  R??? NumberOfData
//This program reports the number of data  in an array.

package Programs;

public class NumberOfData {

    private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};

    private static void NumberOfData()
    {
        System.out.println("Number of entries = " + DATA.length);
    }

    public static void main(String args[]) {

        NumberOfData();

    }
}//End of  Program   NumberOfData
```

To that aim, we will use the following pattern as starting template:

```
//Program  R??? MeanVarianceGP
//This program reports the number of data  in an aarray.

package Programs;

public class MeanVarianceGP {

    private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};

    private static void NumberOfData()
    {
        //AddCodeHere
    }
```

```
    public static void main(String args[]) {

        NumberOfData();

    }
}//End of  Program   MeanVarianceGP
```

To add and evolve code, we will resort to tinkering with concatenation of diverse substrings from among the following *genetic code* that associates strings to strings:

```
A -> System.out.println("Number of entries = " +
B -> DATA.length;
C -> )
D -> ;
```

This setting has been chosen with the intention of posing the problem at one step off the solution. In fact, the solution is given by the chromosome

```
ABCD ->  System.out.println("Number of entries = " + DATA.length);
```

Now, how will we know when we are done? We will use the main compiler `javac` and will inject a fitness criterion which is given by the correct output:

```
 Number of entries = 11
```

Our GP master can found in program `R48 MeanVarianceGP.` Created `.java` classes have as name `MeanVarianceTry + i` where `i` is an index, a number, ranging from 1 to 7.

50 ***Exercise.*** *Run the previous program and play with the code. Play enough to agree else disagree with the Author and bring forward your own appreciations:*

51 ***Our conclusions:***

1. We have proved that GP on top of evolution of appropriately chosen strings is possible and that eventually can produce fruits in some few minutes instead of eons.

2. This is possible because we formulated a problem that is one step away from the solution.

3. But, certainly, a combinatorial explosion expects everyone that bolds to go against the slightest generalization or complication.

4. Besides, our approach is extremely inefficient because most synthesized programs are not accepted by the compiler and produce null output.

5. We also perceived that randomness would outperform evolution: it is faster and cheaper. One reason is that evolution is really void: the fitness criterion is `all else nothing` so, it lacks the possibility to gather small change.

6. Nevertheless, by dividing the task in various subtasks we are informally given the fitness criterion to have gray values other than black else white.

7. Therefore, an intelligent implementation of GP is not in the world of programs but in the realm of tasks that are divided in subtasks.

8. All this seems to imply that the cheapest way to make GP is by implementing an artificial intelligence before everything else.

9. On top of all this, a necessary turn around the hard disc is extremely time consuming.

10. So, as `javac` as the `RuntimeCompiler` runs at disadvantage with respect to those languages, say, Lisp, JavaScript, Scala, that do not need the hard disc to convert strings to code.

11. While we are ready to recognize the strength of other languages, our responsibility here is to explore what Java could be good for. To begin with and up to present work, everything that we do or synthesize is readily understood. By contrast, if one needs GP with totally incomprehensible code, the right choice is Scala. If you want the power of Scala + comprehension, use Groovy.

3.3 Grammars again and again

To somehow battle the inefficiency of previous methodology let us prepare the terrain for the evolution on grammars.

52 ***Our motivation:*** *Can we use grammars to attack GP in order to diminish the fraction of inviable programs?*

53 ***Example.*** *Let us propose a template and a suitable grammar to generate a program that reports the number of data in an array. The desired program reads:*

```
//Program  R??? MeanVarianceTarget
//This program reports the number of data  in an aarray.

package Programs;

public class MeanVarianceTarget {


 private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};


    private static void NumberOfData()
    {
        System.out.println( "Number of entries = " + DATA.length);
    }

    public static void main(String args[]) {

        NumberOfData();

    }
}//End of  Program   MeanVarianceTarget
```

We propose to begin with the following template:

```
 public class LengthTemplate {

    private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5



    private static void NumberOfData()
    {
        System.out.println( NAMEOFVARIABLE + VALUEOFVARIABLE);
    }

    public static void main(String args[]) {
```

```
        NumberOfData();

    }
}
```

This template can be transformed into the desired code by means of the following grammar:

```
"NAMEOFVARIABLE"  -> "\"Number of entries = \""
"VALUEOFVARIABLE" -> "DATA.length"
```

54 ***Example.*** *Program* `R54 MVGrammar` *uses the* `RuntimeCompiler` *to test the previous claim.*

55 ***Exercise.*** *Device the template and the grammar to synthesize the mean of an array of numeric data. Develop the ensuing testing code with the* `RuntimeCompiler`*.* *Answer*

56 ***Criticism:*** *Used grammars do not generate recursive calls: each instruction is executed once and that is all to it. So, this way of working cannot support evolution.*

3.4 Functions

We have implemented grammars inside a HashMap and this is the perfect option for grammars whose elements are intended to be applied just once. But for complex affairs we need the possibility to make recurrent applications of the same rules. That is why we are going to examine functions that are specially targeted for composition.

57 ***Example.*** *Our aim is to synthesize a program that accepts a data array and outputs their number of data and the mean such as* `Program R55a MeanVarianceTarget2`*. We begin with a rather simple template and concoct a grammar that we pack into a set of functions. This is shown in Program* `R57 MVGrammar3` *in which functions + grammars are put to operate together.*

58 ***Exercise.*** *Use BiFunctions to rewrite the previous program. Our answer can be seen in program* `R58 MVGrammar4`*.*

59 ***Exercise.*** *The replace operator over strings functions like this:* `t.replace(a, b)`*, where we see 3 arguments. So, use TriFunctions to rewrite the previous program. Our answer can be seen in program* `R59 MVGrammar5`*.*

60 ***Example.*** *Program* `R60 MVGrammar6` *shows how to solve the same task (synthesis of the code for reporting the number of data and their mean) but beginning from more elementary items and trying to use functions.*

61 ***Exercise.*** *Run the previous program and play with the code.*

62 ***Example.*** *Program* `R62 MeanVarianceGP2` *shows how to use evolution + functions to synthesize the code to print the number of data. Warning: fitness is all else nothing so evolution is really inoperative. Our purpose is just to see how to merge functions and evolution.*

63 ***Exercise.*** *Run the previous program and play with the code.*

3.5 GP with functions

We will face to the problem of implementing the possibility of gathering small change along the use of grammars + functions.

64 ***Our model***

We are assuming with respect to GP that:

1. The code under synthesis is itself shielded from selection. So, fitness is associated to just the output of the tested code.

2. A large program is given a fitness with varying degree by diving it into various subunits whose fitness could be all else nothing.

3. By looking at method `applyFunctions()` of program `R60 MVGrammar6` one can see that the elaboration of the template by means of functions is composed of blocks with varying number of functions from 3 to 7. Each block is responsible for one subunit of code. That is why we work with the following correspondence: a call to a function is like a DNA codon, and a block of functions is like a chromosome. A string of chromosomes is a genome or an individual. So, we recover the usual genetic structure: codon, chromosome, genome.

4. There are 37 tokens at all but blocks use from 6 to 12 tokens. This means that instead of working with the complete set of tokens, we can choose for each chromosome a varying subset of tokens from 6 to 12.

5. We will reuse the code of Program `Q100 BinaryAdder3LH3` that we use in Vol XVI for the synthesis of a BinaryAdder because that program has a similar structure to the one we need.

65 ***Exercise.*** *We have proved that the calculus of the mean can be done on two functions, replacement and concatenation. But a grammar consists of replacement rules alone. Do we need to change the word grammar by, say, extended grammar to refer to this set of two functions?* *Answer*

66 ***Example.*** *Program `R66 MeanVarianceGP3` attempts at using `GP = evolution of grammars` with functions and the usual genetic structure to synthesize the code for the number of data and the mean.*

67 ***Exercise.*** *Run the previous program and play with the code. Play enough to agree else disagree with the Author: the program is pretty useless.*

68 ***Improving the approach.***

In our previous program the template was introduced as an ordinary token and exactly as the last one. This causes the template to be almost no touched at all, contrary to what is seen in program `R60 MVGrammar6`. In fact, one can see that each block is indeed responsible for a subunit of code but that moreover each block has the following structure:

A string is produced as result of chaining concatenations and replacements of various tokens that are all different than the template and at last the resultant string is incorporated by a replacement inside the template. Example:

The block that is responsible for the line of code:

```
N = DATA.length;
```

is the following:

```
String s7 = conc.apply(tokens.get(17), tokens.get(5));
s7 = conc.apply(s7, "        //addCodeHere44");
System.out.println("s7 = " + s7 );
template = inTReplaceAbyB.apply(template, "//addCodeHere4", s7);
```

And it is equivalent to a chained replacement inside the template:

```
template = inTReplaceAbyB.apply
           (template, "//addCodeHere4"
              , conc.apply(
                           conc.apply(tokens.get(17), tokens.get(5))
                                 , "        //addCodeHere44"
                          )
           );
```

So, the generic structure of each blocks is:

```
 template = inTReplaceAbyB.apply(template, "//addCodeHere", s);
```

where `s` is a compound string and the template does not appear in its construction.

69 ***Exercise.*** *Implement this aforementioned generic structure of each block to see what happens. Beware: delete the template from the list of ordinary tokens, improve the fitness measure that it could take on values different than all or nothing, keep in mind that the fitness function must be an increasing function of the matching extension of the output of each generated class. Our answer can be seen in program* `R69 MeanVarianceGP4`*. This program was run during 1200 generations on 7 individuals: no error was detected and apart from null nothing was produced.*

70 ***Exercise.*** *Run the previous program and play with the code.*

71 ***Challenge.*** *Run the previous program on one thousand individuals and during one million generations. If you like this game, it would be a good idea to run the program on a free computing service of the web.*

72 ***Graduation.*** *Implement the GP of the variance.*

73 ***Challenge.*** *Grammars have allowed us to do some simple exercises in GP. How much are we left to do? To answer this question prove that independent grammars cannot be listed: for every list you try, some GP grammar will be lacked. And if you include it, other will appear and so on for ever and ever. So, grammars will always provide for new work.*

3.6 Conclusion

We have shown on some problems that are one step away from being solved that Easy Java Genetic Programming is feasible. Preparing the path for those that want to go one step farther, we have shown how to develop the appropriate software. Nevertheless we stumbled with one problem: for one thousand generations no result was found so a doubt remains: is the program correctly designed? This question shows that the mathematical science of software verification is welcome. Else, that we need to dive into the world of supercomputers.

Chapter 4

The JVM (Java Virtual machine)

`Behind the scenes`

74 ***Introduction and purpose.*** *Everyone in our community has the right to become an expert so, we have the responsibility to motivate every brain. That is why we formulate here an invitation to examine for the sake of GP the entrails of the JVM (Java Virtual Machine). For those that do not fear combinatorial explosions, this might serve, for instance, to optimize an extant code for high velocity, a virtue that all highly used pieces of code must have. For those that aim at a strong Java formation we present here some necessary information and programs. For those that are pursuing a PhD dissertation, we present enough material to make them feel more educated and secure and so they will be able to read very specific and sophisticated articles.*

4.1 How a Computer functions

A computer is a machine that spins around the processing of information, an operation that is done by the CPU, the Central Processing Unit (Woodford, [101] 2016).

75 ***The functions of the CPU.***

The task of the CPU is to execute lists of instructions that are compounded of control orders and the corresponding data (Reference, [71] 2017). For each instruction, the CPU must:

1. Read the next instruction (code + data) from memory, specifically from RAM (Random Access Memory, a memory that can be accessed directly at any address).

2. Decide if the instruction is related to arithmetic (3 + 4) else to logic (if $a > b$ do this else do that) and delegate the task to the corresponding service.

3. Execute the posed task.

4. Report the output to the console or save it to RAM or...

76 *Heterogeneity problem*

There are infinitely many ways of implementing a CPU, infinitely many forms of programming it and infinitely many forms of filling the communication gap in within the human being and the machine. What to do?

A very smart solution has been to design a huge but harmonious, coherent set of programs that ease the communication of the man with the machine in regard with the most varied tasks. They are called **operating systems**.

Next, we have high level programming languages that allow to program the CPU in human friendly terms: they focus on the human perspective of tasks. Suppose we have a high level language like Basic, C or Java. Given a high level language, the usual solution to the heterogeneity problem is to design a particular solution for each operating system, which in its turn must develop a particular solution for each hardware or type of computer. By contrast, the solution adopted by Java is rooted on the **JVM (Java Virtual Machine)** (Joseraj [39] 2016): this is a simulated CPU that executes directly the Java code. From the stand of the User, the JVM has two interfaces. The first faces to the User: it is responsible for the work of `Javac`, the main compiler. This translates the program that has been written in Java to bytecode. This interface is the very same for all operating systems and all types of computers and this explains the independence of Java with respect to platforms and processing units. The second interface of the JVM is the **executor machine** that takes the bytecode, releases a translation appropriate for the specific operating system which in its turn uses those instructions to manage the hardware, the specific computer where the bytecode is run. That hardware produces and output that is shown to the console or redirected to a file. All this is triggered when one calls `java`, the executor that converts bytecode into actions .

The executor machine functions lie this: if a method is called for the first time, it is translated by JIT (Just In Time compiler) to native code, i.e., code that both the operational system and the hardware can understand. The resultant code is then executed. If the same method has not been changed and is invoked again, the JVM calls the JIT compiled file that is executed directly.

We will learn the fundamentals of bytecode to estimate how much it takes to make GP on the JVM.

4.2 Fundamentals about .class files

Let us see some generalities about files and then we pass to `.class` files that contain the bytecode.

77 ***Bits and bytes.***

A file is a stream of bits that contains ones and zeros. These acquire meaning by a process of interpretation. To ease this task, bits are associated in bytes which are strings of 8 bits. A byte has the capacity to encode $2^8 = 256$ numbers that if are assumed to be positive integers are those from from 0 to 255, included. Because 16 * 16 = 256, the natural way to represent a byte is by its hexadecimal (base 16) equivalent. Let us learn how to do this: suppose we have the byte 10101111. This can be separated in two words of 4 bits: 1010 1111. The first word represents the decimal number

$1010 = (((1 \times 2 + 0) \times 2 + 1) \times 2 + 0) = 10$

while the second represents 15. Now, numbers in base 16 are:

$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a(10), b(11), c(12), d(13), e(14), f(15)$.

Hence, the binary number 10101111 is represented as the hexadecimal number af. In effect: the binary number 10101111 and the hexadecinal number af both represent the same decimal number 175. This can be done automatically with Java:

78 ***Notation: to specify that a number comes in hexadecimal format when confusion may arise the prefix `0x` is added. Example: `0xab` is an hexadecimal number while `ab` is a string.***

A `.class` file is produced by the main Java compiler `javac` working over a `.java` file.

79 ***A non semantic `.class` file reader is presented by program `R79 ClassReader`.***

80 ***Exercise.*** *Run the previous program and play with the code. Verify that the first four bytes of a `.class` file contain the magic word `cafebabe`, the distinctive of `.class` files. Verify that in regard with positive outputs every text in the same line has the same number in binary, base 10 and base 16. For instance, the number 23 in base ten is represented in base 16 as 17 because* $17 = 1 \times 16 + 7 = 23$. *The decimal number `23` is represented in binary by `10111`. Beyond positive integers a `.class` file has diverse types of information with different encodings: say, a number of type double demands more memory than a number of type int. That is why the type of each variable must always be declared in the bytecode.*

In short: No one likes to see zeros and ones, the native bit content of a file. By contrast, the int and hexadecimal content are perfectly legible. Nevertheless, human developers do not like to work with numbers. Rather, they prefer a human friendly representation, one faithful to the semantic content and that works with diverse encodings for the different types of information. Java provides one that includes **opcodes** (codes for elementary operations) or **mnemonics**. Executing `javap -c` will print the whole bytecode content of the file.

81 ***Opcode equivalents of bytecode programs can be seen with `javap`. Its main available options are the following:***

```
-help  --help  -?        Print this usage message
-version                 Version information
-v  -verbose             Print additional information
-l                       Print line number and local variable tables
-public                  Show only public classes and members
-protected               Show protected/public classes and members
-package                 Show package/protected/public classes
                         and members (default)
-p  -private             Show all classes and members
-c                       Disassemble the code
-s                       Print internal type signatures
-sysinfo                 Show system info (path, size, date, MD5 hash
                         of class being processed
-constants               Show final constants
-classpath <path>        Specify where to find user class files
-cp <path>               Specify where to find user class files
-bootclasspath <path>    Override location of bootstrap class files
```

82 ***Exercise.*** *Modify program `R37 JavacTest` that it could show up the opcode in a clean way. Play with the diverse compiling options. Our answer can be found in program `R82 JavacTest3`.*

83 ***Understanding a JVM program.***

Let us consider the Java expression

```
int a = 2 + 5;
```

For a Java user it renders a = 7 and that is all to it. Simple and clear. That is why Java is a high level language. For a low level language like bytecode, things

are quite different. To get a flavor of that world, let us imagine that we have a computer made of wood: there is a 2 sculptured in wood, a 5, a box with that sort of things, a great table that serves as workbench, another box with machines, one of which is an adder. In a lab like this one might run a program as the following

```
Fetch the 3
fetch the 5
fetch the adder machine
plug the 3 and the 5 into the adder
read the returned value
report the returned value
return back the 3 to its box
return the 5 back to its box
return the adder to its box
clean the workbench and
get ready for a new operation
```

Such mechanistic instructions are proper of **assembly languages** (low level languages that deal directly with the hardware computing machine) and bytecode is one of them although for a virtual or abstract machine. So, let us be prepared for very detailed and technical stuff whereof only the tip of the iceberg will be touched in the present introduction.

Now, there are many types of computers. Which is the type of computer of our JVM?

84 ***The JVM is a stack machine.***

The first ingredient that we need to understand the JVM is the notation. The flavor is gotten with simple examples. Say, 2 + 3 is denoted in **Polish notation** as `add, 2, 3` (this is actually a program for a machine that uses that notation). The same 2 + 3 will be denoted as `2, 3, add` in **Reverse Polish notation** which is the notation adopted by the JVM: if a program is written as usually in lines along a page, variables and arguments appear always on top of the acting operations.

A **stack machine** reads a program instruction by instruction and consequently updates a stack of data. Thus, `2, 3, add` is executed as follows:

Input	2	3	add
stack	2	3	5
		2	

The execution can be paraphrased as follows:

```
- push 2 into the stack
- push 3
- read and remove the two values from the stack,
  add them and push the result
  into the stack for further operations.
```

In general, two active operations can be executed over a stack: **push**, which adds an element at the uppermost place, and **pop**, which removes the element from the top: Adding and removing are operations that are always executed at the top of the stack. That is why the last to come in is the first to go out, as in an ordinary stack of books that are moved and removed one by one. Hence, we have the official definition of a **stack machine**: is a computer that uses a Last-in, First-out stack to hold short-lived temporary values (ReelLearning, [70] 2012, [96] 2017c; Arhipov [3] 2011).

85 ***Exercise.*** *Show on a table the execution over a stack machine of the compound operation (2 + 3) * 7 + 11. Add a verbalization of the execution.* Answer

Software and evolution mean the very same thing: reuse. So, what to do if the machine destroys everything inside the stack over the computation? The solution is to keep apart declared constants and variables. That is why a stack machine has a list of constants, the **constant pool**, and another of variables, the **LocalVariableTable**. So, an object can be stored in a local variable and then loaded from that local variable into the stack. If constants are numbers 1, 2, 3, 4, 5 no pool is necessary, they go as a suffix to the reserved word `iconst`, say, `icost_3` . If the number of variables is less than 3, no table is necessary, say, the value of variable 2 will be loaded with `aload_2`.

We need a dictionary that explains some very usual terms. For the JVM a prefix *i* indicates that the operation is over integers and the prefix `a` refers to an arbitrary object.

86 ***My first opcode dictionary.***

```
iconst_n : Push int constant. Integers n = 0..5.
invokestatic  #3 : call a static method
astore_n: store object in local variable n : n = 0..3
bipush        8 : Push a signed byte.
aload_n : Load object from local variable n : n = 0..3
getstatic     #5   : Push a static object's variable.
invokevirtual #6   : call any other method
ldc           #7   : Push a single word constant.
return : return from method with nothing
```

87 ***References.*** *A full opcode dictionary can be found in (homepages, [34] 2017).Items can be organized by name, by function and by number. Other useful information is presented by Wikipedia ([97] 2017d). Since Authors might slightly contradict one another, the original source turns to be necessary to decide a fine detail affair (Oracle, [61] 2015).*

88 ***Challenge.*** *Is there something better than a HashMap to encode a dictionary?*

89 ***Criticism.*** *Bytecode is far from being human friendly. But it is a extremely machine-friendly. So, what to do? One option is to practice day and night. Another is to invent a parallel human friendly language (Engel, [24] 1999; Lilac, [49] 2017). Or much better, if you are interested in GP, devise a completely new brand DSL (Domain Specific Language) with all the virtues of bytecode but without any of its defects. In fact, Bytecode consists of three modules that work coherently so, evolution must run over coherent changes in various parts of each program. So, it is not very evolutionary. The solution has been to invent a language in which all mutations are viable for instance Slash (Arturadiv [4], 2011) and Concatenative Languages ([59], 2017). Moral: if something dislikes you, turn off your mouth and instead turn on your mind that it could work in a solution.*

Among all those possibilities, our present choice has been to work on Bytecode directly. So, let us continue.

90 ***Example.*** *Let us make a Java program to calculate the expression* `(2 + 3) * 7 + 11`*. Next, let us connect it to master program* `JavacTest3` *to unveil its bytecode content.*

The program is the following, in which some line numbers have been inserted:

```
/*
 * Program R90 SimpleArithmetic
 * Executes an arithmetic expression
 * in various steps to watch the function of the JVM as a stack machine.
 */
package Programs;

/**
 *
 *
 */
public class SimpleArithmetic {
```

```
    public static void main(String[] args)
    {
        (16)int i1 = 2;
        (17)int i2 = 9;
        (18)int result0 = i1 + i2;
        (19)int i3 = 7;
        (20)int result1 = result0 * i3;
        (21)int i4 = 13;
        (22)int result2 = result1 + i4;
        (23)System.out.println("Result of (" + i1 + " + " + i2 + ") * "
" + " + i4 + " = " + result2);
    }(24)


}//End of Program R90 SimpleArithmetic
```

The associated bytecode has two parts, the first deals with arithmetic the second with the strings that respond for printing the output. The arithmetic part is the following:

```
 0: iconst_2
         1: istore_1
         2: bipush        9
         4: istore_2
         5: iload_1
         6: iload_2
         7: iadd
         8: istore_3
         9: bipush        7
        11: istore        4
        13: iload_3
        14: iload         4
        16: imul
        17: istore        5
        19: bipush        13
        21: istore        6
        23: iload         5
        25: iload         6
        27: iadd
```

```
        28: istore        7
        30: getstatic     #2                  // Field
java/lang/System.out:Ljava/io/PrintStream;
        94: return
```

and the LineNumberTable is this:

```
        line 16: 0
        line 17: 2
        line 18: 5
        line 19: 9
        line 20: 13
        line 21: 19
        line 22: 23
        line 23: 30
        line 24: 94
```

This allows us to propose the following equivalence among the Java code and the corresponding starting instruction of the Java bytecode:

```
int i1 = 2;  ->iconst_2
int i2 = 9; ->bipush        9
int result0 = i1 + i2; ->iload_1
int i3 = 7; ->bipush        7
int result1 = result0 * i3; ->iload_3
int i4 = 13; ->bipush        13
int result2 = result1 + i4; ->iload         5
System.out.println.... ->getstatic     #2

    // Field java/lang/System.out:Ljava/io/PrintStream;
} ->return
```

91 ***Exercise.*** *Run previous programs and play with their code to verify our assertions.*

92 ***Modularity.*** *Human developers demand from code to be structured in order to ease its understanding and maintenance. Java assumes this and allows methods to be defined at no cost. Modularity is accepted by the JVM without too much problem: for each method Java produces the same output as for a class: code, constant pool and variable table. This ensemble is called a* ***frame****. Frame of private methods are made visible with the parameter* `-p`*.*

93 ***Example.*** *If we run program* `R21a addTwoIntegers` *from inside* `JavacTest3`*, we find the frame associated to the method* `addTwoIntegers` *as follows:*

The method `addTwoIntegers` is called from line 27 of the class `AddTwoIntegers`, whose frame is the following:

```
private static java.lang.Integer addTwoIntegers(java.lang.Integer,
java.lang.Integer);
    descriptor: (Ljava/lang/Integer;Ljava/lang/Integer;)Ljava/lang/Inte
    flags: ACC_PRIVATE, ACC_STATIC
    Code:
      stack=2, locals=2, args_size=2
         0: aload_0
         1: invokevirtual #2                  // Method
java/lang/Integer.intValue:()I
         4: aload_1
         5: invokevirtual #2                  // Method
java/lang/Integer.intValue:()I
         8: iadd
         9: invokestatic  #3                  // Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
        12: areturn
      LineNumberTable:
        line 17: 0
```

We see that the method deals with two objects that next are identified as integer variables and added. The result is identified as integer and returned.

94 ***Recursiveness can produce surprising results.*** *If we call* `JavacTest3` *from inside* `JavacTest3`*, one can see that the recurrent calls creates recurrent representations of JavacTest3. Nevertheless, that is not always the case:*

95 ***Exercise.*** *Where and how is recursiveness encoded in the program below* `R95 RecursiveSum`*?*

A recursive encoding of the sum of the first n integers is presented in program `R95 RecursiveSum`:

```
/*
 * Program R95 RecursiveSum
 * This program uses a recursive sum to add the positive integers
```

```
 * less than n.
 */
package Programs;

public class RecursiveSum {

    private static int sum;

    private static int add(int n)
    {
        if (n < 1) return 0;
        else
            sum
                    =
                        n
                            +
                                add(
                                        n
                                            -
                                                1);
        return sum;
    }


    public static void main(String[] args)
    {
        int n = 3;
        System.out.println("The sum of numbers until " + n + " is " +
                  add(n));
    }

}//End of Program R95 RecursiveSum
```

If we run this program from NetBeans or from Eclipse, we get the expected result:

```
The sum of numbers until 3 is 6.
```

If we run it from inside `Program R82 JavacTest3`, we receive the following output:

```
public class Programs.RecursiveSum
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #14.#28        // java/lang/Object."<init>"
   #2 = Methodref          #13.#29        // Programs/RecursiveSum.add
   #3 = Fieldref           #13.#30        // Programs/RecursiveSum.sum
   #4 = Fieldref           #31.#32        //
java/lang/System.out:Ljava/io/PrintStream;
   #5 = Class              #33            // java/lang/StringBuilder
   #6 = Methodref          #5.#28         //
java/lang/StringBuilder."<init>":()V
   #7 = String             #34            // The sum of numbers less th
   #8 = Methodref          #5.#35         //
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBu
   #9 = Methodref          #5.#36         //
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
  #10 = String             #37            //  is
  #11 = Methodref          #5.#38         //
java/lang/StringBuilder.toString:()Ljava/lang/String;
  #12 = Methodref          #39.#40        //
java/io/PrintStream.println:(Ljava/lang/String;)V
  #13 = Class              #41            // Programs/RecursiveSum
  #14 = Class              #42            // java/lang/Object
  #15 = Utf8               sum
  #16 = Utf8               I
  #17 = Utf8               <init>
  #18 = Utf8               ()V
  #19 = Utf8               Code
  #20 = Utf8               LineNumberTable
  #21 = Utf8               add
  #22 = Utf8               (I)I
  #23 = Utf8               StackMapTable
  #24 = Utf8               main
  #25 = Utf8               ([Ljava/lang/String;)V
  #26 = Utf8               SourceFile
  #27 = Utf8               RecursiveSum.java
  #28 = NameAndType        #17:#18        // "<init>":()V
  #29 = NameAndType        #21:#22        // add:(I)I
```

```
  #30 = NameAndType        #15:#16        // sum:I
  #31 = Class              #43            // java/lang/System
  #32 = NameAndType        #44:#45        // out:Ljava/io/PrintStream;
  #33 = Utf8               java/lang/StringBuilder
  #34 = Utf8               The sum of numbers less than
  #35 = NameAndType        #46:#47        //
append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
  #36 = NameAndType        #46:#48        // append:(I)Ljava/lang/StringBuilde
  #37 = Utf8                is
  #38 = NameAndType        #49:#50        // toString:()Ljava/lang/String;
  #39 = Class              #51            // java/io/PrintStream
  #40 = NameAndType        #52:#53        // println:(Ljava/lang/String;)V
  #41 = Utf8               Programs/RecursiveSum
  #42 = Utf8               java/lang/Object
  #43 = Utf8               java/lang/System
  #44 = Utf8               out
  #45 = Utf8               Ljava/io/PrintStream;
  #46 = Utf8               append
  #47 = Utf8               (Ljava/lang/String;)Ljava/lang/StringBuilder;
  #48 = Utf8               (I)Ljava/lang/StringBuilder;
  #49 = Utf8               toString
  #50 = Utf8               ()Ljava/lang/String;
  #51 = Utf8               java/io/PrintStream
  #52 = Utf8               println
  #53 = Utf8               (Ljava/lang/String;)V
{
  private static int sum;
    descriptor: I
    flags: ACC_PRIVATE, ACC_STATIC

  public Programs.RecursiveSum();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method
java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
```

```
        line 8: 0

  private static int add(int);
    descriptor: (I)I
    flags: ACC_PRIVATE, ACC_STATIC
    Code:
      stack=3, locals=1, args_size=1
         0: iload_0
         1: iconst_1
         2: if_icmpge     7
         5: iconst_0
         6: ireturn
         7: iload_0
         8: iload_0
         9: iconst_1
        10: isub
        11: invokestatic  #2                  // Method add:(I)I
        14: iadd
        15: putstatic     #3                  // Field sum:I
        18: getstatic     #3                  // Field sum:I
        21: ireturn
      LineNumberTable:
        line 14: 0
        line 16: 7
        line 20: 11
        line 24: 18
      StackMapTable: number_of_entries = 1
        frame_type = 7 /* same */

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: ACC_PUBLIC, ACC_STATIC
    Code:
      stack=3, locals=2, args_size=1
         0: bipush        15
         2: istore_1
         3: getstatic     #4                  // Field
java/lang/System.out:Ljava/io/PrintStream;
         6: new           #5                  // class java/lang/Strin
         9: dup
```

```
        10: invokespecial #6                  // Method
java/lang/StringBuilder."<init>":()V
        13: ldc           #7                  // String The sum of numbers les
than
        15: invokevirtual #8                  // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        18: iload_1
        19: invokevirtual #9                  // Method
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
        22: ldc           #10                 // String  is
        24: invokevirtual #8                  // Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
        27: iload_1
        28: invokestatic  #2                  // Method add:(I)I
        31: invokevirtual #9                  // Method
java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;
        34: invokevirtual #11                 // Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
        37: invokevirtual #12                 // Method
java/io/PrintStream.println:(Ljava/lang/String;)V
        40: return
      LineNumberTable:
        line 30: 0
        line 31: 3
        line 32: 28
        line 31: 37
        line 33: 40
}
SourceFile: "RecursiveSum.java"
/home/jose/jdk1.8.0_101/bin/javap  -? -sysinfo  -c -p -s -l -verbose
/home/jose/NetBeansProjects/eJVolu17P/src/Programs/RecursiveSum.class
exitValue() 0


Working Directory = /home/jose/NetBeansProjects/eJVolu17P


Running the .class file with java
The sum of numbers less than 15 is 120
[/home/jose/jdk1.8.0_101/bin/java, -cp,
```

```
/home/jose/NetBeansProjects/eJVolu17P/src/, Programs.RecursiveSum] exit
```

Answer

We consider that we have explored the JVM enough to think of concrete experiments with GP.

4.3 GP over the JVM?

We have enough reasons to believe that applied GP on Java is a matter of supercomputing + high creativity. Or, in terms of a desktop user, it is a failure. Now, what must we expect from GP on the JVM? Let us argue now that the JVM is advantageous for GP with respect to Java:

96 ***Theorem: GP on Bytecode is better than on Java.***

Proof: Java is a human friendly interface that must support bytecode content. So, it is clear that Java is more constrained than the JVM. Hence, Java is less evolvable than bytecode. From this consideration alone, we can predict that seas of nonsense in Java are by far larger than those in Bytecode. Therefore, the JVM is better than ordinary Java with respect to GP.

97 ***Is this important?***

The Author imagines that a small difference in the width of nonsense seas cause sensible differences in wasted resources.

98 ***Challenge.*** *The proposed discussion is not that academic. Can you provide an answer as professional as the question?*

4.4 Cloning a .class file

Our aim is to make GP with the JVM. Our first idea to achieve this is as follows:

99 ***Our Plan.*** *Since every* `.class` *file contains no more than bytes, one can edit them to implement mutation and recombination plus a filtering process to make GP at the level of the JVM. Nothing else is necessary, in particular no understanding of the code is demanded.*

100 ***Intrigue.*** *We need a code of interpretation to pass from bytes to the different types of data. The order as these data are saved is also necessary to understand a file. Can we go on without that information in order to be faithful to the spirit of evolution that demands no knowledge but only actions? Let us see.*

In whatever case, the first step to make GP is to produce a clone of a given `.class` file.

101 ***Example.*** *Program* `R101 ClassReader2` *reads the file* `AddTwoIntegers.class` *as an array of bytes that are shown as int numbers. Next, the new file is saved as* `AddTwoIntegers2.class`*. Its content is read and published. To finalize, the executor* `java` *is called on the copied file to see what happens.*

102 ***Exercise.*** *Run previous programs and play with the code. Verify that the copied file gets damaged.*

We need new options. Package `io (input/output)` has some:

103 ***Example.*** *How to use input and output streams can be learned from tutorials* `R103a DataInputStreamDemo`*,* `R103b DataOutputStreamDemo` *and* `R103c WriteFileExample`*.*

104 ***Exercise.*** *Run previous programs and play with the code. To verify that the program* `R103c WriteFileExample` *correctly works, read saved file with program* `R28b ReadAFile.`

105 ***Example.*** *Program* `R105 ClassReader3` *reads the file* `AddTwoIntegers.class` *using an* `InputStream`*. Next, the new file is saved as* `AddTwoIntegers2.class` *using an* `OutStream`*. To finalize, the new saved class is run to see what happens.*

106 ***Exercise.*** *Run the previous program and play with the code. Verify that it does not work.*

We need to explore other approaches.

107 ***Example.*** *Program* `R107 ClassReader4` *reads the file* `AddTwoIntegers.class`*. Next, the new file is saved as* `AddTwoIntegers2.class` *using another* `OutStream`*. To finalize, the new saved class is run to see what happens. We use* `fileChannels` *from package* `nio (new input/output)` *(Jenkov, [37] 2014).*

108 ***Exercise.*** *Run the previous program and play with the code. Verify that it does not work.*

109 ***Exercise.*** *Modify the previous program in order to print the content of the input file as well as that of the output file. An immediate comparison must be allowed and done. Our answer can be found in program* `R109 ClassReader5`*.*

110 ***Exercise.*** *Study the tutorials by Saryada ([72] 2013a, [73] 2013b) about* `NewBufferedReader` *and* `NewBufferedWriter`*. Use them to develop a program to read and duplicate a* `.class` *file. Test whether or not the copied file is fully functional. Our answer can be found in programs* `R110a FilesNewBufferedReader`*,* `R110b FilesNewBufferedWriter` *and* `R110c ClassReader6`*.*

111 ***Exercise.*** *Package* `java.nio.file` *(codejava, [14] 2017) has many procedures to read and write files, various of which we have used thus far. One of them is straightforward:* `copy()`*. Scan the web for suitable tutorials and test whether or not these procedures can be used to faithfully duplicate* `.class` *files. Our answer can be found in programs* `R111 ClassReader7`*. Our implementation does not produce functional copies.*

112 ***Recognizing a failure.*** *We have been unable of using the Java 8 inbuilt machinery to make a functional clone of a* `.class` *file. We made diverse attempts, some of which were severely chastised by the StackOverflow Community (joser, [36] 2017), but received hints lead to nothing. So and to the best of our understanding, we are dealing with a bug, or maybe a family of bugs in Java 8.1. This bug hinders the work of NetBeans and Eclipse alike. That implies that we must wait for some time until the bug is fixed in order to be able to make clean Darwinian Evolution on the JVM.*

4.5 Conclusion

JVM is no more a sublime mystery. It is rather a masterpiece that with great effort is being made by human beings that cannot do everything perfect just on time. Or, bugs never can be ruled out. In fact, we were unable to make by 2017 clean Darwinian evolution on the JVM due to a bug somewhere in Java 8. The bug is possibly not in Java itself but in used libraries. The reason of this assertion is that the Ubuntu Linux System also produces damaged copies of `.class` files.

Chapter 5

Parsing .class files

`Adding understanding`

113 ***Introduction and purpose.*** *We expect the GP on the JVM to be more productive than on Java itself. So, to explore this beautiful challenge is a duty for our Community. But we already have been defeated because even the duplication of a* `.class` *file was impossible for us using the Java machinery. That is why we have delayed the study of Darwinian evolution on bytecode indefinitely. Now, the Darwinian evolution is blind by its very nature. That is why we expect combinatorial explosions everywhere. So, an intrigue arises: what could happen if we add some vision, insight and understanding to Darwinian Evolution? We expect a dramatic increase of the population of problems one step off the solution. And of course this depends on the quantity and quality of injected understanding. Although there are infinitely many ways of pursuing this goal, all of them begin with the very same task: to parse or understand a* `.class` *file, i.e., to transform a series of bytes into information about JVM values. This is our goal for the present chapter. So, let us review some free code.*

5.1 The .class format

We already know that everything in computer science rests on strings of zeros and ones. These strings carry meaning by encoding-decoding procedures which are done according to specific formatting rules. So, to know the different types of information carried by a `.class` file, we need to know the corresponding format.

114 ***Exercise.*** *Study the overview of the* `.class` *file format that is presented by Pujos et al, ([67] 2014). Pay attention to the fact that every such file begins with an identifier or magic number that in hexadecimal reads CAFEBABE.*

115 ***Types and values.*** *The specification of the JVM was worth a lengthy document (Lindholm et al., [50] 2015). An extract about Primitive Types and Values follows:*

```
The primitive data types supported by the Java Virtual Machine
are the numeric types, the boolean type (§2.3.4),
and the returnAddress type (§2.3.3).

The numeric types consist of the integral types (§2.3.1)
and the floating-point types (§2.3.2).

The integral types are:

    byte, whose values are 8-bit signed two's-complement integers,
    and whose default value is zero

    short, whose values are 16-bit signed two's-complement integers,
    and whose default value is zero

    int, whose values are 32-bit signed two's-complement integers,
    and whose default value is zero

    long, whose values are 64-bit signed two's-complement integers,
    and whose default value is zero

    char, whose values are 16-bit unsigned integers representing
    Unicode code points in the Basic Multilingual Plane,
    encoded with UTF-16, and whose default
    value is the null code point ('\u0000')

The floating-point types are:

    float, whose values are elements of the float value set or,
    where supported,
    the float-extended-exponent value set, and whose default value
    is positive zero
    double, whose values are elements of the double value set or,
    where supported, the double-extended-exponent value set,
    and whose default value is positive zero
```

```
The values of the boolean type encode the truth values true and false,
and the default value is false.

The First Edition of The Java® Virtual Machine Specification
did not consider boolean to be a Java Virtual Machine type.
However, boolean values do have
limited support in the Java Virtual Machine.
The Second Edition of The Java®
Virtual Machine Specification clarified the issue
by treating boolean as a type.

The values of the returnAddress type are pointers
to the opcodes of Java
Virtual
Machine instructions. Of the primitive types,
only the returnAddress type is
not
directly associated with a Java programming language type.
```

Thus, we see that the diverse types of informations need diverse numbers of bytes ranging from 1 to 4. Numbers of bytes 1, 2 and 4 are respectively matched by type byte, short and int. Text is another deal: utf8 chars need from 1 to 4 bytes (emre, [23] 2012) and the employed number of bytes must be specified somewhere.

116 ***Exercise.*** *The operations that constitute the bytecode instructions, that specify what to do with the items in the stack, come encoded in one byte that makes place for two hexadecimal numbers. Study the correspondence of hexadecimal numbers and* ***opcode*** *(operational code that are strings encoding for a human readable form of an operation for the JVM) presented by Wikipedia ([97] 2017d). Check the fairness of the binary and hexadecimal representaitions and the notation that represents the action of each operation: a* `before` *state is transformed into an* `after` *state. Example: (Oracle, [61] 2015). In a* `.class` *file the Bytecode operations are part of the* `attributes`*.*

How is this information currently used to work with the JVM? To know it let us explore JBE (Java Bytecode Editor), a very popular application that promises to allow developers to read and modify `.class` files of simple Java programs.

117 ***Exercise. Lab on `NetBeans` with JBE (Java Bytecode Editor).*** *To understand how the `.class` format is currently dealt with Java let us make the following laboratory:*

1. Download the `.zip` version of JBE (Java Bytecode Editor) from (Ando, [1] 2006). Extract it to a suitable folder taking care of keeping the directory structure.

2. Import the project. To that aim follow the menus: File $->$ New project $->$ Java $->$ New Project with existing sources $->$ next $->$ give a name to your project, say, JBE. Assign a project folder to your new NetBeans Project (example in Ubuntu Linux: /home/jose/NetBeansProjects) $->$ next $->$ add the folder where your `.zip` JBE file resides (example: /home/jose/JavaBCEditorsss/Java-Bytecode-Editor/Java-Bytecode-Editor-master) $->$ next $->$ OK.

3. To run the project: Expand the tree of the project and select the file:

   /sourcePackages/ee.ioc.cs.jbe.browser/BrowserApplication.java

   Run the selected file with Shift + F6.

4. Play with this wonderful toy: open a `.class` file, examine it, encounter what you expect, try to modify it. Verify that your modifications are or are not saved.

5. Inside the tree browse for the package `org.apache.bcel.classfile`. One can find here a package filled in the know-how to decipher a `.class` file. Thus, pay attention to the file `ClassParser.java`. We find that the Author uses a `DataInputStream` to manipulate the file content and the following instructions to read data:

   ```
   The first 4 bytes are read as integer and represent the .class ID
   or magic number  0xCAFEBABE.
   file.readUnsignedShort(); //Classparser
   file.readInt(); ////Classparser
   file.readUTF(); //ConstantUTF8
   file.readFully(code);//Code file. Each command of the JVM needs a
   ```

   No reading of floats or doubles was found. So, JBE was devised and released for educational purposes only: thanks a lot to its author Ando.

118 ***Exercise: experiment with the Java Bytecode Editor CE (Java CLASS File EDITOR) (CE [9] 2004).*** *Answer*

119 ***Exercise. Lab on `Eclipse` with REJ (REAL EDITOR of JAVA .class files) developed by `sami.koivu (at) gmail.com` (Koivu, [40] 2011).***

1. We will test this application for its capability to edit `.class` files. Thus, the application must read a `.class` file, allow editing of that file by the User and save proposed changes. The saved file must be understandable by the main Java compiler `javac`.

2. We will use as test the `.class` file `AddTwoIntegers` that must be on your `workspace`. If it is not, run program `JavacTest3` to restore it. Now, it must be there although not visible in the tree of the project. The selected file adds two integers. Verify it with the program `R118 ClassFileRunner`.

3. Go to web site of `REJ`: `http://rejava.sourceforge.net/`. Find the link for downloading files and download the latest version (`rej_v0.7_src.zip` by May/2016) of the source code. Unzip or extract it to a suitable folder.

4. Import the content into Eclipse. To that aim follow the menus: File $->$ Import $->$ From folder or archive $->$ next $->$ Import Source (the directory where the expanded downloaded file resides) $->$ Finish. Expand the tree of the project to see how it was populated.

5. Desirable: if it is possible for you add the library `com.sun.jdi`: In that case, select the project, right click and choose Properties. Select `Java Build Path` + `Libraries` + `Add External JAR`. Select the `jdi.jar` file and end with OK. This is convenient by not strictly necessary: most services work without this resource.

6. To run the application over Eclipse: select the project and click over the play-icon of the upper bar. Select `Java Application` and next the `MAIN WINDOW`. Pay no attention to errors and go ahead.

7. Now you can play with your new and excellent toy.

120 ***Exercise: evolution by hand.*** *REJ seems to be very promising for building a lab on evolution over the JVM. Show this by using REJ to make evolution by hand: begin with a `.class` file that adds to integers and modify it appropriately in order to get a `.class` file that multiplies two integers. The new file must be saved and accepted by `javac`. To test this, use program `R118 ClassFileRunner` with the path to the new file.* *Answer*

121 ***Nice challenges presented by REJ:*** *GP on the JVM seems to be a matter for persons that love heavy duty. If you belong in that group, REJ can help you:*

1. *REJ contains an hexadecimal editor. Go to the menu* `View` *and check the box for the Hex Editor. A new tab shall appear with the hex code of the* `.class` *file. Verify that you can modify all numbers including the* `cafebabe` *signature. Undo you modifications with* `Crtl + z`*. Thus REJ can eventually be made into an hex GP tool.*

2. *Go to the page of the application,* `http://rejava.sourceforge.net/`*, and look at the* `Hello world!` *wonderful video and the corresponding tutorial on how to use REJ to synthesize a functional* `.class` *file. This means that one can use REJ to produce by* `blind concatenation + filtering for Java acceptance` *viable* `.class` *files. By the same token, one can add mutation, recombination and selection for a specific function to get full fledge GP. To enjoy a solution one can explore EpochX ([25] 2017). This line of research also has been deeply explored by Orlov and Sipper ([62] 2009), continued by Elyasaf, Orlov and Sipper ([22] 2013), reviewed by Collom ([17] 2014) and recently revisited by Orlov ([63] 2017). The software they use is based on very powerful tools, say, ECJ ([21] 2017) and ASM (Eric Bruneton, Romain Lenglet, Thierry Coupaye [7] 2002). Nevertheless, the Author did not found the code for Finch, which is the application they developed.*

122 ***Challenge: from bytecode to Java.*** *GP on the JVM is intended to output programs in bytecode, which is a mystery for almost everybody. Most people will prefer programs in Java. The gap is filled by a tool that converts bytecode into Java code. These tools are called* ***Java decompilers****. Research about them and implement over Eclipse at least one of them.*

5.2 Conclusion

The Java Community has being working very hard to allow developers to make highly productive GP on bytecode. The very first task is to convert the sequence of zeros and ones of a `.class` file into instructions with meaning for both machine and human. This process of parsing has allowed us to use REJ to make genetic engineering on a simple bytecode file, the initial step of every GP program on the JVM. The training gained in this task has expanded our mind enough to enable the joyfully reading of recent and challenging achievements of experts in the area.

Chapter 6

Clojure

`The power of the Java clan`

123 ***Introduction and purpose.*** *A computer can run programs in diverse languages, say, Java, C, Basic, Ada. All languages boil down to a final translation to language machine, the language that electronic circuits execute. By the same token the* ***JVM*** *is a virtual computer as capable as a real one and can support diverse languages all with the same Bytecode assembler. The number and variants of JVM languages is astonishing (Wikipedia [98] 2017).*

Precisely some languages that use the JVM have been elected as protests against the defects of Java itself. Say, Scala was designed to be functional but at the end it resulted to produce obfuscated almost functional programming so, a developer in Scala can produce perfect code that he or she cannot understand a little while later. Groovy claims to make everything that Scala does but being moreover understandable. Kotlin is proud of had been very well designed with all those characteristics that are needed by the industry. In regard with Kotlin one trouble is well known: `.jar` *classes are heavier than the corresponding Java versions because Java is just the JVM while Kotlin is JVM + Kotlin. On the other hand, due to the interoperability of the JVM, various serious languages have versions for the JVM like JPython that is the JVM version of Python.*

While all JVM languages allow GP, one is special. In fact, our interest in this chapter is in the direction of Lisp, an ancient language that was chosen by John Koza to implement his GP. So, our purpose is to become acquainted with ***Clojure*** *which is a modern JVM, functional version of Lisp and that will expand our horizons and our minds. Additionally, Clojure has an active community producing very recent works in GP. More to the fact, some people seem to sell the idea that GP must become part of the usual life of Clojure developers, something that is unthinkable for Java developers. Our interest springs from a relation in within*

trees and parsing and that is why we begin with it.

6.1 Trees and parsing

Parsing is the task of assigning meaning to a list of chars. In relation with bytecode, we assign meaning to lists of ones and zeros. So, we transform bytecode into a series of instructions that are readily understood by human developers. But, beware, programs must also be executed by electronic circuits. So, a question arises: can we attack parsing and execution as a single problem and produce an elegant and efficient solution? The answer is affirmative. So, Our immediate aim is to illustrate how any Java piece of code (and of any other language) can be described by a tree that moreover conveys the information on how the code must be executed. In second place we will discuss how to use those trees to enable evolution in the space of Java programs. In third place we ask: is the question important enough to try out a fresh beginning?

Let us illustrate how to transform Java code into a tree.

124 ***Example.*** *Let us build the tree associated to the Java expression* $int a = 3+4$.

There are indeed various directives to build a tree to represent a piece of Java code. Two examples:

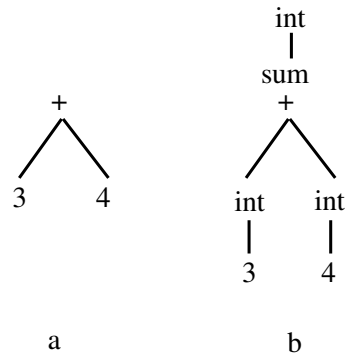

Figure 6.0. The expression `3 + 4` *can be directly represented by the tree at the left. But a programming language must try to optimize, say, memory management. So, one can add the note that variables are of int type as it is done at the tree at the right. By observing all constraints a tree to be a perfect representation of a piece of Java code must have a rather rich structure.*

The very idea of coding is that one can build complex structures by assembling simple ones. So, the previous tree can become a subtree of a more complex expression as the following:

```
if (a > 3) a = 3+4 else a = 1
```

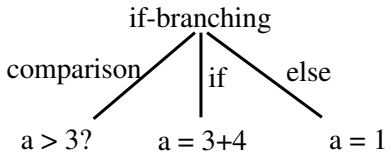

Figure 6.1. Trees can be inserted as subtrees of more complex trees. In that way complexity arises. This is the general structure of an if-branching, in which each expression is represented in its turn by a tree.

How to find the tree of a piece of code is a complex affair that has produced many solutions with a well understood core but with specificities that may demand dedicated work (Tomassetti17, [86] 2017). One of those solutions is ready to be used in Eclipse.

6.2 Two tools of Eclipse

Let us learn how to see the bytecode of a Java class using an Eclipse tool.

125 ***Exercise over Eclipse.*** *Install a plug-in to see the bytecode of a Java program open on the Editor Window:*

1. Go over the menus: help $->$ Eclipse Marketplace $->$ Find.
2. Type: bytecode.
3. Install `Bytecode Outline`. This plug-in allows to see the bytecode of the Java class that is on the Java Editor Window.
4. Mode of use: with an open Java program click on the menus Window $->$ Show View $->$ Other $->$ Java $->$ Bytecode. One can also open a hypertext with the bytecode reference from the same menus and from the window of the plug-in.

The second tools allows to see the tree associated to a Java class. More specifically:

The **AST (Abstract syntactic tree)** is a non redundant tree that contains the totality of information of the Java class. Let us learn how to use an Eclipse tool to visualize the AST.

126 ***Exercise over Eclipse.*** *Install a plug-in to see the AST of a Java program:*

1. Follow the Eclipse main menus: Help $->$ Install New Software.
2. Replace the string `type or select a site` with:

   http://www.eclipse.org/jdt/ui/update-site
3. Wait for some lengthy seconds: a list of available software must appear. Check `Select All`. Click `Next`. Wait for some minutes until the software is installed. `Next` again.
4. Accept the license and `finish`.
5. Verify the lot of information that a Java class contains. Most of it is hidden from the eyes of the User so that he or she could dedicate his or her attention to the design of the code.

127 ***Challenge. GP on ASTs.*** *The AST offers an expedite path to GP: trees allow directly for the possibility to implement mutation and recombination. Apart from GP ASTs also have many other applications (Kuhn [43], 2017) so, Eclipse also has tools to process ASTs. All these facilities present a challenge: to implement GP on ASTs. In that case you have a lot to learn from the rest of this chapter and also from the various works of Orlov cited at the end of the previous chapter. The corresponding Eclipse tool is known as JDT (Java Development tools) which can be installed from the Eclipse MarketPlace. There exist many tutorials and code examples, say, that by Vogel and Scholzs ([93] 2016).*

6.3 Clojure: a fresh start

If the code of every program boils down to a tree that is readily understood by both humans and machines, can we make trees into a programming language? In my opinion, a positive answer is known since many moons ago: it is LISP, the language chosen by John Koza for his GP. LISP, being a great idea, begot too many dialects, one of which is **Clojure**.

Clojure was created by Rich Hickey as the fruit of a program of research during various years and attempts. He worked on his own funding. The main but not

unique goal of his research was to merge Lisp to Java but along a functional implementation. He released the language in 2007 (Wikipedia, [99] 2017) The word Clojure seems to be a non homologous recombination of `closure` and `Java`. As a bonus, `cl` stands for `common lisp`. Closure is related to higher functions in various variables.

128 ***Example of functions in two variables that are enabled by Clojure***

Example of a function in two variables.
$f(x,y) = x+y$
f can be curried as follows:
$f(x,y) = g_y(x) = x+y$
This means that in first place you make y constant and in the second you use that constant to define a function. So:
$g_3(5) = 5+3 = 8$
$g_3(7) = 7+3 = 10$
$g_9(7) = 7+9 = 16$
$g_9(8) = 8+9 = 17$

Other example of a function in two variables that is enabled by Clojure is the following one that makes Clojure luring for GP. Let us imagine that we have a function on two variables, one is a command and the other is a string. Function f executes the command on the string:
$f(command, string) = command(string)$
f can be curried as follows:
$f(command, string) = g_{command}(string) = command(string)$
This means that in first place you choose what command to work with to make *command* constant and in the second you apply the command on the string. So:
$g_{print}(Hi!) = print(Hi!) = Hi!$ (appears on the console)
$g_{print}(Goodby!) = print(Goodby!) = Goodby!$
$g_{reverse}(123) = reverse(123) = 321$
$g_{reverse}(clojure) = reverse(clojure) = erujolc$

129 ***Closure and currying are terms dealing with higher order functions (functions that take other functions as arguments). Currying deals with evaluation, closure deals with accessing scope. In general terms:***

A **closure** is the fact, for a function, to be able to access something in the enclosing context. Currying is the fact of evaluating function arguments one by one, producing a new function with one argument less on each step. These concepts are slippery, see Saumont ([74] 2014).

130 ***Three++ virtues of Clojure:***

1. *It is functional as a protest to Java. Even in Java 8 functions are not that simple. In Clojure they are.*

2. *It is zero-boilerplated -quite contrary to Java. Example: To declare a string put it inside quotations. To concatenate strings Java uses the + operator. In Clojure, no operator is needed: just type the reserved word* `str`*, the first variable, space, the second variable, the third and so on and that is all to it. But the operator + does not concatenate strings because it causes ambiguity with math notation. Other example: the semicolon to end an expression is optional. Expressions are always surrounded by, say, braces, and they end when the number of left braces is compensated by that of right ones. So, an expression can occupy various lines and be indented to show structure. Thus, Clojure programs are naturally hierarchically structured into code blocks delimited by parentheses. To mark a comment use a semicolon.*

3. *It is Java friendly: You can readily import Java code.*

4. *It is more appropriate for GP than Java: Clojure has a command to transform strings into code and moreover recombination of live Clojure code can be easily engineered to produce more viable strings than in Java.*

To learn Clojure we will go over various tutorials.

131 ***Exercise. Execute the following online tutorial about Clojure.***

1. Visit the site `https://clojure.org/` and inspect it passively but thoroughly. Observe that this language uses the enclosed Pollish notation: the Java expression `3 + 4` is written in Clojure as `(+ 3 4)` prefix notation. By the same token, `(+ 3 4 5)` is evaluated in Clojure to `12`. To be enclosed means that, when typing a Clojure expression, start with () and fill inside. Do this recurrently.

2. Find and follow the link `Getting Started`.

3. In the new window find the title `Try Clojure Online` and follow the link `TryClojure`.

4. In the new window find the symbol > and click with the cursor at its right. Type `next` and go on with the tutorial.

5. Return to the page `Getting Started` and under title `Try Clojure Online` click over `Himera`. This is another online compiler. Find the title `ClojureScript at a glance - PDF` and click the `pdf` link. Download the document and save it to a visible place, say, the desktop, where it must be ready for a rapid consult.

Let us go now to a more serious work... on our machine.

132 ***Exercise. Install Clojure over Eclipse.***

1. To get a feel of the work to be done, watch `Clojure Gently` the wonderful video tutorial about Installing Clojure over Eclipse by Jari Lanchia (I apologize for any mistyping) : https://www.youtube.com/watch?v=dQ6hhnitzHg. (Active by 8/V/2016). For Eclipse `Neon.3` we have:

2. Turn on Eclipse.

3. Follow the main menus: Help $->$ Eclipse MarketPlace $->$ type `Clojure` $->$ Select and install `CounterClockWise. Accept license` and `finish`.

4. Restart Eclipse. In the welcome window select `New Clojure Project`. Give a name to your project (my-first-clojure-project) and assign a workSpace.

5. In general, for `creating a new project` follow the main menus: File $->$ New Project $->$ Other $->$ Clojure Project $->$ Next $->$ Give a name to your project and assign a workSpace $->$ Finish.

6. Expand the tree of the Package Explorer: you must see your Clojure project amidst the Java ones.

7. Expand the tree of your Clojure project and open the file `core.clj`. It must read:

```
(ns my-first-project.core)

(defn foo
  "I don't do a whole lot."
  [x]
  (println x "Hello, World!"))
```

8. A window with title REPL must appear. REPL means `Read Evaluate Print Loop`. It is a service that allows to run a piece of code instantly. If REPL gets somehow closed, one can reopen it as follows: right click on the main editor window and choose `Leiningen` $->$ `Launch headless REPL for the project`. The REPL window is divided. One part is for printing code. It contains the invitation to print code <type clojure code here>. The other is for the output. To get in touch, repeat here the previous online tutorial.

9. REPL has various icons with suitable functions. Two are the most important for us. The first is a cogwheel, a wheel with teeth, with a red square that is used for halting a long calculation. It functions very poorly. But if one clicks at it and next to the icon with two counter-wise arrows, then it stops at that very instant. Another icon is a screen with an x for clearing the output window.

10. If REPL crashes, one can turn it off and it obeys immediately.

11. We see that Clojure runs over Eclipse thanks to CCW (CounterClockwise). It uses Leiningen, which is "the de-facto standard build tool used for Clojure projects" (Petit, [64] 2016?).

Now we got ready to follow a tutorial. The Author chose a tutorial that fills in the immediate needs of an impatient person that wants to browse Clojure code with the aim of looking how GP is implemented in this language. Nevertheless,If someone wants to go deeper and slower, he or she can opt for the official tutorial `Clojure for the Brave and True` by Alan Dipert ([31] 2016?). For general consulting: Community $->$ Tutorials and learning material $->$ Clojure SheatSheet. Hypertext will provide you with explanations and examples on every item. Also: you can download a pdf version without hypertext context.

133 ***Exercise. Run the `Clojure by example` tutorial by Hirokuni Kim ([33] 2017).*** *This tutorial can be run both online and over Eclipse. In the last case, use copy&paste. Example: in the right panel of the main page of the tutorial one can read right at the beginning:*

```
user=> (println "Hello, world!")
Hello, world!
nil
```

so, one selects and copies to clipboard the expression `(println "Hello, world!")`, *next one pastes it into the REPL window of Eclipse with* `Ctrl + V` *and runs it with a return.*

Strong warning: *the syntax of Clojure is quite different than that of Java. So, read the explanations and build a library with all the executed lines of code. To that aim, you can clear the content of your first Clojure project (in the main window) and paste lines into it. To insert comments at the end of a line: use a semicolon. Titles are inserted as comments. Save your library from time to time. Hirokuni uses the jargon of Javascript so, it might sound a bit strange. Our library can be found in the package* `Clojure`*, file* `ClojureTutorial.txt`*. All coming files developed by the Author will be attached to this same package in* `.txt` *format.*

Read documentation with no click *just drag the cursor to any Clojure expression in red and wait a second: you shall see the associated documentation. This must be true for the REPL, that must be active, as for the main editor window. If the User defines his or her own code and adds documentation, it will be displayed on the same prompt. Documentation are lines in within quotes that are inserted below the declaration of a function: be attentive to the code. It appears colored in blue. Example: in the following snippet the documentation says that the* `multiply-by-two` *function multiplies a number by two:*

```
(defn multiply-by-two
  "multiplies a number by two"
  [x]
  (* 2 x))
```

This snippet exemplifies the directions of Java perfectionists according to which documentation is superfluous: give the right name to your method and forget documentation. By contrast and according to Clojure standards, documentation is mandatory because they love extremely short names that might be mute or even misleading.

Comment: *Hirokuni Kim assures along his code that the light and the darkness are on equal rank and that God created both. In the sense given by Job this is plainly correct: "Shall we receive good from God, and shall we not receive evil?". Most evil come from other persons as it is explained in Proverbs: "The LORD has made all things for himself: yea, even the wicked for the day of evil." Ecclesiastes recognizes that the disorder caused by the evil is difficult to understand and accept: "In the day of prosperity be joyful, but in the day of adversity consider: God also hath set the one over against the other, to the end that man should find nothing after him".*

134 ***The return of functions.***

Clojure has no `return` statement. So, the return value of a function must be inferred from the code just as the compiler does it. Try the following rule: *if something is evaluated, it disappears.* Henceforth, the return of a function is `nil` else something that is not evaluated. Example: the function `f` below adds 4 to the input and the return value is the input plus four because that output is live, it never is used:

```
(defn f [x]
      (+ x 4))
(println (+ 5 (f 3)))
```

Because the output is live, it can be used further:

```
(println (+ 5 (f 3)))
```

This program output 12 because it adds 4 to 3 and then adds 5. By contrast, a very similar program returns a quite different answer:

```
(defn g [x]
         (println   (+ x 4)))
   (println (+ 5 (g 3)))
```

This program executes `(g 3)` and outputs 7, which is live until it is printed. Once printed it exists anymore. So, in line 2 the `(println (+ x 4))` instruction output `nil`. Hence, the output of `g` is nil for any input. This causes a `NullPointerException` in line 3.

Warning : it is usual to find the live terminals at the last line so, beginners tend to think that the last line contains the output of functions. This is false. For instance, in a branching with an `if` one can decide where to put the output if any, in the `if` execution line or with the else line. Let us notice that Clojure has no `else` statement: Clojure is zero boilerplate so, the `else` option is listed just below the executing line of the `if` clause with no title at all. For example, the program

```
(defn h [x]
        (if ( < x 10) (println (str x " is digit"))
            (println (str x " is not digit"))))
```

will output `3 is digit` for `(h 3)` from the `if` clause while it will output `17 is not digit` for `(h 17)` from the `else` branching.

The return of a Clojure function can be very complex. To create a complex output in Java one must create an object and pack it into a class. In Clojure that is straightforward. Example: the program

```
(defn h []
         (let [ x 10
                y "Hello world!"]
          [x y]))
(println (h))
```

will print `[10 Hello world!]`. To print the first component use `first` and to print the second use `second` as here:

```
(println (first (h)))
(println (second (h)))
(println (nth (h) 1))
```

where we see how to generalize in the last line. Now, if we take off the parenthesis around `h`, the program will crash. Why? It is because parenthesis say that we are invoking `h` as a function and that we need its output. Otherwise, Clojure would not know what to do.

It was good for the Author to make a review of Clojure fundamentals following various tutorials. Here is another one:

135 ***Exercise.*** *Over your own namespace run the Clojure tutorial composed by* `tutorialspoint` *([87] 2017?). A* `.txt` *file with the code can be found in the file* `Clojure-tutorialspoint.txt`*.*

136 ***Challenge.*** *Run the Clojure tutorial composed by* `Adam Bard` *([5] 2017).*

137 ***Exercise.*** *Implement and enjoy the Clojure tutorial composed by* `Lee Spector and Tom Helmuth` *([81] 2016). This tutorial also teach how to processes some files so, it is more difficult than the previous ones.*

To go forward we need to review two important points, the first dealing with the extension of the language and the second with graphics:

138 ***Macros: how to extend the language.***

Clojure employs some reserved words that are used to define building blocks of its grammar: `if else do filter reverse ...` . One uses these blocks in complex manners to devise functions. The problem is that one may desire to extend the language to include new words with new possibilities. In this regard, let us see an example in detail:

```
;if-do in a function
(defn if-pos-divide-by [a b]
           (if  (pos? a) (do (println "positive") (/ b a))))
    (if-pos-divide-by 3 4)
;#'clojure-tutorial-spector.core/if-pos-divide-by
;positive
;4/3

;variation lacking "do" would compile but not run
(defn if-pos-divide-by [a b]
              (if  (pos? a) ( (println "positive") (/ b a))))
;#'clojure-tutorial-spector.core/if-pos-divide-by
 (if-pos-divide-by 3 4)
;positive
;NullPointerException   clojure-tutorial-spector.core/if-pos-divide-by
(form-init5392612804203617273.clj:2)


;Solution = create the macro "when" that is used in functions:
(defn when-pos-divide-by [a b]
           (when (pos? a) (println "positive") (/ b a)))
;#'clojure-tutorial-spector.core/when-pos-divide-by
(when-pos-divide-by 3 4)
;positive
;4/3

;How does the when macro function?
(macroexpand '(when (pos? a) (println "positive") (/ b a)))
;(if (pos? a) (do (println "positive") (/ b a)))


;Let us repeat the creation of the when macro.
;Attempt one: oops!
(defmacro my-first-macro-when [ a b]
     (if  (pos? a) ( (println "positive") (/ b a))))
;#'clojure-tutorial-spector.core/my-first-macro-when
(my-first-macro-when 3 4)
;positive
;NullPointerException   clojure-tutorial-spector.core/my-first-macro-wh
(form-init5392612804203617273.clj:2)
```

```
;....
;many unsuccessful trials
;....


;Solution from
;Daniel Higginbotham (2015)
;Writing Macros | Clojure for the Brave and True
;http://www.braveclojure.com/writing-macros/
(defmacro when-official
     "Evaluates test. If logical true, evaluates body in an implicit do."
     {:added "1.0"}
     [test & body]
     (list 'if test (cons 'do body)))
;#'clojure-tutorial-spector.core/when-official

;The macro is used in a function
(defn when-official-pos-divide-by [a b]
          (when-official (pos? a) (println "positive") (/ b a)))
(when-official-pos-divide-by 3 4)
;#'clojure-tutorial-spector.core/when-official-pos-divide-by
;positive
;4/3


;Complaint: where is here an extension of the language
-on the supposition that the when-macro were lacked?
;Answer: we can use whatever test and whatever action or form.
(defn when-official-even [a]
     (when-official (even? a) (println (str a " is even, it is the double
      of ")) (/ a 2)))
   (when-official-even 6)
;#'clojure-tutorial-spector.core/when-official-even
;6 is even, it is the double of
;3

(when-official-even 5);nil
```

139 ***Exercise.*** *Implement and run over Eclipse the tutorial on Macros by Bard ([6] 2017b).*

140 ***Exercise.*** *Get acquainted with `Quil` ([69] 2017), the official drawing facility of Clojure. Some examples on how to use it in GP will appear below. `Quil` has been built on top of `Processing` ([66] 2017) , which is a Java friendly language for learning graphics and animations. Notice that Processing is good as for desktop applications as for the web. The Processing Community is active and is guided by a very fine sense of perfection.*

6.4 Genetic algorithms

Let us run some genetic algorithms from the web to see the possibilities of Clojure. They might come in the form of projects with various files. So, let us learn a bit about projects in Clojure.

141 ***Exercise: basic information about projects.*** *Every Clojure project has a leading file that is named `project.clj`. It contains the basic information about the project. How to interpret it can be read in Higginbotham ([32], 2017)*

Let us learn now how to mount with the nails an extant complex Clojure project. We do that in order to learn the structure of a Clojure project. Automatic importing procedures exists and will be seen below.

142 ***Using copy&paste to import a project into Eclipse CCW.***

Let us download and run a concrete project that contains a genetic algorithm whose aim is to match a given phrase: `Hello world!`.

1. Go to `https://github.com/yogthos/genetic-algorithm-example`

   This is the site of `Example of a simple GA using Clojure` by yogthos: Dmitri Sotnikov ([75] 2013).

2. You must find a tree with a Clojure project. We will import it into Eclipse as follows:

3. Over Eclipse open a new Clojure project by following the menus: File $->$ new $->$ Other $->$ Clojure Project $->$ Give as name to your project: `clojure-genetic-algorithm` $->$ Finish.

4. Look for your project over the Package Explorer. If it is not visible, use: Window $->$ Show view $->$ Package explorer. Expand the tree of the project and find the file `project.clj`. Replace its content with that of the corresponding file of the Github project. Next, add a package to `src` folder: right click on `src` and select New $->$ Package $->$ Give it the name `clojure-genetic-algorithm`.

5. Add a file (a new namespace) to the `clojure-genetic-algorithm` package: click on the name of the package and select: New $->$ Other $->$ Clojure Namespace $->$ Give a name : `core.clj`. Open the file and replace its content by the corresponding file of the Github repository.

6. Replace the content of the file `core-test-clj` by the content of corresponding file in the repository.

7. To run the project: open the file `core-test-clj` of the `clojure-genetic-algorithm` of the test section and right click over its pane. Select Clojure $->$ Load file in REPL $->$ Wait for a long time that the REPL appears (the appropriate clojure version must be downloaded, expanded and installed). Next, type on the lower part of the REPL:

   ```
   (evolve-string)
   ```

   Press the `Return` or `Enter` key. You have commanded to run the genetic algorithm to match the `Hello world!` phrase.

8. In response and after a little while you will see that the program peacefully ends without much information. Indeed, the program matched the target phrase but only inside the CPU.

9. To see what we expect to see, a genetic algorithm that step by step matches the given phrase, insert a command to print the first element of the population for every generation. To do so, replace the code of the function `evolve-step` by the following:

```
(defn- evolve-step
  "mutate the population, then promote top members
and add mated members to the end"
 [size population mutator threshold fitness target]
 (let [mutated (rank (mutate population mutator threshold fitness target))

        promote-size (/ size 5)
        keep-size (- (/ size 2) promote-size)
```

```
        [xs ys] (split-at keep-size mutated)]
(println (first population))
(concat xs (take promote-size ys) (mate mutated fitness target))))
```

10. Because you changed the code, you must recompile it anew: right click over the pane in use and select: Clojure $->$ Compile file in REPL. The corresponding feedback must appear in the REPL. After seeing it, type:

    `(evolve-string)`

    You shall see a trace of the genetic algorithm showing how the solution is built step by step.

Is the tested procedure for importing Clojure projects boring for you?

143 ***Challenge.*** *Learn from the web how to automatically import a Clojure project into Eclipse CCW. Or maybe you would like to try another code editor. In that case, give a try to* `Visual Studio Code` *([91] 2017) or to* `LightTable` *([48] 2017) with plugins* `clj-light-refactor` *and* `LightTable Parinfer`*. Or to* `clooj` *([10], 2017). Some people also like to work on* `Netbeans` *with the help of* `Enclojure`*. People that want to go one step above Java and to explore the world of Kotlin would go to* `IntelliJ & Cursive`*. And, what about Gorila-REPL ([27] 2017)? This REPL allows to produce written material following the* `Literate Programming Format` *in which text, graphics and code share the same recursive space.*

144 ***Exercise.*** *We have learned how to import and run the Clojure project that matches a phrase. But we are very far from understanding how the program functions. So, write a tutorial to explain it. Answer*

145 ***Exercise. Import and run a GA that solves the salesman problem.*** *The salesman must find a route that minimizes the cost of a round trip along a set of cities that are interconnected by some roads with given cost, say, in kilometers. Mount and run the project that solves this problem from ([15] 2015) Answer*

146 ***Exercise.*** *We have learned how to import and run the Salesman Clojure project. But we are very far from understanding how the program functions. So, write a tutorial to explain it. Answer*

6.5 GP with Clojure

Clojure is by its very nature an open invitation to GP because its immanent facility to convert strings into code: How is that possible?

"Clojure is a homoiconic language, which is a fancy term describing the fact that Clojure programs are represented by Clojure data structures. This is a very important difference between Clojure (and Common Lisp) and most other programming languages - Clojure is defined in terms of the evaluation of data structures and not in terms of the syntax of character streams/files. It is quite common, and easy, for Clojure programs to manipulate, transform and produce other Clojure programs." (Clojure.org, [12] 2017).

Let us see by ourselves how easy it is to transform string into code.

147 ***The eval form.*** *This form or primitive allows to transform a string into code. We can grasp how it works from the following code that was run in a REPL:*

```
(def my-str '(+ 2 3))
(println my-str)
(+ 2 3)
;nil

(eval my-str)
;5
```

148 ***Exercise.*** *Implement in Clojure a program that automatically generates a class that matches the outputs of a given arithmetic operation on two integers. Similar to our Java program* `R32 ArithmeticOperationsGP3`. *Answer*

149 ***Exercise.*** *Develop the code for the Clojure GP of the mean. Answer*

150 ***Exercise.*** *Implement and run the GP Clojure project* `fungp` *composed by Mike Vollmer ([94] 2012). This code is a library for GP in Clojure. Answer*

151 ***Exercise.*** *We have learned how to import and run the* `fungp` *Clojure project. But we are very far from understanding how the program functions. So, write a tutorial to explain it. Answer*

152 ***Exercise.*** *Use* `fungp` *to develop the GP code to synthesize the mean of a set of data of any length. Answer*

We have seen how to attack GP using grammars and trees which are universal strategies. Let us pay attention now to a specific plus of Clojure that was recently added to the language:

153 ***Exercise. GP with `clojure.spec`.*** *Implement and run over Eclipse the GP program invented by Carin Meier ([53] 2017) in which GP is rooted on the `clojure.spec` package.* *Answer*

154 ***Exercise.*** *Work out the understanding of the aforementioned code by Carin Meier.* *Answer*

Answer

154, page 74. This program promises to simulate evolution of creatures. Related code in Artificial Life demands thousands of lines and a great deal of expertise. But we see here some few lines. So, we feel very intrigued. In consequence, we have decided to try it out. To begin with, we burnt out some prerequisites. We have in first place the guiding idea, which is the use of the spec library (Clojure.org, [13] 2017):

```
The spec library specifies the structure of data,
validates or destructures it,
and can generate data based on the spec.
```

So and in regard with GP, the immediate purpose is to use that library to generate data that enclose code that must be evolved to get more code to fit in a given function. More virtues are exposed by the very Rich Hickey ([30] 2016).

To learn how to use this library we went over the introductory tutorial by Carin Meier ([54] 2016b)

So, we added a new namespace `tutorial` to the package `genetic-programming-spec`. That namespace was populated with the code from the blog following the (adapted) instructions given by Carin.

Next, we looked at the video about spec by the same author ([55] 2016c).

We returned back to mentioned site

`http://gigasquidsoftware.com/blog/2016/07/18/genetic-programming-wit`

This is the site of `Genetic Programming With clojure.spec` by Carin Meier. We read the blog trying to follow it along the corresponding code. After commenting the code we copied the resultant files as plane text to the file `Clojure-gp-spec-core.txt` and added to the Clojure package of the present volume.

Last, we went on to study and comment the other related work of Carin on self-healing code ([56] 2016d).

Care must be taken in regard with the `spec` libraries: because we were working with an alpha version, this should had been specified in the requirement list.

The resultant files were copied as plane text to the file `Clojure-gp-healing.txt` and added to the Clojure package of the present volume.

155 ***The wisdom behind `spec`.***

Thanks to Carin Meier we got the opportunity to learn while playing. So, one might think that in `spec` there is nothing besides fun. It is not the case. Quite to the contrary, the spec spirit deeply dives into the hearth of the mathematics behind computation. The best way to grasp this idea is to see how a language that is based on specifications is created. So, let us welcome `TLA+`, a programming language that was developed on top of Java. To see the motivation and fundamental idea one can look at the video `Hard die` by Leslie Lamport ([44] 2015?). Do not get disturbed at the beginning when the Author assumes that you have gone over other three videos: this video is pretty clear. Next, you might go to the official site (Lamport [46] 2017). A Java friendly version of `TLA+` is the language `PlusCal` (Lamport [45] 2015).

6.6 Making school in GP

Le us make a review of the works of an impressive GP Author that because of his endurance, wisdom, professionalism and educational drive is awarded here a high five.

156 ***Exercise. Hello Mr. Spector!*** *Watch the video on `Genetic Programming in Clojure` by Lee Spector ([80] 2015). No doubt, the lecturer is a monster. But, please, fear him not. Rather, think that we need one another. In fact, he can provide domain specific expertise and you can propose clear concepts. For instance, he uses the following expression: `Deliberate, Darwinian evolution of computer programs.` a) Prove that this expression is a contradiction.*

b) Additionally, he preaches: `Darwinian evolution is itself a designer worthy of significant respect if not of devotion.` This is a personal appreciation that is based on something real. What is it?

c) To end, he dictates: `life is the evolution of programs.` Explain the meaning of this slogan and decide whether it is true else false.

Answer

Let us study a bit better the ideas and works of Spector.

157 ***Exercise.*** *Read a short article for the lay public (Lee Spector, [77] 2005).*

This article will allow you to witness that GP with Clojure is at the very front of modern applied science and that the main bound to the application of Evolution is the own imagination.

158 ***Exercise.*** *Implement and run the code by Spector on* `Pucks`*, digital critters that try to survive (Lee Spector, [82] 2016b).* *Answer*

159 ***Example. Pucks continued.*** *Let us devise an agent&world pair such that the world grants eternal life to the agent.*

The Git repository cannot be modified. So, the Git folder was deleted. The Clojure project `clojure-pucks` was also deleted together with its containing folder. A new beginning was set: a new project was declared with the name `pucks`. The Git project was this time downloaded as an ordinary file. Next, it was imported into our Clojure project: the name of the project was right-clicked, the Import menu was selected. The option `Archive file` was chosen and the corresponding zip file (`pucks-master.zip`) was imported. The result was that the original tree was enlarged with a new branch: `pucks-master`. Programs inside this branch do not run because it is not the main one and the mutual calls of the diverse files got wrong. So, the implementation of the native structure was mandatory. To this aim, necessary folders (`pucks pucks.agents pucks.worlds`) were created inside the main branch, the content of the core was substituted by that of the project (from the pucks master branch). The definition of the project in `project.clj` was also updated. The files that accompanied the core were dragged in first place and then the folders `agents` and `worlds` were also dragged -in that order. A world was chosen at random for which a REPL was opened and a simulation was correctly triggered by the command

```
(run-pucks (agents) (settings)).
```

We imagined a paradise for the agent that is in the `pucks.agents.user.clj` file. The paradise is described in the `pucks.worlds.dev.world24` file. It contains a `vent` (source of energy) and a `zapper` (a sink of energy) and a stone that protects the agent from being sunk. To achieve this we tried various modifications until a satisfactory one was found. For each correction a new REPL was needed so, the old one was closed. The resultant file were copied to `Clojure-pucks-user.txt` in Clojure package of the project for this volume.

160 ***Challenge: comment each file of the Pucks project.***

161 ***Challenge. Pucks and adaptation.*** *The Pucks project inherently comes with a challenge for the User: to design his or her own agent to see if it can survive indefinitely in all worlds. Take and solve the challenge. Once you are done, develop the GP to solve the same problem.*

Developing programs is a very difficult enterprise. To make of this task into an automatic and autonomous procedure, using GP or Artificial Intelligence, is surely much more difficult. So, to battle the arisen complexity it is better to come armed with very good weapons. But where are they? The tradition in the GP community is to answer like this: *Weapons? Tools? Do them by yourself!* So, let us get acquainted with one tool to see from first hand how is the deal. The tool to examine is the answer to the problem of putting the JVM to work optimally for GP. This is achieved by creating a new and appropriate language whose entrails we can examine in detail.

162 ***Exercise and challenge: The Push language as a modern tool for GP.***

1. *To understand the motivations and perspectives of the Push language, let us read the article "Genetic Programming and Evolvable Machines" by Lee Spector et al ([76] 2002)..*
2. *Find the program on page* `17 (11 of 34)` *that calculates the factorial of a positive integer. To understand this program, translate it to Java.*
3. *Download and implement* `Clojush` *(the Clojure implementation of Push) from ([83] 2017)*
4. *Challenge: comment the code of* `Clojush`*.*

Answer

163 ***Exercise.*** *Read and digest the PhD dissertation on modern GP by Thomas Helmuth ([28] 2015). It was directed by Spector and shows how to put Push to use.* *Answer*

164 ***Challenge.*** *Show that the approach of learning on examples that is used in this dissertation is good for problems in the P class but that it is useless for problems in the NP class and above. Thus, it is predicted to be useless for general program synthesis but this topic is too big and surely it has infinitely many sub-worlds of low complexity where this approach might be very useful. (To be useless means that an algorithm is sharply outperformed by randomness.)*

Let us dive now a bit more on coding with Push.

165 ***Exercise.*** *Implement and run over Eclipse* `taggp` *a tree based GP library that runs over Push by Lee Spector ([79] 2012). Explain the main idea. Answer*

166 ***Challenge.*** *Comment the code of the aforementioned* `taggp` *project.*

167 ***Exercise.*** *Implement and run over Eclipse the code that Helmuth (2015 l.c.) developed for his dissertation Answer*

168 ***Challenge.*** *Find and study the latest work of Lee Spector. By 2017 it is directed to attack with Push the problem of optimizing algorithm parameters for which he uses SMAC (Sequential Model-based Algorithm Configuration) (Spector, [84] 2017).*

169 ***Exercise.*** *Test the web with the prompt* `clojure music` *to see what happens and implement over Eclipse the project you most like. Answer*

Clojure has many fronts of development. One of them is very interesting for us: 3d shapes and animations with the quality needed by modern Game Technology.

170 ***Exercise.*** *See the video by Bryce Covert ([18] 2016) that explains from scratch how a game is developed in Clojure.*

Now, let us see how Clojure is desirable for those that live in the world of big money that demands high quality games:

171 ***Exercise. High quality games.*** *Give a look to Arcadia ([2] 2017), a Clojure game engine that co-works with Unity3D ([92] 2017). Some elementary things that Arcadia can do are presented by Gardner ([26] 2015). Game development is very complex and demanding. That is why GP is being explored to see, say, how to have it designing strategies for agents. Nasser ([58] 2017) presents how the trade looks like from the stand of a game designer and researcher.*

6.7 A lonely cowboy

Functional programming allows the generation of complex structure by concatenation of modifications. Precisely as it is done in biology, in morphogenesis. Let us see how it looks with Clojure.

172 ***Exercise. Morphogenesis with Clojure.*** *Watch the video on 3d-form-evolution through tree-based transformations developed by the lonely cowboy Karsten Schmidt ([65] 2014). If you are ready to learn a lot follow the link pointing to Github and go further to the home page of this author where you can find a handful of challenging projects.*

6.8 Conclusion

GP is an extremely difficult enterprise but Clojure might be an excellent option if not the best.

Chapter 7

Conclusion: Testing a prediction

The ancient Evolutionary Theory of the Origin of the Species claimed that life on Earth arose by evolution beginning from some few humble principles. In modern terms, this theory reads: the genome is software and evolution is the developer responsible for its existence. So, this theory predicts that evolution shall be useful to develop software. This prediction is true. In fact, all human developers run in their minds a sort of evolutionary algorithm to develop software: from a possible humble beginning more functional complexity is added, tested, corrected and so on. The automation of this program is known as GP (Genetic programming) which has produced fruits since long ago. Java has allowed us to verify over very simple problems that GP is a fact.

On the other hand, the huge variability, extreme complexity and exceeding perfection of living forms make it clear that GP algorithms shall run and produce excellent software just in front of own eyes. This is a mandatory prediction. The Author has been unable to see this or something similar.

Nevertheless, the Author has found a terrible challenge that has generated a lot of work, ideas, dreams, fruits and joy. In fact, many people have being working in GP and now we can enjoy too many fronts where Java developers can work at. Thank them all and very specially to those that gifted us Clojure and Clojure-GP.

In regard with Easy GP, the immediate path might be to think of problems one step off the solution. Example: parametric optimization that groups all those algorithms that produce results whose quality depends on the values of some parameters as in regression. The long path would be to develop an artificial intelligence whose specialty shall be to divide complex problem in small subtasks that must be effectively solvable by evolution.

Answers

Chapter 1

4, page 3. The two settings -teleonomic and non teleonomic- are in principle very different. This suspicion can be defended by common sense as follows: let us think of a young man that does not know what is life for. He comes and goes without a purpose. No person will bet for the idea that this man will earn a living. By contrast, if that young man was taught and trained to be a responsible father, he will study and work hard for this. With a bit of wisdom he will succeed. Now, if we extrapolate this reasoning to the realm of evolution, we can conclude that Directed Evolution (with a purpose) goes faster than Natural Evolution (without a purpose). So, The Evolutionary Theory, which is about Natural Evolution, predicts that Directed Evolution shall be feasible. In fact, artificial selection has produced every sort of races of hens, doves, dogs, cats, pigs and horses. Today, the good idea about Directed Evolution is this: the limits of Directed Evolution are those of your imagination. GP is of course included.

To be more specific, the Evolutionary Theory reads:

```
Variants of the genomes of living beings that were created
by random mutation and recombination were filtered by differential
survival and reproduction and upon recursion sexual barriers
appeared resulting in evolutionary segregation, in speciation
i.e., in the splitting of a species in two or more.
```

So, the correct prediction of the Evolutionary Theory in regard with GP is:

```
Java programs that are mutated and recombined at random
can be tested for acceptance by Java -this is the environment
where Java programs live- and so a library of acceptable
programs can be created, mutated and further recombined
to enrich the library.  In this case, the arisen of species
```

```
is mandatory, i.e., programs will be divided by reproductive
barriers:  recombination inside groups are expected to produce
viable offspring but recombination among groups not.
```

This prediction refers to natural evolution. The synthesis of software to fit a specific function belongs in the realm of Directed Evolution.

Our aim along this volume is to add a year of reality to these comments. For instance, let us deal with the following conundrum: Directed Evolution allows to systematically gather small change faster than Natural Evolution which wanders around without a purpose. But, at the same time, Directed Evolution is defined by specific constraints so, the evolutionary rate shall diminish with respect to Natural Evolution. This sounds like a contradiction. How is it solved? Our invitation is to take common sense as a motivation to think finer but not as a source of truth. So, our proposal is the following:

Both Natural and Directed Evolutions are telonomic. The difference is that Directed Evolution is teleonomic by its very nature while Natural Evolution is teleonomic but only a posteriori, in hindsight: the environment presents challenges and the species tries to resolve them by inventing and perfecting functions. That is why we say that in hindsight the purpose of evolution is to adapt the species to its environment. This is is a tight constraint. If besides we add more constraints to fit a specific target of Directed Evolution, the evolutionary rate shall in principle diminish. Nevertheless, artificial selection can go swiftly over loci that in nature have variants of similar or even depressed selective value. Example: sterile males cause in nature generalized rejection of males by females, so a population with one sterile male is threatened to disappear. By contrast, artificially sterilized males are currently used to combat pests in order to cause female frustration, rejection of males and population damping. On the other hand, practitioners of Directed Evolution usually work on phenotypes that boil down to one of some few loci while the environment in nature always affects the whole genome and possibly with reverse signs over different loci: that is why Directed Evolution is expected to produce faster results than Natural Evolution. Notice that Directed Evolution is anymore a promise. Rather, it is possible in the only case where variability exists. But this fails for nearly 60% of the genome in which genes are monomorphic.

Chapter 2

21, page 12. The word bug appears neatly in hindsight in evolution as follows: once evolution has ended with a marvelous product, everything else in its history that is not perfect can be named bugged. So, a bugged individual is one that is not

perfect. And therefore, `a bug is just a synonym of evolutionary activity from raw material to perfection`. So, bugs are immanent to evolution. That is why the Evolutionary Theory is obviously false.

Chapter 3

55, page 27.
A code that calculates a mean may look like this:

```
//Program  R55a MeanVarianceTarget2
//The mean of some data is calulated.
package Programs;

public class MeanVarianceTarget2 {

    private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5};
    private static final int N = DATA.length;


    private static void NumberOfData() {
        System.out.println("Number of entries = " + N);
    }

    private static void mean() {
        double sum = 0;
        for (int i = 0; i < N; i++) {
            sum = sum + DATA[i];
        }
        double mean = sum / N;
        System.out.println("Sum = " + sum);
        System.out.println("Mean = " + mean);
    }

    public static void main(String args[]) {
        NumberOfData();
        mean();
    }
}//End of  Program  R55a  MeanVarianceTarget2
```

In consequence, we begin with the following template:

```
//Program  R???? MeanTemplate
//The mean of some data is calulated.
package Programs;
public class MeanTemplate {

    private static final int DATA[] = {1, 5, 2, 4, 7, 8, 9, 5, 6, 3, 5
    private static final int N = 0;


    private static void NumberOfData() {
        System.out.println(NAMEOFVARIABLE1 + VALUEOFVARIABLE1);
    }

    private static void mean() {
        double sum = 0;
        for (int i = 0; LOOPCONDITION; i++) {
            sum = sum + DATA[i];
        }
        double mean = 0 / 1;
        System.out.println(NAMEOFVARIABLE2 + VALUEOFVARIABLE2);
        System.out.println(NAMEOFVARIABLE3 + VALUEOFVARIABLE3);
    }

    public static void main(String args[]) {
        NumberOfData();
        mean();
    }
}
```

The desired code can be restored thanks to the following grammar:

```
"N = 0"  ->  "N = DATA.length");
"NAMEOFVARIABLE1"  ->  "\"Number of entries = \"");
"VALUEOFVARIABLE1"  ->  "N");
"LOOPCONDITION"  ->  "i < N");
"NAMEOFVARIABLE2"  ->  "\"Sum = \"");
"VALUEOFVARIABLE2"  ->  "sum");
"0 / 1"  ->  "sum / N");
"NAMEOFVARIABLE3"  ->  "\"Mean = \"");
"VALUEOFVARIABLE3"  ->  "mean");
```

The program that tests our claim is `R55b MVGrammar2`.

65, page 29. Concatenation can be rewritten as replacement: concatenation of `a` to `b` can be expressed as a replacement of `a` by `a+b` inside `a`. So, no extension to grammars is needed.

Chapter IV

85, page 38. `((2 + 3) * 7) + 11 = 2, 3, add, 7, multiply, 11, add` is executed as follows:

Input	2	3	add	7	multiply	11	add
stack	2	3	5	7	35	11	46
		2		5		35	

The execution can be verbalized as follows:

```
- push 2 into the stack
- push 3
- read and remove the two values from the stack,
  add them and push the result
  into the stack for further operations.
- push 7
- read and remove the two values from the stack,
  multiply them and push the result
  into the stack for further operations.
- push 11
- add
```

95, page 42. The method `add(...)` is called from line 32 of the class `RecursiveSum`. From line 28 of code we see that it is invoked as static:

```
line 32: 28  invokestatic  #2                  //Method add:(I)I
```

The characteristics of the method are exhaustively declared here:

```
#2 = Methodref          #13.#29        // Programs/RecursiveSum.add:(I)I
#13 = Class              #41            // Programs/RecursiveSum
#29 = NameAndType        #21:#22        // add:(I)I
#21 = Utf8               add
#22 = Utf8               (I)I
```

The method `add(...)` invoked itself in line 20 of the add.code.

```
line 20: 11: invokestatic  #2                  // Method add:(I)I
```

And the same cascade of effects are triggered as before:

```
#2 = Methodref          #13.#29        // Programs/RecursiveSum.add:(
#13 = Class              #41            // Programs/RecursiveSum
#21 = Utf8               add
#22 = Utf8               (I)I
#29 = NameAndType        #21:#22        // add:(I)I
#41 = Utf8               Programs/RecursiveSum

Code:
 27: invokestatic  #2                  // Method add:(I)I

LineNumberTable:
 line 25: 27
```

We find that recursiveness is implemented transparently.

118, page 55. This Java Bytecode editor shows the structure of `.class` files very clearly. The Java code is not provided but only a `.jar` file so, no IDE is needed to run it:

1. Go to site of CE `http://classeditor.sourceforge.net/`, and download the file with the application.
2. Extract the file to a suitable folder.
3. Look for the `ce.jar` file. Launch it with double click. No IDE is required. On failure, revise and change permissions to run the application.
4. Command the application to open a `.class` file to examine, say, `AddTwoIntegers.class`.
5. To view the content of the file in very organized way: click on the `Summary` service.

120, page 55. To make evolution by hand, or genetic engineering, we use coherent changes that modify at the same time various naturally tied items. For instance, and in relation with the `AddTwoIntegers .class` file, the full procedure is the following:

1. Over Eclipse under the Projects' tab, right click over the name of the file `AddTwoIntegers.java`, select properties and determine the directory that could be something like:

   `/home/jose/workspace/eJVol17P/src/Programs/AddTwoIntegers.java`.

2. Use the menu of `REJ` to open the file `AddTwoIntegers.class`. Look for it in the same directory of the corresponding `.java` file. If the `.class` file does not exist, run program `JavacTest3` to create it. Once the file appears in the listed tree, select it and press `return`. The bytecode must appear.

3. Modify it to multiply two integers: Inside the editor of `REJ` find the method `AddTwoIntegers(...)` and localize therein the command to add: `iadd`. Right click and select: `Insert after`. Select the command for integer multiplication `imul`. Click OK. The new command must had been added to your bytecode. Next, delete the command to add: right click on `iadd` and select `Remove`.

4. Now a lot of bookkeeping follows.

5. To change the name of the class: select, refactor and rename it: `MultiplyTwoIntegers123.class`. Check that this operation affects the code below which is updated automatically.

6. Use the same procedure to update the name of the method that before added and now multiplies: `multiplyTwoIntegers(...)`.

7. To change + by $\star$: go to the menu `View` and check the box for the `Constant Pool`. A new tab with title `Constant Pool` must get open. Therein, look for the + symbol, double click over it and in the opening dialogue replace + by $\star$. You can verify that in the Editor pane, the symbol + appears anymore and that in its place the symbol $\star$ appears.

8. To save the file one has two options: `Save` for replacement of the old file by the new content but with the name of the old file. The second is `Save as..` for a brand new file. Test your saved file for multiplication with the program `R118 ClassFileRunner` to which you must give the directory of the new file.

9. It might be difficult to put the application to work but modifications on lonely items in the Constant Pool are always possible so, keep trying. You can change something, save, check the new file with Program `R118 ClassFileRunner`, modify, check again and so on. The author suffered a lot before succeeding.

144, page 72. To make the code more accessible to our understanding we have inserted comments right where an explanation is needed and directly on the code. An example of a comment is the following:

```
 (comment
(defn plus-one [x]
  (inc x))
)
```

A Clojure comment is similar to a `/* */` Java block with the difference that everything must be compilable.

All inserted examples have been taken from (clojuredocs [11] 2017). Every piece of code inside a comment can be run on the REPL of the corresponding namespace. The resultant code can be found on the file `Clojure-genetic-algorithm.txt`.

145, page 72. We followed the next procedure:

1. Go to

   http://codyshepp.com/articles/3-exploring-genetic-algorithms-with-

   This the site of `Exploring Genetic Algorithms with Clojure` by Cody Shepp (2015). Go around looking here and there.

2. Go to the end of the document and look for a link to Github that says: `You can find the full code on Github here.` Click `here`.

3. You must find a tree with a Clojure project. We will import it into Eclipse as follows:

4. Over Eclipse open a new Clojure project by following the menus: File $->$ new $->$ Other $->$ Clojure Project $->$ Give as name to your project: `clojure-genetic-algo` $->$ Finish.

5. Look for your project over the Package Explorer. If it is not visible, use: Window $->$ Show view $->$ Package explorer. Expand the tree of the project and find the file `project.clj`. Replace its content with that of the corresponding file of the Github project. Next, add a package to `src` folder: right click on `src` and select New $->$ Package $->$ Give it the name `evo`.

6. Add a file (a new namespace) to the evo package: click on the name of the package and select: New $->$ Other $->$ Clojure Namespace $->$ Give a name : `core.clj`. Open the file and replace its content by the

corresponding file of the Github repository. Do the same with the files `distances.clj`, `draw.clj`, `genetics.clj`, `util.clj`. In that way, you are cloning the tree structure of your project.

7. To run the project: open the file `core.clj` of the `evo` package and right click over its pane. Select Clojure $->$ Load file in REPL $->$ Wait for a long time that the REPL appears $->$ Type on the lower part of the REPL:

   ```
   (main 450 2000)
   ```

   Press the `Return` or `Enter` key. You have commanded to run the genetic algorithm during 450 generations and with 2000 individuals.

8. A panel will appear with a drawing depicting the quality of the solutions. The best solutions appear in red. As the algorithm advances, more and more solutions classify as good and eventually good solutions amount to a sensible fraction of the population. After finishing the given task, the best cost is reported.

9. You can close the drawing with the red square of the console tab.

146, page 72. The first thing we do is to understand the problem to be solved. This is explained in the link given by the Author inside the namespace `distances.clj`:

https://www.codeproject.com/Articles/259926/Introduction-to-Genetic-Algorit

By following the link we see that the aim is to find an acceptable solution for the salesman problem: "A salesman, Johnny Walker, is supposed to visit 10 cities in Camelot. Each traveling connection is associated with a cost (i.e., the time for the trip). The problem is to find the cheapest round-route that visits each city exactly once and returns to the starting point.". The site presents a beautiful diagram of the cities and its distances measured in arbitrary units of time. This is clearly an optimization quantitative problem and so an acceptable solution can be found with the help of evolution in the form of genetic algorithms.

To make the code more accessible to our understanding we have inserted comments. The resulting code can be seen in the file `Clojure-genetic-algo.txt`.

148, page 73. We cannot speak of a translation of mentioned Java program to Clojure because these languages are too different. Instead, we attack the given problem in Clojure directly applying what we have learned from the studied genetic algorithm. Anyway we need some tools and a plan:

1. A tool to generate random arithmetic operations.

2. A seed code to begin a substitution game of one arithmetic operation by another chosen at random.

3. A loop for trying new arithmetic operations. The loop ends when a match has been achieved.

4. To test a given operation 20 pairs of random numbers are picked up. The output is compared to the expected one pair by pair. Absolute values of differences are added together. Zero total difference means matching. The probability of error is not zero.

To develop the code we opened a new project in Clojure with name: `clojure-arithm-operations`. A REPL was open. To begin encoding we developed the Clojure code for a generator of random operations. After testing it with the REPL, we copied it into the pane of our project. Tests were added as comments. And so on. The code with comments is contained in the file `ClojureArithOperations.txt`. The clean code is in `ClojureArithOperationsCleanCode.txt`. Another more professional, Clojure-idiomatic version is presented in `Clojure-arithm-operations.pro.txt`

149, page 73. We do not want to burn too much computer time. So, we will reproduce a human made version of that code. It is:

```
(defn mean-one-line
 "The mean of a data vector"
 [x]
   (/ (reduce + x) (count x)))


(comment
  (mean-one-line [1 2 3]);2
  (mean-one-line [0.1 0.2 0.3]);0.20000000000000004
 )
```

So, we can begin a project `clojure-mean` with the operators { /, reduce + count} and the variable `x` or a convenient replacement. Random Clojure expressions were created by a suitable grammar in order to fit the code for the mean. We found a problem: the created expressions were not in general correct and so executions were halted by the resulting run-time exceptions that was not caught by Java -contrary to what one would desire but in agreement with the general experience:

complex software inescapably comes with mortal bugs. Anyway, to verify that the program can indeed produce the expected code, trial Clojure expressions were compared with that of the mean and the program was given to run until matching. The code was saved to the text file `Clojure-the-mean-simple-attack.txt`.

150, page 73.

1. We went to the mentioned site

   https://github.com/vollmerm/fungp

   This the site of `fungp, A genetic programming library for Clojure` by Mike Vollmer (2012).

2. A tree with a Clojure project was found. We imported it into Eclipse as follows:

3. Over Eclipse we opened a new Clojure project by following the menus: File $->$ new $->$ Other $->$ Clojure Project $->$ Gave as name to the project: `fungp` $->$ Finish.

4. The tree of the project was expanded and the file `project.clj` was found. Its content was replaced by:

```
(defproject fungp "0.3.2"
  :description "Genetic programming in Clojure"
  :jvm-opts ["-XX:ReservedCodeCacheSize=128m" "-server"]
  :dependencies [[org.clojure/clojure "1.5.1"]]
  :url "http://gaia.ecs.csus.edu/~vollmerm/gp/"
  :license {:name "GNU General Public License v3"
            :url "http://www.gnu.org/licenses/gpl.html"}
:plugins [[lein-marginalia "0.7.1"]])
```

5. The content of the file `core-test-clj` was replaced by:

```
(ns fungp.test.core
  (:use [fungp.core])
  (:use [clojure.test]))

(defn list-or-cons? [tree]
  (or (list? tree)
      (= clojure.lang.Cons (type tree))))
```

```
(def test-functions '[[+ 2][- 2][* 2][inc 1][dec 1]])

(def test-terminals '[x])

(deftest test-create-tree
         (let [tree-type (type (create-tree 5 test-terminals [] te
:grow))]
           (is (or (= tree-type clojure.lang.Symbol)
                   (= tree-type clojure.lang.Cons)))))

(deftest test-module-tree
         (let [tree (create-module-tree 5 test-terminals [] test-f
0
0 :grow)]
           (do (is (list-or-cons? tree))
               (is (= (first tree) 'let))
               (is (vector? (second tree))))))

(deftest test-tree-operations
         (let [tree (create-tree 5 test-terminals [] test-function
               subtree (rand-subtree tree)]
           (do (is (seq? (replace-subtree tree '(+ x x))))
               (is (seq? subtree)))))

(deftest test-truncate
         (let [tree (create-tree 5 test-terminals [] test-function
           (do (is (= tree (truncate tree 10)))
(is (not (= tree (truncate tree 2)))))))
```

6. A new package with name `fungp` was added to `src` folder.

7. The src folder of the repository was cloned into Eclipse by copy&paste. To that aim, a new folder was added to that package. Its name was: `sample`. Its content was filled with the corresponding files (namespaces) at the Github repository: `cart.clj`, `compile_ants.clj`, robot.clj and sinbowl.clj.

   The same was done with the other 4 namespaces `core.clj`, `defined_branches.clj`, `tutorial.clj` and `util.clj`. These 4 files were added not to the sample folder but to the fungp package.

8. The project contains a general GP library in the file core.clj. How to use that library is exemplified with various case studies that come in the sample folder. Each one of these files needs a brand new REPL so, one must close ancient ones. Else one must require the needed namespace from the active REPL. Samples run very slowly and can be triggered as follows:

   - To run `cart.clj` we opened the file `cart.clj` of the `sample` package and right clicked over its pane. Selected Clojure $->$ Load file in REPL $->$ Waited for a long time that the REPL appears $->$ Type on the lower part of the REPL:
     `(test-cart 3 4)`
     We pressed the `Return` or `Enter` key. With this we commanded to run the genetic algorithm during 3 iterations and with 4 migrations.
   - To run compile-ants.clj in a new REPL:
     `(test-compile-ants 3 4)`
   - To run `interpret-ants.clj`:
     `(test-ants 3 7)`
   - To run `robot.clj`:
     `(test-robots 2 7)`
   - To run `sinbowl.clj`:
     `(test-sinbowl 3 9)`

151, page 73. In first place we tried to understand the `tutorial.clj` that can be run with, say,

```
(test-genetic-program 7 7)
```

This tutorial exemplifies how to solve a general regression problem in which some data on the plane is fitted by a function that results as combination of some functions of a given list. We have inserted many comments inside the code of this and of every namespace. The modified content of files was copied as plane text to the file `ClojureFungp.txt` and appended to the Clojure package.

152, page 73. The mean of a data in a vector `x` in Clojure is calculated as follows:

```
(defn mean-one-line
  "The mean of a data vector"
  [x]
    (/ (reduce + x) (count x)))
```

So, we imagined that the GP of the mean should had been a straightforward exercise. The reality was otherwise: we thought at first that the + operator corresponded with a function with 2 arguments since it is a binary operation. But this choice leaded to nothing. Next we noticed that the + operator plays in this program the role of a constant but this option produced runtime NullPointerExceptions that we were unable to get rid of.

153, page 74.

1. We went to the mentioned site

   http://gigasquidsoftware.com/blog/2016/07/18/genetic-programming-w

2. The link to Github was followed.

3. A tree with a Clojure project was found. We imported it into Eclipse using our nails:

4. Over Eclipse we opened a new Clojure project by following the menus: File $->$ new $->$ Other $->$ Clojure Project $->$ Gave as name to the project: `genetic-programming-spec` $->$ Finish.

5. The tree of the project was expanded and the file `project.clj` was found. Its content was replaced by:

```
(defproject genetic-programming-spec "0.1.0-SNAPSHOT"
 :description "FIXME: write description"
 :url "http://example.com/FIXME"
 :license {:name "Eclipse Public License"
           :url "http://www.eclipse.org/legal/epl-v10.html"}
 :dependencies [org.clojure/clojure "1.9.0-alpha17"])
```

6. The content of the file `core-test-clj` was replaced by:

```
(ns genetic-programming-spec.core-test
  (:require [org.clojure.test :refer :all]
            [org.clojure/test.check "0.9.0"]
            [genetic-programming-spec.core :refer :all]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

7. A new package with name `genetic-programming-spec` was added to `src` folder.

8. A new file (new namespace) was added to the `genetic-programming-spec` package. Its name was: `core.clj`. Its content was replaced by the corresponding file of the Github repository. In that way we cloned the tree structure of our project. This content appears in the file `Clojure-gp-spec-core.txt`.

9. To run the project we opened the file `core.clj` of the `genetic-programming-spec` package and right clicked over its pane: Clojure $->$ Load file in REPL $->$ Waited for a long time that the REPL appears $->$ we typed on the lower part of the REPL:

   ```
   (evolve 100 100 7 ["hi" true 5 10 "boo"])
   ```

   We pressed the `Return` or `Enter` key. With this we commanded to run the genetic algorithm during 100 individuals and during 100 generations and with an initial creature that must be evolved.

10. An abundant answer was displayed.

156, page 75. a) Darwinian evolution has no purpose at all. But deliberate or directed evolution has a purpose given by the person or system in charge that imposes the fitness criterion. So, the expression `Deliberate, Darwinian evolution` is a contradiction. Technically, this type of contradiction is called oxy-moron that happens when two contradictory terms are concatenated to conform a new meme. The word oxymoron comes from the Greek *acute-dull*, which is an oxymoron in itself and so the epitome of all of them. Oxymora illustrate the complexities of human languages and that is why they must be described by context sensitive grammars, a fact that causes automatic translation to be a very difficult problem originating in its turn the need of human supervising. That is how many companies in the linguistic trade earn a life. b) The appreciation that "Darwinian evolution is itself a designer worthy of significant respect if not of devotion" is based on a real fact: to design and make living beings is extremely difficult no matter whatever technique, method or hint you can use. c) The slogan `life is the evolution of programs` means in first place that living organisms are built by software that can evolve. True. To be exact we must keep in mind that software consists of programs + initial conditions. Now, all that we really know about life seems to imply that it needs initial conditions over which the genetic algorithm develops organismic life. In second place the slogan is the very Evolutionary Theory of the origin of species. This is obviously false: were life the result of the

evolution of software, it would be visible in our own body the battle of evolution against bugs in the form of thousands of malfunctions and malformations. Since those tracks are lacked, the evolutionary theory is false. Everyone knows this by instinct.

Answer:

158, page 76. We created a Clojure project in Eclipse CW with name `clojure-pucks`. Cloned the file `project.clj`. Next we looked for the core but it happens to be a bundle of files. So, cloning with own nails was not a good idea. We discovered in that way the need to use an automatic methodology. That is why we followed `collab-uniba` ([16] 2015) to import a GitHub project into Eclipse. We followed through the menus: Window $->$ Show view $->$ Other $->$ Git repository

And we were done by just typing the link to the Github project:

```
https://github.com/lspector/pucks.git
```

Eclipse opened a new pane for Github with name `Git Repositories` and we found there all the files of the project. To run this project we executed the instructions given in the page of the project:

```
you can open a "world" file (in src/pucks/worlds/*),
load the file into a new
read-eval-print loop (REPL),
and evaluate an expression such as:

(run-pucks (agents) (settings))
```

This was done but no result was seen. To understand why we looked at the properties of the project: it has nothing to do with Clojure. But nevertheless it became clear that the System opened a folder for Git. So, we import that folder into the old Clojure project we opened at the beginning. To that aim we right clicked on the name of the Clojure project `clojure-pucks` and imported our project with the option: Import $->$ From folders or archive $->$ Source (`/home/jose/git/pucks`) $->$ Finish. No change was observed at the Clojure project. But it became possible to open a REPL for the file `src/pucks/worlds/core` at the Git Repositories pane. We closed this REPL because according to the instructions given by the README document, there are many namespaces that can be run directly. These are located in the folder `worlds.ai` (8 variants) and `worlds.dev` (23 variants). Example: the tree of the repository was expanded up to the file

`pucks.worlds.ai.world1`
for which a REPL was opened and (after a long time) we typed therein:
`(run-pucks (agents) (settings))`
In response a wonderful graphic simulation was triggered. The very same procedure serves to run anyone of the worlds as in the `ai` as in the `dev` folders. Every REPL was closed before opening a new one and before a new launching.

To enjoy the simulations, the `Description` section in the README file was read. To know the nature of a puck, one must click on it in order to open a descriptive pane.

162, page 77.
In regard with the program on page `17 (11 of 34)` of the mentioned article, it reads:

```
(QUOTE (POP 1)
 QUOTE (DUP 1-DO*)
 DUP 2 < IF)
```

We can understand what this piece of code does if we translate it to Java:

```
if (( n = 0) ||(n = 1)) return 1;//(QUOTE (POP 1)    (DUP 2 < IF))
else
return  (n * factorial(n-1)); // ( QUOTE ((DUP 1 -) DO*))
```

In regard with running the regression demo on Clojush we went through the menus
Window − > Show view − > Other − > Git repository
and typed

`https://github.com/lspector/Clojush.git`

After restoring the view, we expanded the project tree and selected the file: `workingtree.src.clojush.problems.demos.simple-regression`. It was opened and a REPL was commanded to be open. But nothing happened. So, we open a new Clojure project with name `clojush` and imported the Git repository after right clicking on the name of the project:
Import − > From folders or archive − > Source (/home/jose/git/clojush) − > Finish.
This time a REPL was possible to be opened and therein we typed:
(pushgp argmap)
after running the program, it ended with the following output:

```
 step: 1000
program: (in1 integer_mult in1 in1 integer_add 2 integer_add integer_su
integer_mult in1 integer_add)
errors: (0 0 0 0 0 0 0 0 0 0)
total: 0
size: 13
```

To try to understand better we rewrote the program as:

```
(x * x x + 2 + - x * x +)
(x2 x + 2 + - x * x +)
((x2 + x + 2))     - x * x +)
```

For another run the output was

```
(in1 integer_mult in1 integer_sub 2 integer_sub in1 integer_sub in1
integer_mult in1 integer_add)
(x * x - 2 - x - x * x +)
```

A longer expression was output for third run. A fourth one showed no convergence.

To try to make sense of these programs we made use of two rules:

First: When things are clear, execute them. Otherwise, make use of two additional rules:

1 2 3 +

becomes

5 1

and

Second: 3 + is immutable.

The Author was unable to apply these rules successfully to end with the demanded expression $x^3 - 2x^2 - x$.

163, page 77.

We read chapters 1 and 2 and other selected parts of the Helmuth's Dissertation. These showed us that GP with `Clojure + Push` is right now a nice and realizable option for working for a Ph.D. degree. The relevant features of Push and the compelling need for its evolution begin at page 19 of the dissertation.

It is a good idea to read the overall idea as presented in a short article by (Helmuth and Spector, [29]:

One can also give a look to the official page of the Push language (Push-language, [68] 2017). It has been implemented in many languages and Spector choose -not in vain- the Clojure implementation for his works.

165, page 78. A Clojure project was opened with name `clojure-taggp`. The definition of the project (`project.clj`) was updated from the Git page. The core (`taggp.core`) was also updated. A new folder was added to the folder `taggp/src/taggp/examples/`. The four namespaces of the original `examples` folder were also copied with exactly the same name. Thus, we cloned the tree-structure of the project with all its mutual callings. To run an example we opened it with a REPL (and closed any other) and typed

(-main)

In this case, the minus symbol is an ordinary character and not an operator. A long output is printed with a final answer.

The main idea of this work is to use Push to develop " a new technique for evolving modular programs with genetic programming. The technique is based on the use of "tags" that evolving programs may use to label and later to refer to code fragments. " (Spector et al. [78] 2011).

167, page 78. We downloaded the `source code zip file` from
`https://github.com/thelmuth/Clojush/releases/tag/benchmarksPaper`

Next, we followed the menus File $->$ Import $->$ From folder or archive $->$ next $->$ Import Source (the directory where the expanded downloaded file resides) $->$ Finish.

Next we opened the namespace `clojush.problems.software.even-squares`, added a REPL to it and in response the namespace was run and an abundant output was displayed.

169, page 78. Many projects that seemed to be wonders appeared by July/2017. We implemented that by `taylodl` ([85] 2014) which allows to make music on the fly. The corresponding code was saved to the text file `Clojure-scale.txt`.

Bibliography

[1] ANDO SAABAS (2011) JBE: Java Bytecode Editor
http://cs.ioc.ee/~ando/jbe/
Verified 12/VIII/2017 54

[2] ARCADIA (2017)
https://arcadia-unity.github.io/
Verified 12/VIII/2017 78

[3] ARHIPOV ANTON (2011) Java Bytecode Fundamentals
http://arhipov.blogspot.com.co/2011/01/java-bytecode-fundamentals.html
Verified 12/VIII/2017 38

[4] ARTURADIB (2011) Slash-a : Programming language and C++ library for (linear) genetic programming
https://github.com/arturadib/slash-a
Verified 12/VIII/2017 39

[5] ADAM BARD (2017) (2011) Learn X in Y minutes Where X=clojure
https://learnxinyminutes.com/docs/clojure/
Verified 12/VIII/2017 67

[6] ADAM BARD, HANDSOME WEB DEVELOPER AND A COMMUNITY (2017B) (2011) Learn X in Y minutes Where X=clojure macros
https://learnxinyminutes.com/docs/clojure-macros/
Verified 12/VIII/2017 70

[7] ERIC.BRUNETON, ROMAIN.LENGLET, THIERRY.COUPAYE (2002) ASM: un outil de manipulation de code pour la réalisation de systèmes adaptables
http://asm.ow2.org/current/asm-fr.pdf
Verified 12/VIII/2017 56

[8] ELI BENDERSKY (2017) Clojure: The Perfect Language to Expand Your Brain?

https://dzone.com/articles/clojure-the-perfect-language-to-expand-
Verified 12/VIII/2017 vi

[9] CE (2004) Java File Class Editor
http://classeditor.sourceforge.net/
Verified 12/VIII/2017 55

[10] CLOOJ (2017)
https://github.com/arthuredelstein/clooj
Verified 12/VIII/2017 72

[11] CLOJUREDOCS (2017)
https://clojuredocs.org/
Verified 12/VIII/2017 90

[12] CLOJURE.ORG (2017) The Reader
https://clojure.org/reference/reader
Verified 12/VIII/2017 73

[13] CLOJURE.ORG (2017B) Spec guide
https://clojure.org/guides/spec
Verified 12/VIII/2017 74

[14] CODEJAVA (2017) JAVA NIO - COPY FILE OR DIRECTORY EXAMPLES
http://www.codejava.net/java-se/file-io/java-nio-copy-file-or-dire
Verified 12/VIII/2017 50

[15] CODY SHEPP (2015) EXPLORING GENETIC ALGORITHMS WITH CLOJURE
http://codyshepp.com/articles/3-exploring-genetic-algorithms-with-
Verified 12/VIII/2017 72

[16] COLLAB-UNIBA15 (2015) How to import a GitHub project into Eclipse
https://github.com/collab-uniba/socialcde4eclipse/wiki/How-to-impor
Verified 12/VIII/2017 98

[17] BRYCE COVERT (2016) Applying Genetic Programming to Bytecode and Assembly
http://digitalcommons.morris.umn.edu/cgi/viewcontent.cgi?article=10
Verified 56

[18] ERIC C. COLLOM (2014) USE lisp WITH game - Making an Adventure Game with Clojure.

https://www.youtube.com/watch?v=lql2yFXzKUs
Verified 78

[19] DARWIN CHARLES (1859) ON THE ORIGIN OF SPECIES BY MEANS OF NATURAL SELECTION, OR THE PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE. LONDON: MURRAY. [1ST ED.]
http://darwin-online.org.uk/converted/pdf/1859_Origin_F373.pdf
Verified 12/IV/2017 1

[20] DAVEY MIKE (2010) A TURING MACHINE - OVERVIEW
https://www.youtube.com/watch?v=E3keLeMwfHY
Verified 12/VIII/2017 9

[21] SEAN LUKE. (2017). ECJ Then and Now. In Proceedings of GECCO '17 Companion, Berlin, Germany, July 15-19, 2017, 8pages. DOI: http://dx.doi.org/10.1145/3067695.3082467
https://cs.gmu.edu/~sean/papers/gecco17-ecj.pdf
Verified 12/VIII/2017 56

[22] ACHIYA ELYASAF, MICHAEL ORLOV, MOSHE SIPPER (2013) A HeuristicLab Evolutionary Algorithm for FINCH
https://www.cs.bgu.ac.il/~orlovm/papers/heuristiclab.pdf
Verified 12/VIII/2017 56

[23] EMRE, (2012) how-many-characters-can-utf-8-encode
http://stackoverflow.com/questions/10229156/how-many-characters-can-utf-8-
Verified 12/VIII/2017 53

[24] ENGEL JOSHUA (1999) Programming for the Java™ Virtual Machine
https://www.safaribooksonline.com/library/view/programming-for-the/0201309
Verified 12/VIII/2017 39

[25] EPOCH (2017)
http://www.epochx.org/quickstart-guide.php
Verified 12/VIII/2017 56

[26] TIMS GARDNER (2015) Quick Arcadia demo with floating REPL
https://www.youtube.com/watch?v=_p0co13WYPI
Verified 12/VIII/2017 78

[27] GORILA-REPL (2017) gorilla-repl.org
http://gorilla-repl.org/
Verified 12/VIII/2017 72

[28] THOMAS HELMUTH (2015) Dissertation: GENERAL PROGRAM SYNTHESIS FROM EXAMPLES USING GENETIC PROGRAMMING WITH PARENT SELECTION BASED ON RANDOM LEXICOGRAPHIC ORDERINGS OF TEST CASES. University of Massachusetts Amherst September 2015 College of Information and Computer Sciences
https://web.cs.umass.edu/publication/docs/2015/UM-CS-PhD-2015-005.p
Verified 12/VIII/2017 77

[29] THOMAS HELMUTH, LEE SPECTOR (2015) General Program Synthesis Benchmark Suite
http://thelmuth.github.io/GECCO_2015_Benchmarks_Materials/
Verified 12/VIII/2017 100

[30] RICH HICKEY (2016) Introducing clojure.spec
http://blog.cognitect.com/blog/2016/5/23/introducing-clojurespec
Verified 12/VIII/2017 74

[31] DANIEL HIGGINBOTHAM (2016) Clojure for the Brave and True
http://www.braveclojure.com/clojure-for-the-brave-and-true/
Verified 12/VIII/2017 64

[32] DANIEL HIGGINBOTHAM (2016B) Clojure for the Brave and True,, Apendix A
http://www.braveclojure.com/appendix-a/
Verified 12/VIII/2017 70

[33] HIROKUNI KIM (2016) Clojure By Example
https://kimh.github.io/clojure-by-example/#about
Verified 12/VIII/2017 64

[34] HOMEPAGES (2017) JVM Opcode Reference
http://homepages.inf.ed.ac.uk/kwxm/JVM/index.html
Verified 12/VIII/2017 39

[35] IBM WATSON: HOW IT WORKS (2017)
https://www.youtube.com/watch?v=_Xcmh1LQB9I
Verified 12/VIII/2017 8

[36] JOSER(2017) program-copies-class-file-but-distorts-magic-value. StackOverflow
http://stackoverflow.com/questions/43232532/program-copies-class-fi
Verified 12/VIII/2017 50

[37] JENKOV JAKOB (2014) Java NIO FileChannel
http://tutorials.jenkov.com/java-nio/file-channel.html
Verified 12/VIII/2017 49

[38] JOSER (2016) Convert a number from base 10 to N
https://stackoverflow.com/questions/24563998/convert-a-number-from-base-10
Verified 9/VIII/2017 7

[39] JOSERAJ JACKSON (2016) The JVM Architecture Explained An overview of the different components of the JVM, along with a very useful diagram
https://dzone.com/articles/jvm-architecture-explained
Verified 12/VIII/2017 34

[40] SAMI.KOIVU (AT) GMAIL.COM (2011) reJ
http://rejava.sourceforge.net/
Verified 9/VIII/2017 55

[41] KOZA J, (1980, 1996) Genetic programming. On the Programming of Computers by Means of Natural Selection. A Bradfort Book. The MIT Press. First printing: 1980; Fifth Printing: 1996. v

[42] KOZA J, (2007) General information about Genetic Programming.
http://www.genetic-programming.org/
Verified 12/VIII/2017 v

[43] THOMAS KUHN, OLIVIER THOMANN (2017), ABSTRACT SYNTAX TREE
http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html
Verified 12/VIII/2017 60

[44] LESLIE LAMPORT (2015?) Hard die
http://video.ch9.ms/ch9/7625/d068b1a0-08be-4547-81ef-5e0048117625/video4.m
Verified 12/VIII/2017 75

[45] LESLIE LAMPORT (2015B) The PlusCal Algorithm Language
http://lamport.azurewebsites.net/tla/pluscal.html
Verified 12/VIII/2017 75

[46] LESLIE LAMPORT (2017) The TLA Home Page
http://lamport.azurewebsites.net/tla/tla.html
Verified 12/VIII/2017 75

[47] LEARN PYTHON (2016)
https://www.learnpython.org/en/Hello%2C_World%21
Verified 12/VIII/2017 8

[48] LIGHT TABLE (2017) (2016)
http://lighttable.com/
Verified 12/VIII/2017 72

[49] LILAC - A JAVA ASSEMBLER (2017)
http://lilac.sourceforge.net/index.html
Verified 12/VIII/2017 39

[50] LINDHOLM TIM, FRANK YELLIN, GILAD BRACHA, ALEX BUCKLEY (2015) ORACLE Data Types The Java Virtual Machine Specification. Java SE 8 Edition
https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html#jvms-
Verified 12/VIII/2017 52

[51] LOCEY KENNETH J. AND JAY T. LENNON (2016) Scaling laws predict global microbial diversity PNAS vol. 113 no. 21, 5970–5975, doi: 10.1073/pnas.1521291113
http://www.pnas.org/content/113/21/5970.abstract
Verified 12/IV/2017 1

[52] SAR MAROOF (2016) JAVA QUIZ: PASSING OBJECTS TO METHODS
https://dzone.com/articles/java-quiz-passing-objects-to-methods
Verified 12/IV/2017 11

[53] CARIN MEIER (2016) Genetic Programming With clojure.spec
http://gigasquidsoftware.com/blog/2016/07/18/genetic-programming-wi
Verified 12/IV/2017 74

[54] CARIN MEIER (2016B) One Fish Spec Fish Squid's Blog
http://gigasquidsoftware.com/blog/2016/05/29/one-fish-spec-fish/
Verified 12/IV/2017 74

[55] CARIN MEIER (2016C) ClojureTV: Genetic Programming with clojure.spec
https://www.youtube.com/watch?v=xvk-Gnydn54&t=7s
Verified 12/IV/2017 74

[56] CARIN MEIER (2016D) An experiment with self healing functions and clojure.spec
https://github.com/gigasquid/self-healing
Verified 12/IV/2017 74

[57] MOINUL ISLAM (2015) HOW TO GET YOUR NETBEANS PROJECT INTO ECLIPSE
http://stackoverflow.com/questions/21535023/how-to-get-your-netbeans-proje
Verified 12/VIII/2017 vii

[58] RAMSEY NASSER (2017) SYMBOLIC ASSEMBLY: USING CLOJURE TO META-PROGRAM BYTECODE
https://www.youtube.com/watch?v=eDad1pvwX34
Verified 12/VIII/2017 78

[59] NEWDAVE (2017) CONCATENATIVE LANGUAGE
http://concatenative.org/wiki/view/Concatenative%20language
Verified 12/VIII/2017 39

[60] PRESSLER RON (2016) Why Writing Correct Software Is Hard . . . and why math (alone) won't help us
http://blog.paralleluniverse.co/2016/07/23/correctness-and-complexity/
Verified 12/VIII/2017 12

[61] ORACLE (2015) THE JAVA VIRTUAL MACHINE SPECIFICATION, JAVA SE 8 EDITION
https://docs.oracle.com/javase/specs/index.html
Verified 12/VIII/2017 39, 53

[62] MICHAEL ORLOV MOSHE SIPPER (2009) Genetic programming in the wild: Evolving unrestricted bytecode. DOI: 10.1145/1569901.1570042 Conference: Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009
https://www.researchgate.net/publication/220741836_Genetic_programming_in_t
Verified 12/VIII/2017 56

[63] MICHAEL ORLOV (2017) Evolving Software Building Blocks with FINCH. In Proceedings of GECCO '17 Companion, Berlin, Germany, July 15–19, 2017, 2 pages. DOI: 10.1145/1569901.1570042 Conference: Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009
https://doi.org/10.1145/3067695.3082521
Verified 12/VIII/2017 56

[64] LAURENT PETIT (2016)Counterclockwise User Guide
http://doc.ccw-ide.org/documentation.html#_install_counterclockwise
Verified 12/VIII/2017 64

[65] POSTSPECTACULAR (2014) Morphogen virus structure.
https://www.youtube.com/watch?v=vXlOB4NfAE0
Verified 12/VIII/2017 78

[66] PROCESSING FOUNDATION (2017) Processing (2014)
https://processing.org/
Verified 12/VIII/2017 70

[67] PUJOS BRUNO AND NASSIM EDDEQUIOUAQ (2014) 0xCAFEBABE ? - java class file format, an overview - LSE Blog
https://blog.lse.epita.fr/articles/69-0xcafebabe-java-class-file-f
Verified 12/VIII/2017 51

[68] PUSHLANGUAGE (2017)
http://pushlanguage.org
Verified 12/VIII/2017 101

[69] QUIL (2017) ClojureScript library for creating interactive drawings and animations
http://quil.info/
Verified 12/VIII/2017 70

[70] REELLEARNING (2012) Data Structures: Stack (Abstract Data Type)
https://www.youtube.com/watch?v=XSdXSmwb550
Verified 12/VIII/2017 38

[71] REFERENCE (2017) What is the function of the processor?
https://www.reference.com/technology/function-processor-aca06f931c9
Verified 12/VIII/2017 33

[72] SARYADA WAYAN (2013A) HOW TO READ FILE USING FILES.NEWBUFFEREDREADER?
https://kodejava.org/how-to-read-file-using-files-newbufferedreade
Verified 12/VIII/2017 50

[73] SARYADA WAYAN (2013B) HOW TO WRITE FILE USING FILES.NEWBUFFEREDWRITER?
https://kodejava.org/how-to-write-file-using-files-newbufferedwrit
Verified 12/VIII/2017 50

[74] SAUMONT PIERRE-YVES (2014) WHAT'S WRONG WITH JAVA 8: CURRYING VS CLOSURES?
https://dzone.com/articles/whats-wrong-java-8-currying-vs
Verified 12/VIII/2017 61

[75] YOGTHOS: DMITRI SOTNIKOV (2013) Example of a simple GA using Clojure
https://github.com/yogthos/genetic-algorithm-example
Verified 12/VIII/2017 70

[76] LEE SPECTOR (2002) "Genetic Programming and Evolvable Machines"
http://hampshire.edu/lspector/pubs/push-gpem-final.pdf
Verified 12/VIII/2017 77

[77] LEE SPECTOR (2005) And now, digital evolution. —The Boston Globe— August 29, 2005,
http://archive.boston.com/news/globe/editorial_opinion/oped/articles/2005/0
Verified 12/VIII/2017 76

[78] SPECTOR, L., B. MARTIN, K. HARRINGTON, AND T.HELMUTH. 2011. (2011) Tag-Based Modules in Genetic Programming. In Proceedings of the Genetic and Evolutionary Computation Conference(GECCO-2011). ACMPress.pp.1419-1426.
http://faculty.hampshire.edu/lspector/pubs/spector-tags-gecco-2011-with-ci
The correction to an error in this paper can be seen at
http://faculty.hampshire.edu/lspector/tags-gecco-2011/
Verified 12/VIII/2017 101

[79] LEE SPECTOR (2012) `taggp` Tree-based genetic programming with tagging.
https://github.com/lspector/taggp
Verified 12/VIII/2017 78

[80] LEE SPECTOR (2015) Genetic Programming in Clojure
https://www.youtube.com/watch?v=HWMJdO4klIE
Verified 12/VIII/2017 75

[81] LEE SPECTOR (2016) Clojinc. Steps toward Clojure, starting from zero.
https://github.com/lspector/clojinc
Verified 12/VIII/2017 67

[82] LEE SPECTOR (2016B) Pucks: An environment for experiments and education in artificial intelligence and artificial life.
https://github.com/lspector/Pucks
Verified 12/VIII/2017 76

[83] LEE SPECTOR (2017) Clojush.
https://github.com/lspector/Clojush
Verified 12/VIII/2017 77

[84] LEE SPECTOR (2017B) SMAC-PushGP-GECCO-2017. A paper for EuroGP-2017 on using SMAC with PushGP.
https://github.com/lspector/SMAC-PushGP-GECCO-2017
Verified 12/VIII/2017 78

[85] TAYLODL (2014) Making Music with Clojure – An Introduction to MIDI.
https://taylodl.wordpress.com/2014/01/21/making-music-with-clojure-
Verified 12/VIII/2017 101

[86] GABRIELE TOMASSETTI (2017) PARSING IN JAVA (PART 1): STRUCTURES, TREES, AND RULES. JAVA ZONE
https://dzone.com/articles/parsing-in-java-part-1-structures-trees
Parsing in Java (Part 2): Diving Into CFG Parsers. Java Zone
https://dzone.com/articles/parsing-in-java-part-2-diving-into-cfg-
Verified 12/VIII/2017 59

[87] TUTORIALSPOINT (2017) CLOJURE TUTORIAL
https://www.tutorialspoint.com/clojure/index.htm
Verified 12/VIII/2017 67

[88] TZIMOULIS NIKOLAS (2015) A SIMPLE IMPLEMENTATION OF A TURING MACHINE IN JAVA
https://gist.github.com/NikolasTzimoulis/2846116
Verified 12/VIII/2017 9

[89] UDIPROD (2013) THE HALTING PROBLEM. 8. SO CAN HUMANS SOLVE THE HALTING PROBLEM?
http://www.zutopedia.com/halting_problem.html#faq
Verified 12/VIII/2017 9

[90] UNDEFINED BEHAVIOR (2016) IMPOSSIBLE PROGRAMS (THE HALTING PROBLEM)
https://www.youtube.com/watch?v=wGLQiHXHWNk
Verified 12/VIII/2017 9

[91] VISUAL STUDIO CODE (2017) (2014)
https://code.visualstudio.com/
Verified 12/VIII/2017 72

[92] UNITY3D (2017) (2017)
https://unity3d.com/
Verified 12/VIII/2017 78

[93] LARS VOGEL, SIMON SCHOLZ (C) (2009 - 2016) ECLIPSE JDT - ABSTRACT SYNTAX TREE (AST) AND THE JAVA MODEL
http://www.vogella.com/tutorials/EclipseJDT/article.html
Verified 12/VIII/2017 60

[94] MIKE VOLLMER, (2012), FUNGP: A GENETIC PROGRAMMING LIBRARY FOR CLOJURE
https://github.com/vollmerm/fungp
Verified 12/VIII/2017 73

[95] WIKIPEDIA. (2017A) HALTING PROBLEM
https://en.wikipedia.org/wiki/Halting_problem
Verified 12/VIII/2017 8

[96] WIKIPEDIA CONTRIBUTORS, (2017C) 'STACK MACHINE', WIKIPEDIA, THE FREE ENCYCLOPEDIA,
https://en.wikipedia.org/w/index.php?title=Stack_machine&oldid=767832463
Verified 12/VIII/2017 38

[97] WIKIPEDIA CONTRIBUTORS (2017D), "JAVA BYTECODE INSTRUCTION LISTINGS," WIKIPEDIA, THE FREE ENCYCLOPEDIA,
https://en.wikipedia.org/wiki/Java_bytecode_instruction_listings
Verified 12/VIII/2017 39, 53

[98] WIKIPEDIA CONTRIBUTORS (2017E), "WIKIPEDIA CONTRIBUTORS. LIST OF JVM LANGUAGES " WIKIPEDIA, THE FREE ENCYCLOPEDIA,
https://en.wikipedia.org/wiki/List_of_JVM_languages
Verified 12/VIII/2017 57

[99] WIKIPEDIA CONTRIBUTORS (2017E), ""CLOJURE," WIKIPEDIA, THE FREE ENCYCLOPEDIA,
https://en.wikipedia.org/w/index.php?title=Clojure&oldid=778191584
Verified 12/VIII/2017 61

[100] WILLIAMS L, SMITH B (2007) IMPORTING AND EXPORTING JAVA PROJECTS IN ECLIPSE
http://agile.csc.ncsu.edu/SEMaterials/tutorials/import_export/
Verified 12/VIII/2017 vii

[101] WOODFORD CHRIS. (2016) COMPUTERS
http://www.explainthatstuff.com/howcomputerswork.html
Verified 12/VIII/2017 33

[102] WYLIE R (2015) BBC - Earth - Hawaii: The islands where evolution ran riot. 25 June 2015
http://www.bbc.com/earth/story/20150625-islands-where-evolution-ra
Verified 12/VIII/2017 12