Copyright Notice

On a cultural gap

On the typical university campus, the typical mathematician and the typical computing scientist live in different worlds: they don't know each other or, if they do, they are not on speaking terms. The purpose of this essay is two-fold, viz. to give a historical explanation of this phenomenon and to argue that we should do something about it.

Exhorting the world to mend its ways is always a tricky business, for implicit in the exhortation is always the verdict that the world's ways leave plenty of room for improvement, a suggestion that is always offensive to the touchy. One way of gilding the pill is to qualify one's sentences by all sorts of softeners such as "typical", "in general", "on the average", "usually", "not uncommonly", etc.. For brevity's sake I shall not do so.

At the heart of my historical explanation lies the thesis that when, now four decades ago, the electronic computer was sprung on us, we were not ready for it and that --people and computers being what they are-- widespread confusion was unavoidable.

By far the most common way in which we deal with something new is by trying to relate the novelty to what is familiar from past experience: we think in terms of analogies and metaphors. (Even the 5th Edition of the Concise Oxford Dictionary still defines a typewriter as a "machine for printing characters on paper as substitute for handwriting"!) As long as history evolves along smooth lines, we get away with that technique, but that technique breaks down whenever we are suddenly faced with something so radically different from what we have experienced before that all analogies, being intrinsically too shallow, are more confusing than helpful.

The only feasible way of coming to grips with really radical novelty is orthogonal to the common way of understanding: it consists in consciously trying

not to relate the phenomenon to what is familiar from one's accidental past, but to approach it with a blank mind and to appreciate it for its internal structure. The latter way of understanding is far less popular than the former one as it requires hard thinking (and, as Bertrand Russell has pointed out, "Many people would sooner die than think. In fact they do."). It is beyond the abilities of those --and they form the majority-- for whom continuous evolution is the only paradigm of history: unable to cope with discontinuity, they cannot see it and will deny it when faced with it.

But such radical novelties are precisely the things technology can confront us with. The automatic computer was one of them; from the same era, the atom bomb and --perhaps to a lesser extent-- the pill were two others.

* * *

To make matters worse, a few accidents of history increased during the first two decades the confusion as to what automatic computing was about still further.

When, for instance, computers became available as industrial products, it was a commercial imperative for the budding computer industry to dissociate its products as far as possible from any form of mathematics, the latter being viewed as the pinnacle of "user-unfriendliness". Its sales force accordingly brainwashed the public, including computing scientists, mathematicians, politicians, and managers. (The computer industry even brainwashed its onw management, which to this very day, even if willing to admit that they find themselves in the "high-tech" business, would be horrified to learn that its leading technology is a very mathematical one.)

A confusion of even longer standing came from the fact that the unprepared included the electronic engineers that were supposed to design, build, and maintain the machines. The job was actually beyond the electronic technology of the day, and, as a result, the question of how to get and keep the physical equipment more or less in working condition became in the early days the all-overriding concern. As a result, the topic became --primarily in the USA-- prematurely known as

"computer science" —which, actually, is like referring to surgery as "knife science"— and it was firmly implanted in people's minds that computing science is about machines and their peripheral equipment. Quod non.

We now know that electronic technology has no more to contribute to computing than the physical equipment. We now know that a programmable computer is no more and no less than an extremely handy device for realizing any conceivable mechanism without changing a single wire, and that the core challenge for computing science is hence a conceptual one, viz. what (abstract) mechanisms we can conceive without getting lost in the complexities of our own making.

Remark The above terse summary of computing science's core challenge calls for a slight elaboration. I have tried it out at a number of occasions. Computing colleagues that know only too well from their own experience what I am talking about react with "That's aptly put." or a similar form of agreement. But I have also learned that, no matter how apt, it conveys little to those who have never seen the kind of complexity I am referring to. Let me, therefore, try to sketch its nature.

Blocks are made of buildings, buildings of walls, walls of bricks, bricks of crystals, etc., if you so desire down to the elementary particles consituting the nuclei. That is, we view the whole as composed of parts, which are in some sense "smaller" than the whole, and then apply the process recursively to the parts. We thus arrive at a hierarchical decomposition, the depth of which is some sort of logarithm of the ration between the "sizes" of the whole and of the ultimate parts. But the ratio between an hour (for the whole computation) and several hundred nanoseconds (for an individual instruction) is 10^{10}, a ratio that nowhere else has to be bridged by a single science, discipline, or technology. Compared with the depth of the hierarchy of concepts that are manipulated in programming, traditional mathematics is almost a flat game, mostly played on a few semantic levels, which, moreover, are thoroughly familiar. The great depth of the conceptual hierarchy —in itself a direct consequence of the unprecedented power of the equipment— is one of the reasons why I consider the advent of com-

puters as a sharp discontinuity in our intellectual history. (End of Remark.)

Computers being a radical novelty, it is not amazing that we were not ready for them. The mathematicians, however, were more than unready.

Mathematicians have the social disadvantage of living in an introverted world with very much its own values and its own standards for judging merit and significance, a world which is fertile ground for inbreeding. And that is exactly what happenend. The well-known definition of geometry (viz. "what the geometricians do") can safely be generalized to their conception of mathematics: as none of them had been thinking about automatic computing, automatic computing couldn't be mathematics.

As a subculture, they are not only narrow-minded, they are also very conservative. They know this themselves, but feel that, as heirs and custodians of an impressive tradition, they have indeed a lot to conserve. We all grant them that, but I suspect there is another mechanism at work. Mathematics differs from the other sciences in that the vast majority of its practitioners --including most of the outstanding ones-- are heavily engaged in teaching. Teachers regard the effort spent on instilling habits as their investment and shudder at the thought of having to undo it. Consequently, they equate without hesitation "convenient" with "conventional" and, if it had not been for others, we would still be doing arithmetic in Roman numerals, because that was "easier" since people were used to it.

The mathematician had a technical disadvantage as well, his professional values and prejudices having their roots in the late 19^{th}, early 20^{th} century. He was full of analysis, loved the continuum and the complex plane, and considered infinity as a prerequisite for mathematical depth. How in the world could computing have anything to do with him? If he acknowledged the existence of computers at all, he viewed them as number crunchers, of possible use as a tool for his colleague in numerical analysis --if he had one-- . (For numerical analysis, of which he knew very little, he had at best a mild contempt.) The long and the short

of it is that he disregarded computing completely, strengthened in that attitude when he noticed that machines were primarily used for business administration, which was a trivial pursuit anyhow.

Part of the blame should be put on the early scientists that got involved in computing. They came --of necessity-- from other disciplines: they had mostly been trained as physicist, chemist, or crystallographer (with an occasional astronomer and meteorologist). They were the computer users of the first hour, but regrettably the vast majority of them did not transfer their scientific quality standards to the programming that became their major occupation: as soon as programming is concerned, otherwise respectable scientists suddenly accept to live by the laws of the mathematical jungle. A generation of "scientific" machine users has approached the programming task more as solving a puzzle posed by the machine's manufacturer than as an activity worthy of the techniques of scientific thought. Their logical catch-as-catch-can must have been repulsive to the orderly mind that the mathematician rightly or wrongly regards as his specialty.

Part of the blame for the poor image of computing should also be put on the early Departments of Computer Science, as they were called. They were not very illuminating for the rest of the world as they were very uncertain themselves about the true nature of their calling. Usually these departments were little more than ill-considered cocktails of locally available desciplines (and semi-disciplines) that had some connection with automatic computing: electronic engineering, communication and switching theory, business administration, numerical analysis, numerical control, library science, artificial intelligence and the like, in short, such an incoherent bunch of disciplines that the resulting cocktail hardly appealed to the intellectually discerning palate. These premature departments were concerned with construction or the various possible application areas of computers; their problem was that, to begin with, they had to operate in the periphery of the science that was yet to emerge.

This discipline, which became known as Computing Science, emerged only when

people started to look for what be common to the use of any computer in any application. By this abstraction, computing science immediately and clearly divorced itself from electronic engineering: the computing scientist could not care less about the specific technology that might be used to realize machines, be it electronics, optics, pneumatics, or magic. At the same stroke, computing science separated itself from all the specific problems of embedding computers meaningfully in some segment of some society --concerns that, societies being as different as they are, are almost unavoidably parochial-- .

Let me mention --without any claim to completeness-- some of the highlights that were to shape computing science as it emerged.

In 1960, trans-Atlantic cooperation resulted in the design of a new programming language ALGOL 60. As a vehicle for program design it was a great improvement over its existing competitor FORTRAN and it was immediately accepted as a standard where the latter was not already entrenched. Much more important than the programming language itself, however, was the way in which ALGOL 60 had been defined. The formalism that became known as "Backus-Naur-Form" (or BNF for short) and was used for a complete recursive definition of ALGOL 60's context-free syntax embodied a quantum leap in the rigour of language definition. From that moment, formal grammars were a corner-stone of computing science.

Then LISP emerged as programming vehicle for symbol manipulation. LISP can be viewed as a highly successful exercise in the mechanization of applied logic. For computing science, its cultural significance is that it liberated computing form the narrow connotations of number crunching by including all formal processes --from symbolic differentation and integration to formal theorem proving-- as computational activities.

In the mid-sixties, computing's scope was once more extended with a new dimension, when we learned how to program nondeterministic machines. The origin of the problem came from the task of designing "operating systems" that would en-

able the central processor to cooperate with a possibly large number of coupled but otherwise unsynchronized activities, communication with which would occur at unpredictable moments. The problem was of a humble, technical origin, but it was of great significance for computing's culture. The moments of communication between central processor and peripherals were not only unpredictable, but consequently also irreproducible, and as a result the experimental aoproach to programming was obviously no longer tenable.

It was at this stage that the distinction between pragmatic programming and scientific design began to appear in the world of programming. What is now known as "iterative design" is the paradigm for the pragmatist, who believes that his design will work properly unless faced with evidence to the contrary, upon which he will seek to improve his design. Among programmers, this finding and fixing of malfunctionings is known as "debugging" and the (unjustified) faith in its ultimate convergence was at the time widespread. As a result, the inability of repeating the experiment in the case of an observed malfunctioning (so as to identify its cause) came as a shock. The scientific designer, in contrast to the pragmatist, believes in his design because he knows why it will work properly under all circumstances.

Thus the ground was prepared for what happened in the late sixties and early seventies, when firm foundations were laid for reasoning about algorithms. It has changed the relation between programs and machines: was it formerly the task of the programs to instruct our machines, now it became the task of the machines to execute our programs. It has also changed the status of programs: had they formerly the status of conjectures supported by some experimental evidence, now they could (and sometimes did) attain the status of rigorously proved theorems. (Formerly, books on programming used to be recommended by the author's assertion in the preface that all the programs in his book had been tested on a computer. In the mid-seventies, the first book appeared that was recommended by the author's assertion that he had tested none of the programs in his book.) The fact that we continue to use the same term "program" for both the conjecture and the theorem is regrettable, for it is the source of much confusion.

Computing's last decade is sometimes called the decade of modularity, but that covers only part of the story. With aforementioned firm foundations for reasoning about algorithms we were in principle on familiar mathematical grounds, but only in principle: any ambitious sophisticated program, such as, say, a high-quality compiler, becomes daunting when viewed as a mathematical object. It is huge, in its design many, sometimes conflicting, goals have been pursued, and its final justification often requires many a subtle argument. Some form of "divide-and-conquer" is clearly indicated. But it is not a fixed amount of work that has to be partitioned: an unfortunate choice of parts can greatly increase the amount of work at both sides of their boundary! The original impuls for modularity came from a desire for flexibility, in particular how to subdivide a sizeable program text into "modules", i.e. clearly confined text segments that could be replaced by an alternative without compromising the correctness of the whole program, in very much the same way as we can replace in a mathematical theory the proof of one of its theorems without affecting the theory as a whole. But the emphasis has shifted from such mere replaceability to the question of how to break down the whole task most effectively: the demands are such that elegance is no longer a dispensable luxury, but decides between success and failure.

So much for a bird's eye view of the emergence of computing as an activity worthy and in need of the techniques of scientific thought. As mathematics is unique in the way in which it combines generality, precision, and trustworthiness, it is not surprising that computing science emerged as a discipline of a distinctly mathematical flavour.

* * *

Like all transitions from craft to science, also this one caused tension and anxiety, and was not universally welcomed. Computing's craftsmen felt threatened by it and it has been bitterly resented by many a department of mathematics that had formerly assumed that computing would never distract or divert its best students. But even without that element of competition, some feeling of uneasiness among the mathematicians is understandable, for computing science, the topic they have ignored so completely, might have a profound influence on mathematics in

general. It is indeed expected to do so, and to understand why this expectation is held, we should consider the following.

All through the ages, the spectrum of educational practice has known two extremes. At the one end we have the guilds, in which knowledge is kept as a well-guarded secret, and for that reason is never formulated explicitly; the apprentice joins a master for seven meagre years and absorbs the craft by osmosis, so to speak. At the other end we have the university, where the students listen to the professor, who tries to formulate his knowledge and the key elements of his abilities as explicitly as possible, thus bringing it all out in the public domain. Along this scale, mathematics occupies a curious, double position. While mathematical results are published and taught quite openly --to the extent that many mathematical curricula are very much "knowledge-oriented"-- , how to do mathematics is hardly taught explicitly. Mathematical methodology is not a topic of explicit concern and, as we shall see in a moment, when mathematicians feel threatened, they do so in their capacity of members of their guild. In passing we note that it is not so much that they are unwilling to teach how to do mathematics, but that they are unable to teach it, not knowing how they do it themselves.

Along comes computing science as a mathematical discipline, but in a few important aspects very different from the average one. Firstly it is much less knowledge-oriented; this is probably a consequence of the fact that, while knowledge is always about a specific area, the computer truly deserves the epitheton "general-purpose". Secondly, formal logic, and formal techniques in general, play a much more important role; this is (i) because formal techniques are indispensable to control the type of complexity a programmer has to deal with, (ii) because, by virtue of its mechanical interpretability, any programming language represents a formal system of some sort, and (iii) because symbol manipulation as mechanical processing of uninterpreted formulae comes very natural to the computing scientist. Thirdly, it is a discipline in which methodological questions are a central, explicit concern. The latter two characteristics are, of course, closely related: in a formal argument --whether mechanized or not--

the structure of the argument is given so explicitly that you cannot avoid seeing it. Or, to put it in another way, the reason why today's average mathematician does not know how he does mathematics and hence is unable to teach how to do it, is because he relies so heavily on informal arguments: informality is the hallmark of the mathematical guild member.

In the relation between mathematics and computing science, the latter has been for many years at the receving end, and I have often asked myself if, when, and how computing would ever be able to repay its debt. And slowly, the picture of the answer emerged.

The liberation of logic from the philosophical obligation of mimicking how people are wont to reason has opened the way for the design of effective calculi, thereby greatly extending the range of applicability of formal techniques. Besides being of technical importance, that development is of cultural significance, as arguments made fully explicit in such neutral fashions provide the tangible subject matter of a teachable mathematical methodology: the secret craft of the guild is on the verge of being taken into the public domain.

In a remarkable instance of foresight, this was seen as early as 1967 by John McCarthy (of Stanford University) when he wrote:

> "It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."

The only thing to add is that now, almost two decades later, we dare to go a bit further and dare to include mathematics in general. This is because the analogy between programs and proofs is getting closer and closer. This has been seen on theoretical grounds by the logician Per Martin-Löf and made him write:

> "It [= the creation of high-level languages of a sufficiently clean logical structure] has made programming an activity akin in rigour and beauty to

that of proving mathematical theorems. (This analogy is actually exact in a sense that will become clear later.)"

and also

"In fact, I do not think that the search for high-level programming languages that are more and more satisfactory from a logical point of view can stop short of anything but a language in which (constructive) mathematics can be adequately expressed."

In addition and independently, the analogy between programs and proofs has been convincingly forced upon us by quite practical experiences, which go beyond existence proofs in the form of programs that construct the objects in question. Properties of possibly non-constructively defined objects have been elegantly derived from programs constructing those objects, using the standard techniques of designing and manipulating programs (e.g. showing that a function is its own inverse by manipulating a program computing it until the program is symmetric in its input and output). And, one step further, the techniques of program design have equally successfully been applied to proof design; it is in particular that last experience that strongly suggests that programming methodology and mathematical methodology are not that far apart at all.

Posing as a respectable scientist, I should abstain from crystal gazing in public, but I do see the possibility of a fascinating future in which the quality of the work of the computer scientists and of the mathematicians will be an order of magnitude better than it is now. It is a future in which we don't only agree that mathematical elegance is important but will also teach its conscious pursuit. It is a future in which we don't only agree that a good notation helps, but in which we actually teach how to design notations that are geared to the manipulative needs at hand. It is a future in which programs will display all the beauty of a crisp argument, and in which the dictionaries will no longer define mathematics as the "abstract science of space, number, and quantity" (Concise Oxford Dictionary) but as the "art and science of effective reasoning".

Austin, 13 September 1985

prof.dr.Edsger W.Dijkstra
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712 - 1188
United States of America.

The above text, for which EWD913 was a draft, has been written for "The Mathematical Intelligencer". I have thought about the inclusion of the Game of Stanley Gill (by way of gem):

The Game of Stanley Gill is played with four integer variables x, y, u, and v and two positive integer constants X and Y. The opening position of the game is $x = X$, $y = Y$, $u = X$, and $v = Y$ and consists in playing the following move as often as possible:

if $x < y$, decrease y by x and increase v by u;
if $y < x$, decrease x by y and increase u by v.

The game ends with $x = y$, $(x + y)/2 = \gcd(X,Y)$, and $(u + v)/2 = \mathrm{scm}(X,Y)$, conclusions which follow from the fact that each move maintains:

$0 < x$
$0 < y$
$\gcd(x,y) = \gcd(X,Y)$
$x.v + y.u = 2.X.Y$.

EWD