What Computing Science is about.

When automatic computing equipment began to become available, it quickly became clear that the naive approaches of the day were grossly inadequate for the fruitful exploitation of the new technologies, and Departments of Computing Science were founded in response to that need. They were founded, however, prior to a consensus about what Computing Science as intellectual discipline should encompass. (During its first decade, Computing Science suffered, in fact, from an "identity crisis".)

The discipline was in part shaped by the constraints of academic viability, in particular those of permanence and of coherence.

The requirement of permanence rules out all concerns whose loss of significance is foreseeable in the near future: hence, particular properties of equipment currently on the market could not be allowed to play any role.

Since automatic computing equipment truly deserves the characterization "general purpose", whatever was specific to some application area had to be excluded: for the sake of its own coherence, Computing Science had to focus its attention on the application-independent aspects of automatic computing.

As time went by it became clear that, instead of "automatic computing", the more general term "automatic symbol manipulation" was more appropriate. The act of automation, however, requires programming; programming evolved from a naive activity to the application of formal techniques, and program design itself became more and more a matter of symbol manipulation. Consequently, computing science found itself concerned with the interplay between automatic and human symbol manipulation.

This interplay has profound logical aspects - such as decidability - , but has been shaped in more detail by its quantitative aspects, which in more than one way are overwhelming.

Firstly, there is the question of what we leave to the automaton and what we do ourselves: it may be quite simple to state a desired result as the solution of a mechanically solvable equation, whose mechanical solution, however, would require the storage of more bits of information than there are atoms in the universe. (Also, even a computation that with current technology would take only 1000 years is not very interesting.) In such cases some more human involvement is clearly indicated.

Secondly, if we view programming as the human formal labour done in order to keep the mechanical symbol manipulation within doable proportions, there is the urgent question of how to do our programming with enough efficiency.

There is no doubt that programming, viewed as a mathematical activity, has to be a very (if not totally) formal one. Mathematical theorems intended for human usage are appealed to in a modest frequency and, moreover, by mathematicians, who are wise enough not to appeal to a theorem in a situation in which it obviously does not hold. A programmer's theorem that a given repetition maintains a given invariant, however, is appealed to in high frequency ~ thanks to mechanization easily up to 1000 Herz ~ and, moreover, these appeals take place without a wise mathematician as witness. In this sense, Programming = Mathematics + Murphy's Law. (The latter states "Anything that can go wrong will go wrong.".)

Hence, the required trustworthiness makes it mandatory that programs are derived formally. (I think it can be argued that computing scientists have been the first to make intensive, large-scale use of formal logic as an indispensable tool for their daily reasoning.) Viewed as formal objects, however, programs quickly become of an awesome size, representing big, and at the same time compact deposits of our intellectual labour. Hence the great stress on the efficiency with which the programmer does his part of the symbol manipulation.

Computing science, therefore, finds itself confronted in full force with the problem of the economics of formal reasoning. This can be viewed as a long-standing problem of mathematical methodology, but the point is that in automatic computing it can no longer be ignored: there it becomes the central issue. In addition, the possibility of mechanization adds a totally new dimension to the cost/benefit analysis.

Historical Remark The earlier computer applications were of an obviously repetitive nature: the payroll had to be done each month, and each month the size of the job was linear in the number of employees. What to mechanize was in those days a clear-cut problem. With mechanical proof verification, or a certain amount of mechanized reasoning as component of a general-purpose programming system, the decision what to mechanize is no longer obvious at all. A shift in ambition has caused a shift in perception. (End of Historical Remark.)

A final remark about the rôle to date of (formal) program derivation, since that rôle has been played down by the observation that relatively few new algorithms have been designed — or "discovered", if you are a Platonist — that way. The new ones that were designed were way beyond the conceptual limits inherent in the traditional intuitive approach, but, more importantly, it has completely changed the intellectual status of many a published algorithm, viz. from an incompletely formulated or erroneous conjecture to a crisply stated and proven theorem.

Pasadena, 25 May 1987

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

[Written for Harvey Friedman]