

Abstraction Barriers in Mathematics and Computer Science

Uri Leron <uril@tx.technion.ac.il>

Technion – Israel Institute of Technology

Plenary talk at LME3, 1987, Concordia University, Montreal, Canada

0. Introduction

Abstraction is a major tool in both mathematics and computer science for the organization and control of complexity, hence its importance in enhancing problem solving, understanding of and communication about complex systems.

The talk falls into three parts, the first two being mainly expository, the third containing the main thesis. The first part is a (brief and much oversimplified) discussion of abstraction in mathematics, dwelling especially on its relation to generalization. The second part turns to computer science, where abstraction is mainly concerned with hierarchical organization of complex systems and the suppression of low-level detail. It turns out that in computer science much of the struggle for reducing complexity and achieving good communication has to do with choosing the "right" level of abstraction for a given situation, and erecting *abstraction barriers* (hence the pun in the title) to guard against the intrusion of low-level detail. In the third part I put forward the suggestion that these insights from modern computer science have implications for mathematics education, namely that communication, learning and problem solving can be enhanced by carefully choosing an appropriate level of abstraction for a given situation. There is nothing new in saying that too much abstraction can be a barrier to communication and learning in a given situation; but it is less well known that we can (and do) also err in the other direction, i.e., that of *too little abstraction*. Thus we might be well-advised to look for a suitable *intermediate-level abstraction* to match each given situation.

1. Abstraction in Mathematics

Abstraction vs. generalization. In mathematics, abstraction is closely related to generalization, but each can also occur without the other. The formula

$(a + b)^2 = a^2 + 2ab + b^2$ is *generalized* (but not abstracted) from natural to rational to real to complex numbers. *Abstraction* occurs when the formula is seen to hold for any two commuting elements in a ring. This is also a generalization, for it encompasses all the previous cases and many more (e.g., a and b can be polynomials or two commuting matrices). The description "all the prime numbers less than 20" is more abstract than "the numbers 2, 3, 5, 11, 13, 17, 19" but not more general, since both specify the same set.

Groups. The group concept is abstracted (epistemologically and historically) from many relatively concrete examples – such as groups of permutations – by ignoring the specific features of each example and stressing the common properties, especially those that are seen to feature in the proof of important theorems. This process goes through many steps, some involving generalization, some not. Rigid motions in the Euclidean plane, symmetries and permutations, are all abstracted (and much generalized) in the concept of a *transformation group*. Then transformation groups are further abstracted by the concept of an (“abstract”) group. However, this last remarkable step in abstraction (first suggested by Cayley around 1850, but assuming its modern form and significance only in the 1890's and beyond) doesn't entail generalization: a theorem of Cayley states that every (abstract) group can be represented as a transformation group; thus the extension of the two concepts is the same. Mac Lane (1986), who gives this example, also cites the example of Stone's representation theorem of abstract Boolean algebra as an algebra of subsets (p. 436). It is noteworthy (and relevant to the theme of this talk) that “more concrete” doesn't necessarily mean “simpler”. Thus many of the important theorems of group theory (e.g. Lagrange's Theorem) gain in simplicity and insight when proved for an abstract group rather than for a transformation group.

Abstract vs. concrete. In general (according to Mac Lane) there seems to be a tendency in the history of mathematics to create ever more abstract structures, but then turn back and look for *representation theorems* that describe them in terms of more concrete systems. Two more examples of this phenomenon would be the abstraction n -tuples in terms of vector spaces, then the “concrete” representation of any (finite dimensional) vector space as an n -tuple space relative to a given basis; and the abstraction of matrices in terms of linear transformations, then the representation of those back in terms of matrices relative to given bases.

Another vast family of examples of abstraction-without-generalization consists of the many cases in which mathematical objects are given two (or more) equivalent characterizations, one more abstract than the other. Thus, consider the characterization of the rationals as a minimal field containing the integers; of the reals as a complete ordered field; of the complex numbers as the minimal field containing the reals and a square root of -1; of the subspace spanned by a set of vectors as the minimal subspace containing them (rather than the more “concrete” definition as the set of all their linear combinations); of the determinant as a multilinear alternative function of the matrix's rows.

In general we may say that in mathematics, “abstract” is opposed to “concrete”. Abstraction occurs when objects, sets and operations are characterized in terms of their properties and the relationships among them, rather than by concretely describing the objects or the operations themselves. Abstract descriptions are often also more functional, stressing how to *use* the described object rather than how to *construct* it. Thus the abstract description of the complex numbers as a (minimal) field containing the reals and square root of -1, tells us more about what can be done with them (e.g. solve equations) than the more concrete description in terms of pairs of real numbers (or numbers of the form $a + bi$). We shall come back to this distinction of "function vs. structure" in the third section.

2. Abstraction in Computer Science

Though abstraction in mathematics and abstraction in computer science are closely related, there are subtle differences in their meaning and in the epistemological place they occupy within the two disciplines.

A first example. I start with an extremely simple example which illustrates some of the main features of abstraction as used in computer science. Of course, its very simplicity prevents it from convincingly demonstrating the full *power* of the methodology. For a comprehensive treatment of the issues mentioned here see Chapters 1 and 2 of Abelson & Sussman (1985/1996).

Suppose we want to insert an instruction in a Logo procedure that will cause execution to pause for 10 seconds. Here are three different ways to achieve this. The first method (much used in the early Logo days) is to insert the instruction *Repeat*

7000 [] at the appropriate place in the procedure. The second and third methods are to insert in the procedure the instructions *Wait1 10* or *Wait2 10*, where the subprocedures *Wait1* and *Wait2* have been defined as follows:¹

<i>To Wait1 :Seconds</i>	<i>To Wait2 :Seconds</i>
<i>Repeat :Seconds * 700 []</i>	<i>If :Seconds < 0 [Stop]</i>
<i>End</i>	<i>Wait2 :Seconds - .025</i>
	<i>End</i>

It is revealing to compare the three methods, looking at them alternatively from the computer and from the human points of view. From the computer (i.e. formal language) point of view, the first two instructions are entirely equivalent since *Wait1 10* is evaluated by Logo as *Repeat 7000 []*. From the human point of view, *Wait1 10* and *Wait2 10* ought to be considered equivalent since both serve the same function (i.e., pause 10 seconds).

In computer science, *Wait1 10* is considered more abstract than *Repeat 7000 []*, even though both describe the same process. Abstraction here consists in creating a higher-level entity by means of chunking and naming, and then suppressing details of implementation. It is in this abstract sense that *Wait1* and *Wait2* are considered equivalent. Suppressing implementation detail highlights their common functionality, which in the end is all that matters in their use to help solve problems.

The forgoing discussion has demonstrated abstraction as it appears in a bottom-up piece of programming: The higher-level procedure *Wait1* was first created (by chunking and naming) from lower-level elements, in this case *Repeat* with certain values for the input parameters. Then we insisted on suppressing all implementation detail to promote thinking of the new procedure as an independent entity doing a well-defined piece of work.

Alternatively, we could approach abstraction in a top-down fashion: When programming a procedure in which a pause is needed, just insert a WAIT instruction, using it for the moment as if it were one of the language primitives. Here, again, one insists on thinking at the abstraction level appropriate for the situation, suppressing

¹ These numbers are approximately correct for LCS1's Apple Logo II, for which, however, this particular application is not very realistic since a version of *Wait* already exists as a primitive procedure.

low-level implementation detail. Note, however, that suppressing does not mean discarding altogether, only temporarily putting aside to be dealt with later.

Abstraction barriers. Programming at an appropriate level of abstraction means choosing *primitives* (basic terms) that are appropriate for the problem (and to the programmer) at hand. Since there is no reason to expect that the suitable primitives will just happen to be there as primitives of the language we are using, this mostly means *creating* our own primitives – in essence creating a special-purpose language for the problem at hand. The process of abstraction is essential if these are to function as true primitives and if we are not to be bogged down by the details of implementing our own language on top of the existing programming language. In refusing to deal with lower-level detail while thinking of the (high-level) problem to be solved, we have constructed a metaphoric *abstraction barrier* that separates the programming task into two nearly independent tasks: Above the barrier we deal with solving the original problem in terms of the *abstract primitives* we have selected; below, we deal with the details of implementing these abstract primitives. This methodology can greatly reduce the complexity we have to deal with at any particular moment. Of course, in dealing with complex programming projects, there are not just two but many levels of abstraction and many levels of abstraction barriers. Thus implementing the top-level abstract primitives may require constructing another set of abstract primitives and another abstraction barrier (and then yet another, etc.), before hitting rock bottom at the level of the computer language primitives.

Abstraction for sixth-graders. The power of abstraction is mainly manifest in large programming projects, where complexity becomes a serious issue. However, choosing an appropriate language for the problem at hand has important educational implications even with elementary school children programming with the Logo turtle. Here is an example, based on ideas of P. Mendelsohn (1986), which I have actually tried out with six-graders with very satisfactory results.

Suppose we ask children to write a procedure to draw the *Tower* displayed in Figure 1. Many children who try to build the tower with the turtle primitives *Forward* and *Right* (or even with the usual *Square*) get mixed up with messy interface bugs. Even those who eventually succeed in accomplishing the task, usually leave traces of a rather messy *style* of programming and, apparently, thinking too.

In contrast. When we asked them to build the *Tower* using Mendelsohn's abstract primitives (Figure 1) *SquareRight*, *SquareLeft*, *SquareMiddle* (abbreviated *SqR*, etc.) and the various *Jumps*, they did manage to negotiate it quite gracefully. Also, when asked, in a separate session, to actually implement these various “tool” procedures, they easily did so. Thus it appears that what really helped them deal with what was for them a complex problem, was our help in identifying the abstract primitives appropriate for the problem, and in erecting the abstraction barrier separating the solution of the problem in terms of these tools, from the details of implementing them on top of Logo. From the educational perspective, it may be valuable to discuss these issues with the students, and lead a class discussion in which the appropriate abstract primitives are actually *discovered*.

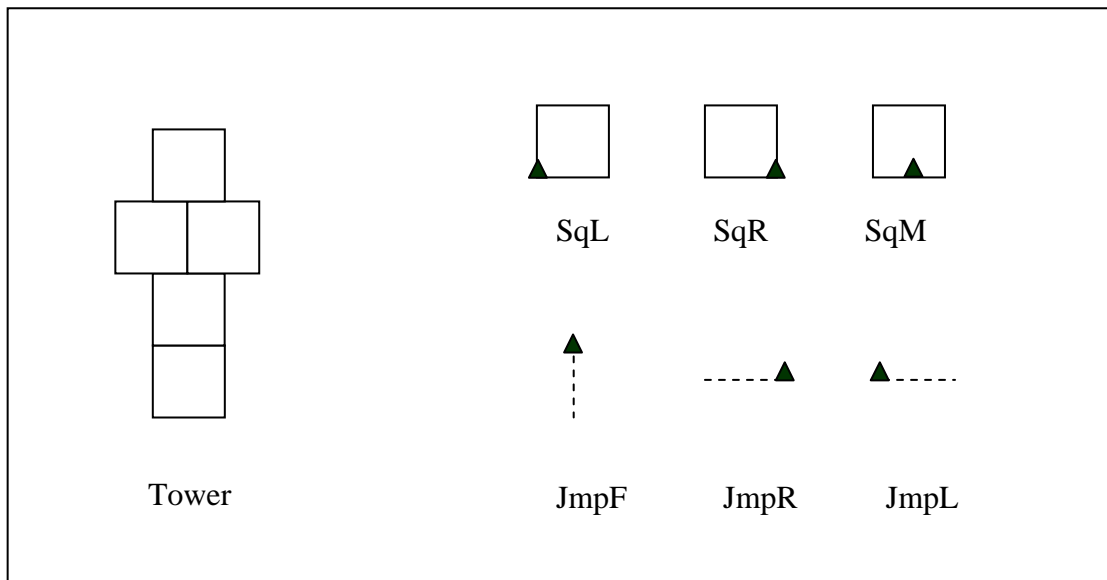


Figure 1

Procedural abstraction and data abstraction. While the first two examples (*Wait* and the *Tower* project) illustrate *procedural abstraction*, i.e. creating a higher-level procedure from simpler ones, the next example illustrates *data abstraction*, i.e. creating a higher-level *object* from simpler ones. This example, adapted from Abelson & Sussman (1985) p. 126ff, sketches the implementation in Logo of a system of complex number arithmetic. Approaching the problem abstractly, we postpone any decisions as to the representation of complex numbers in Logo. Instead we assume we already have available operations on complex numbers *Re :Z* and *Im :Z* (and possibly also *Abs :Z* and *Arg :Z*), called *selectors* (they *select* the parts from the whole) and an

operation on pairs of real numbers $MakeComplex :X :Y$, called *constructor* (it *constructs* the object from its parts), and we define complex arithmetic the usual way, using the real and imaginary parts of the complex numbers involved (or their magnitude and argument). For example, here is the definition of complex addition:

To ComplexAdd :Z :W

Output MakeComplex (Re :Z + Re :W) (Im :Z + Im :W)

End

By insisting that implementation issues be dealt with in a lower level, we have erected an abstraction barrier to separate the mathematical issues regarding complex numbers from implementation detail concerning the computer and the programming language.

Since we refuse to specify, at this level, what *Re*, *Im* and *MakeComplex* actually are, how can we insure that these operations behave the same way as the actual mathematical operations they seek to model? We can do this *abstractly* by postulating that:

MakeComplex (RE :Z) (Im :Z) outputs :Z for all complex :Z

(This also implies the reverse relations

Re MakeComplex :X :Y outputs :X for all real :X and :Y

Im MakeComplex :X :Y outputs :Y for all real :X and :Y).

It is noteworthy that these relations, together with the addition and multiplication defined on the abstract data, entirely determine the complex number field. In particular, they suffice to prove all the standard theorems.

The rules of thumb regarding the power of abstraction to control complexity and enhance communication, reflect the cumulative experience and wisdom of the computer science community over decades. Recently, the merits of working with a suitable *middle-level abstraction* has also found support from theoretical considerations. I am referring to Marvin Minsky's (1987) *Society of Mind*, especially his *K-lines* (knowledge lines) theory of memory (Ibid, Ch. 8). This is not the place to go into his theory in detail, so let me just quote one sentence from his *level-band theory* (Ibid, Section 8.5, p. 86):

The basic idea is simple: we learn by attaching agents to K-lines, but we don't attach them all with equal firmness. Instead, we make strong connections at a certain level of detail, but we make weaker connections at higher and lower levels.

Mind over machine. Suppression of detail, communication barrier, guarding against the intrusion of low-level detail – whence the aggressive overtones? They seem to indicate the metaphorical existence of some “monster” that is trying to force the innocent programmer into an improper style of thinking and programming. Indeed, if we look carefully at the kind of detail we need to suppress (namely, the kind of detail that will spontaneously occur in programming if we are not aware of these abstraction issues), we find that the beast is none other than the computer itself, as manifested through both hardware and software limitations. The “primitives” of the computer language are indeed too primitive to serve as efficient thinking tools for specific complex problems, and we need to use abstraction to fashion more suitable, higher level “thinking primitives”. In other words, abstraction is our weapon in the struggle to maintain human-level thinking against the pressures of computer-language-level thinking.

3. Implications for Mathematics Education: The Need for Intermediate-Level Abstraction

We have seen that abstraction is a major methodology in modern computer science for controlling complexity and improving communication. The use of abstraction in computer science was seen to involve a hierarchical view of complex systems, and appropriate use of naming, and the suppression of low-level detail. We now ask: Can this methodology be used beneficially in mathematics education as well?

We have also noted that abstraction can be seen as a weapon in the human programmer's struggle against the pressures of the computer (or computer language) to think in too low a level. Is there an analogous “monster” in math, which tends to force us to think and communicate in too low a level of detail? My answer to both these questions is “yes”, and I shall elaborate with the aid of some examples.

Roughly speaking... As a first example of how abstraction in the sense used in computer science might look in a mathematical context, consider the following “game” I sometime play with students in my teacher education classes. Suppose you

were to explain to someone, intelligent but with little mathematical training, what a prime number was. How would you do it in one brief sentence? Typically, students give answers which are mathematically correct but, in my view, carry little explanatory power. Two examples are: "A prime number is one which has exactly two distinct factors" (a clever definition that manages to cunningly exclude both 1 and trivial factorizations via a technical trick, but which completely obscures the meaning of the concept); and "a number, different from 1, that has no factors other than 1 and itself" (a definition that emphasizes relatively irrelevant detail and omits the most relevant feature). My own favorite choice runs something like the following: "A prime number is one that cannot be decomposed". Then, when elaboration is in order, I tend to first explicate what I mean by "decomposing", then mention (ever so casually!) that actually we consider only non-trivial decompositions (which is easy to explain and motivate) and, finally, remark on the exclusion of 1 (which is both hard to justify and relatively insignificant at this level).

Question: How can we, at any stage, allow mathematically imprecise statements like my first formulation above?

Answer (indirect): Mathematical rigor, or its human guise "mathematical conscience", is part of the communication "monster" I was talking about at the end of the previous section. At any rate, we can satisfy our mathematical conscience without disrupting communication by saying: "A prime number is, *roughly*, one that..." Indeed, note that "roughly" usually indicates the omission of subtleties (i.e., low-level detail) which are deemed disruptive of good communication at the present stage.

Thinking primitives. When doing, discussing or applying mathematics, mathematicians prefer relatively vague terms like "decompose", "non-trivial", "unique representation" and "for sufficiently large n ", over their more detailed and formal explications. These are in fact the mathematical analogues of the computer science *thinking primitives*, discussed above under the title *Mind over machine*. This preference of mathematicians represents their intuitive choice of the level of abstraction which provides for the best communication and understanding. Indeed, in the sense of suppressing irrelevant detail, the vague, incomplete description of prime numbers is more abstract than its complete, rigorous, formal explication. It is also easier for students to process and make sense of in early learning stages. Finally, it

highlights the context in which primes are important, namely their function as the “atoms” from which all numbers are built.

Unfortunately, the same is not true for the more formal communication channels such as lectures, textbooks and research journals. There, the (level-wise) undifferentiated detailed and formal version – analogous to the primitive expressions of the programming language – reigns. Students, who are mostly unable to extract the higher-level thinking primitives from this kind of presentation, are reduced to working in a level of detail which is quite unsuitable for meaningful mathematical activity – somewhat like programming in machine language.

I emphasize that these remarks should not be taken as suggesting the abandon of rigor or formal detail in mathematics; after all, “low level” does not necessarily mean less important. My point is, rather, that we should be more aware of the hierarchical nature of our mathematical thinking, and make more effort at maintaining abstraction barriers between the various levels in the hierarchy. Again, low-level detail is not ignored, but only temporarily suppressed to be dealt with later.

Another example: Fields. The next example has also come up in a course for prospective high school teachers, while working on an elementary proof that the three classical ruler-and-compass construction problems of the ancient Greeks (trisecting the angle, duplicating the cube and squaring the circle) are, in fact, unsolvable. Here the power and use of abstraction – in both senses – can be naturally and effectively demonstrated. One of the principal tools used in this solution is an extension of the familiar methods from analytical geometry for finding various intersection points of lines and circles. In the usual treatment, all coordinates of points and coefficients of (equations of) lines and circles are assumed to be in \mathbb{R} , the field of real numbers. We may express this by saying that we are doing analytical geometry *over* \mathbb{R} . However, we can apply the same methods for doing analytical geometry over *any* subfield F of \mathbb{R} (e.g., over the rational numbers). Here we consider only points with coordinates in F , and only lines and circles with coefficients in F . This is mathematical abstraction par excellence.

Now in the course of developing this tool, students need to be reminded of the definition of a field. Many have told me of the relief they felt when I had written on the board that “a set of real numbers is called a *field* if it is closed under the four

arithmetical operation”. Can it be that simple? Didn't the “official” definition of a field usually involve some monstrous list of “postulates”? However, this is not the main point here. The main point is that my definition is, like that of a prime number, intentionally left both vague and incomplete. In fact, these omissions, are exactly what makes it so easy to remember and understand. It is vague because I've used the thinking primitive “closed under the four operation”, and it is incomplete because I have “suppressed” some low-level, relatively trivial requirements (e.g., the set must have at least two distinct elements). This is very much in the spirit of abstraction in computer science. Of course, later we would explicate “closed” and “the four operations”, and fill in the missing details. The emphasis here is, again, on maintaining an abstraction barrier that shields the top-level notion of a field from the host of details that threaten to obscure its basic simplicity.

Groups again. Next are two examples from higher mathematics: group theory and calculus. Similar examples can be given in practically all branches of mathematics, which hints at the feasibility of restructuring the teaching of undergraduate mathematics around better “thinking primitives” than we have at present.

I have often noticed, through observation of student difficulties as well as reflection on my own thinking, that understanding and solving problems about quotient groups G/N is greatly facilitated by thinking of them as the result of an identification process, the “kernel” of which (i.e. the elements identified with the unit element) is the normal subgroup N . This is in fact the essence of the so-called “fundamental homomorphism theorem” (cf. e.g., Theorem 2.7.1 in Herstein, 1975). Conversely, I have noticed that understanding and problem solving is often hindered by thinking of the quotient group in standard terms, that is, through the definition in terms of cosets. This observation is very much in line with the above insights from computer science. Since the “identification” approach is the more abstract one in that it stresses the essential attributes of the quotient group (i.e., turning congruence into equality) and suppresses “implementation detail”, namely the specific way the quotient group is constructed so as to achieve those properties.

What we have seen in the case of groups is, in fact, quite typical. The more abstract approach often stresses function over structure. But stressing function over structure is precisely what helps in understanding and problem solving, since both deal with what the concept at hand is *good for*. Minsky (1987, pp. 88, 123, 131, 222), who discusses

the function vs. structure distinction extensively, illustrates it with the concept of “chair”. A functional description is “a chair is something used for sitting”; a structural description would specify its parts, its shape, what it is made of, etc. (Actually, for everyday objects like a chair, a satisfactory structural definition cannot generally be given; instead we usually come up with a description of a “typical” chair, and other chairs are recognized by their similarity to the prototypical chair.) Initially, a functional description may be more useful, but eventually, the best understanding of a concept comes from knowing both its function and structure and, most importantly, the way its structure helps to serve its function.

Calculus. The second example from higher mathematics has to do with how we conceptualize “nearness” in discussing convergence, continuity and the like in the differential calculus. Here again the standard epsilon-delta formalism, actually the “implementation details” of the important concepts of the calculus, tends to dominate students' thinking, leading them to conceptualize in too low a level of detail. In my experience, understanding and problem solving are enhanced when one erects an abstraction barrier to hide those implementation details, using instead higher-level notions like “neighborhoods”, “as close as we wish” and probably also the Robinson-Keisler modern treatment of infinitesimals (cf. Harnik, 1986). For example, I have often asked my students to decide (and prove their answer) whether the following statement is true or false: *Between any two real numbers there exists a rational number.* Many students experience great difficulties trying to solve this problem with the aid of inequalities and the epsilon-delta formalism. However, this question becomes very easy when conceptualized in terms of the more “abstract” (in computer science sense) relationship: *You can find rational numbers as close as you wish to any given real number.* Once solved in those terms, one can attend (if needed) to the low-level implementation of “as close as you wish” in terms of epsilon-delta. As before, maintaining this abstraction barrier helps keep our problem solving at the appropriate level, shielding us from the intrusion of low-level detail.

Status. In conclusion, it is interesting to contrast the positions that the topic of abstraction occupies within the two disciplines, mathematics and computer science. Though everybody agrees that mathematics deals with abstractions, the topic is not normally considered as part of the mathematical subject-matter proper. In fact, you cannot read about it in mathematical textbooks (even ones with titles such as *abstract*

algebra...); instead, you'll have to search for it in books on the philosophy, psychology or history of mathematics or in popular books on the nature of mathematics. The opposite is true in (modern) computer science, where abstraction is a methodology which is explicitly discussed, taught and practiced. To substantiate the above claim, I have even designed a little “armchair experiment”, in which I searched the index section of many mathematical textbooks and confirmed that *abstract* or *abstraction* are practically nonexistent there. In contrast, just take a glance at the index (or even the table of contents) of Abelson & Sussman and you'll find *abstraction* everywhere.

Acknowledgement: I am grateful to the following friends and colleagues for helping in the preparation of this talk: Hal Abelson, Brian Harvey, Nira Krumholz, Tami Lapidot, David Pimm, Dick Tahta, Rina Zazkis.

References

- Abelson, H. and Sussman, G., 1985. *Structure and Interpretation of computer programs*, MIT Press (2nd edition 1996).
- Harnik, V., 1986. ‘Infinitesimals from Leibnitz to Robinson: Time to bring them back to school’, *The Mathematical Intelligencer*, Vol. 8 No. 2.
- Herstein, I.N., 1975. *Topics in algebra* (2nd edition), Wiley.
- Mac Lane, S., 1986. *Mathematics: Form and function*, New York: Springer-Verlag.
- Mendelson, P., 1986. *The “computer programming objects” project: From prescription to description of geometrical figures*, Proceedings of the second international conference for Logo and mathematics education, University of London Institute of Education.
- Minsky, M., 1987. *The society of mind*, Simon and Schuster.