EWD 1036—On the cruelty of really teaching computing science

Edsger W. Dijkstra
Transcription by Javier Smaldone

Austin, 2 December 1988

The second part of this talk pursues some of the scientific and educational consequences of the assumption that computers represent a radical novelty. In order to give this assumption clear contents, we have to be much more precise as to what we mean in this context by the adjective "radical". We shall do so in the first part of this talk, in which we shall furthermore supply evidence in support of our assumption.

The usual way in which we plan today for tomorrow is in yesterday's vocabulary. We do so, because we try to get away with the concepts we are familiar with and that have acquired their meanings in our past experience. Of course, the words and the concepts don't quite fit because our future differs from our past, but then we stretch them a little bit. Linguists are quite familiar with the phenomenon that the meanings of words evolve over time, but also know that this is a slow and gradual process.

It is the most common way of trying to cope with novelty: by means of metaphors and analogies we try to link the new to the old, the novel to the familiar. Under sufficiently slow and gradual change, it works reasonably well; in the case of a sharp discontinuity, however, the method breaks down: though we may glorify it with the name "common sense", our past experience is no longer relevant, the analogies become too shallow, and the metaphors become more misleading than illuminating. This is the situation that is characteristic for the "radical" novelty.

Coping with radical novelty requires an orthogonal method. One must consider one's own past, the experiences collected, and the habits formed in it as an unfortunate accident of history, and one has to approach the radical novelty with a blank mind, consciously refusing to try to link it with what is already familiar, because the familiar is hopelessly inadequate. One has, with initially a kind of split personality, to come to grips with a radical novelty as

a dissociated topic in its own right. Coming to grips with a radical novelty amounts to creating and learning a new foreign language that can *not* be translated into one's mother tongue. (Any one who has learned quantum mechanics knows what I am talking about.) Needless to say, adjusting to radical novelties is not a very popular activity, for it requires hard work. For the same reason, the radical novelties themselves are unwelcome.

By now, you may well ask why I have paid so much attention to and have spent so much eloquence on such a simple and obvious notion as the radical novelty. My reason is very simple: radical novelties are so disturbing that they tend to be suppressed or ignored, to the extent that even the possibility of their existence in general is more often denied than admitted.

On the historical evidence I shall be short. Carl Friedrich Gauss, the Prince of Mathematicians but also somewhat of a coward, was certainly aware of the fate of Galileo—and could probably have predicted the calumniation of Einstein—when he decided to suppress his discovery of non-Euclidean geometry, thus leaving it to Bolyai and Lobatchewsky to receive the flak. It is probably more illuminating to go a little bit further back, to the Middle Ages. One of its characteristics was that "reasoning by analogy" was rampant; another characteristic was almost total intellectual stagnation, and we now see why the two go together. A reason for mentioning this is to point out that, by developing a keen ear for unwarranted analogies, one can detect a lot of medieval thinking today.

The other thing I can not stress enough is that the fraction of the population for which gradual change seems to be all but the only paradigm of history is very large, probably much larger than you would expect. Certainly when I started to observe it, their number turned out to be much larger than I had expected.

For instance, the vast majority of the mathematical community has never challenged its tacit assumption that doing mathematics will remain very much the same type of mental activity it has always been: new topics will come, flourish, and go as they have done in the past, but, the human brain being what it is, our ways of teaching, learning, and understanding mathematics, of problem solving, and of mathematical discovery will remain pretty much the same. Herbert Robbins clearly states why he rules out a quantum leap in mathematical ability:

> "Nobody is going to run 100 meters in five seconds, no matter how much is invested in training and machines. The same can be said about using the brain. The human mind is no different now from what it was five thousand years ago. And when it comes to mathematics, you must realize that this is the human mind at an

extreme limit of its capacity."

My comment in the margin was "so reduce the use of the brain and calculate!". Using Robbins's own analogy, one could remark that, for going from A to B fast, there could now exist alternatives to running that are orders of magnitude more effective. Robbins flatly refuses to honour any alternative to time-honoured brain usage with the name of "doing mathematics", thus exorcizing the danger of radical novelty by the simple device of adjusting his definitions to his needs: simply by definition, mathematics will continue to be what it used to be. So much for the mathematicians.

*
* *

Let me give you just one more example of the widespread disbelief in the existence of radical novelties and, hence, in the need of learning how to cope with them. It is the prevailing educational practice, for which gradual, almost imperceptible, change seems to be the exclusive paradigm. How many educational texts are not recommended for their appeal to the student's intuition! They constantly try to present everything that could be an exciting novelty as something as familiar as possible. They consciously try to link the new material to what is supposed to be the student's familiar world. It already starts with the teaching of arithmetic. Instead of teaching $2 + 3 = 5$, the hideous arithmetic operator "plus" is carefully disguised by calling it "and", and the little kids are given lots of familiar examples first, with clearly visible such as apples and pears, which are *in*, in contrast to equally countable objects such as percentages and electrons, which are *out*. The same silly tradition is reflected at university level in different introductory calculus courses for the future physicist, architect, or business major, each adorned with examples from the respective fields. The educational dogma seems to be that everything is fine as long as the student does not notice that he is learning something really new; more often than not, the student's impression is indeed correct. I consider the failure of an educational practice to prepare the next generation for the phenomenon of radical novelties a serious shortcoming. [1] So much for education's adoption of the paradigm of gradual change.

The concept of radical novelties is of contemporary significance because, while we are ill-prepared to cope with them, science and technology have now

[1] When King Ferdinand visited the conservative university of Cervera, the Rector proudly reassured the monarch with the words; "Far be from us, Sire, the dangerous novelty of thinking.". Spain's problems in the century that followed justify my characterization of the shortcoming as "serious".

shown themselves expert at inflicting them upon us. Earlier scientific examples are the theory of relativity and quantum mechanics; later technological examples are the atom bomb and the pill. For decades, the former two gave rise to a torrent of religious, philosophical, or otherwise quasi-scientific tracts. We can daily observe the profound inadequacy with which the latter two are approached, be it by our statesmen and religious leaders or by the public at large. So much for the damage done to our peace of mind by radical novelties.

I raised all this because of my contention that automatic computers represent a radical novelty and that only by identifying them as such can we identify all the nonsense, the misconceptions and the mythology that surround them. Closer inspection will reveal that it is even worse, viz. that automatic computers embody not only one radical novelty but two of them.

The first radical novelty is a direct consequence of the raw power of today's computing equipment. We all know how we cope with something big and complex; divide and rule, i.e. we view the whole as a compositum of parts and deal with the parts separately. And if a part is too big, we repeat the procedure. The town is made up from neighbourhoods, which are structured by streets, which contain buildings, which are made from walls and floors, that are built from bricks, etc. eventually down to the elementary particles. And we have all our specialists along the line, from the town planner, via the architect to the solid state physicist and further. Because, in a sense, the whole is "bigger" than its parts, the depth of a hierarchical decomposition is some sort of logarithm of the ratio of the "sizes" of the whole and the ultimate smallest parts. From a bit to a few hundred megabytes, from a microsecond to a half an hour of computing confronts us with completely baffling ratio of 109! The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology. He has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before. Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.

Again, I have to stress this radical novelty because the true believer in gradual change and incremental improvements is unable to see it. For him, an automatic computer is something like the familiar cash register, only somewhat bigger, faster, and more flexible. But the analogy is ridiculously shallow: it is orders of magnitude worse than comparing, as a means of transportation, the supersonic jet plane with a crawling baby, for that speed ratio is only a thousand.

The second radical novelty is that the automatic computer is our first large-scale digital device. We had a few with a noticeable discrete component: I just mentioned the cash register and can add the typewriter with its individual keys: with a single stroke you can type either a Q or a W but, though their keys are next to each other, not a mixture of those two letters. But such mechanisms are the exception, and the vast majority of our mechanisms are viewed as analogue devices whose behaviour is over a large range a continuous function of all parameters involved: if we press the point of the pencil a little bit harder, we get a slightly thicker line, if the violinist slightly misplaces his finger, he plays slightly out of tune. To this I should add that, to the extent that we view ourselves as mechanisms, we view ourselves primarily as analogue devices: if we push a little harder we expect to do a little better. Very often the behaviour is not only a continuous but even a monotonic function: to test whether a hammer suits us over a certain range of nails, we try it out on the smallest and largest nails of the range, and if the outcomes of those two experiments are positive, we are perfectly willing to believe that the hammer will suit us for all nails in between.

It is possible, and even tempting, to view a program as an abstract mechanism, as a device of some sort. To do so, however, is highly dangerous: the analogy is too shallow because a program is, as a mechanism, totally different from all the familiar analogue devices we grew up with. Like all digitally encoded information, it has unavoidably the uncomfortable property that the smallest possible perturbations—i.e. changes of a single bit—can have the most drastic consequences. [2] In the discrete world of computing, there is no meaningful metric in which "small" changes and "small" effects go hand in hand, and there never will be.

This second radical novelty shares the usual fate of all radical novelties: it is denied, because its truth would be too discomforting. I have no idea what this specific denial and disbelief costs the United States, but a million dollars a day seems a modest guess.

Having described—admittedly in the broadest possible terms—the nature of computing's novelties, I shall now provide the evidence that these novelties are, indeed, radical. I shall do so by explaining a number of otherwise strange phenomena as frantic—but, as we now know, doomed—efforts at hiding or denying the frighteningly unfamiliar.

A number of these phenomena have been bundled under the name "Software Engineering". As economics is known as "The Miserable Science", software engineering should be known as "The Doomed Discipline", doomed

[2] For the sake of completness I add that the picture is not essentially changed by the introduction of redundancy or error correction.

because it cannot even approach its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot.".

The popularity of its name is enough to make it suspect. In what we denote as "primitive societies", the superstition that knowing someone's true name gives you magic power over him is not unusual. We are hardly less primitive: why do we persist here in answering the telephone with the most unhelpful "hello" instead of our name?

Nor are we above the equally primitive superstition that we can gain some control over some unknown, malicious demon by calling it by a safe, familiar, and innocent name, such as "engineering". But it is totally symbolic, as one of the US computer manufacturers proved a few years ago when it hired, one night, hundreds of new "software engineers" by the simple device of elevating all its programmers to that exalting rank. So much for that term.

The practice is pervaded by the reassuring illusion that programs are just devices like any others, the only difference admitted being that their manufacture might require a new type of craftsmen, viz. programmers. From there it is only a small step to measuring "programmer productivity" in terms of "number of lines of code produced per month". This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent": the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger.

*
* *

Besides the notion of productivity, also that of quality control continues to be distorted by the reassuring illusion that what works with other devices works with programs as well. It is now two decades since it was pointed out that program testing may convincingly demonstrate the presence of bugs, but can never demonstrate their absence. After quoting this well-publicized remark devoutly, the software engineer returns to the order of the day and continues to refine his testing strategies, just like the alchemist of yore, who continued to refine his chrysocosmic purifications.

Unfathomed misunderstanding is further revealed by the term "software maintenance", as a result of which many people continue to believe that

programs—and even programming languages themselves—are subject to wear and tear. Your car needs maintenance too, doesn't it? Famous is the story of the oil company that believed that its PASCAL programs did not last as long as its FORTRAN programs "because PASCAL was not maintained".

In the same vein I must draw attention to the astonishing readiness with which the suggestion has been accepted that the pains of software production are largely due to a lack of appropriate "programming tools". (The telling "programmer's workbench" was soon to follow.) Again, the shallowness of the underlying analogy is worthy of the Middle Ages. Confrontations with insipid "tools" of the "algorithm-animation" variety has not mellowed my judgement; on the contrary, it has confirmed my initial suspicion that we are primarily dealing with yet another dimension of the snake oil business.

Finally, to correct the possible impression that the inability to face radical novelty is confined to the industrial world, let me offer you an explanation of the—at least American—popularity of Artificial Intelligence. One would expect people to feel threatened by the "giant brains or machines that think". In fact, the frightening computer becomes less frightening if it is used only to simulate a familiar noncomputer. I am sure that this explanation will remain controversial for quite some time, for Artificial Intelligence as mimicking the human mind prefers to view itself as at the front line, whereas my explanation relegates it to the rearguard. (The effort of using machines to mimic the human mind has always struck me as rather silly: I'd rather use them to mimic something better.)

So much for the evidence that the computer's novelties are, indeed, radical.

*
* *

And now comes the second—and hardest—part of my talk: the scientific and educational consequences of the above. The educational consequences are, of course, the hairier ones, so let's postpone their discussion and stay for a while with computing science itself. What is computing? And what is a science of computing about?

Well, when all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations. From our previous observations we should recall that this is a discrete world and, moreover, that both the number of symbols involved and the amount of manipulation performed are many orders of magnitude larger than we can envisage: they totally baffle our imagination and we must therefore not try to imagine them.

But before a computer is ready to perform a class of meaningful manipulations—or calculations, if you prefer—we must write a program. What is a program? Several answers are possible. We can view the program as what turns the general-purpose computer into a special-purpose symbol manipulator, and does so without the need to change a single wire (This was an enormous improvement over machines with problem-dependent wiring panels.) I prefer to describe it the other way round: the program is an abstract symbol manipulator, which can be turned into a concrete one by supplying a computer to it. After all, it is no longer the purpose of programs to instruct our machines; these days, it is the purpose of machines to execute our programs.

So, we have to design abstract symbol manipulators. We all know what they look like: they look like programs or—to use somewhat more general terminology—usually rather elaborate formulae from some formal system. It really helps to view a program as a formula. Firstly, it puts the programmer's task in the proper perspective: he has to derive that formula. Secondly, it explains why the world of mathematics all but ignored the programming challenge: programs were so much longer formulae than it was used to that it did not even recognize them as such. Now back to the programmer's job: he has to derive that formula, he has to derive that program. We know of only one reliable way of doing that, viz. by means of symbol manipulation. And now the circle is closed: we construct our mechanical symbol manipulators by means of human symbol manipulation.

Hence, computing science is—and will always be—concerned with the interplay between mechanized and human symbol manipulation, usually referred to as "computing" and "programming" respectively. An immediate benefit of this insight is that it reveals "automatic programming" as a contradiction in terms. A further benefit is that it gives us a clear indication where to locate computing science on the world map of intellectual disciplines: in the direction of formal mathematics and applied logic, but ultimately far beyond where those are now, for computing science is interested in effective use of formal methods and on a much, much, larger scale than we have witnessed so far. Because no endeavour is respectable these days without a TLA (= Three-Letter Acronym), I propose that we adopt for computing science FMI (= Formal Methods Initiative), and, to be on the safe side, we had better follow the shining examples of our leaders and make a Trade Mark of it.

In the long run I expect computing science to transcend its parent disciplines, mathematics and logic, by effectively realizing a significant part of Leibniz's Dream of providing symbolic calculation as an alternative to human reasoning. (Please note the difference between "mimicking" and "providing an alternative to": alternatives are allowed to be better.)

Needless to say, this vision of what computing science is about is not uni-

versally applauded. On the contrary, it has met widespread—and sometimes even violent—opposition from all sorts of directions. I mention as examples

(0) the mathematical guild, which would rather continue to believe that the Dream of Leibniz is an unrealistic illusion

(1) the business community, which, having been sold to the idea that computers would make life easier, is mentally unprepared to accept that they only solve the easier problems at the price of creating much harder ones

(2) the subculture of the compulsive programmer, whose ethics prescribe that one silly idea and a month of frantic coding should suffice to make him a life-long millionaire

(3) computer engineering, which would rather continue to act as if it is all only a matter of higher bit rates and more flops per second

(4) the military, who are now totally absorbed in the business of using computers to mutate billion-dollar budgets into the illusion of automatic safety

(5) all soft sciences for which computing now acts as some sort of interdisciplinary haven

(6) the educational business that feels that, if it has to teach formal mathematics to CS students, it may as well close its schools.

And with this sixth example I have reached, imperceptibly but also alas unavoidably, the most hairy part of this talk: educational consequences.

*
* *

The problem with educational policy is that it is hardly influenced by scientific considerations derived from the topics taught, and almost entirely determined by extra-scientific circumstances such as the combined expectations of the students, their parents and their future employers, and the prevailing view of the role of the university: is the stress on training its graduates for today's entry-level jobs or to providing its alumni with the intellectual bagage and attitudes that will last them another 50 years? Do we grudgingly grant the abstract sciences only a far-away corner on campus, or do we recognize them as the indispensable motor of the high-technology industry? Even if we do

the latter, do we recognize a high-technology industry as such if its technology primarily belongs to formal mathematics? Do the universities provide for society the intellectual leadership it needs or only the training it asks for?

Traditional academic rhetoric is perfectly willing to give to these questions the reassuring answers, but I don't believe them. By way of illustration of my doubts, in a recent article on "Who Rules Canada?", David H. Flaherty bluntly states "Moreover, the business elite dismisses traditional academics and intellectuals as largely irrelevant and powerless.".

So, if I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see the depressing picture of "Business as usual". The universities will continue to lack the courage to teach hard science, they will continue to misguide the students, and each next stage of infantilization of the curriculum will be hailed as educational progress.

I now have had my foggy crystal ball for quite a long time. Its predictions are invariably gloomy and usually correct, but I am quite used to that and they won't keep me from giving you a few suggestions, even if it is merely an exercise in futility whose only effect is to make you feel guilty.

We could, for instance, begin with cleaning up our language by no longer calling a bug a bug but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, viz. with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer's own creation. The nice thing of this simple change of vocabulary is that it has such a profound effect: while, before, a program with only one bug used to be "almost correct", afterwards a program with an error is just "wrong"(because in error).

My next linguistical suggestion is more rigorous. It is to fight the "if-this-guy-wants-to-talk-to-that-guy" syndrome: *never* refer to parts of programs or pieces of equipment in an anthropomorphic terminology, nor allow your students to do so. This linguistical improvement is much harder to implement than you might think, and your department might consider the introduction of fines for violations, say a quarter for undergraduates, two quarters for graduate students, and five dollars for faculty members: by the end of the first semester of the new regime, you will have collected enough money for two scholarships.

The reason for this last suggestion is that the anthropomorphic metaphor—for whose introduction we can blame John von Neumann—is an enormous handicap for every computing community that has adopted it. I have now encountered programs wanting things, knowing things, expecting things, believing things, etc., and each time that gave rise to avoidable confusions. The analogy that underlies this personification is so shallow that it is not

only misleading but also paralyzing.

It is misleading in the sense that it suggests that we can adequately cope with the unfamiliar discrete in terms of the familiar continuous, i.e. ourselves, quod non. It is paralyzing in the sense that, because persons exist and act *in time*, its adoption effectively prevents a departure from operational semantics and thus forces people to think about programs in terms of computational behaviours, based on an underlying computational model. This is bad, because operational reasoning is a tremendous waste of mental effort.

Let me explain to you the nature of that tremendous waste, and allow me to try to convince you that the term "tremendous waste of mental effort" is not an exaggeration. For a short while, I shall get highly technical, but don't get frightened: it is the type of mathematics that one can do with one's hands in one's pockets. The point to get across is that if we have to demonstrate something about *all* the elements of a large set, it is hopelessly inefficient to deal with all the elements of the set individually: the efficient argument does not refer to individual elements at all and is carried out in terms of the set's definition.

Consider the plane figure Q, defined as the 8 by 8 square from which, at two opposite corners, two 1 by 1 squares have been removed. The area of Q is 62, which equals the combined area of 31 dominos of 1 by 2. The theorem is that the figure Q cannot be covered by 31 of such dominos.

Another way of stating the theorem is that if you start with squared paper and begin covering this by placing each next domino on two new adjacent squares, no placement of 31 dominos will yield the figure Q.

So, a possible way of proving the theorem is by generating all possible placements of dominos and verifying for each placement that it does not yield the figure Q: a tremendously laborious job.

The simple argument, however is as follows. Colour the squares of the squared paper as on a chess board. Each domino, covering two adjacent squares, covers 1 white and 1 black square, and, hence, each placement covers as many white squares as it covers black squares. In the figure Q, however, the number of white squares and the number of black squares differ by 2—opposite corners lying on the same diagonal—and hence no placement of dominos yields figure Q.

Not only is the above simple argument many orders of magnitude shorter than the exhaustive investigation of the possible placements of 31 dominos, it is also essentially more powerful, for it covers the generalization of Q by replacing the original 8 by 8 square by *any* rectangle with sides of even length. The number of such rectangles being infinite, the former method of exhaustive exploration is essentially inadequate for proving our generalized theorem.

And this concludes my example. It has been presented because it illustrates

in a nutshell the power of down-to-earth mathematics; needless to say, refusal to exploit this power of down-to-earth mathematics amounts to intellectual and technological suicide. The moral of the story is: deal with all elements of a set by ignoring them and working with the set's definition.

*
* *

Back to programming. The statement that a given program meets a certain specification amounts to a statement about *all* computations that could take place under control of that given program. And since this set of computations is defined by the given program, our recent moral says: deal with all computations possible under control of a given program by ignoring them and working with the program. We must learn to work with program texts while (temporarily) ignoring that they admit the interpretation of executable code.

Another way of saying the same thing is the following one. A programming language, with its formal syntax and with the proof rules that define its semantics, is a formal system for which program execution provides only a model. It is well-known that formal systems should be dealt with in their own right, and not in terms of a specific model. And, again, the corollary is that we should reason about programs without even mentioning their possible "behaviours".

And this concludes my technical excursion into the reason why operational reasoning about programming is "a tremendous waste of mental effort" and why, therefore, in computing science the anthropomorphic metaphor should be banned.

Not everybody understands this sufficiently well. I was recently exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen it was such an obvious case of curriculum infantilization that its author should be cited for "contempt" of the student body", but this was only a minor offense compared with what the visualizations were used for: they were used to display all sorts of features of computations evolving under control of the student's program! The system highlighted precisely what the student has to learn to ignore, it reinforced precisely what the student has to unlearn. Since breaking out of bad habits, rather than acquiring new ones, is the toughest part of learning, we must expect from that system permanent mental damage for most students exposed to it.

Needless to say, that system completely hid the fact that, all by itself, a program is no more than half a conjecture. The other half of the conjec-

ture is the functional specification the program is supposed to satisfy. The programmer's task is to present such complete conjectures as proven theorems.

Before we part, I would like to invite you to consider the following way of doing justice to computing's radical novelty in an introductory programming course.

On the one hand, we teach what looks like the predicate calculus, but we do it very differently from the philosophers. In order to train the novice programmer in the manipulation of uninterpreted formulae, we teach it more as boolean algebra, familiarizing the student with all algebraic properties of the logical connectives. To further sever the links to intuition, we rename the values true, false of the boolean domain as black, white.

On the other hand, we teach a simple, clean, imperative programming language, with a skip and a multiple assignment as basic statements, with a block structure for local variables, the semicolon as operator for statement composition, a nice alternative construct, a nice repetition and, if so desired, a procedure call. To this we add a minimum of data types, say booleans, integers, characters and strings. The essential thing is that, for whatever we introduce, the corresponding semantics is defined by the proof rules that go with it.

*
* *

Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification. While designing proofs and programs hand in hand, the student gets ample opportunity to perfect his manipulative agility with the predicate calculus. Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs. And this concludes the sketch of my proposal for an introductory programming course for freshmen.

This is a serious proposal, and utterly sensible. Its only disadvantage is that it is too radical for many, who, being unable to accept it, are forced to invent a quick reason for dismissing it, no matter how invalid. I'll give you a few quick reasons.

You don't need to take my proposal seriously because it is so ridiculous that I am obviously completely out of touch with the real world. But that kite won't fly, for I know the real world only too well: the problems of the

real world are primarily those you are left with when you refuse to apply their effective solutions. So, let us try again.

You don't need to take my proposal seriously because it is utterly unrealistic to try to teach such material to college freshmen. Wouldn't that be an easy way out? You just postulate that this would be far too difficult. But that kite won't fly either for the postulate has been proven wrong: since the early 80's, such an introductory programming course has successfully been given to hundreds of college freshmen each year. [3] So, let us try again.

Reluctantly admitting that it could perhaps be taught to sufficiently docile students, you yet reject my proposal because such a course would deviate so much from what 18-year old students are used to and expect that inflicting it upon them would be an act of educational irresponsibility: it would only frustrate the students. Needless to say, that kite won't fly either. It is true that the student that has never manipulated uninterpreted formulae quickly realizes that he is confronted with something totally unlike anything he has ever seen before. But fortunately, the rules of manipulation are in this case so few and simple that very soon thereafter he makes the exciting discovery that he is beginning to master the use of a tool that, in all its simplicity, gives him a power that far surpasses his wildest dreams.

Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding. Within a few months, they find their way in a new world with a justified degree of confidence that is radically novel for them; within a few months, their concept of intellectual culture has acquired a radically novel dimension. To my taste and style, that is what education is about. Universities should not be afraid of teaching radical novelties; on the contrary, it is their calling to welcome the opportunity to do so. Their willingness to do so is our main safeguard against dictatorships, be they of the proletariat, of the scientific establishment, or of the corporate elite.

[3] Because, in my experience, saying this once does not suffice, the previous sentence should be repeated at least another two times.