# Educational Matters

**Hans-Jörg Kreowski**

**May 1992**

The history of computer science is short. No wonder that there is not much clarity and agreement among the computer scientists what the substance of the field is. Is computer science a scientific or engineering discipline? Is it a science at all? Or is it a commercial affair with public-relation branches in the universities to give it a serious touch? How can computer science be taught? What is the core of computer science curricula? The debate is going on. And I hope that the *Educational Matters Column* in the EATCS Bulletin, starting in this issue, can be established as an open forum for contributions to the debate.

Today I invite you to a fake panel discussion with Edsger W. Dijkstra and David L. Parnas as panelists. Their answers to my questions are taken from two papers:

> E.W. Dijkstra: *On the cruelty of really teaching computing science,*
> Communications of the ACM 32,12 (1989), 1398-1404.
>
> D.L. Parnas: *Education for computing professionals*,
> IEEE Computer 23,1 (1990), 17-22.

*Ladies and gentlemen, dear Colleagues, I have the pleasure to welcome two of the most prominent computer scientists. Thank you, Edsger Dijkstra, thank you, David Parnas, that you are willing to present your opinions on teaching computer science – or should I better say computing science or even better informatics as many in Europe do? But let me start with the first question: Is computer science a scientific or engineering discipline? What is the core of it?*

**EWD**: "The underlying assumption..... is that computers represent a radical novelty in our history..... Coping with radical novelty..... 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 history 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 *cannot* be translated into one's own mother tongue. (Anyone 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..... 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 misconception, and the mythology that surround them....."

**DLP**: "It has been a quarter century since universities began to establish academic programs in computing science. Graduates of these programs are usually employed by industry and government to build useful objects, often computer programs. Their products control aircraft, automobile components, power plants, and telephone circuits. Their programs keep banking records and assist in the control of air traffic. Software helps engineers design buildings, bridges, trucks, etc. In other words, these nonengineering graduates of CS programs produce useful artifacts; their work is engineering. It is time to ask whether this back door to engineering is in the best interests of the students, their employers, and society..... Each new curriculum proposal includes more "new" computer science and, unavoidably, less "classical" material..... I reject that trend and propose a program whose starting point is programs that were in place when computing science began."

*Although your statements sound quite controversial, you seem to agree that something is wrong with the state of computer science and its academic education. Can you make your views more precise?*

**EWD**: "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 is 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 it 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. First, it puts the programmer's task in the proper perspective: he has to derive that formula. Second, 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 formulae; 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, these days, no computing endeavor is respectable without an acronym, I propose that we adopt for computing science VLSAL (Very Large Scale Application of Logic), and, to be on the safe side, we had better follow the shining examples of our leaders and make a trademark of it....."

**DLP**: "In the early 1960s, those of us who were interested in computing began to press for the establishment of computing science departments. Much to my surprise, there was strong opposition, based in part on the argument that graduates of a program specializing in such a new (and, consequently, shallow) field would not learn the fundamental mathematical and engineering principles that should form its basis..... My colleagues and I argued that computing science was rapidly gaining importance and that computing majors would be able to study the older fields with emphasis on those areas that were relevant to computing. Our intent was to build a program incorporating many mathematics and engineering courses along with a few CS courses. Unfortunately, most departments abandoned such approaches rather early. Both faculty and students were impatient to get to the "good stuff". The fundamentals were compressed into quick, shallow courses that taught only those results deemed immediately relevant to computing theory.....
As I look at CS departments around the world, I am appalled at what my younger colleagues – those with their education in computing science – don't know. Those who work in theoretical computing science seem to lack an appreciation for the simplicity and elegance of mature mathematics..... Further, many of those who work in the more practical areas of computing science seem to lack an appreciation for the routine systematic analysis that is essential to professional engineering. They are attracted to flashy topics that promise revolutionary changes and are impatient with evolutionary developments. They eschew engineering's systematic planning, documentation, and validation. In violation of the most fundamental precepts of engineering design, some "practical" computing scientists advocate that implementors begin programming before the problem is understood. Discussions of documentation and practical testing issues are considered inappropriate in most CS departments....."

*This focuses our discussion to educational matters. If I understand you right, both of you complain about misconception and disorientation. What is the problem?*

**EWD**: "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 is a disguise 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 behaviors, based on an underlying computational model. This is bad because operational reasoning is a tremendous waste of mental effort....."

**DLP**: "The preparation of CS undergraduates is even worse than that of graduate students. CS graduates are very weak on fundamental science; their knowledge of technology is focused on the very narrow areas of programming, programming languages, compilers, and operating systems. Most importantly, they are never exposed to the discipline associated with engineering. They confuse existence proofs with products, toys with useful tools. They accept the bizarre inconsistencies and unpredictable behavior of current tools as normal. They build systems of great complexity without systematic analysis. They don't understand how to design a product to make such analysis possible. Whereas most engineers have had a course in engineering drawing (also known as engineering graphics), few CS graduates have had any introduction to design documentation. Because they lack knowledge of logic and communications concepts, CS graduates use fuzzy words like "knowledge" without the vaguest idea of how to define such a term or distinguish it from older concepts like "data" and "information". They talk of building "reasoning" systems without being able to distinguish reasoning from mechanical deduction or simple search techniques. The use of such fuzzy terms is not merely sloppy wording; it prevents the graduate from doing the systematic analyses made possible by precise definitions....."

*What can be done about the misery in the education of computer science? Where are changes needed?*

**EWD**: "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 "behaviors".....
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 hightlighted 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 conjecture is the functional specification the program is supposed to satisfy. The programmer's task is to present such complete conjectures as proven theorems."

**DLP**: "Most CS departments were formed by multidisciplinary teams comprising mathematicians interested in computing, electrical engineers who had built or used computers, and physicists who had been computer users. Each had favorite topics for inclusion in the educational program, but not everything could be included. So, the set of topics was often the intersection of what the founders knew, not the union. Often, several topics were combined into a single course that forced shallow treatment of each. The research interests of the founding scientists distorted the educational programs. At the time computing became an academic discipline, researchers were preoccupied with language design, language definition, and compiler construction..... Today, it is clear that CS departments were formed too soon. Computing science focuses too heavily on the narrow research interests of its founding fathers. Very little computing science is of such fundamental importance that it should be taught to undergraduates. Most CS programs have replaced fundamental engineering and mathematics with newer material that quickly becomes obsolete. CS programs have become so inbred that the separation between academic computing science and the way computers are

actually used has become too great. CS programs do not provide graduates with the fundamental knowledge needed for long-term professional growth....."

*At the end of our panel discussion I would like to ask you what you recommend to change the situation.What does a computer science course look like that would meet your ideas? What does a curriculum look like that would get your appreciation?*

**EWD**: "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 are 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..... 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....."

**DLP**: "I believe the program proposed below would provide a good education for computing professionals. It is designed to draw heavily on the offerings of other departments, and it emphasizes mature fundamentals to prepare our graduates for a life of learning in a dynamic field. Wherever possible, the courses should be existing courses that can be shared with mathematicians and engineers. Students should meet the strict requirements of engineering schools, and the programs

should be as rigid as those in other engineering disciplines..... (*It follows a detailed proposal for a computer science program that is unfairly cut off here.)*
CS students are burdened by many courses that require hours of struggle with computing systems. Programming assignments include small programs in introductory courses, larger programs in advanced courses, and still-larger projects that comprise the main content of entire courses. This "practical content" is both excessive and inadaquate. Much effort is spent learning the language and fighting the system. A great deal of time is wasted correcting picayune errors while fundamental problems are ignored. "Practical details" consume time better spent on the theoretical or intellectual content of the course..... Properly run cooperative education programs provide the desired transition between academia and employment. Students produce a real product and get feedback from interested users. Review and guidance from faculty advisors is essential to integrate the work experience with the educational program. Project courses can and should be replaced by such a program. The use of the computer in academic courses can be greatly reduced....."

*Thank you very much for your drastic comments and – hopefully – provocative statements. I would be happy if your critique leads to a fruitful discussion how the education of computer science, computing science or informatics can be done better. Clearly, this is more a process than a goal. But the computer science community should turn soon if things go in the wrong direction.*

All of you are invited to contribute to the column *Educational Matters*. Interesting information, position papers, critical and controversial statements, hints, suggestions and ideas should be sent to

Prof. Dr. Hans-Jörg Kreowski
Universität Bremen
Fachbereich Mathematik/Informatik
Postfach 33 04 40
D-2800 Bremen 33
Tel.: ++49-421-218-2956
Fax: ++49-421-218-4322
email: kreo@informatik.uni-bremen.de

I am looking forward to your reactions.