Programming notations. There is no escape.

Liesbeth de Mol

Programming notations. There is no escape.

Text prepared for a talk at the NewCrafts conference in Paris, 25-26 may 2023.
This work is based on a collaboratively written chapter for a forthcoming book "What is a computer program?", written by the PROGRAMme collective.[1]

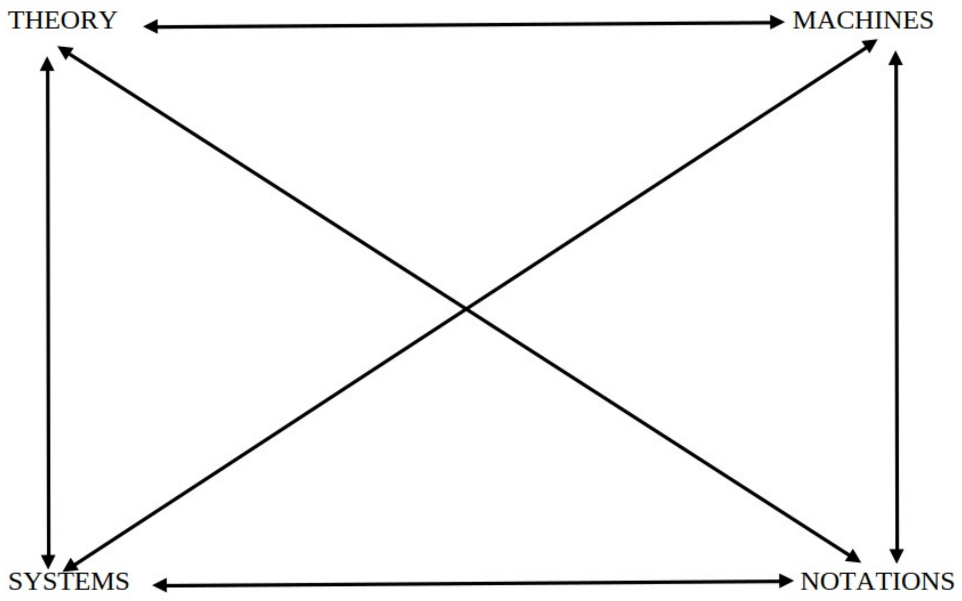
Prepared and presented by Liesbeth De Mol.

PROGRAMme is an open collective of people with diverse backgrounds, convinced that we need a more profound reflection on the programming field at large, without commitment to any dogmatic perspective and this under the banner of anti-disciplinarity. Our practice is to challenge one another in an environment in which the aim is not to undermine or oppose, but to question in order to reach new insights about programs. From this perspective, PROGRAMme stands for the idea of *re-programming* oneself, facing our own disciplinary obstacles, with the help of others, all considered on equal footing.

One of the methodological axes of our forthcoming book is the proposal to view programs as an open and diverse totality. That is, a program cannot be understood through one perspective only. We say that programs have different modalities. These modalities determine the main chapters of our book: in the sense that programs can be written down in a notation, they *are* textual; to the extent that programs can be handled as abstract objects, they *are* formal; in the sense that programs are stored and executed on a computing device, programs *are* physical; and to the extent that programs are made, shared and used in our human world, they *are* socio-technical.

The following diagram prompts us to explore the dynamics and evolving relations between the different modalities.

1 Active contributors to the chapter on which this text is based are: Troy Astarte, Maarten Bullynck, Felice Cardone, Martin Carlé, Liesbeth De Mol, Marie-José Durand-Richard, Tomas Petricek, Mark Priestley, Henri Stephanou. The chapter was repeatedly discussed within the PROGRAMme collective. Active members include: Nicola Angius, Troy Astarte, Maarten Bullynck, Selmer Bringsjord, Felice Cardone, Martin Carlé, Edgar Daylight, Liesbeth De Mol, Marie-José Durand-Richard, Endy Hammache, Simone Martini, Baptiste Mélès, Elisabetta Mori, Pierre Mounier-Kuhn, Alberto Naibo, Mael Pégny, Tomas Petricek, Mark Priestley, Giuseppe Primiero, Julian Rohrhuber, Henri Stephanou, Ray Turner, Nick Wiggershaus.



As can be seen from that diagram, each node, representing one of the modalities is in a relation to each of the other nodes. Because of the dynamics that develops between the different modalities, the meaning of programs and their relations to humans may constantly shift and cannot be precisely located. As such, the diagram does not achieve a closure and, therefore, does not provide a static structure in which those meanings can be fixed once and for all. Instead, it sketches a space in which the open and diverse totality of programs can come to the fore, escaping any attempt that tries to reduce the question "What is a computer program?" to conventional answers.

Let me now turn to the presentation of the chapter on the notational modality. The main argument I would like to make is that there is no escape to the multiplicity of programming notations, unless we want to give up on “real” programming and accept the losses that would come with such an option. In order to make that argument we need to have a firmer grip on what it means to program. Is pressing a sequence of buttons to control a floor turtle programming? How about making a macro in some spreadsheet package? Or what if we could just use natural language to specify our desired program?

To answer these questions we need to be a bit more liberal about our notion of a programming notation. A traditional view is that notations are a textual interface between a programmer, the expert-user, and a machine. One then writes programs in some programming language like Python or Rust. But such view cannot help us if we want to differentiate between a user programming through a notation or one who is merely interacting.

To tackle that problem, we develop the notion of notational programmability. It allows us to make a continuous and finer-grained differentiation between different notations, including the less conventional ones. Central to that notion is that a notation’s programmability is deeply connected to its ability to express new so-called operational meanings which are enabled through a notation that results from a process of negociating between a number of different trade-offs such as programming efficiency or user-friendliness.

The argument is developed in two parts. First, I establish the notion of notational programmability which will require me to jump back a couple of centuries and then move forward to the 1940s. This allows me to introduce the metaphor of Pandora’s box of notations to support, in the second part of the talk, that any succesful attempt at trying to stop the never-ending need for new notations to solve all kinds of problems – would be the true curse.

In order to establish the notion of notational programmability, we need, first of all, a clearer idea of the meaning of operational meaning. But how can we speak of meaning with regards to notations which, from a certain viewpoint need to be without any meaning. That is, a notation that enables so-called blind computations? Here we need to go back to a passage in the history of mathematics and logic in which the question of meaning went hand in hand with the idea of developing a notation without any meaning. That passage is sometimes viewed as the historical foundation of computer science and it is by opposing that viewpoint, offering another reading of that passage, that we can recover our notion of operational meaning.

The idea of a notation without meaning was fully developed by the english algebraists of the early 19th century. In earlier practices, one relied on a notion of meaning that was anchored in properties of numerical quantities. I give a simplified example. If we have 7-3 then the meaning of that term clearly is 4, but he operation 3-7 was then considered illegitimate since its meaning would be an impossible quantity. When we then turn to the algebraic notation a – b, it becomes impossible to distinguish between legitimate and illegitimate operations. The scope of the notation seemed to exceed its accepted semantic domain.

In the 17th century, despite these problems, the successful practice of a symbolic calculus led to the idea of a "blind knowledge". As the ever more complex manipulation of symbols was no longer only grounded in human intuition alone, one could not immediately "see" or understand the meaning of what was being done. Computation was considered "blind" in the sense that the correctness of the results was no longer guided and controlled by the meaning and interpretation of the symbols used. To develop a science of operations then, the English algebraists of the early 19th century proposed a universal symbolic algebra as the language of symbolical reasoning operating only by *laws of combination* on arbitrary symbols, independent of any concrete meanings. This guaranteed the correctness of the deductive process, not the following of the intuitive meaning of each step in a demonstration. The results did not have to depend on their meanings at all, but only on the laws of combination which were grounded in the laws of thought. But at the same time, the laws could be carried out without reference to meaning at all, making the mechanization of symbolical calculus possible.

This idea of a formal calculus independent of any external meaning, was then pushed to the extreme in the context of early 20th century mathematical logic in the works of people like Emil Post or Haskell Curry. Rougly speaking, their idea was to find the ultimate formalisation of finite symbolical reasoning without *any* reference to meaning at all. Here, the idea of linking up mechanisation with human thinking and its symbolizations reached an ultimate form through what is known as the universal Turing machine. This machine is both a formalisation of everything that can be computed but, at the same time, also imposes a limit on what *can* be computed. In a sense, it is an ultimate formalization of the very idea of blind calculus.

If we admit that any computation can be formalized in this manner, one might be tempted to believe that these formalizations are the perfect notation to instruct non-human computers. But this viewpoint can only be a myth. Indeed, if that would be true, why then are we not using the Turing notation as a notation for instructing the machines that are supposed to be the materializations of those very same formalizations? The answer is that we need to make a basic distinction between programmabiity and computability.

To do that, we need to understand how we can recover meaning from meaningless notations. The fact that "meaningless" notations enabled mechanized symbol manipulations does not imply that they cannot contribute to the generation of meaning. On the contrary, if suitably chosen, they have the ability to *create new meanings* as mediators that *transfer* certain properties from one domain to another in an open-ended network of *operational analogies* across different semantic domains.
The observation that notations are instrumental to establish new meanings was supported by a longer tradition of algebraic practices. To give a highly simplified example, these practices enable to establish a connection between, on the one hand, the series that you can see at the top of the slide and which concerns the domain of multiplication and, on the other, the series 1, 2, 3, 4, etc by introducing a new notation: a to the n-th power. By establishing that connection, it is possible to compute also with powers and, in generalizing them, with functions over integers and with functions as such.

The notation thus becomes the instrument that enables to establish certain analogies to become operational, they accompany a transfer of properties. We say that the notation has an ability to express a new operational meaning. It is one way of recovering meaning from meaningless notations.

We can then also apply this to Turing’s notation for the universal machine which established an operational analogy between data and operations, allowing to manipulate operations as if they were just numbers. By consequence, the manipulation of symbols gets as "blind" as it can get. And the operational meanings one could express in that manner becomes nearly unlimited in the sense that it could carry out *any* kind of computational operation.

Within the realm of pure symbol manipulation the operationalization of symbolizations, becomes the basic key to an understanding of how notations for blind symbol manipulation can gain a new meaning *through* their use. But that "meaning" is found in the ideal world of meta-mathematics.

As we argue here, this does not suffice for our notion of notational programmability. Once "true" programming happens -- that is, when we move to real machines -- then the purpose and concern of employing notations as well as the effect of developing and deploying them flips. Rather than getting as 'blind' as possible, the concerns are that programs *will* halt and the purpose is to gain insight and control of the machine's behavior and to be comprehensible by and to the user. The program, by means of its notation, relates the user to the effectiveness of the computation set to work. By exploring this relation of the human and the machine as enabled by a notation, new meaning is created as it operationally connects different domains in much the same way as introduced by the principle of transfer mentioned. The effect, however, namely to act on the *real* world or, to impose new laws on things, is fundamentally different. It is here where we can start to see why such things as the universal Turing machine can never suffice as a notation to program *real* machines to serve *real* humans.

The fundamental difference between human computors and the machines that were originally built to mimick them requires, a different approach. The analogies that need to be made between what is desired at the human end and what is possible at the machine's end requires a careful negotiation through the creation or use of a programming notation.The new meanings that can be created in that manner can neither be reduced to pure machine execution – the materialization of blind calculus, if you want -- nor to "pure thought". They are an effect of the continued interaction between the human and the machine *through* the notation.

Now that you have an idea of our notion of operational meanings, I develop now in more detail our notion of notational programmability by exemplifying it in the context of one of the first computers, the ENIAC. It was around that machine that we locate what we have called the initial surprise of programming and the problems it resulted in.

Without going too much into the details, the ENIAC was built in a war context in which the work of human computers and analog machines was not quick enough to keep up with the demands of the military. It was in that setting that Eckert and Mauchly proposed to build a high speed electronic computer. The result would be the ENIAC.

The reality of high speed computation made it necessary for program execution to become *fully* automatic. Human intervention during computation simply does not ake any sense in a high speed context.

But this fully automated high-speed computation became an unexpected source of errors, namely the failure of the human programmers to accurately anticipate the effects of the instructions given. An early example is given by Douglas Hartree for an ENIAC problem. In not having anticipated that a certain quantity might become negative during computation, the program created an error. On that basis, Hartree concluded that there is a fundamental distinction to be made between human and machine computation. A human computer following Hartree's computational plan "*faced with this unforeseen situation, would have exercised intelligence, almost automatically and unconsciously*". ENIAC, on the other hand, had a purely mechanical “understanding” of Hartree's instructions and continued to compute "blindly". This initial surprise of programming for Hartree was clearly anchored in the consequences of having to pro-gram a high speed automatic machine. It turned programming into a problem for and by itself.

The surprise is analyzed here as arising from the interplay of two different “meanings” latent in a program, that intended by humans and that developed by machines. The gap between these two was to be negotiated by creating a new meaning through a notation that could be "shared" by both and was anchored in making desired analogies operational. This recurring situation motivates our notion of notational programmability.

ENIAC's original programming interface was controversial and the machine was even described by its operators as a "son of bitch to program". In electro-mechanical machines such as the Harvard Mark I instructions were coded, expressed in a machine-readable medium. The coded program thus formed a *notational interface* between human and machine. The operational meanings that are created in this manner are expressed through the notation, as it ties together these two sides.

ENIAC took quite a different approach. Individual instructions, corresponding to the basic operations of ENIAC's many different functional modules were set up on switches. The sequencing of operations was defined by physically connecting these switches to a program bus rather than by reading a sequence of instructions. By consequence, ENIAC was a highly parallel machine. ENIAC programmers used various forms of notation to help design programs, but the end result was not a list of instructions, but rather a plugging diagram that showed how ENIAC should be physically set up to run the program.

The physical process of setting up a program was laborious and time-consuming implying a fundamental incompatibility between the time it took to set-up a program compared to the high speed of the computation itself. The ENIAC team fully recognized this problem, and had been thinking about a successor machine in which this programming problem would be solved by storing instructions in a fast memory from which they could be extracted and decoded at a rate compatible with the machine's electronic speed of computation.

The canonical presentation of these ideas was given in the EDVAC design and became a basic feature of many later architectures. The first time instructions were made available at electronic speeds was in early 1948 when an EDVAC-style instruction code, known as the "converter code", was provided for ENIAC. We view this episode as an extended and sometimes controversial negotiation in which one of ENIAC's basic attributes, its parallelism and extremely fast speed of computation, was compromised in favour of providing the machine with a notational interface and the gains that came with it. The implementation of the converter code reduced the speed of computation by at least an order of magnitude, but even with this loss, ENIAC was still much faster than its competitors.

The new approach enabled novel programming techniques such as closed subroutines. That is, a subroutine that can be accessed from different points in the main program. Moreover, the trade-off in favor of the convertor code made the process of programming and using ENIAC as a whole more "economical".

The ENIAC conversion and the trade-offs that came with it, showcase the significance and consequences of notational programmability: the choice for a particular notational system always comes with a carefully negotiated trade-off between a number of different factors. These factors imply that the programmability of a particular notational system is always contextual: the gains one might have by the decision to use a certain notation, always need to be carefully weighed against what might be lost.

This need to contextualize is clear from the ENIAC conversion itself. As well as compromising ENIAC's speed, the switch to a notational interface for programming restricted its computational flexibility to some users of the machine. In its first life, ENIAC’s parallelism was exploited for number-theoretical computations by Derrick H. Lehmer. From his perspective, the conversion was not an improvement but "spoiled" the ENIAC and, in particular, its parallelism. To put it differently, from the numbertheoretic perspective of Lehmer, there was no new operational meaning to be found in the serialization of instructions. It was a loss.

The initial surprise of programming was strongly grounded in problems resulting from a combination of full automation with high-speed computation. It led pioneers like von Neumann or Hartree to the conclusion that programming required a panoptic foresight in which everything must be foreseen. This not only required high planning but also an appropriate approach to programming itself. A key realization was that programming was not a static process of translation but of providing a dynamic background for the automatic evolution of a meaning. The reason for this is that the initial sequence of instructions in the machine code does not represent the actual order of actions upon execution nor the actual instructions used, since these might change as the computation develops. In other words, it is because there is no one-to-one correspondence between the sequence of instructions and the actually executed sequence of actions that the relation between what we want the machine to do and what it actually does, becomes a dynamical one.

It is here then that we situate the original problem of expressing operational meanings in the context of notational programmability: the fact that a notated program, on the one hand, expresses that which is intended by the human user of the notation and, on the other, needs to be such that it also provides the required "dynamic background" for the machine to develop its meaning, implies that the notation must carry with it meanings that can be expressed, developed and understood by both in their own "interpretational" systems. The ability of a notation to express such meanings feeds directly into notational programmability: as explained before, the choice for or the development of a notational system, also involves a possible loss and gain in terms of operational meaning.

Applying this to the case of the ENIAC order code, it becomes clear that this notation, while a good match to the machine's internal system was not the most suitable for direct use by most human coders. While it enabled new operational meanings around machines like ENIAC, such as the closed subroutine, it was not very "meaningful" from the human perspective and so, "*laid on coders an exceedingly heavy burden of extensive and complex efforts towards understanding, assessing, and reformulating in machine terms a problem that was presented in conventional [...] terms.* In response, two notational techniques appeared in the 1940s. The first is the flow diagram approach as described by Goldstine and von Neumann, which I cannot discuss here, the other is the use of an assembler-like notational system that was developed around the

EDSAC, an early British machine developed by Wilkes and his group. The EDSAC approach was to create a new notational level that was a step further away from pure machine code to a notation that was more amenable to human use. But it could also be "read" by the machine.

I exemplify this here with one notational technique from EDSAC, the so-called interpretive subroutine. These *"enable the programmer to write the whole, or part, of his program in an order code which may be quite different from the basic order code of the machine."* The order codes used in those routines *"do not enter the control circuit of the machine but are extracted from the program, one by one, by the interpretive subroutine, which examines them, and carries out the appropriate operations."* This technique affects notational programmability by providing a notation that can be understood and interpreted by the machine but also relieved the human programmer of a number of complications. In that way they enabled new operational meanings, such as the notation for computation with complex numbers or indirect addressing.

But, as before, switching to other notational systems always comes with a number of trade-offs. In this case, the gain in speed on the programming side resulted in a decreased efficiency of program execution. Because of their computational costs, interpretive routines were *"of real value only when applied to relatively simple problems in which the total running time is short."* These kind of trade-offs between what might be better for different human users and what can be achieved using a particular machine are a recurring phenomenon in the history of programming notations.

As shown, early attempts at bridging the gap between what the machine can do and what humans want it to do, resulted in particular notational approaches to solve programming problems. This did not resolve the *surprise of programming,* however. Many pioneers soon recognized the numerous specific problems that particular notations had to address. We argue here that this surprise and the challenges it resulted in, never stopped, but continued and evolved over time. This development is driven by local changes in notational programmability: to develop, modify or use notations as the result of a negotiated trade-off between a number of different factors and the consequent potential to express new operational meanings and, possibly, to restrict others. These trade-offs are always contextual and the gains that the use of a notation brings are never absolute. The different factors at play in such trade-offs exemplified in the previous sections can be systematized through a number of core dimensions which *together* feed into notational programmability.

1. *User dimension:* different users with different backgrounds, needs and preferences have different uses in mind when using a notation. This dimension not only relates to the domain of use but also involves such things as convenience, the ability of a notation to be re-used and modified by other humans or the comprehensibility of a notation.

2. Efficiency dimension: different notations might result in a loss of or gain in efficiency at the machine's end (memory and/or speed) and at the user's end.

3. Control dimension: different notations result in different levels of control over the users -- humans and machines -- of the notation. Some notations are better suited as a means to avoid error. Some notations are constructed in such a way that the user's ability to change them is highly restricted.

These three interrelated dimensions *are* historical. Or, to put it differently, the variety and depth of programming problems, one might see them as curses, occurs as a function of a historically developing trade-off between efficiency, use and control. The introduction of or choice for a particular notation is then always a local solution to one or more of such "curses" determined by the particular configuration of the different core dimensions and the weight one assigns to them locally. Clearly, the manner in which these dimensions are weighed and balanced against one another, contribute to a notation's ability to explore and express (shared) operational meanings.

The contextual and historical nature of this negotiating process across these core dimensions and the gains *and* losses that come with it, resulted in an ever increasing number of different notations. A frequently used metaphor in programming circles to refer to this multiplication of notations is that of the Tower of Babel and its common interpretation that God punished humankind by breaking up its unity of language to prevent cooperation. We introduce instead the myth of Pandora's box: because Prometheus stole fire, a symbol of technology, from the Gods to give it to humankind, the Gods punished us by sending Pandora, the curious one, with a box which she should never open. She was too curious though, opened it, and sent all kinds of curses upon us. When seeing what happened, Pandora hastened to close the box and could keep one thing closed up, Hope.

We turn this myth around to use it as a metaphor to argue that what might be seen as curses, the ongoing multiplication of notations, is in fact a blessing and this anchored in notational programmability. That is, there is no escaping of the multiplication of notations unless we give up on our ability to explore new operational meanings and so, as we say here, to give up on real programming. What appears then as a curse is in fact a blessing which we should accept as a gift from the Gods, if I can stick to the metaphor, rather than a punishment. The hope of locking up those apparent curses, would be the true curse, if succesful. It would give the control over the technology back to the Gods.

In the second and much shorter part of this talk, I look at a number of cases from the history of programming which can be seen as attempts to achieve an improved if not ultimate notation anchored in a renegotiation between the different dimensions that result in new paths for exploring operational meanings and closing off others. Of course, I cannot give a full analysis of each of these cases but will instead use these to make the more general argument.

I start with automatic programming, which is, generically speaking, the idea to automate programming itself. This idea has emerged repeatedly throughout the history of programming and takes different forms that correspond to changing expectations of the computer user. But its point of origin is really when, in the late 1940s and earlier 1950s, the use base for computers diversified, and with it, the need for other types of programs *and* more programmers. The need for automatic programming in this context was clear. If a notation was more suitable to the domain in which a program was supposed to be used, then perhaps the specialist programmer would no longer be needed and the “user” could become the programmer.

By the mid 1950s, the idea of automatic programming had become more common and notations were considered an essential aspect of such approaches. Glennie, the author of AUTOCODE for the Manchester Mark I believed that what *"we need to do to make programming and coding easy"* is to *"make coding comprehensible"* which may *"be done only by improving the notation of programming."*

One key insight was that aspects of human coding processes could themselves be programmed, thus, establishing an operational analogy between the human so-called clerical processes and the procedure-like working of a machine, an analogy that was mediated through a notational system that supported this analogy to become operational.

Automatic programming systems made it possible to come up with notations that fitted different purposes. Already in the 1950s, this led to an explosion of notations which was seen as a major problem to be controlled. It was deepened by the fact that these systems were specific to particular machines, making it very hard to share programs across different systems. The idea of a machine-independent and universal notation then became a major hope for closing Pandora's box of notations.

ALGOL and COBOL are two well-known notations that appeared in the 1950s and aimed to be universal for a particular use. But both of these were still quite user-specific. A natural extension of this ideal was a notation that could be *general in its purpose*, one that could be used by *any* user going from novices to systems programmers. This would bring obvious benefits: users need learn only one notation, only one compiler would need to be maintained, and programs could be easily shared. Furthermore, a perfectly universal notation would stop the concerning proliferation of notations.

A classic example of such an ‘ultimate notation’ is IBM’s PL/I which was intended to “meld and displace FORTRAN and COBOL”. But its ambition would prove too big if not unrealistic. The fact that it was also supposed to be approachable by novice users, led to a syntax that was deliberately kept loose. This led to a remarkably complicated notation. It is one of the reasons why it never arrived in the academic community. While it might increase IBM’s *control* over the market, it was detrimental to another aspect of the control dimension: writing error-free programs in PL/I turned out to be a huge challenge and one of the reasons why it was famously called a fatal disease bij Edsger Dijkstra.

Moreover, there was an assumption that people would continue to built and use so-called mainframe computers and in particular mainframes built by IBM. It was not prepared for the microcomputer.

But of course this is just one highly simplified example. It is introduced here as an illustration of a more structural observation: the user, efficiency and control dimensions cannot be frozen in time nor space. The dream of an ultimate notation is always one that is historical and so anchored in current hardwares, current needs, current competencies and current markets. Hence, while possibly ideal for that very particular context, it must fail hugely, unless we freeze history itself. It is a failure that we should celebrate: if we would stick to just one notation, we would remain restricted to the operational meanings that can be expressed in that one notation, we would, essentially, limit and restrict our own thinking.

But what if we instead had a notation that can be adapted to *become* an ideal notation for each particular context? This became a possibility in the context of what we call self-programmable notations. The original idea of automatic programming was to have a notation that is closer to the human programmer, but is still close enough to the machine operationally wise. LISP and Smalltalk are two prominent programming notations that make a further abstraction step to have the notation appear *independent* of the actual machine. It is then no coincidence that both of these were anchored in the idea of personalizing the computer so that *any* user would have their own metamedium to work with and do whatever they want it to do.

LISP, which I cannot discuss here, was still very much focused on users-as-programmers. This was very different in the context of SmallTalk which was anchored in the vision of Alan Kay and Adele Goldberg of the computer as a metamedium, a medium that can be all other media. Their ideal was a system that would work not just for so-called user-programmers but for *any* user, most notably, children. It was in fact at this point that they went against the idea of time-sharing systems of the time and for which LISP had been key:

> The kids [...] are used to finger-paints, water colors, color television, real musical instruments, and records. If the “medium is the message,” then the message of low-bandwidth timesharing is “blah.”

Personal computers for millions of potential users of all ages *"meant that the user interface would have to become a learning environment."* The complexity of the system would need to be reduced *"and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior."* The computer became a means to liberate oneself as a human, a view that contrasts deeply with the vision of the computer as a means to control and automate society.

A crucial idea that enabled such liberation was to equip the computer with a self-adaptable notation that could be used for a variety of computer tasks, including, and this is core, unanticipated ones. This then had the potential of eliminating the strict distinction between a user and a programmer: at the simplest level, one could simply "play around" with buttons or icons and pictures. But it was *always* possible to move up, first to programming within a specific microworld like Turtle graphics and, later, to a the full power of Smalltalk.
The exploration of the Smalltalk environment was made possible by a philosophy that *everything is an object.* Every object can be inspected, manipulated and aggregated to create new objects. In an ideal Smalltalk world, everything is at the same level and has equal access to everything else in the system. But, in that philosophy of objects, there is no longer any direct operational analogy between the programming notation

and the real machine code. The possibilities of a structure of interacting objects which can be seen as a computer in and by themselves, also close up the actual computer on which they are being implemented. It is one of the losses one had to accept in such a framework.

In its ambition to serve every possible user, Smalltalk opened up the possibility of another kind of notation which we shall call concrete notations. Such notations aim for a notation that resembles particular human worlds, restricting the domain of operation to what is already familiar. For a child, it can be a turtle or a sprite moving around, for an electrical engineer it might be predefined circuit components that can be combined and parametrized; for an accountant, it might be spreadsheets as the computerized analog of worksheets. And indeed, based on SmallTalk a number of such environments was developed such as Pygmalion or ThingLab, turning Smalltalk into a kind of interactive programming laboratory.

However, many concrete programming notations, which first appeared as Smalltalk experiments or were inspired by it, were turned into stand-alone systems (think of Scratch). These kind of concrete notations, in being close to the lived world of particular users (or, better, the imagined lived worlds as conceived by the developers) are by definition also limited in their use. In being so close to the familiar they are hard to use if one wants to explore the unfamiliar.

The ideal of personal computing as promoted by Kay and Goldberg, the possibility of creating ones own personal environment through the self-programmable notation, then steadily evolved into another kind of personal computing. It is one that focused *only* on concrete notations for so-called end-users to exclude them from truly personalizing those notations to their own needs by restricting their programmability. It was the time when personalization steadily evolved towards customization through parametrization.

Such concrete notations removed the underlying foundation that made them historically possible -- the self-adaptable metamedium that SmallTalk was. What was considered to be but the first step into the metamedium, the concrete notation, became a main paradigm to, ultimately, restrict the users freedom, locking them out of the system through black boxing and code obfuscations.

Here then we see how we moved from ENIAC, whose original programmability was, for most, low compared to the reworked ENIAC, towards a notational paradigm in which programmability has been heavily restricted under the banner of having something that is supposed to be immeidately intuitive for all. In both situations one can maintain that programming is happening when using the notation, but, in both, the shared operational meanings that can be expressed are heavily restricted by the systems supporting them. The consequences are clear. Just as a focus *only* on the machine heavily restricted most people in their use of the machine, a focus *only* on having something that has to be intuitive for *all*, has steadily resulted in a control structure which seperated end from expert user and an inability of the end-user to do anything else but exploring and exploiting what are, essentially, predefined operational meanings.

If high programmability is what we want, then it no longer suffices to think only in terms of what is familiar in human terms. Self-alientation becomes a necessary pre-condition in order to negotiate meanings across the space between humans and machines. If one refuses to pay that price, that is, if one has to rely on a notational interface that is anchored on mimicry only (either at the machine end or at the human end), then, as we argue here, the space for negotiation shrinks and with it, the ability to create new meanings.

I would now like to return briefly to automatic programming to move towards a conclusion and to talk about the new hype: machine learning. The main idea behind the original work on automatic programming in the 1950s was mostly to automate part of the labour involved when programming directly in machine code. But there was also another ideal underpinning some of these approaches: the idea that one could replace the expensive programmers by regular users.

Since that time, the idea of automating the programmer has re-occurred time and again in the history and industry of programming, think for instance of the relatively recent popularity of low and no coding frameworks focused on the so-called citizen-developer. The result has not been the end of programming but more often than not, the creation of a new division of labor and knowledge between the “real” developers and the others.
Today, yet another pronouncement has been made about the end of programming: there is a hope and a promise that systems like ChatGPT will one day reduce programming to prompt engineering. We are told that the new programmers will only have to specify in human terms what they need, and the system will return to them the desired solution. Besides the many societal risks that these systems bring there are two basic questions to be asked here in terms of our notion of notational programmability.

First of all, there is a question of feasability: given the historicity of notational programmability and the fact that, hopefully, we cannot freeze change in use, control and efficiency, is it rational to think that finally we would have reached the end stage with all programming problems resolved by an AI which is fully dependent on past knowledge?

Secondly, there is a question of desirability. If one day, developers and programmers would be reduced to end-users too, then the true curse of Pandora’s box of notations would be launched upon us. Given that such systems have no real means to explain to us what they throw at us, our ultimate notation migh become what we always knew, our own natural language. But such notation would be unable to provide for the creation of any new shared operational meanings. We would remain locked up in our own language, no longer having a notation that allows to negotiate meanings between two distinct semantic domains, that of the machine and that of the human world.

In the meantime, we would give control not so much to these technologies, which are very capable and could be great tools, but to the businesses that call themselves open but decided to close everything up. Being locked up in our own language, we would no longer have an ability to actually change the system but merely enable and feed it. There is then a basic difference to be drawn in such contexts of automatic programming for a user, whether you are entitled to enact the law or whether you are entitled to change the law. Notational programmability, I hope to have shown to some extent, is oriented towards the latter. If the hope of fully automated programming would ever be fulfilled, then we would give up on that ability. It is then the only curse that should forever remain locked up inside of Pandora's box.

It is my personal conviction that in the current discourse, more work should be done in exploring that path: rather than being impressed and jump on the next bandwagon, or just to panick about the consequences without going to the core of the problem, we should dare to be more outspoken and insist that programmability cannot and should never be given up, to become a human right *for all*. It requires an effort from *all* users, including developers, to become more literate about programs and their histories. Making that effort is the price we must pay if we want to program, rather than being programmed. It is the price we must pay if we want to change the laws.