FALSIFICATION OF THE EVOLUTIONARY THEORY WITH MICROLISP

José del Carmen Rodríguez Santamaría
https://www.ejristos.com
Bogotá, Colombia

February 28, 2022

Contents

1 INTRODUCTION **2**

2 NOTEBOOK COMPUTERS AND MICROLISP **4**
2.1 A MicroLisp 0.1 Computer 4
2.2 Antecedents of MicroLisp 5
2.3 MicroLisp 0.1 over binary strings 6
2.4 MicroLisp 0.1 over digits 6
2.5 MicroLisp 0.0 7
2.6 GP (Genetic Programming) with MicroLisp 0.0 8
2.7 GP with MicroLisp 0.1 10

3 GP WITH REGULATION **11**
3.1 MicroLisp 0.2 14
3.2 GP with MicroLisp 0.2 14
3.3 MicroLisp 0.3 15
3.4 Loops 15
3.5 Fault tolerance 16
3.6 MicroLisp 0.3 Advanced Editor 17
3.7 GP with MicroLisp 0.3 20
3.8 The Nonsense Sea 21

4 THE COST OF APPROXIMATIONS **22**

5 CONCLUSION **22**

Abstract

A falsification of the **Evolutionary Theory (ET)** with **Genetic Programming (GP)** over **Flipp** is already available (Rodríguez, [13] 2021). Since Flipp is an extremely simple programming language, one naturally asks: What could happen when one uses a more powerful language? Our answer is **MicroLisp**, a family of programming languages that extends Flipp and that admit loops. These eventually produce shorter codes but add syntactic rules. The first factor accelerates GP, but the second one overloads it. The net result is, according to our abstractions and simulations, negative: GP on MicroLisp runs slower than on Flipp. By the same token, the **Evolution of Perfection** is also mandatory. On the other hand, loops include regulatory signals so, these are useful to model gene regulation and therefore, their study illuminates the question: Do genes accelerate Evolution? Since genes are regulated entities, our answer is negative and so, the falsification of the ET gets stronger.

1 INTRODUCTION

When one makes eye contact with a bat that hangs from a beam, and one observes its beautiful tiny hair together with its tender face, one feels that it is near to our heart. The materialist resolution of this feeling is the **Evolutionary Theory (ET)**: the bat and the man are brothers because they have a common ancestor. Our formulation of that theory is as follows:

We have in the first place that the instructions of the genome to continue life are verbal receipts, i.e., instructions that are mediated by a code, the so-called genetic code, that can be redefined in millions of different forms. In modern terminology: **the Genome is software**. By the same token, **Evolution is a software developer** because it can produce the instructions of the genome of the offspring from those of parents and usually with some modifications. **The Verbal ET** now reads: Evolution is the sole software developer that is responsible for the origin of bat and man.

The Verbal ET is obviously false. This is human terminology to say that given the experience of any human being above 1 year, it is immediate to reject the verbal ET. In effect, our life spins around inventing, using, testing, teaching receipts. Say, a baby must invent many procedures to express him or herself. In formal language, **a receipt is an algorithm**. When an algorithm is transcribed to a programming language, we have **software**. So, algorithms are for human beings more primitive than software.

The point is that we know how difficult life is: composing well-suited algorithms is an extremely complex task. This means that the route to success is paved in failures that are followed by more failures. This is so crude that most products of whatever nature are somehow incomplete or dysfunctional. Software companies exemplify this quite perfectly and that is why they update their programs every day.

To convert the subjective experience of complexity into a scientific prediction that involves Evolution and Genetics, we must digest our daily suffering:

When something does not function, we make changes and incorporate foreign knowledge. In Genetics, changes are called mutations and the incorporation of know-how is called recombination. These two operations form **Evolution**. We make mutation and recombination at every moment, so Evolution is immanent to our lives as it is the **Evolution of Perfection**: mediocre solutions are evolved to ascend to good quality to next get engaged in a battle for excellence. This is universal observation

and so, it amounts to a prediction: **The Evolution of Perfection is mandatory** in every Evolutionary Process -apart from simple cases that nobody cares. Thus, to test the Evolutionary Theory, that we appeared by Evolution, we look for evidences of the Evolution of Perfection. To that aim, we look at ourselves: are we filled in incomplete and dysfunctional features? Although we have defects, say, we are mortal, our definitive answer to this question is simple: No, we are not filled in incomplete and dysfunctional organs, instead, we are perfect. If Evolution were the correct explanation of my existence, terrible defects would abound everywhere in my body, form, and function. This is false. Such an assertion amounts to a rejection of the ET.

Now, **our responsibility has been to generously show how to pass from intuitive claims to Science**, i.e., how to make natural, objective, and clear-cut tests. Now, since the ET deals with software, the natural frame to test it is **Genetic Programming (GP)**, the use of Evolution to develop software. We implemented this program over Flipp, a programming language for a computer with one memory cell. This game for children was enough to falsify the ET: our simulations show that Evolution indeed functions, but this happens at so a low pace that a prediction arises: as our bodies as the fossil record must be filled in every type of imperfections. Or, the Evolution of Perfection is unavoidable. Since this is clearly false, the ET has been falsified (Rodríguez, [13] l.c., 2021, this work will be referred to in the sequel as **GP1**).

The reason for the present article is the following:

We found in our study that Evolution can exactly match small binary strings but that it is unable to match long ones, whose approximations navigate around mediocrity. From this, we speculated that small chunks will acquire a function and evolve into perfection. In Molecular Biology, small chunks with functions are called genes. Therefore, our speculation reads: Given that Evolution can not match long strings, its capability of matching short ones is naturally translated into the arising and proliferation of genes.

We consider at this moment that this speculation cannot be effectively falsified thus, we accept it. So, let us suppose that genes exist: Do genes accelerate Evolution? With this, we mean that the introduction of genes lowers the cost of Perfection from being exponential in the length of the input to, say, polynomial or maybe, logarithmic.

Before studying the aforementioned question, we must make a precision: the genes we know are not chunks of DNA. A gene is a string that encodes for a polypeptide and whose synthesis can be regulated according to the needs at the moment. Thus, we must fabricate an evolutionary environment in which regulation is available. We present a solution to this demand in the following form: We extend Flipp into a family of programming languages that include regulative instructions to study further the battle in GP of Evolution against complexity. More explicitly: We will fabricate MicroLisp, a family of programming languages, some of which include loops. These say how many times a certain task must be executed. Thus, loops include regulative signals that eventually allow for shorter codes. This shall accelerate Evolution. But on the other hand, loops overload Evolution with syntactic rules. This diminishes the velocity of Evolution. What could be the net result? To produce an authoritative answer, let us here make some simulations. So, let us begin with the construction of MicroLisp.

2 NOTEBOOK COMPUTERS AND MICROLISP

Sheets of paper and pencil are used everywhere by human beings to make calculations. Suppose for instance that we need to make an accounting of the movement of our store. In reference to a notebook, a pseudo-code for that task looks like this:

1. Assign the first page to apples, the second to oranges, and the third to bananas.
2. All pages are initialized with zero.
3. An apple arrived: go to the first page and add 1.
4. An orange arrived: go to the second page and add 1.
5. Take out an apple: go to the first page and subtract 1.
6. A banana arrived: go to the third page and add 1.
7. The store was closed at 6 pm. An inform of the activity in the day is in order. So, print the number of apples, oranges, and bananas that are in stock.
8. End.
9. Output: 0 1 1

We have here elementary operations that are readily abstracted by simple commands: go one page forward, go one page backward, add one to the current page, subtract one from the current page, print the content of the current page. This observation allows us to devise a computer together with its operational system.

Our computer in mind is conceptually too simple. This is wonderful because DNA is also conceptually simple and not only it but the operation of the whole biochemical and genetic machinery. Complexity arises from the very large number of types of components that always are at the stage.

2.1 A MicroLisp 0.1 Computer

Let us introduce a **Notebook Computer** and its **MicroLisp** programming language. The computer has a memory array with 5 cells in a row and a pointer to the one and only active cell. A cell must be selected, active, at every instant. To navigate circularly across the array, use the command $<$ to select a cell to the left, $>$ to select a cell to the right.

Memory cells keep numbers from a given set. We will work with two numeric alphabets. First: binary numbers 0, 1. Second: digits 0, 1, 2,... .9. These sets are called **cell alphabets** and are the same for all cells. The memory array is circular, as it is the alphabet for the memory cells. At the start, all cells of the Notebook Computer keep a zero.

Challenge. Make a computer as specified using an array of wheels, which have a symbol in each tooth. An active wheel turns around step by step according to the MicroLisp commands. The pointer is at the top and activates the cell that is also on top of the wheel.

MicroLisp 0.1 has 5 commands (we imagine here that the memory array is immobile, like in solid-state computers, while the pointer moves across the array):

+ Add one to the current cell.

− Subtract one from the current cell.

> Move the pointer one cell to the right.

< Move the pointer one cell to the left.

. Reads the value of the active cell and prints it.

2.2 Antecedents of MicroLisp

The name **MicroLisp** is a concatenation of Micro and Lisp. Micro means that our computer is elementary. The programming language Lisp was invented by John McCarthy, circa 1955 and derives from "LIST Processor" (Two-Bit History [15] 2018). That name is used here because every MicroLisp program is a list of programming commands. On the other hand, Lisp is a powerful high-level programming language whose influence pervades the whole world and its name is a challenge for us: can we make of MicroLisp something as powerful and delightful as Lisp?

MicroLisp 0.1 is a simplistic abstraction of the accounting procedure that everyone makes on a notebook. Its simplicity is explained because it is all we need to do GP for matching binary passwords. To understand the importance of our abstraction, we need a context. The appropriate one is that of Computing Science and its various abstractions of computers, among which the following have been highly influential for our work:

1. Turing Machines are almighty computers. They abstract the procedure: under this situation, do this. So, they have branching and decision-making as an inbuilt property. They were invented by Alan Turing in 1936 and are considered as the starting event of Computing Science (BitMerge [5] 2016, Davey [8] 2010).

2. P" was an academic, mystical variation of Turing Machines made by Corrado Böhm in 1964. It converted computation in a game for kids (McCaffrey [11] 2018). Our work follows him just as many others, two of which follow.

3. Intercal was created just for fun by Don Woods and James M. Lyon in 1972, a language that was mysterious but influencing (Bell [4] 2021).

4. Brainfuck is a variation of P", was created in 1993 by Urban Müller. It puts the wisdom of Böhm in simple terms, without his mysticism. The result is that everyone loves it. Precisely, the important work for us was its implementation in pure JS + HTML + CSS due to Michael Duerinckx ([9] 2014). The idea of using this language for GP came to us from Kory Becker and Justin Gottschlich ([3] 2017). That language has given rise to many dialects, and our MicroLisp is just a creative deconstruction that pursues common sense and GP. Nevertheless, we consider that the original abstraction made by Böhm lends itself to direct comparison with mathematical works, which are abundant.

2.3 MicroLisp 0.1 over binary strings

Let us list some MicroLisp 0.1 programs and their output

PROGRAM	OUTPUT
.	0
+.	1
>+.	1
+.−.	10
+.>.	10
+..−.	110
+.>+.>.	110
+...−..	11100
+.>+.>+.>.>.	11100
+.−.+.−.+.−.	101010
>+.><−.>>>+.><−.>+.><−.	101010

By looking at this table, we notice three facts that will influence our work:

1. There are many equifunctional programs, that produce the same output over the same input. For instance, void loops $+-$ or $<>$ can be inserted in whatever site without causing a modification of behavior.

2. There is no upper bound on the length of programs whose output matches a given target string. It is enough to insert void loops to produce ever longer programs.

3. For every binary string, there exists a MicroLisp 0.1 program that outputs it. More to the point, programs with around the double length of the target binary string can readily be found.

2.4 MicroLisp 0.1 over digits

The following are examples and exercises of separated, independent programs when the cell alphabet is the set of digits:

- . reads the content of cell c0 and prints it. Output 0.
- +. adds one to the cell c0 and prints its content. Output 1.
- ++. adds 1 + 1 to cell c0 and prints its content. Output 2.
- +++. Output 3.
- ++++. Output 4.
- 5?
- +.. Output 11.
- 111?
- −.>−−.−. Output 987.
- We lack the blank symbol. So, we can output 11 also as 1 from cell c0 followed by a 1 from cell c1: +.>+.

We see that there are many forms as a sequence can be chased by MicroLisp 0.1. Moreover, every string formed with the symbols of our alphabet of commands is a viable computer program. But some programs, like $++++$, produce no output.

1 ***Exercise.*** *Practice MicroLisp 0.1 with program* `MicroLispComputer01.html`. *Take a test over binary strings and then over decimal ones. Accompany yourself with children and teenagers. They will find it cool. Our material is the most comprehensible mechanistic introduction to computers and their theory.*

2 ***Exercise.*** *Our Notebook Computer implements an interpreter, i.e., a system that executes each command as soon as it is typed. Implement a compiler, a system that only processes complete programs. Our answer will be found below.*

Challenge. Do cells of living beings have a DNA interpreter or a compiler?

Note for purists. We can work with decimal or binary sequences so, we got a problem because the program

$+>+++.>.-.<.$

outputs 3093 for decimal sequences, but 1011 for binary. Thus, in the strict sense, we have two different languages. If you prefer to be strict, use R and L for decimal sequences, but $<$ and $>$ for binary. With this convention, we get:

The program $+R+++.R.-.L.$ outputs 3093, while the program $+>+++.>.-.<.$ outputs 1011. This may look nice, but below we will possibly discover that this directive overburdens our head and that context-dependent sense is more suitable for human beings.

Let us establish now the relation of Flipp with MicroLisp 0.1.

2.5 MicroLisp 0.0

We have promised to extend Flipp.

3 ***Exercise.*** *Prove that MicroLisp 0.1 extends Flipp (Rodríguez, [13] l.c., 2021). Hint: work with a subset of MicroLisp 0.1 and use just one memory cell. The resulting language is called* ***MicroLisp 0.0****. There are at least two versions, and they are quite similar. Answer*

The following are some examples of programs in **MicroLisp 0.0** and their outputs:

PROGRAM	OUTPUT
.	0
.....	00000
+.	1
+++.	1
+.+.	10
+.+++.	10
+..+.	110
+..+++.	110
+...+..	11100

‘

This table illustrates how the program $++$ makes nothing and how it creates variability.

2.6 GP (Genetic Programming) with MicroLisp 0.0

Flipp and MicroLisp 0.0 are very similar. Are they equally fast?

Our task is to decide whether GP over MicroLisp 0.0 is faster than over Flipp to chase random binary strings. Our solution goes as follows:

As MicroLisp 0.0 as Flipp have two symbols that represent commands so, in this regard they are on equal footing. Nevertheless, to simulate Flipp over MicroLisp, one needs in general longer programs. This must augment the cost of GP. But for every string, there is one and only one Flipp program that chases it, while MicroLisp admits many variants. This might diminish the cost of GP. What might be the overall result?

Our plan is to use Evolution to guess a MicroLisp program to reproduce a given target binary string. Our abstraction is not that far from biology: we are using Evolution to guess a sort of DNA that is an algorithm to synthesize proteins whose content is encoded as binary sequences.

Our GP (Genetic Programming) is defined as follows:

- We declare a binary string, say, 100110, as the target string.
- We give Evolution the task of synthesizing a MicroLisp 0.0 program, something like $+.-..+..$, that must output the target string or password.
- Once the target string is matched, a longer output is asked for, say 1110011.
- We use gradual matching, in which passwords are guessed step by step, as in a game for kids. So, we left all-else-nothing matching outside, and therefore we cannot say anything about security passwords such as those used in ATMs.

To implement this plan, we need to compose a compiler for MicroLisp 0.0. Until now, we have an interpreter for MicroLisp 0.1, a program that executes a command as soon as it arrives. But a compiler executes programs, a list of commands. Our choice was to adapt the program `GPwithFlipp.html` of GP1 to MicroLisp 0.0. Pursuing easiness, we proceeded as follows: We designed a JS function that automatically produces the shortest MicroLisp 0.0 program that chases a given string. For instance, the binary string 10010 is matched by the output of the program $+.+..+.+.$ that has length 9. Similarly, the string 11111 is chased by $+.....$ that hast length 6. So, the length of the program that was developed automatically was measured and it was given to GP the task of finding that program among all programs of precisely that length. This was done by program `GPwithMicroLisp00.html`. This program reported that the cost for GP was $51.8e^{0.34x}$, for randomness $0.6e^{1.04x}$, for the worse case searching $1.3e^{1x}$. All values were higher than for Flipp, which for GP was $49.7e^{0.3x}$ according to GP1. The reason is that strings with a mix of zeroes and ones need longer programs and so, they are harder to find, be it by randomness or by Evolution. Nevertheless, GP is by far the best in comparison with randomness and the worst-case searching.

The previous solution is not very honest because it includes a hint, the length of the solution. What could happen if one gives no hint? When we tried to attack this question, we found various problems:

1. If the binary string has length n and has zeroes and ones, no program of length n is good to match it. For instance, the string 1010 of length 4 can be output by the program $+.+.+.+.$ of length 8 and this is the shortest one. But on the other hand, the string 1111, of the same length, is chased by program $+....$ that has length 5. So, when Evolution does not know the length of the solution program, it must have access to different lengths.

2. Many MicroLisp 0.0 programs can match the same binary string but they might have different lengths. For instance, insertion of $++$ produces no change of the output but changes the length of its program.

3. Programs like $+++++$ render no output.

Our solution was to change the implementation of Evolution. Instead of working with chromosomes of the same length, we allowed length to vary within x and $2.5x$, for a binary string of length x. Recombination was redefined to accept recombinants of different lengths, and mutation was implemented with three options: punctual, change of a symbol by another, and insertion or deletion of a symbol at a given site. We also redefined the score or fitness of a given program: besides a reward for matching a site, we added a bonus for length approximation. The reason is that a program can output a string 15 chars long to match a string of length 8. The unnecessary tail deserves a chastise.

Officially, the fitness of a program was defined as the sum of the number of coincidences plus a bonus for length approximation. Example:

```
//Fitness of a MicroLisp program in  gradualism is calculated
//by comparison of its output with the target password:
//fitness =  coincidences + bonus for length matching.
//Example:
//password = 1010
//output =   1111000
//coincidences = 2 (1010 and 1111 have two chars in common).
//Chastise for length difference = 7-4 = 3.
//Maximal bonus for length matching = password length = 4
//Actual bonus = Maximal bonus - chastise = 4 - 3 = 1
//Fitness = coincidences + actual bonus = 2 + 1 = 3.
```

Unless otherwise stated, we will use this fitness in the sequel.

All this was incorporated into the program `GPwithMicroLisp00v2.html`. The net result is that GP over MicroLisp 0.0 v2 (without hints) runs in the exponential regime: a run in the interval [1, 18) rendered a cost for GP of $25.7e^{0.67x}$, for randomness of $2.7e^{0.78x}$ and for the worse case searching $1.2e^{0.99x}$. These values are by far higher than for Flipp. Nevertheless, let us recall that the costs for MicroLisp 0.0, that was hinted, were: for GP $51.8e^{0.34x}$, for randomness $0.6e^{1.04x}$, for the worse case searching $1.3e^{1x}$. This shows that the length hint is very effective but is unable of dethroning the exponential regime.

Mathematical note. The cost of GP over Flipp was $49.7e^{0.3x}$ according to GP1. This contrasts with the cost found here for GP over MicroLisp 0.0 without hints, which was $25.7e^{0.67x}$. To compare extrapolations, we pay attention to the exponents. In the first case, it was $0.3x$ and in the second $0.67x$ that we approximate to $0.6x$. The difference is not that large. Nevertheless, this small difference causes huge disproportions in the cost, the full exponential expression. Let us notice that $(e^{0.3x})^2 = e^{0.6x}$. This means that for $x = 47$, the first cost in individuals is $e^{0.3\times 47} = 10000000$ while the second must be its square, 100000000000000. That is why we were allowed to swiftly run simulations over Flipp, but here one must be armed with a lot of patience. And even more for the next versions of GP.

In general, most differences found by us lie in the quadratic realm and rarely in the cubic one. There is, nevertheless, one important exception: Darwinism at its face value is so a powerful hint that the cost of a Genetic Algorithm that matches binary strings is logarithmic instead of exponential. Its cost is even cheaper than a polinomic one. That is pure magic: it is very interesting to clearly understand where it comes from (Rodríguez, [14], 2021).

4 ***Exercise.*** *Compare cost $e^{0.3x}$ with $e^{0.31x}$.* Answer

5 ***Exercise.*** *We always claim that the cost of GP is exponential. Resolve the objection that the natural fitting of every finite set of data is a polynomial and not an exponential function.* Answer

Challenge. We are used to imagining that we can discern by inspection when a difference has statistical significance. Nevertheless, we have no solid foundation to claim that differences found in this work have any predictive value. Would you resolve this deficiency? Never forget that what is important are the extrapolations of values from ranges less than 100 to the realm of billions.

Let us pass now to assess the cost of GP with the complete set of MicroLisp commands.

2.7 GP with MicroLisp 0.1

Our ordinary behavior with notebooks to make accounting procedures has been abstracted into MicroLisp 0.1: navigation across a memory array, navigation across the cell alphabet, and printing the content of a cell. We have the option of working with digits besides binary numbers. Nevertheless, we prefer here binary sequences because they allow us to see regression lines faster. These lines are the tool that possibly enables us to extrapolate results from our very humble simulations to the unreachable world of large numbers, the usual one in Molecular Biology.

6 ***Exercise.*** *Decide whether GP over MicroLisp 0.1 is faster than over Flipp to chase random binary strings. The question is interesting because on one hand, MicroLisp allows for more options to build the same string so, it might be faster. But on the other hand, the alphabet of MicroLisp is longer so, it might be slower. Decide what is the net result.* Answer

We have the pretension that the costs reported by our regression lines predict the response to long inputs as those required by Molecular Biology. The problem is that things are not that simple. In that regard, the next exercise expands our vision.

7 ***Exercise.*** *Program* `FlippDynamics.html` *of GP1 shows how GP tries to find Flipp programs that chase binary strings associated with chess boards of predefined size. Adapt that program to MicroLisp 0.1 and explore its execution for increasing size until boards of side with 256 cells. Describe what happens, propose an explanation, and test it. Answer*

Challenge. Program `MLisp01GPDynamics.html` presented Evolution as a stagnating force that frustrates the Evolutionary Process. Can this negative result be extrapolated to long lengths in order to proclaim that Evolution is unable to reach perfection under high complexity? Our option is the following: that program shows that Evolution is stagnating, but that it has fluctuations. We consider that these never die because they come from randomness and that thanks to them, Evolution can produce perfection if only it has enough resources, say, infinite time. Thus, we accept the extrapolations given by the exponential functions that we have found. Test our belief: If fluctuations are the cue to perfection, then they can be boosted by augmenting the sampling effects. To do this, diminish the population size to 2, and test whether the Evolutionary Process to perfection is faster.

In general, the following is our credo: Evolution has two components, one that battles for stagnation and the other for liberty. The first is deterministic, the second random. We imagine that the creativeness of randomness is not silenced by the deterministic effects. Now, the fact that Evolution is better than randomness for the synthesis of software says to us that successions of tiny mutations with gradualic effects play a role that is small but observable and that creates a deterministic boost to Evolution. So, GP puts determinism to work on top of randomness.

8 ***Exercise.*** *Determine the cost of GP over the version of MicroLisp whose commands are:* $< + - >$ *.... that makes the probability of printing equal to 0.5. This alphabet was shown above to be good for battling complexity. Answer*

3 GP WITH REGULATION

Every mutation in a program on MicroLisp 0.1 is viable. The reason is that MicroLisp 0.1 was designed to be too simple. But contrary to this, most mutations are deleterious or lethal in Biology. So, we have a question: what single extension must be considered to furnish MicroLisp with more possibilities and also with more brittleness that some mutations might cause death? Our answer: to include regulation is the single extension to MicroLisp 0.1 that causes maximal power and brittleness at the same time. Our election makes justice to Biology because regulation is omnipresent in Life, even at the level of bacteria. Its discovery was worth a Nobel Prize to Jacob, Lwoff, and Monod ([12] 1965), who showed that genes can be turned on and off by changing the food given to bacteria.

Definition. For every programming language, DNA included, a regulative instruction is one that refers to other instructions or to a property of design.

DNA is a high level language that admits regulation (Bodine [6] 2014 , Virtual Genetics Education Centre [16] 2015).

In the DNA language, regulative effects multiply the average velocity of expression by a real number that can range from 0 to 10 or more (multiply by 0 means turn off, multiply by 1 means turn on, multiply by 0.5 is an inhibition, multiply by 2 is an enhancement). Gene regulation ensures that the appropriate genes are expressed at the proper times and places. Gene regulation can also help an organism to optimally respond to its environment.

Regulation is a nightmare in engineering. A very practical way of learning this is to try to fix in a bike a problem with its shifter, derailleurs, chain, sprockets, or of their position in relation to the seat tube. Or, think of the tension of the brake or shifter cables. Too much tension and the bike will suffer deleterious or lethal damage, but too low a tension and the function will disappear with possibly dangerous consequences. All this is so difficult that with pleasure, I pay a technician. But I find very instructive to make at least one try until success.

Let us look at an example of biology.

Consider plants with leaves. In the underside over their epidermis, leaves have stomata, which are pores that open to the external milieu. But these pores are not static. Every stoma has a variable geometry, and thanks to that it can be open or closed in diverse degrees. By using this freedom, leaves regulate the velocity of interchange of matter with the exterior milieu. Stomata are the doors for absorbing carbon dioxide and for expelling oxygen and water that was absorbed by the roots. Water is expelled in the form of vapor.

Stomata give rise to many considerations with specific engineering questions or functional architecture:

- What about no stomata at all?
- Is the interchange passive or active?
- Is the transport a global, indivisible problem, or is it substance-dependent?
- Do the pores communicate with the intercellular, nude milieu or with a membrane of specialized cells?
- Since the metabolism of every plant is directed to absorb carbon oxide and to expel oxygen, will the transport of these two components be ligated? In an affirmative case, the design will be overloaded, but the energy efficiency will increase.
- What about size and form that guarantee maximal openness and total closure?
- What about the density of the stomata? Notice that to some degree, size and density are independent because stomata might be tiny and separated.

- What about the material to do cells with variable geometry?
- What about the thickness of walls?
- How many walls must cover a stoma cell? Would you divide the functions of a wall into one or two main roles and assign each role to a different wall? For instance, you might work with one wall and put everything in there. Or you might consider that protection and the rest of the functions are two goals so immiscible that they deserve two separate walls. Or, would you crack trying to explain why plant cells have three walls plus another one to separate cells? (Bailey, [2] 2019).
- What about the stability of your design under diverse stresses, say, too low carbon dioxide pressure or prolonged drought? Let us notice that we have plants that live inside water, like algae, and others that can absorb water from the clouds, as some in the high Andes of Peru.

In our opinion, most of these properties are gradually evolvable, at least within certain ranges. But if one remembers of the bike, ranges are expected to be very narrow. For instance, a variable geometry depends on the elasticity of the layers of the walls plus meta-stability, the property of adopting a completely different form due to a small change. Too low elasticity and the response could be absent or too late, but too much, and you would need fine-tuning of the sensors of surface tension of the cellular plasma. Thus, our discussion predicts multifarious gradual Evolution that is bordered from every side by a loss of function, i.e., by death.

Another example from Biology: gene regulation of the mRNA concentration.

A chromosome is like a book with many chapters. A gene is like a chapter. You are not allowed to take home any volume. Instead, you are allowed to make a photocopy of a chapter. This is the mRNA transcript with the m of messenger. With your photocopy, go to a ribosome and ask for a translation of your RNA into a polypeptide chain. If your photocopy does not disappear, ribosomes would continue to reuse it for doing more polypeptide chains. Until when? Until one of the used amino acids is exhausted. At that moment, the whole machinery of life halts because almost every amino acid is essential. Thus, if a single molecule of mRNA is not destroyed, as a minimum a very deleterious mutation is created. On the other extreme, your mRNA can disappear so rapidly that it never arrives at a ribosome, or the ribosome could be too slow to get ready.

The problem is thus to regulate the mRNA decay to provide the highest stability and efficiency. The solution adopted in Nature is to enjoy the long life of mRNA but to cause an induced degradation with a set of enzymes: each molecule of mRNA is marked with a signal that conveys the message of how rapidly must it be degraded. The velocity of degradation must be finely tuned with the needs of the organism. Thus, an enzyme reads and executes the message by preparing the mRNA for rapid degradation by exonucleases or by exosomes (Hoof and Parker [10] 2002).

Challenge. Discuss a case of gene regulation that is historically very important: the Finches of Darwin (Arhat [1] 2009, Zandonella [17] 2015). Do you consider that this case of verified Evolution converts the ET in a scientific truth?

Let us return now back to MicroLisp. We have the occasion of use regulation to solve a frustration with MicroLisp 0.1: we can not separate numbers with a space. For that reason, we have no idea of the meaning of the string 1158. Is it 1 1 5 8? Or rather 1 15 8? Or just 1158? Let us implement a solution to this problem to measure the cost of making PG in the ensuing setting.

3.1 MicroLisp 0.2

To print separate numbers, we need to include blank as a symbol of the alphabet of the language. We are dealing with a regulative problem because we are asking to print blank after each number. Our solution is to concatenate blank to every number to next print the resultant string. The easiest way of making blank part of the language is to declare the cell alphabet as the digits joined to the symbol blank. The new variant is called MicroLisp 0.2. So, the new cell alphabet is 0123456789*b*, with a blank in the last site that has been replaced by *b*. A memory cell can keep one and only one symbol of this list. In this new alphabet, the sum and subtraction are not defined everywhere. Therefore, navigation along the cell alphabet must be adapted a bit. Let us see how:

MicroLisp 0.2 has 5 commands:

\+ move one place forward in the cell alphabet.

\- move one place backwards in the cell alphabet.

$>$ Move the pointer of activation one cell forward.

$<$ Move the pointer one cell backwards.

. Print the value of the active cell.

MicroLisp 0.1 and 0.2 are related as follows: To add one in the old version is an operation that coincides in the new one with moving one place forward in the alphabet for numbers 01...8. But the two commands differ in regard with add one to 9. In the old version, 9 + 1 results in 0, but now, it returns blank. Therefore, we have two dialects of MicroLisp. In general, for every decision we take, there are many possible variants that might create different dialects. So, we are dealing with a great family of computing programming languages, but we will study only a few exponents. The form to compare their behavior over binary strings are analogous.

9 ***Exercise.*** *Practice MicroLisp 0.2 with program* `MicroLispComputer02.html`.

3.2 GP with MicroLisp 0.2

Let us assess now the cost of introducing regulation into MicroLisp.

10 ***Exercise.*** *Compare the cost of guessing a random binary password with that of guessing the same string to which b (of blank) has been attached at the end. This exercise teaches us how expensive it is to introduce the simplest item of regulation in GP.* *Answer*

3.3 MicroLisp 0.3

Our MicroLisp languages 0.0, 0.1, and 0.2 are mutation tolerant: all mutations are viable and can be understood unambiguously by the corresponding compilers. This is unlike Biology, in which almost all mutations are either deleterious else lethal. This explains the following summoning:

Concourse. Find the simplest extension to MicroLisp that enhances its expressiveness, but also that creates enough syntactic troubles that lethal mutations will be generated in a great proportion.

Our answer to this challenge is to introduce loops: they might produce shorter codes so, it is not excluded, that the average cost of GP will diminish. But, as demanded, the syntax will be overloaded enough to naturally allow the simulation of lethal mutations. In fact, the language will contain a larger set of commands and this causes the cost to explode because more combinations are needed to test. And, on the other hand, loops imply correlations among diverse symbols that can be widely separated. For instance, what shall we do with a symbol that commands to close a loop when none has been open? Shall we ignore it? Will we assume that there is a loop opening symbol 3 chars backward? Must we chastise the string with lethality? So, the syntax of loops produces a great rate of programs that cannot be interpreted in a single way. Our decision is to use them as opportunities to introduce lethal mutations. This is the purpose of the loops in MicroLisp 0.3.

3.4 Loops

Repeated calls on the same portion of code are common everywhere in programming.

Example. The following pseudocode commands to do the same task 5 times

```
task
task
task
task
task
```

This code can be abstracted to the following one:

```
Repeat the task 5 times
{ task }
```

The first instruction is regulative because it refers to the second.

How is this loop or repeater executed? When the task is done for the first time, the 5 is changed to 4, then to 3, 2, 1, 0, and then the loop is exited. By the same token, the corresponding code in MicroLisp looks as follows:

```
5 (Loop counter)
[task]
```

and is executed in the same way. Formally:

```
MicroLisp 0.3 has  7 commands,
two refer to the navigation across the cell alphabet,
two to the navigation along with the array of memory cells,
two are loop controls,
one is the printing command:
+ advance one site in the alphabet in the current cell.
- move back one site in the alphabet in the current cell.
> Move the pointer one cell forward in the array.
< Move the pointer one cell backward in the array.
[ start of loop.
] end of loop.
. Print  the value of the active cell.
```

Example. The program

```
+++
[>+.<-]
```

puts loop counter on 3 on cell C0, $>+$. is the task to repeat that affects cell C1. The subroutine $<-$ returns the pointer to cell C1 and decrements the counter. The Loop is exited when the counter equals 0. When the cell alphabet is the digits and with 8 memory cells, the program outputs 123. In more detail: the counter is set to three on cell C0. Next and inside the loop, cell C1 acquires the value 1 that is next printed, and the counter is decremented to 2. The second round decrements the counter to 1, and increments in 1 the value of C1 that acquires the value 2, which is printed. The third round decrements the counter to 0 and increments 1 to the value of C1 that now keeps a 3, which is printed. Since the counter is 0, the loop is exited. Output: 123.

3.5 Fault tolerance

To embark into GP 0.3, that admits loops, we must take into account that we will find ill-concocted programs. Examples:

```
+++[>++++.              Lonely [
+++]>++++.              Lonely ]
+++]>++.[               Inverted pair ][
>[>+++].                A void loop
+++[++.]                An infinite repeater
```

The question is: How much fault-tolerant must be our language? Until now, the only ill concocted programs were those with no output, such as $+>>>+++$, and these were executed without any checking test. So, we have been fault-tolerant and this might be good for Evolution: a single mutation that incorporates a printing command is enough to produce an output. Another form as we have introduced fault tolerance is by avoiding memory overflow. To that aim, we have declared the memory array to be circular. This is also a complexity killing measure because specific symbols can be found in two different ways.

Now, in general and in our opinion, Biology welcomes all possible choices in regard to fault tolerance. To see this, just consider that animals do not discriminate themselves by physical perfection but can do it by chemical recognition and by behavior.

Example. A perfect and corpulent baby was born to a couple of wolves. It had an extreme tendency to be dominant and disobedient. It was so trouble-making that the father killed him. Now, in hindsight, we ask: Was nature fault-tolerant? Our answer is negative. For wolves that need to be disciplined and highly cooperative, that was a very wise decision because a problematic wolf could drive the pack into extinction.

Example. The immense majority of children are born perfect. That allows us to claim that nature hates fault tolerance. But, we see no explicit mechanism to achieve that. Instead, zero fault tolerance arises from fundamental mechanisms that must be very complicated and extremely brittle. In any case, registered variation at the molecular level is an ultramicroscopic fraction of all possible variations. We know that because most mutations at the molecular level are lethal. In fact, the Neutral Theory of Molecular Evolution says that mutations are in their immense majority lethal, but a minor fraction is neutral.

Adding up everything together, we decide our policy:

- MicroLisp 0.3 includes only the pair of square brackets [] as signals for starting and ending loops. A program to be syntactically correct must be paired: the number of open signals must be equal to that of closing ones, and a closing bracket must always pair with an opening one to the left.
- Managing the loop counter is a duty of the developer, in this case of GP. Warning: the freedom to modify the loop counter is also enjoyed by JS and other languages and is a cause of terrible bugs.
- To prevent infinite looping, programs are allowed to run for a predefined limit of rounds, say, 100.
- Syntactic incorrect programs can participate in recombination but are not eligible for cloning to generate the next generation.

Notice that our rules guarantee the expected execution of syntactically correct programs, but do not interdict all illegal flows. This policy allows finding nice surprises as the program $+[+.]$ that sounds like an infinite loop because it does not include any decrementing command. Nevertheless, it halts over digits because 0, the loop terminal signal, is recurrently found along with successive increments. Also, the program $-[-.]<$ outputs 8,7,6,5,4,3,2,1,0. What does the program $-[.-]<$ over digits?

Because we are making a language that is getting complicated and whose results will be used to support important conclusions, we need to make sure that everything is as we say. Thus, a debugger for MicroLisp 0.3 is an urgent need. Here comes one.

3.6 MicroLisp 0.3 Advanced Editor

Program `MicroLispComputer03.html` presents an Integrated Development Environment for MicroLisp 0.3.

The following functions are included:

- An advanced editor that allows tabulation and comments. It converts the complexity into an artistic game because a correct tabulation makes programs ordered and human-readable.
- The cell alphabet can be selected to be binary 01, digits 0123456789, or digits with blank 0123456789b, where the b can be substituted by a space or by a dash.
- Symbols in memory cells must be interpreted as chars. Nevertheless, in the present implementation, we use non-negative integer numbers as intermediaries.
- The programming language is MicroLisp 0.3 that admits loops.
- Copy-paste operations are enabled.
- Programs can be run. Copy-paste the programs of this and the next section directly into the editor panel to easily test what is claimed.
- To help to debug, every step of the Interpreter is published to the Execution Stack that must be read from the bottom up.

This IDE was evolved by the Author from a program by Michael Duerinckx l.c., ([9] 2014). Let us see how programs look in this editor. All programs in this subsection were run with 16 memory cells and Digits as cell alphabet.

Example. The following program puts 3 in cell c0, 4 in cell c1, and adds the two numbers to cell c2. Data are next printed: $3+4=7$. Numbers are printed without spacing: 347 is the output.

```
+++ >> +++            Add 3 to cell c0, and 3 to cell c2
<                     Moves pointer to cell c1
++++>++++             Add 4 to cell c1, and add 4 to cell c2
<<                    Moves pointer to cell c0
.>.>.                 All values are printed without spaces
```

Comments will be ignored by the editor but if you have installed an orthographic corrector in your system, it also will work here (tested over Ubuntu 20.4). Comments cannot have any symbol from the language, else they will be interpreted as part of the programming. So in your comments, periods are forbidden. You can use upper case letters. You can use here commas but, beware, in other similar languages, the comma is part of the language so, make sure that you know the commands of your language.

11 ***Exercise.*** *Develop and run a MicroLisp program to subtract 2 from 7. Input numbers and the final answer must be printed (725).* Answer

12 ***Exercise.*** *Compose a program for Digits that calculates* 2×3*, prints input numbers and result. Desired output 236.* Answer

Challenges. First: Compose a program for a computer with 2 memory cells to add 3 to 5. Input numbers and the final answer must be printed. Expected output = 358. Second: Do the same, but insert separators. Expected output 3 5 8. According to your personal experience, is memory size a necessity, or is it just a human-friendly characteristic?

Warning about loops: The repeating of a task adds modifications to modifications. Hence, it is not expected to output the same result for each round. A loop or repeater is delimited by the pair of rectangular brackets []. The loop counter is the number that is contained in the cell that is active at the moment when the symbol [is read. The modification of the loop counter is not done automatically. Instead, that must be done by the code itself that must be written by the developer.

We calculated 2×3 as the result of adding twice 3. Where is an example of genuine multiplication?

Example. The following macro, or program, uses a loop to find 2×3. Remember that periods are not allowed in comments. Only the result is printed.

```
++                      Cell c0 initializes the loop counter = 2
[                       Beginning of Loop
    > +++               Move  to cell c1 and add 3
    < -                 Back to cell c0 and decrement counter
]                       Ending of loop
>.                      Print result
```

13 ***Exercise.*** *Add to the previous program the possibility to print incoming numbers and the result of their multiplication.* Answer

14 ***Exercise.*** *Devise a program to calculate* 2×4*. Incoming numbers and the result of their multiplication must be printed without spaces (248). Add clear comments. They must not contain any reserved symbol of the language. Pay special attention to the period.* Answer

Example. Let us compose a program that illustrates the possibilities of MicroLisp 0.3 to be human-friendly. The multiplication 2×3 is divided into three tasks: print incoming 2, print incoming 3, multiply 2×3, and print the result. We expect the problem to be trivial if we reuse a previous macro. Reusing extant material is the *conditio sine qua non* of both Software development and Evolution. The following is our proposal in which tasks are allotted to different memory sections:

```
++.                     Add 2 to cell c0 and print result
> +++.                  Add 3 to cell c1 and print result
>                       Move to cell c2

                        Macro to find 2 X 3 is inserted
                        The enumeration of cells is shifted to 0

++                      Cell c0 initializes the counter = 2
[                       Start of repeater to  find 2 x 3
    > +++               Add  3 to cell c1
    < -                 Return to cell c0 and decrement counter
]                       End of repeater
 >.                     Print result
```

Warning. One might think of developing a MicroLisp language and editor that allows integers as cell alphabet. This dream is not realizable: all languages truncates integers at some point, be directly or by the incapability of the system of dealing with extremely large numbers and strings.

15 ***Exercise.*** *Run program* $++++[>.++[>.+<-]+.<-]$ *to test the capacity of MicroLisp 0.3 to handle nested loops. Rewrite this program in a human-readable form and add comments. The program halts and, with digits and 8 memory cells, it outputs 00111234115671. Halting is possible because in our circular cell alphabets, incrementing the counter returns at the end to 0.* Answer

16 ***Exercise.*** *Make your best to develop a program for the multiplication* 4×3. *Input and output values must be printed without spaces. Expected output 4312.* Answer

Intrigue. Is it really impossible to develop a MicroLisp 0.3 program to calculate without tricks 4×3 with answer 12?

17 ***Exercise.*** *Use loops to write a program for MicroLisp 0.3, that outputs 0-1-2-3-4-5-6-7-8-9. Compare its length with the solution without loops presented by the corresponding demo in the IDE.* Answer

3.7 GP with MicroLisp 0.3

Our task is to chase a list of random binary strings with MicroLisp 0.3 programs. It is solved by the program `MicroLispGP03.html`. Our conclusion is that most programs produce no output, especially because they are syntactically ill so, the cost of GP 0.3 is very high as it is shown by the next report:

```
Exponential cost of matching a binary string of length x
with MicroLisp 0.3

x       Evolution                Randomness
1       NaNeˆ(NaNx)              NaNeˆ(NaNx)
2         7.14eˆ(2.64x)             4.41eˆ(1.29x)
3         55.7eˆ(1.1x)              0.59eˆ(2.79x)
4       152.75eˆ(0.49x)             1.84eˆ(2.12x)
5        52.33eˆ(1.03x)             1.63eˆ(2.18x)
6        35.13eˆ(1.2x)              2.05eˆ(2.08x)
7        63.54eˆ(0.98x)             3.04eˆ(1.93x)
8        98.23eˆ(0.83x)             6.28eˆ(1.69x)
9        99.38eˆ(0.83x)             9.14eˆ(1.57x)
10       178.4eˆ(0.67x)            11.17eˆ(1.52x)
11      328.98eˆ(0.52x)            10.07eˆ(1.55x)
12      253.09eˆ(0.58x)            13.19eˆ(1.48x)
13      209.02eˆ(0.62x)            15.72eˆ(1.45x)
14      208.73eˆ(0.62x)            16.16eˆ(1.44x)
15      164.18eˆ(0.66x)            17.98eˆ(1.42x)

Repetition:
15      245.16eˆ(0.59x)             6.12eˆ(1.51x)
```

In regard to the first run, a desk computer needed half an hour to reach 12, but 7 additional hours to make 13 and 14, and 15 more hours to compute 15.

Assessment. Regulation is all to Biology. Therefore, if regulation is introduced into a living kingdom by whatever cause, it must remain there forever. Borrowing the jargon of Evolutionary Game Theory (Cowden [7] 2012), we might say: regulation is an ESS or Evolutionary Stable Strategy in the game of Evolution vs complexity. That is why we have expanded our MicroLisp to make space for regulation, in our case for loops. The problem is that MicroLisp 0.3 shows us that regulation is extremely expensive for Evolution.

18 ***Exercise.*** *Depict a random walk in the space of MicroLisp 0.3 programs. Analyze the results.* *Answer*

3.8 The Nonsense Sea

To reconstruct the geography of the space of programs according to the results of the last exercise, let us make the following convention. Programs without output correspond with water, and programs with output correspond with the land. The emerging landscape is then that of a great sea that is splashed out by some few tiny islands.

19 ***Exercise.*** *Run the program* `NonsenseSeaEnglish.html` *to verify that in any human language, sentences with sense conform to small islands that are surrounded by an immense sea of strings with nonsense.*

Since programs with no output correspond with nonsense strings so, a simple depiction results: in the space of MicroLisp 0.3 programs, the world consists of small islands with a sense that are surrounded by an immense sea of nonsense.

20 ***Exercise.*** *Depict the tapestry of perfection for GP over MicroLisp 0.3. Do you consider that the emerging picture is compatible with the idea of the nonsense sea?* *Answer*

4 THE COST OF APPROXIMATIONS

We know that the cost of exact matching in GP 0.3 is exponential. What about the cost of approximated matching?

Challenge. Study the cost of approximations in GP with MicroLisp 0.3 and discuss the ensuing consequences. Hint: Follow the corresponding procedure that appears in GP1.

5 CONCLUSION

When the work is intelligent, conclusions are clear cut.

21 ***Exercise.*** *Propose a conclusion in regard to the question we face: what is the cost of GP with regulation?* *Answer*

22 ***Exercise.*** *What is in your opinion the most important conclusion of the present work?* *Answer*

Answers

Exercise 3, page 7. Flipp consists of a computer with one memory cell and a language to program it. The memory cells keep a bit, a zero else a one. The language has two commands. The first is the Point command . that prints the content of the memory cell. The second command is the Exclamation Mark ! that flips the memory value and prints the result: 0 is flipped into 1 and vice versa. MicroLisp 0.1 extends Flipp because one can restrict the first to just one memory cell and because one can express the Flipp commands in MicroLisp 0.1. In fact, the print operators are just the same over the first memory cell, and the Exclamation Mark Command over bits can be simulated in MicroLisp 0.1. as $+.$ or as $-.$ because the arithmetic operations are cyclic and produce the very same results as flipping over a bit. Thus, we define **MicroLisp 0.0** as the language that operates over a Notebook Computer and whose set of commands is composed of the two MicroLisp 0.1 commands $+.$ that respectively adds one and prints.

Exercise 4, page 10. If we divide the two costs, we get:
$r = \frac{e^{0.31x}}{e^{0.3x}} = e^{0.01x}$.
For $x = 1400$, $r = e^{14} = 1000000$. And the disproportion grows exponentially with x.

Exercise 5, page 10. We study the tendency of the exponential fitting over increasing ranges of the input length. We see that after long fluctuations at the start, the exponent tends to a quasi-stabilization. This is impossible for polynomials that need to recur to ever higher degrees of the dominant term. The extrapolation to long ranges of an exponential fitting is done through the exponent c. It is in the vicinity of 1. Thus, we consider that we have the right to claim that GP over all versions of MicroLisp run in the exponential regime because of the observed universal tendency to quasi-stability of exponents.

Exercise 6, page 10. Program `GPwithMicroLisp01.html` rendered over the interval [1,18) in a single run the following results:

```
Exponential cost of matching a binary string of length x
with MicroLisp 0.1.

x    Evolution          Randomness         Worst case searching
1    NaNe^(NaNx)        NaNe^(NaNx)        NaNe^(NaNx)
2      35e^(0.69x)    10.3e^(0.15x)       0.2e^(2.08x)
3    55.6e^(0.35x)     2.2e^(1.31x)       0.6e^(1.39x)
4      99e^(0x)        2.2e^(1.3x)          1e^(1.11x)
5    73.3e^(0.15x)     2.8e^(1.19x)         1e^(1.11x)
6    32.7e^(0.5x)      2.1e^(1.32x)       0.9e^(1.15x)
7    22.8e^(0.63x)       2e^(1.34x)       0.8e^(1.19x)
8    16.6e^(0.74x)     2.1e^(1.31x)       1.1e^(1.11x)
9    18.3e^(0.71x)     2.7e^(1.24x)       2.1e^(0.9x)
10   16.2e^(0.74x)     1.9e^(1.33x)       1.6e^(0.97x)
11   15.3e^(0.76x)       4e^(1.15x)       1.3e^(1.03x)
12   22.4e^(0.67x)     2.9e^(1.22x)       1.3e^(1.03x)
14   15.2e^(0.75x)     3.2e^(1.2x)          1e^(1.09x)
15   10.4e^(0.82x)     1.9e^(1.3x)        0.8e^(1.13x)
16   13.7e^(0.77x)       2e^(1.29x)       0.9e^(1.1x)
```

Another run:

1	NaNe^(NaNx)	NaNe^(NaNx)	NaNe^(NaNx)
2	23.3e^(1.1x)	5.5e^(1.47x)	1e^(1.39x)
3	48.5e^(0.55x)	26.9e^(0.28x)	1.6e^(1.04x)
4	40.4e^(0.66x)	10.6e^(0.84x)	2.8e^(0.69x)
5	35.2e^(0.73x)	10.9e^(0.82x)	2.5e^(0.76x)
6	66.1e^(0.46x)	12.3e^(0.77x)	3e^(0.67x)
7	86.8e^(0.36x)	17.7e^(0.64x)	2.7e^(0.72x)
8	61.3e^(0.47x)	7.1e^(0.94x)	2.2e^(0.79x)
9	68.2e^(0.44x)	6.7e^(0.96x)	2.3e^(0.77x)
10	92.8e^(0.36x)	8.8e^(0.88x)	1.4e^(0.91x)
11	80.8e^(0.39x)	5.7e^(0.99x)	1.4e^(0.91x)
12	44.7e^(0.53x)	3.9e^(1.08x)	1e^(0.99x)
13	82.8e^(0.4x)	4.1e^(1.07x)	1.2e^(0.95x)
14	44.1e^(0.52x)	2.2e^(1.2x)	0.7e^(1.04x)
15	29.6e^(0.6x)	2.1e^(1.21x)	0.6e^(1.07x)
16	19.2e^(0.67x)	2.5e^(1.18x)	0.6e^(1.09x)
17	21.1e^(0.66x)	2e^(1.21x)	0.7e^(1.06x)

The cost of the worst-case searching is lower than that of randomness because the automatic composition of solution programs uses only two symbols, those of MicroLisp 0.0. We conclude that GP on MicroLisp 0.1 is quadratically more expensive than over Flipp: it is for Flipp proportional to $e^{0.3x}$ and for MicroLisp 0.1 is proportional roughly to $e^{0.6x}$. We see that everything happens in the exponential regime.

In relation to the first run, the form as we perceived the effect of an exponential cost was the following: beginning with passwords of length 1, one arrived at length 10 almost instantaneously, next came 11, 12, and 13 in 29 seconds. But reaching 14 needed 44 minutes, while 15 and 16 needed 10 minutes each one, and solving 17 needed 4 additional hours. Upon consideration of the escalation of time that appears over and over, the Author feels a strong tendency to think that the ET is stupid. That is why I must be on guard. In the first place, Applied Evolution produces fruits every day. But on the other hand, the ET is a scientific theory that is false, an assertion that means that the default verdict for the Evolutionary relation of any pair of species is negative and that only the Scientific Method can positively argue over specific cases.

Exercise 7, page 11. Program `MLisp01GPDynamics.html` shows the dynamics of the Evolutionary Process that allows MicroLisp programs synthesized by Evolution to chase a password representing a chessboard. Found problem: MicroLisp 0.1 is unable to chase binary strings of length 256 because it produces too short outputs. When complexity is large, output binary sequences are too short to specify the full board. There are three solutions. First: in our program for a binary string of length L, we allowed MicroLisp 0.1 programs to have a length in within L and $2.5L$. We can enlarge the allowed range. Second: to modify the MicroLisp alphabet to augment the probabilities of printing. That is, instead of the alphabet $< +- >$. we can various commands to print, say $< +- >$ .. or $< +- >$ ... or $< +- >$ ..... Third: to reward longer programs. Our election was to test the modified alphabet $< +- >$ .... over the same program. It worked fine: outputs of length sufficiently large are now produced, although not necessarily in the first generation. Programs show that the dynamics stag-

nate at a frustrating value. Nevertheless, there are fluctuations due to mere randomness, and thanks to them, it is not excluded that the goal might be achieved.

Exercise 8, page 11. We just modified the MicroLisp alphabet in program `GPwithMicroLisp01.html`. Instead of $< + - > .$, we used $+ - <> ....$, with four printing commands. It rendered over the interval [1,18) the cost of GP as $19.3e^{0.71x}$, for randomness $22.9e^{0.69x}$ and for the worst case searching $2.1e^{0.98x}$. A repetition over the same interval rendered the cost of GP as $7.7e^{0.9x}$, for randomness $4.3e^{0.89x}$ for the worst case searching $0.9e^{1.07x}$. Our conclusion is that the performance of the new alphabet in regard to GP is not better than that of the initial one. But we are very intrigued because randomness appeared to be better than Evolution. This may the result of fluctuations that are always present.

Exercise 10, page 14. We use MicroLisp 0.2 with the subset of commands $+ .$ that works over binary strings. Our program `MicroLispGPBlank02.html` compares the cost of guessing a binary string against that of guessing the same string to which b has been attached at the end. For the first task, the used cell alphabet is 01 while for the second, it is $01b$. We used a memory array of 200 cells that probably would prevent circularity interference. As expected, including blank spaces is more expensive. Verified for the interval [1, 14) : the cost of GP without regulation was found to be $21.8e^{0.66x}$ and with regulation $12.8e^{1.19x}$. A difference of the exponent above 0.3 was observed in all subintervals. A repetition rendered for the interval [1, 18): without regulation $6.7e^{0.89x}$, with regulation $8.7e^{1.31x}$. In short: the cost of introducing regulation is less than quadratic.

Exercise 11, page 18. The following program subtracts 2 from 7:

```
+++++++.                Add 7 to cell c0, print value
>>                      Move pointer to cell c2
+++++++                 Add 7 to cell c2
<                       Move pointer to cell c1
++.                     Add 2 to cell c1, print value
>                       Move to cell c2
--.                     Subtract 2 from cell c2, print value
```

Exercise 12, page 19.
The following program calculates 2×3:

```
++.             Print 2 from cell c0
> +++.          Print 3 from cell c1
> +++ +++ .     Add 2 times 3 to cell c2 and print
```

Exercise 13, page 19. The following program uses a loop to find 2×3. Incoming numbers and result are printed without spaces (236):

```
++.                     Cell c0 initializes the loop counter = 2
[                       Beginning of Loop
    > +++.              Move  to cell c1 and add 3 and print content
    < -                 Back to cell c0 and decrement counter
]                       Ending of loop
```

Exercise 14, page 19. The following program calculates 2×4:

```
++.                     Cell c0 initializes the loop counter = 2
[                       Beginning of Loop
    > ++++.             Move  to cell c1 and add 4 and print content
    < -                 Back to cell c0 and decrement counter
]                       Ending of loop
```

The output is 248

Exercise 15, page 20. In human-readable form plus comments, the program $+++[>.++[>.+<-]+.<-]$ reads:

```
+++                 External loop with counter = 3
[                   Begin mark of external loop
     >.             Move to c1 and prints its content
     ++             Internal loop with counter =  2
        [           Begin mark of internal loop
           >.       Move to cell c2 and print content
           +        Add one to cell c2
           <-       return to c1 and decrement counter
        ]           End mark of internal loop
     +.             Add one to counter in C1 and print
     <-             Return to C0 and decrement counter
]
```

Output 00111234115671 when the cell alphabet is Digits and the memory arrays has 16 cells.

Exercise 16, page 20. Our election is to reuse a previous human-friendly program that shows that this characteristic is also the best for evolvability:

```
 ++++.                    Add 4 to cell c0 and print result
> +++.                   Add 3 to cell c1 and print result
>                        Move to cell c2

                         Macro to find 4 X 3 is inserted
                         The enumeration of cells is shifted to 0

++++                     Cell c0 initializes the counter = 4
[                        Start of repeater to  find 4 x 3
    > +++                Add  3 to cell c1
    < -                  Return to cell c0 and decrement counter
]                        End of repeater
>.                      Print result
```

The output is 342. The expected value was 3412. The failure is explained by the cell alphabet that counts circularly from 0 to 9. Let us notice that various coordinated mutations must be involved to produce a new, useful program. This is how complexity

in programming arises: a program has various functions that work coordinately and that therefore must be mutated coordinately.

Exercise 17, page 20. The program must run with a cell alphabet equal to digits with blank (or dash) and with 16 memory cells. Firstly, we devise an idea, i.e., a program without loops that does the work and that lends itself to looping:

```
.                    Print 0
-.                   Print dash
++.                  Print 1
--.                  Print dash
+++.                 Print 2
---.                 Print dash
++++.                Print 3
----.                Print dash
+++++.               Print 4
-----.               Print dash
++++++.              Print 5
------.              Print dash
+++++++.             Print 6
-------.             Print dash
++++++++.            Print 7
--------.            Print dash
+++++++++.           Print 8
---------.           Print dash
++++++++++.          Print 9
```

This idea demands a loop for each line and if one tries to use a double loop to include all lines, one sees that more than one memory cell would be necessary. But in that case, the following idea is more suited to looping:

```
>-<          Move to cell c1 Reach dash Return
.            Print 0 from C0
>.<          Print dash from C1
+.           Print 1 from C0
>.<          Print dash from C1
+.           Print 2 from C0
>.<          Print dash from C1
+.           Print 3 from C0
>.<          Print dash from C1
+.           Print 4 from C0
>.<          Print dash from C1
+.           Print 5 from C0
>.<          Print dash from C1
+.           Print 6 from C0
>.<          Print dash from C1
+.           Print 7 from C0
>.<          Print dash from C1
+.           Print 8 from C0
>.<          Print dash from C1
+.           Print 9 from C0
```

This idea was next implemented with one loop. Three cells were used. The first, c0, to keep the loop counter, next c1 to print numbers, and c2 to print dash:

```
>>-           Move to cell c2 and reach dash
<<            Return to c0

+++++ ++++    Put loop counter = 9
[             Start of loop
                  Print numbers 123456789
    >         Move to cell c1
    +.        Add one to cell c1 and print
                  Print dash
    >.<       Move to cell C2 Print dash Return to cell c1
    <-        Back to cell c0 and decrement  counter
]             End of c0 loop
.           Print 0
```

We have illustrated that loops might make the code of repetitive tasks to be sensationally shorter: the first idea employs 107 commands, the second 49, and its loop implementation 25. On the other hand, the fact that we have been able to devise complex programs with MicroLisp 0.3 authorizes us to follow with the next step: to study the possibilities of Evolution to develop programs in this language.

Exercise 18, page 21. Program `MLisp03RandomWalk.html` shows a random walk in the space of MicroLisp 0.3 programs. It is dominated by programs without output that are rarely interspersed by programs with output.

Exercise 20, page 22. Program `MLisp03EvoOfPerfection.html` shows us the Tapestry of Perfection, a drawing that shows the Evolution of fitness in GP over MicroLosp 0.3. It shows us that Evolution can leave the sea of nonsense to frequently walk amid islands with a sense. But after some few steps, less than 7, one falls into the Nonsense Sea. And begin anew over and over. All this is so chaotic that the perfect solution rarely comes from a good exponent. Instead, it may come from programs with no output that are allowed to recombine.

Exercise 21, page 22. We have accepted that the cost of GP without regulation is exponential with an exponent lower than 1. We find no reason to think that regulation lowers the cost. The cause is that syntax restrictions overcome the saving of trials that loops possibly offer. Moreover, the fraction of binary strings that lend themselves to loops is low because a random string has ones and zeroes in random order and positions, and thus no pattern is expected to be repeated with the exception of some infrequent events. Thus, the overall conclusion of our work is: GP with or without regulation works in the exponential regime.

Exercise 22, page 22. In our opinion, no single result makes a point. Instead, the Evolution of Perfection, depicted by the Tapestry of Perfection, and the exponential cost together allow to formulate a mandatory prediction: if Evolution created me, my body must be filled in every kind of defects, malformations, and malfunctions. False.

Bibliography

All cited links were active by 28/II/2022

References

[1] ABZHANOV ARHAT 2010
Darwin's Galápagos finches in modern biology Phil. Trans. R. Soc. B3651001–1007
https://royalsocietypublishing.org/doi/10.1098/rstb.2009.0321 13

[2] REGINA BAILEY 2019
The Structure and Function of a Cell Wall
thoughtCo.
https://www.thoughtco.com/cell-wall-373613 13

[3] KORY BECKER AND JUSTIN GOTTSCHLICH 2017
AI Programmer: Autonomously Creating Software Programs Using Genetic Algorithms
https://arxiv.org/pdf/1709.05703.pdf 5

[4] ADAM GORDON BELL 2021
INTERCAL, YAML, And Other Horrible Programming Languages
https://earthly.dev/blog/intercal-yaml-and-other-horrible-programming-lang 5

[5] BITMERGE 2016
How Turing Machines Work
https://www.youtube.com/watch?v=cDc6Gfo3egk 5

[6] DAVID M. BODINE 2014?
Gene Regulation, nhgri.
https://www.genome.gov/genetics-glossary/Gene-Regulation 12

[7] COWDEN C.C. (2012)
Game Theory, Evolutionary Stable Strategies and the Evolution of Biological Interactions, Nature NEWS 29 April 2021
https://www.nature.com/scitable/knowledge/library/game-theory-evolutionary- 21

[8] MIKE DAVEY 2010
A Turing Machine - Overview
https://www.youtube.com/watch?v=E3keLeMwfHY
http://aturingmachine.com 5

[9] MICHAEL DUERINCKX 2014
JavaScript Brainfuck interpreter
https://github.com/michd/brainfuck-interpreter-js 5, 18

[10] AMBRO VAN HOOF, ROY PARKER 2002
Messenger RNA Degradation: Beginning at the End, *Current Biology, Volume 12, Issue 8, 2002, Pages R285-R287, ISSN 0960-9822,*
https://www.sciencedirect.com/science/article/pii/S0960982202008023
13

[11] JAMES D. MCCAFFREY 2018
The P" Programming Language
https://jamesmccaffrey.wordpress.com/2018/09/05/the-p-programming-language,
5

[12] THE NOBEL PRIZE IN PHYSIOLOGY OR MEDICINE 1965.
NobelPrize.org. Nobel Prize Outreach AB 2021. Tue. 29 Jun 2021.
https://www.nobelprize.org/prizes/medicine/1965/summary/
11

[13] JOSÉ RODRÍGUEZ 2021 *Falsification of the Evolutionary Theory with Flipp*.
Text without programs:
http://www.ejristos.com/GPfalsifiesEvo/FalsificETwithFlipp.pdf
(English)
http://www.ejristos.com/GPfalsifiesEvo/FalsifTEconFlipp.pdf
(Spanish)
Texts and programs:
http://www.ejristos.com/GPfalsifiesEvo/FlippETfalsification.zip
2, 3, 7

[14] JOSÉ RODRÍGUEZ 2021 *The Darwinian Evolutionary Theory at its face value*.
Text without programs:
https://ejristos.com/dl/DarwinAtFaceValue.pdf
https://ejristos.com/dl/DarwinEnSerio.pdf (Spanish)
Text and programs:
https://ejristos.com/dl/DarwinAtFaceValue.zip 10

[15] TWO-BIT HISTORY 2018
How Lisp Became God's Own Programming Language
https://twobithistory.org/2018/10/14/lisp.html 5

[16] VIRTUAL GENETICS EDUCATION CENTRE 2015?
Gene Expression and Regulation https://www2.le.ac.uk/projects/vgec/highereducation/t
12

[17] CATHERINE ZANDONELLA 2015
A gene that shaped the evolution of Darwin's finches Princeton News February 2015
https://www.princeton.edu/news/2015/02/11/gene-shaped-evolution-darwins-fi
13