Introducing a course on program design and presentation.

Historically, the nature of the programming challenge has been widely misunderstood and the difficulty of programming has been grossly underestimated.

The first program fed into the first stored-program controlled computer -the EDSAC in Cambridge, England- was supposed to cause the machine to print a table of squares, which was duly printed. The second program, which was for table of prime numbers, was, however, wrong. Consequently, right from the beginning, the academic community had been warned and should know better than to belittle the programming challenge. It did know better and warned, but for decades its warnings have been suppressed and drowned out by the propaganda machines of the computer industry - in particular IBM- and of the Pentagon. For the protection of their interests, these organisations thought it vital to convince the general public that programming was no serious problem, that it required no intelligence and certainly no education. With the appropriate programming languages - such as Cobol, PL/I, and Ada - programming's problems would melt as snow in the

sun.

Under these pressures it has been difficult to get Programming Methodology accepted as a topic of academic research and teaching. The barriers were a little less unsurmountable in those societies in which IBM and the Pentagon had less power; the fact that, in recent developments, both organizations have lost all of their surplus credibility (if not more) should in this respect assist American Departments of Computing Science in catching up.

* * *

This course is given from the position that an academically educated computing scientist should be able to program at least an order of magnitude better than the average programmer or the hacker without formal training. Note that this is a direct consequence of my positions that (i) the programming challenge is of sufficiently high calibre to merit academic concern and that (ii) having absorbed an academic education or not should make a substantial difference. (Note that neither of these positions is necessarily shared by all my academic colleagues.)

There are at least three different ways in

which the term "computer program" is used; since I shall use it in this course only in the last of the three senses, let me list them. They could be called the artistic, the legal, and scientific interpretation.

(i) Artistic. Here the program is an automaton created out of curiosity, with the intention of observing its behaviour when exercised. A program in this sense has the intellectual status of The American Flag – "Let us hoist it and look how many salute." –. People in AI and in Experimental CS have a tendency to consider such "artistic programs" worthy objects of academic concern.

(ii) Legal. Here the user is not supposed really to know what he wants or needs, and the programmer is not supposed really to know what the user wants or needs. Consequently the programmer does not really know what he delivers, nor does the user really know what he gets. Here programs and their documentation have the intellectual status of the texts lawyers compose: of course you cannot expect these texts to be clear, unambiguous and free from contradictions, but if due care has been applied, you cannot complain. The major shortcoming of the legal

interpretation — which is the predominant one in today's software industry — is that it precludes technical improvement of the product. People in Software Engineering - should I say SE ? - tend to adopt the legal interpretation.

(iii) Scientific Here the program is viewed as a statement in a formal language, i.e. as a formula. When the program's specification has been stated in an equally formal language, the question of whether the program meets its specification is then a mathematical question. Here, the program is one of the triple : program, specification, and proof that the program meets the specification. Because such a triple is a sound and solid mathematical edifice, I had expected in my innocence that this view of programs would appeal to the mathematical community, but it does not, the reason being that the required proof style leaves no room for handwaving and that the mathematical guild has developed no appreciation for such a style. We stick to this third interpretation of the term "program": we are heading for programs that are part of the statement of proven theorems.

* * *

The idea of mathematically proving the cor-

rectness of programs is now more than 30 years old. American computing science has tried to realize this in the form of "program verification" -automated or not-: given a program and its specification, prove that the former satisfies the latter, the idea being that the programmer (whom you cannot bother with mathematics) would provide the program, while the mathematician (whom you cannot bother with the tedious trivialities of programming) would provide the correctness proof. This ridiculous division of labour - it is really putting the cart before the horse - condemned the verification project to failure. The proper way of doing it is the other way round; given the specification, one designs the correctness proof and, as that design progresses, derives the program to which the proof is applicable, a method known as "designing proof and program hand in hand". That this is a much more promising way of doing things than the "a posteriori verification" was well-known and understood a quarter of a century ago, but victimized by IBM, the Pentagon, and its failing system of primary and secondary education, the USA has, alas, been unable to hear the message. Be it slowly, things seem to be changing, which could be a consequence of growing internationalization (with my apologies for this appalling noun).

I expect the main ingredients of this semesters course to be the following ones.

- A replacement of the (still!) traditional "operational semantics" by what is known as "postulational semantics", as they provide a compact basis for correctness proofs without exploding case-analyses.
- The essentials of the predicate calculus, which I intend to present as a very simple and general, and thus powerful calculational tool for the actual construction of proofs and programs.
- The application of the above to the design and presentation of (far from trivial) sequential (and if time permits, parallel) programs.
- And, all the time, the critical consideration of decisions taken and conventions adopted.

Austin, 25 August 1993

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