Mihai Nadin

The Timeliness and "Future-ness" of Programs

A computer program (henceforth, program) is a machine. There is no way around this condition of programs. But until we understand what this entails, the statement bears as much knowledge as any other truism.

Why is a program a Machine? And if this is so, what are the consequences for our understanding of the time dimension (timeliness), in particular, "future-ness," of programs? The question has pragmatic implications: Today we are engulfed in programming more than in any other form of human activity. Behind almost all activities—production of goods, machines, processed foods, medicine, art, games and entertainment—programming is involved in a broad range and variety of forms. We invent new materials in computational form before we actually "make" them; we explore new medicines; we design the future (architecture, urban planning, communication, products) using programs; we redefine education, politics, art, and the military as we express our goals through programs. All the invisible computers (embedded in our world) that make up our universe of existence were programmed and keep undergoing reprogramming. Therefore, to address time aspects of programming is to account for the meaning and efficiency of a form of praxis that defines the human being in humankind's current new age[1].

But what does it mean *to program*? Let us take a simple program procedure: the factorial, which is frequently present somewhere in the larger scheme of things, though not of particular importance. It is part of the mathematical description of the world. The factorial of a number n is denoted by n! and is defined in mathematics as

$$n! = [(n-1) \cdot (n-2) \ldots 3.\ 2.\ 1] = n \cdot (n-1)!$$

Even those who refuse to look at a formula ("Mathematics is not for me!") could, if they spare one second, notice that to calculate the factorial of n one would calculate the factorial of (n-1) and multiply the result by n, that is, $n! = (n-1)! \cdot n$. (Obviously, if n=1, the factorial is 1.) This means that in order to calculate the factorial, we multiply 1 by 2, the result by 3, the new result by 4 and so on until we reach n. A counter keeps track of how the numbers increase from 1 to n.

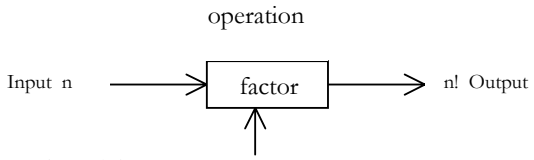
---

[1] Mihai Nadin, *The Civilization of Illiteracy,* Dresden University Press, 1998 and the German translation, *Jenseits der Schriftkultur,* Dresden University Press, 1999. (See also Endnotes.)

How does a computer program handle this? We can, as I did above, define the factorial computationally:

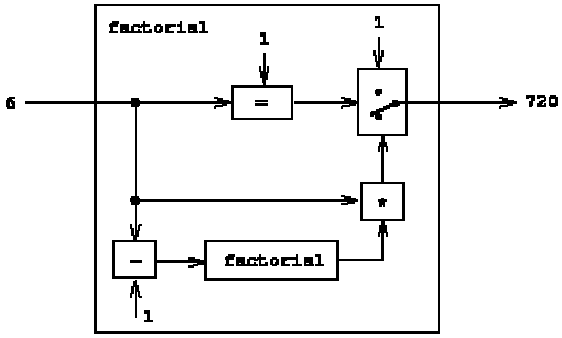
Program lines	What the program lines mean
(define (factorial) n)	
(if (=n1),	which means if n=1,
1	the value is 1
(*n (factorial (-n1)))))	multiply (*) n by the factorial of (n-1).

"Where is the machine here?" some will ask (and not only those with no broad knowledge of computers). As we know from literature or from using machines, a machine takes something, called Input, does something with it, and produces the result as Output:



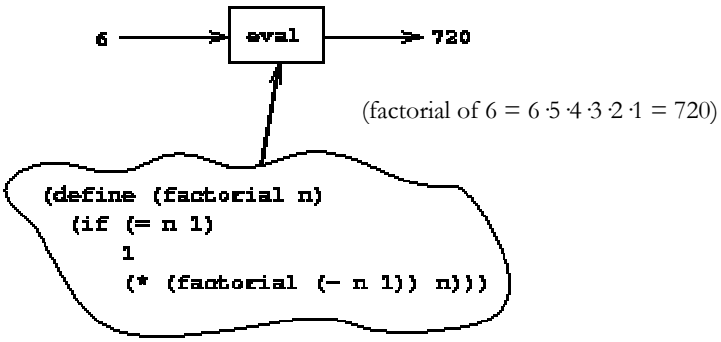
1. The machine "factorial"

This applies regardless whether the machine crushes stones, makes salami from raw meat, or keeps a certain rhythm controlled by the weights turning a wheel (the mechanical clock). Let us consider a rendering (2) of the machine called the factorial program. (In this case, the input is the number 6, whose factorial will be calculated).



2. The factorial program (see above) can be viewed as an abstract machine.

We notice here parts that decrement (6, 5, 4,...), multiply (*), and test for equality. We also notice a 2-position switch and a factorial machine (it takes a number and calculates its factorial). It is obvious that our machine contains another machine, the one called factorial. This qualifies it as an infinite machine, in the language of machine theory. Moreover, if we want to have our program evaluated, we need to submit it to an evaluator or interpreter. (In simple language, this means to test it in order to find out what to expect from the program.)



3. The evaluator emulating a factorial machine and producing the value for 6!

(The two examples are presented here with the kind permission of Harold Abelson and Gerald Jay Sussman, who, with Julie Sussman, wrote a fundamental book on computer programs[2].)

The evaluator takes as input a description of a machine (the program) and emulates its functioning. Accordingly, the evaluator appears as a universal machine, that is, it "knows" how all programs work. Imagine such an evaluator as an entity that can look at the plan for your future house and return something like a validation stamp, or draw attention to the fact that the bathroom on the second floor has no connection to the water pipe. Or the same evaluator can check out your Webpage design, your new recipe for chicken soup, or the plans for a new car, and return a meaningful interpretation that will guide your actions. No human being can contain the knowledge, not to mention the broad scope of such diverse projects, that it takes to validate the "program" we call

[2] Abelson, Harold, Gerold Jay Sussman, with Julie Sussman. *Structure and Interpretation of Computer Programs.* Cambridge MA: MIT Press, 1996.

architectural plan, Webdesign, chicken soup recipe, or automobile engineering (to name a few examples). To evaluate an open-ended gamut of programs is impossible. That program evaluation is possible when we have a limited domain of knowledge, and indeed necessary in all computations corresponding to the limited domain defined by the program, corresponds to the nature of knowledge representation in the form of programs and to what we expect from a program. To state it briefly: Gödel's incompleteness theorem[3] guarantees that for a limited domain the description can be complete and consistent.

Have you noticed any reference to *time* up to this point? No. The reason is simple. Machines are timeless; even the machine we call "evaluator" (or "interpreter") is timeless. Unless something breaks down in a machine, it will endlessly and uniformly perform the function for which it was conceived. The best machine repeats itself *ad infinitum* (or at least until its physical breakdown) without taking note of it. If it has a counter, the number on the counter shows "How far from infinity" it is. Its reason for existing is this orderly behavior, its predictability. Its underlying principle is determinism: the cause-and-effect sequence. There is, however, an implicit time factor here: the cause precedes the effect. But in fact, this time factor is also reducible to a machine, more precisely, to one that measures intervals. That time is more than an interval—just as space is more than distance—is an idea we will eventually have to entertain as we progress in our discussion of the timeliness (and *future-ness*) of programs.

Once we acknowledge determinism as the underlying principle of the human-made entities we call machines—in the form of artifacts or as mental machines—we also acknowledge that the "patron saint" of this perspective of the world is René Descartes (1596-1650)[4]. Western civilization adopted his views, albeit some (Is the human being reducible to a machine?) slightly modified. (For reasons apparently having to do with a healthy survival instinct, or with opportunism, Descartes claimed that only animals, but not human beings, are reducible to machines.) Consequently, western civilization adopted the rationality of his fundamental contribution—reductionism—and gave up any claim to a holistic understanding of the world. According to Descartes and others, all there is, in its amazing complexity, can be managed by breaking the whole into its parts and then describing each and every component from the deterministic viewpoint of the cause-and-effect sequence. Within this

---

[3] Gödel, Kurt. Über formelle unentschiedbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatsh. Math. Phys.*, 38 (1931), pp. 173-198.

[4] Descartes, René. *Discourse de la méthode pour bien conduire sa raison et chercher la vérité dans les sciences*. Leiden, 1637.

encompassing view of the unified world, machines are a good description of the living as embodiment of a functionality achieved from lower-complexity elements. The Cartesian conceptual revolution extends well into the computer age, although as we advance in our understanding of the difference between the living and the physical, we are starting to question some of its tenets. Moreover, with the computer, the inherited notion of the machine starts being challenged: no more gears, weights, and coils, no more an exclusively energy-controlled entity, but rather one in which information (mainly in the form of data) plays the crucial role.

Obviously, this is not the place to rewrite the chronicle of determinism or to start yet another anti-determinism crusade. Neither is it the place for an account of the many questions it has left unanswered. However, without understanding the fundamental perspective it establishes, and the challenges we face in questioning this perspective, moreover in establishing a new perspective, we could not answer even the most trivial questions regarding the future of computation. Indeed, in looking at the time aspects of programming, we are looking (aware of it or not) at the future of computation. Some see this future already sown into the programs we write today; others in the new technologies of computation (computing with light, DNA computing, quantum computation, etc.). And yet others see this future in a computation understood as a living entity, or at least as a hybrid entity (involving living components) able to reach anticipatory performance.

Early on, some of us realized that the computer, as a machine, is by no means more interesting than an abacus. We did not doubt that an automated abacus is faster than any expert in using this relatively old arithmetic machine. We did not doubt that an automated abacus could perform many operations per time unit (i.e., that it can be fast), that it could store data (even in primitive registers) beyond our own memory performance, that it could become the functional repository of all our arithmetic needs. In other words, it would "know," for us, all there is to know about arithmetic. What really drew my attention to the machine called computer—the term *computer* was used in the 19th century to denote a profession carried out by human beings—was a totally different question: Does the abacus know arithmetic? (The human being called computer by his profession knew arithmetic and probably more than that.) Furthermore: Does the computer know—as much as the human called *computer*—what it is processing? If it does, then does this understanding of what is computed affect the outcome? And again, if it does know, then how? (The human *computers* learned it and continued learning as they were exposed to new data.) Where does the machine's knowledge come from? (The program carried out by the professionals called *computers* took the form of astronomical tables.) But before ending

this personal aside, I want to add one more piece of information. My computer—the one I was supposed to study and program—did not exist; it was on paper[5]. This sounds absurd today, but that was the reality in a part of the world—Romania—where the absurd was "invented." (Just recall Eugen Ionesco's theater of the absurd, and Tristan Tzara's Dada movement.) And that was my good fortune because, after all, computation is about programs, not about switches, tubes, electrons, storage, keyboard, and all that makes up the necessary, but by no means sufficient, hardware.

Computation emerges as nothing more than a way of automating mathematics. Yes, human *computers* were slow, made errors, got sick, took time off. An automated procedure was by far more adequate and economical. Before digital computation, others tried to reach the same goal by using means corresponding to the pragmatics of their time. John Napier (1550-1617), the Scottish inventor of logarithms, tried (around 1610) to simplify the task of multiplication. (Napier's rods or bones, as they were called, served the purpose.) Blaise Pascal (1623-1662) worked on adding machines (1641); Gottfried Wilhelm Leibniz (1646-1716), who I consider "the father of the digital," introduced binary code; Wilhelm Schickart (1592-1635) built a machine (described by Kepler) that performed sophisticated operations; Joseph-Marie Jacquard (1752-1834) built the loom, able to generate complicated patterns (computer graphics before the age of computers!). Many have tried to write the history of these early attempts at automatic calculations. And many made up all kinds of stories since the subject is conducive to fictional accounts. Obviously Charles Babbage (1791-1871) figures high in such books (and stories) through his two machines—the Difference Engine and Analytical Engine (which apparently were never built)—as well as William Stanley Jevons (1835-1882), who in 1869 built a machine to solve logic problems[6]. Through various stories, we became aware of E.O. Carissan (1880-1925), lieutenant in the French infantry who made up a mechanical contraption for factoring integers and testing them for primality. And we know of Leonardo Torres y Quevedo (1852-1936), who assembled (or is famed for allegedly having done so) an electromechanical calculating device that played chess endgames.

---

[5] At the Polytechnic Institute in Bucharest (1955-1960), I programmed on paper and carried out debugging on paper. It could just as well have been done in my mind.

[6] Peirce, in Logical Machines (November 1887, published in *The American Journal of Psychology*) described the Jevons machine, as well as Marquand's machines. He also pointed out that the study of the transition from such machines to the Jacquard loom would "do very much for the improvement of logic." See *The Writing of Charles S. Peirce,* Peirce Edition Project, Vol. 6, 1982, p. 72.

This short account (which omits many details) is only indicative of an understanding of temporality that extends well beyond the current use of the word "program." Each of the individuals—scientist or not—mentioned above programmed, but more in the sense in which the abacus is programmed, not the digital machine, which is at the heart of computers today. One can say that the abacus is "hardwired;" in other words, it is its own program. Dependence on the hardware is implicit in mechanical and electro-mechanical contraptions. Napier's rods and Pascal's adding routines are timeless. Indeed, today they would allow us to carry out operations with the same degree of precision as in the days in which they were conceived. Even in our days, the Jacquard loom serves as a model of programmed patterns, the only difference being that in a program we can handle more data and readjust the "digital loom" in almost no time.

Since I mentioned Babbage, two things deserve to be highlighted: He extended the meaning of the word *engine,* corresponding to the machine of the Industrial Age, to a processing unit of mathematical entities. As a cognitive instrument, the metaphor affected the future understanding of machines meant to process information. Some[7] attributed to Charles Sanders Pierce a computer using the electro-mechanical switches of a hotel system (pointing to rooms reserved, occupied, vacant). Peirce went far beyond Babbage, as he wrote upon the latter's death[8]:

> But the analytical engine is, beyond question, the most stupendous work of human invention. It is so complicated that no man's mind could trace the manner of its working through drawings and descriptions, and its author had to invent a new notation to keep account of it (p. 458).

He also pointed out very precisely that "Every reasoning machine [as Peirce called them] . . . is destitute of all originality, of all initiative. It cannot find its own problems; . . ."[9]

---

[7] Ketner, Kenneth L. The Early History of Computer Design: Charles Sanders Peirce and Marquand's Logical Machines, *The Princeton University Library Chronicle,* Vol. XLV:3, Spring, 1984. And: Gardner, Martin. *Logic Machines and Diagrams,* 1959.

[8] Peirce, Charles Sanders. Logical Machines, *The American Journal of Psychology*, November 1887, p. 70.

[9] Peirce, Charles Sanders. Charles Babbage, *Nation* 13 (9 November 1871): 207-208, reproduced in the *Peirce Edition Project,* Vol. II, 1984, pp. 457-459. (See also Endnotes.)

The automation of calculations for ballistics in Howard Aiken's (1900-1973) Mark I (1944) and in the artillery calculations on a general-purpose electronic machine, on the ENIAC (at the Moore School of the University of Pennsylvania) are in fact the identifier for computation. Indeed, automated mathematics is the shorthand for the initial computer. Behind this not trivial observation we find the origin of almost all the questions preoccupying us today in respect to computation. This begs some explanation. Descartes proclaimed the reduction of everything to the cause-and-effect sequence and the reduction of the living to the machine as the embodiment of determinism. This reduction resulted in the description of time as duration, and of the living as functionality (which the machine expressed). The program of the Descartes type of machine is given once and for all time. It does not change, as performance does not change unless the components break down. In the deterministic machine, the implicit time dimension pertains to its functioning, which is dictated by the physical characteristics of the components. Such a machine exists, like everything else in Descartes' world, in the time dimension of existence reduced to duration.

With the advent of the computer, the implicit assertion is reasonable: For the class of mathematical descriptions of the physics of ballistics, artillery calculations in particular, we can conceive of a machine that will automate the calculations. In other words, the determinism of the physics described in mathematical equations of ballistics is such that we can automate their processing. One can generalize from such equations to many other phenomena. Space exploration, as well as the trivial description of playing soccer, comes easily to mind. One can take the mathematics of the particular ballistics problem as an attempt at modeling many phenomena of practical impact. If we know how to handle such difficult descriptions, we already know how to handle simpler cases, ranging from simulating a game of billiards, to building games driven by the same program, and to building a control device to guide a rocket. The abstraction of mathematical descriptions, to which I shall return, makes them good candidates for an infinite variety of concrete applications.

This is no small accomplishment. But it is by far not yet what we understand when we use the words "computer" and "programs." We need to be even more specific. Ballistic equations, as complex as they can get, are but a small aspect of mathematics. (In the meanwhile, they have been substantially improved.) For all practical purposes, a dedicated machine (driving a cannon, for instance) is nothing more than a description of the task to which it is dedicated. The implicit assumption is that of Descartes' machine: it performs within a world that is regular, repetitive, and predictable. Even the variety of applications it might open is treated the same way. But once we transcend the specialized machine

and enter the domain of computation as a universal process, we transcend the boundaries of the reductionist perspective. And we are forced either to accept the model of a permanency that extends from the physical to the living and to society, or to acknowledge dynamics and take up the challenge of understanding knowledge as process. More questions to come our way and guide our endeavor are:

- Is everything reducible to mathematical description?
- Is everything describable in the language of 0, 1 (precision vs. expressiveness)?
- Is Boolean logic the expression of all there is to the logical decisions we make in life (whether it is a matter of deciding what to have for breakfast or how to understand the genetic code)?

The computer understood as automated mathematics carries a Cartesian curse with it: If something is reducible to a mathematical description—or if our mathematical descriptions are abstract enough—it can be computed. Many scientists and engineers live by this notion. They simulate life in computational form and study it as though it were real, as real as our skin, pain, birth, death—only on a more global scale. Others draw our attention to a simple realization: Our descriptions, regardless of the medium in which we produce them, are constructs. Their condition is not unlike the condition of everything else we construct, subject to physical limitations, but also nothing more than products of our minds. Such mind products are focused on understanding the context in which our individual and collective unfolding as human beings takes place.

Today we call the mathematical description clause a necessary condition for something to be expressed through computation as we know and practice it. We add to it the expectation of a logical description. To be computational means to be expressed through a computational function. That this condition is ultimately insufficient is due to the time factor. Imagine that we have captured whatever is of interest to us (for our work, enjoyment, inquiries, etc.) in computational form. And imagine that we have enough computational resources to process our data. Time limits the endeavor since many computational functions are undecidable or *intractable,* which basically means that it will take time beyond what we dispose of (the lifespan of an individual or a generation) to perform the computation. Complexity has its price. Descartes was not in a hurry. For him the clock's rhythm was fast enough to guide his descriptions of the living as being no more than a machine that processed sensorial information. Now that our clocks have become very fast—the beat of the digital engine at the heart of our desktop machines reached the gigaherz level—we hope to recover some of the complexity that in more steady

times was done away with. Still, the most intriguing questions we would like to address remain outside the domain in which automated mathematics and automated logic (sometimes called reasoning) supports many of our current theoretical and practical endeavors.

This brings up the more probing question of the language describing our inquiry. If we want to acknowledge time, we have to deal with process. The word describes a dynamic entity (something taking place over time). And as it takes place, it affects something else. In the mechanical age, to process meant to affect the physical and chemical appearance of substances we wanted to change, preserve, combine, or extract. Today the verb *to process* applies to an abstract entity called information. Indeed, computers can be understood only in association with the object of their functioning, only as processing data (the concrete form of information). To ignore that our notion of information stems from thermodynamics means to move blindly through a world that is constructed on the very premise of our acceptance of the laws of thermodynamics.

A closer look at how information is defined might tell us more about the time dimensions of programs than programs themselves. Shannon's genius (and direction) is probably comparable to that of Descartes. He considered information strictly from the engineer's perspective: Give me an input that my machine can expect, and I will make sure that the processing of this input will not alter it beyond recognition. His focus is on communication; and accordingly, he does not concern himself with anything but the physical properties of the carrier. Meaning is ignored, which means that, specifically, the semantic dimension is of no concern. What I describe here is well known; I myself have addressed this issue of the exclusive use of syntax more than once[10]. But there is one more thing to be added here: Syntax—as we know it from semiotics (to which I shall return) is timeless. It captures only the description of the carrier, not the meaning of the message, and even less the pragmatic dimension. Working for the Bell Telephone Company, Shannon was concerned with the price of sending messages through a telephone line. He noticed that a great deal of what makes up a message is repetition (what we call redundancy, i.e., that part of what we exchange in communication that carries nothing new with it). Actually, for Shannon, information was the inverse of redundancy,. If prior to reading these lines your knowledge of Shannon's theory was that a) "He was the founder of information theory;" and if after reading these lines you double your knowledge to b) "Information

[10] Nadin, Mihai. Consistency, completeness, and the meaning of sign theories: The semiotic field, *The American Journal of Semiotics,* 1:3, 1982, pp. 79-88.

theory is reductionist theory," then I contributed a bit (pun intended) to your knowledge.

Recently, the American public were glued to their television sets watching the outcome of a celebrity trial (Martha Stewart, a household name in the USA, was on trial before a jury). It was a typical Shannon experiment. There were four counts on which the jury had to render a verdict of guilty or not guilty. Outside the courthouse, thousands waited to hear the outcome. TV cameras from around the world focused on the exit from the Manhattan courthouse. And as the foreperson was reading the jury's verdict, strange messengers started a bizarre show: They ran ahead waving above their heads colored scarves—red for guilty on count 1, blue for guilty on count 2, etc. The TV viewers had no access to details of the color code prepared in advance by journalists hurrying to be the first to make the verdict known. Prior uncertainty—guilty or not on count 1, etc.—was halved each time a runner with a colored scarf ran down the stairs. Ultimately, if the jurors themselves had been on the stairs and used all the sentences read inside the courtroom, the information would have been the same. The text they would have read would be informationally equivalent to the color of the flag. One *bit* is defined as the information needed to reduce the receiver's uncertainty by half, no matter how high that prior uncertainty was. It is a logarithmic measure, and the formula behind the whole thing is

$$H = -\sum_{i=1}^{n} P_i \log_2 P_i \text{ (bits per symbol)}$$

This says that information, defined on the premise of the reductionist machine model, is commodity, quantifiable like energy consumption, or like currency flows. In this respect, every program that is based on the assumption that entropy (a concept originating in thermodynamics and describing the disorder of a system) is a good model for information dynamics remains fundamentally in the realm of physical entities and their respective deteministic laws. This is the model of the carrier (the sign) reduced to its appearance (the syntax).

In the realm of the living, which always includes the physical but is *not* reducible to it, entropy only partially qualifies the dynamics of the whole. Accordingly, for all programs pertaining to artificial machines, Shannon's information theory is an appropriate foundation. But once we enter into living computations, or into the promising hybrid computation (living and artificial in some functional connection), the notion of information itself is no longer adequate. With the living, time, in its richness—that is, no longer only duration and no longer a one-directional vector—has to be acknowledged and indeed accounted for in the programming.

In recent years, the Boltzmann equation

$$S = k\log W$$

—in which S stands for entropy, k is a constant (the Boltzmann constant), and the logarithm of the states of a system W ("elementary complexions")—which stands behind Shannon's work, underwent scrutiny. Constantin Tsallis[11] belongs to those who noticed that under certain circumstances, some systems will actually undergo a reduction in their entropy. This new theory of disorder takes into consideration the dynamics of self-organization. Moreover, Leo Szilard[12], in describing biological processes, took note of the decrease in entropy in living systems. With all this in our minds, it is important that we realize that sooner or later information theory itself will have to be redefined in order for us to account for the fundamentally different dynamics of life.

My own position is that anticipation is what distinguishes the living from the non-living, and that anticipatory computation can be achieved only by effectively redefining information as to include not just the semantic dimension, but foremostly to make the pragmatics possible. From reaction-based computation expressed in programs that are machines, to anticipatory computation, we will have to redefine many of our fundamental premises. Together with the *bit,* an *antebit* will describe the process. The bit will effectively describe the probabilistic domain (and all the post-fact statistical information), while the *antebit* will describe the possibilistic domain (and all the pre-factual opportunities). This brings us back to the implementation of information processing in what we call computers (information processing machines, in particular, programs).

Mathematics allows us, among other things, to describe what we call reality. It is not the only means of description. So-called natural language can be used for the same purpose. Images, in one form or another, are also descriptions. So are sounds. Some domain-specific means of description constitute the "language" of those domains: the formalisms of chemistry (chemical formulas are a well-defined means of description), of genetics, or of logic (focused on thinking). That such means of description of what there is can, at the same time, be a means of

---

[11] Tsallis, Constantine, V. Latore, M. Barager, A. Rapisarda. Generalization to non-extensive systems of the rate of entropy increase: the case of the logistic map, *Physics Letters* A 273, 2000.

[12] Szilard, Leo. Über die Entropieverminderung in einem thermodynamischen System bei Eingriffen intelligenter Wesen, *Z. Phys.* 53, pp. 840-856. (See also: On the decrease of entropy in a thermodynamic system by the intervention of intelligent beings, *Behavioral Science,* 9, pp. 301-310.)

synthesizing something that is only in our minds will not preoccupy us here. But we should never ignore the complementary nature of the analytical (description) and the synthetical (design). Programs are a true exemplification of this condition.

Mathematical description expresses what is called declarative knowledge. One can generalize to the declarative knowledge expressed in logic, chemistry, genetics, etc. It concerns what is, as captured from a certain perspective. For instance, the ballistic equations whose solution prompted automation in a computer program describe the physics upon which artillery is based. That there is more to a cannon than only trajectory is well known, even by those who have never operated a cannon. But for all practical purposes, the program to guide artillery actually describes the physics of throwing a ball from A to B. This program is no longer an expression of declarative knowledge, but of imperative knowledge: how to hit a target (which might even move from B to C as we try to target it from A).

In relation to mathematics, computer science, which has programs as a goal (among others, of course), belongs, not unlike machine engineering, to the domain of imperative knowledge. It consists of procedures that are descriptions of how to perform activities. And as any other procedure—let's say hammering a nail into a wall—it is based on recursion: it has its own actions as a reference, it is self-referential. There is an implicit circularity here: all of it is repetitive (defined in terms of what is repeated, i.e., in terms of itself). Simply stated, what governs the whole endeavor is a strategy of decomposition: divide the action into parts. If each part has a well-defined identifiable task, this task can become a module for other procedures. The abstraction process guarantees efficiency: One does not have to reinvent the hammering of the nail each time this action becomes necessary. By the same token, recursivity speaks of the successful reduction of the task into independent procedures. This is why computers are made of machines that contain machines that contain machines, etc. Every detail is suppressed in the process. The meaning of such modules ought to be independent of parameters not essential to the task. (Remember the program called *factorial*? The factorial procedure is independent of the size of the number n. Think about a procedure returning the volume of a complex object. The parameters of the object, or the nature of its surface, or the density of the material should not affect the calculation).

Let's be clear about the following: Declarative and imperative knowledge can be conceived only as interrelated. We can easily realize how declarative knowledge (take a mathematical equation describing the reflection of a ray of light on a mirror) is "translated" into imperative knowledge (the computer program that shows the reflection). It is by far

more complicated to use imperative knowledge (a description of a scene) in order to infer from it declarative knowledge (a deduction: "There must be something blocking the reflection"); but it is an operation that is relatively often performed (for instance, in interpreting images captured by digital cameras).

There is only one reason to insist on these aspects: to make it clear that the more abstract the procedure, the more effective it is (provided that it is an adequate procedure). Abstraction ultimately means the squeezing out of time. We can build programs from modules only if they are time independent. Compound procedures have no internal time; their reference to duration is a reference to their internal dynamics, not to that of the world. The space and time effectiveness of programs within the reductionist view of computation concerns the space (storage) and time (synchronization mechanisms) implicit in the processing, not in the entities described. However, with computation, machines open up to time through the dimension of interactivity. While every other known machine is timeless, computers open up the possibility of being driven by data from the order of events (as in playing a game), better yet, of reflecting the order of events, or of introducing (in robotics, for example) an order of events that allows for the performance of a specific task, or of achieving a complex behavior. This new dimension renders the deterministic mold relative.

In order to achieve interactivity, programs rely on mathematical and logical descriptions that reflect the dynamics of the expected activity. If you want a real-time facility for correcting spelling in a word processing program, you have to provide a dynamic view of the activity called *writing*. Obviously, the more complex the activity—let's say target recognition at microscopic levels (after labeling an ingredient of a medication, following its "moving" through the tissue subject to treatment, and even "guiding" it so that it reach a desired region), or at the interplanetary scale (the landing on Mars resulted in many examples)—the more elaborate the description and implementation in programs. At this level we no longer distinguish durations (the determinsitic reduction of time), but time variability: slower than real time, real time, faster than real time. We also identify the depth of time, as in synchronicity; or as in parallel streams of time (while process 1 unfolds, process 2, related or unrelated, unfolds within the same time scale or not, etc. etc.); or as in different directions (the time vector is bi-directional and probably even multi-directional).

Within this view, the machine model has to be revisited in the sense that the clear-cut distinctions characteristic of the black box (Input, Output, States) be redefined. In particular, the local state variable describing the actual state of the computational object has to be defined in such a way that it allows for change. Computational objects with state variables that

change correspond not only to trivial tasks subject to automation (such as bank account management), but also to very complex tasks (such as cooperative design over networks). Functional programming is not adequate is such cases. Imperative programming, which introduces new methods for describing and managing abstract modular program entities, is but one of the many methods developed for this purpose. Obviously, the description of the living that we call genetics is better adapted to the task of supporting interactivity. This explains why new forms of computation, which are based on the abstractions of particular fields of knowledge (e.g., genetics, quantum mechanics, DNA analysis), emerge as we try to capture time-based phenomena. From the viewpoint of time processes, a stone is in a different situation from a living entity. With the advent of anticipatory computing[13], this becomes more and more evident.

Max Bense, the mercurial prophet of rationalist aesthetics in a country perverted by speculation, correctly noticed that "Nicht die mathematische Beschreibung der Welt ist das Entscheidende, sondern die aus ihr gewonnene prinzipielle Konstruierbarkeit der Welt"[14]. (It is not the mathematical description of the world that is decisive, but the principle of the constructability of the world that is gained from it, [translation ours].) Unfortunately, as was the pattern of his activity (and private life), he did not stop in due time. He went on to speak of "die planmäßige Antizipation…einer zukünftigen künstliche Realität," (anticipation according to plan…of a future artificial reality [translation ours]); and finally, to ascertain: "Nur antizipierbare Welten sind programierbar, nur programierbare sind konstruierbar und human bewohnbar." (Only worlds that can be anticipated are programmable, only programmable [worlds] are capable of being constructed and inhabited by human beings, [translation ours].) None of his illustrious students (whom he managed to antagonize through acts bordering on the irrational) noticed how deterministic thought in the end led their master to an upside-down image of the role of computation. But at least Bense had the stature of an *agent provocateur*. In comparison, the new theoreticians of "aesthetic computation" are at best pygmies too self-important to read what others wrote long before they did.

Concurrent processes, i.e., taking place in parallel, although not necessarily along the same time metric, are only an image of the complexity of the time identity of interactive programs. The timeliness of programs that by now have adaptive qualities and display evolutionary

---

[13] Nadin, Mihai. Anticipation—A Spooky Computation, *CASYS, International Journal of Computing Anticipatory Systems* (D. Dubois, Ed.). Liege: CHAOS, Vol. 6, 1999, pp. 3-47.
[14] Bense, Max. *Einführung in die informations theoretische Ästhetik* (Introduction to Information Theory Aesthetics). Reinbeck, 1969.

characteristics is quite different from that of "canned" programs. Their future is different from what industry understands today when they produce the next version of a program or an operating system. In fact, in addressing the issue of timeliness and *future-ness* of programs, we soon come to the conclusion that technology, as the embodiment of the successful use of programs, can limit performance but should not, as still happens, drive content. As interactivity becomes the driving force, we should be able to move beyond task-based computation to a pragmatic foundation. In other words, instead of the routine of launching "canned" programs of timeliness applications (word processing, paint program, browser, etc.) under the guidance of operating systems, we should be able to execute pragmatic functions—I want to represent my data for colleagues all over the world—that would interactively select the appropriate applications and use them as we, users in a deterministic role, do today. This role change will render the overhead of training operators obsolete, and our question regarding program timeliness will take on new meaning: Is the co-evolution of the living and the programs it conceives possible?

Deep down, in the digital engine, there are two elements controlling and making computation possible: an "alphabet" and a "grammar." These two together make up a language—machine language. The alphabet consists of two letters (0 and 1); the grammar is the Boolean logic (slightly modified since Boole, but in essence a body of rules that make sense in the binary language of *Yes* and *No* in which our programs are written). The assembler—with a minium of "words" and rules for making meaningful "statements"—come on top of this machine language; and after that, the level of "formal language" performance in which programs are written or automatically generated. Such programs need to be evaluated, interpreted, and executed. Here I submit to the reader structural details we all know (some in more detail than others) but which only rarely preoccupy us. My purpose is very simple: to lend meaning to my point that computers are semiotic machines (which I first articulated over 20 years ago[15]). Too many scholars took over my formulation (with or without quotation marks or attribution) without understanding that as a statement, it is almost trivial. What my colleagues—some of them respectable authors and active in semiotic organizations claiming legitimacy—totally missed is the need to realize that such a description makes sense only if it advances our understanding of what we describe.

[15] As both computer scientist and semiotician, Mihai Nadin went on record that computers are semiotic machines as early as 1976 (see Nadin, Mihai. The repertory of signs, *Semiosis*, Heft 1, 1976, pp. 29-35; Sign and fuzzy automata, *Cahiers de Linguistique Théorique et Appliquée*, XIV, 1, 1977).

To say that the computer is a semiotic machine means to realize that what counts in the functioning of such machines are not electrons (and the in the future, light or quanta) but information expressed in semiotic forms, in programs, in particular. We take representations (which reflect the relation between what a sign represents and the way we represent it) and process them. This is the Shannon-based model. (Or should I say the Bell Telephone model?) Moreover, as we use computation, we try to assign a meaning to our representation. Since in the machine itself, or in the program that is a machine, there is no place for a semantic dimension, we build ontologies (which are databases not dissimilar to encyclopedias or dictionaries) and effect association. This is how search engines frequently work; this is what stands behind the new verb "to google" and our actions when we start research by identifying sources of knowledge in the world-wide Web.

But even our ontologies, hand-made or automatically generated, stand on the "shoulders" of the language of zeros and ones (of *Yes* and *No*) and of the Boolean algebra that is the grammar of this primitive language. Any semiotician worth his salt should by now know that the means of representation actively influence and affect the representation. They are not neutral, but constitutive of interpretant processes that make up our way of thinking, that influence our actions. To say that computers are semiotic machines means to realize that the interpretant, i.e., infinite semiosis (the sign processes through which we become part of the signs we interpret) causes us to act differently, to think differently, to express ourselves differently from the way we did when language (and literacy) were the dominant means of expression, communication, and signification.

The extreme precision brought about by an alphabet of two letters and a grammar of clear-cut logic comes at the expense of expressiveness. The more precise we are, the less expressive the result. In terms of program timeliness—*future-ness,* in particular—this means that we could capture time and make it a part of programs only—but only when computation will transcend, as it partially does, not just its syntactic dimension, but also the semantic dimension of the signs making up programming languages. Indeed, at the moment when computation will be pragmatically driven by what we do, it will acquire a time dimension coherent with our own time—and will reflect the variability of time. We are what we do, and accordingly, if we could integrate programs in what we conceive, plan, execute, and evaluate, and thus in our own self-constitution, we would establish ourselves not just as users, but also as part of the program. For this to happen, many conceptual barriers need to be overcome. We would have to address the need to redefine information, to redefine the alphabet and the logic, to rediscover quality as the necessary complement of

quantity, and to understand the digital and analog together. Within my program, which is a bit more radical, this means to practice not only the deterministic reaction mode of the physical, but also the anticipatory characteristic of the living.

Endnotes

Footnote 1. *The Civilization of Illiteracy* focuses on what makes the humankind's successive ages (referred to as pragmatic framework) possible and necessary. The German translation is an abridged version of the English

Footnote 9. In the same article, Peirce gave some details regarding Babbage:

> About 1822, he made his first model of a calculating machine. It was a "difference engine," that is, the first few numbers of a table being supplied to it, it would go on and calculate the others successively according to the same law (p. 457).
>
> He discovered the possibility of a new *analytical* engine to which the sufference engine was nothing; for it would do all the *arithmetical* work that that would do, but infinitely more; it would perform the most complicated *algebraical* processes, elimination, extraction of roots, integration, and it would find out for itself what operations it was necessary to perform; . . . (p. 458)

Footnote 15. As both computer scientist and semiotician, Mihai Nadin went on record that computers are semiotic machines as early as 1976 (see Nadin, Mihai. To ward off any claims to the opposite, the pertinent articles are listed as follows: The repertory of signs, *Semiosis*, Heft 1, 1976, pp. 29-35; Sign and fuzzy automata, *Cahiers de Linguistique Théorique et Appliquée*, XIV, 1, 1977). In the USA, Nadin announced that "the computer is the semiotic machine *par excellence"* (cf. *Visual semiotics: methodological framework for computer graphics and computer-aided design.* Lecture presented at the conference, *The Designer and the Technology Revolution* at the Rochester Institute of Technology, Rochester, New York, May 13-15, 1982). He was the first to offer classes at the Rhode Island School of Design, Brown University (both in Providence, RI), the Rochester Institute of Technology, and the Ohio State University. He did semiotic consulting for Apple's Lisa computer (user interface evaluation) in 1981-82. Further documentation can be found in: Interface design and

evaluation, *Advances in Human-Computer Interaction*, vol. 2 (R. Hartson, D. Hix, Eds.). Norwood NJ: Ablex Publishing Corp., 1988; The bearable unbearability of the rational MIND, *Ästhetik, Semiotik, Informatik* (F. Nake, Ed.). Baden-Baden: Agis Verlag, 1993, pp. 63-102.

Until about 1995, the semiotic establishments at the leading centers in the USA, North and South America, and Europe, were still emphasizing semiology (as opposed to Peircean semiotics). They had not even come so far as to consider the visual as a domain worthy of semiotic research. Once so-called semioticians decided to consider the computer, they did so without checking whether anyone had written anything before they had their "revelation." (Sadly, this is the case with many fields of academic endeavor, such as the aesthetics of computer science.) And, as has been the case with most endeavors in semiotics, the scholars/scientists either misunderstand the subject or, blinded by their brilliant discovery, lose all sense of academic rigor and lead others into the pit.

Some examples of people who took over the term "semiotic machine" without giving due and proper credit are: Pedro Barbosa, with José Manuel Torres (O computador como máquina semiótica, no date); Dr. Gerd Döben-Henisch (Semiotic Machines. Theory, Implementation, Semiotic Relevance, Aug. 6, 1996, 8th International Semiotic Congress of the German and the Netherlands Semiotic Societies); Karin Wenz (Representation and self-reference: Peirce's sign and its application to the computer, *Semiotica* 143–1/4, 2003, 199–209). Among the conferences and congresses held on the subject, and where, again, Nadin's fundamental work was totally ignored, are: 10th International Congress of the German Association for Semiotic Studies (DGS), University of Kassel (Germany), July, 19-21, 2002; Section "Semiotics and the Computer" (under the auspices of Winfried Nöth, Kassel University, Department of English and Romance Language Studies [of all things!]).