A Curriculum Proposal for Computer Science

There are many signs of a growing awareness of the need to introduce a more analytical, mathematical approach to the teaching of computers and computer programming. These signs appear in reports of adverse effects of programming on young minds [1], in proposals for the introduction of mathematical topics in the early computer science curriculum [2], and in attempts to fix the blame (commonly on the programming languages used) for present deficiencies.

Language is indeed a crucial issue. Conventional mathematical notation, which fosters the development of analytical ability and sets a high standard for the use of language in analysis, is not used in programming, primarily because it is not executable. Commonly used programming languages, on the other hand, discourage the use of analysis in any sense that would be considered significant in mathematics.

The present letter makes specific proposals for changes in the computer science curriculum, and discusses a number of issues in the choice of language. It is based on some long-term experience in the use of an analytic approach (e.g., at Pomona College) and on a recent experimental introduction and assessment of a third-year course in Applied Mathematics for Programmers— referred to below as AMFP—at T.H. Twente, Netherlands.

First-Year Programming Courses

Any change in first-year courses (even such as proposed by Ralston) is difficult to introduce because of its potential effects on the subsequent program. In the case of programming courses this is doubly difficult because: (a) any applications treated in new programming courses may seriously overlap topics in later established courses and (b) later courses may rely on the introduction of specific languages. To meet these objections, we propose:

1. the use of a suitable executable, analytic notation (to be called EAN) in the treatment of topics drawn largely from elementary mathematics. This will introduce the essential notions of executability, precision of expression, checking and revision, recursive definition, etc. It will also provide a review of elementary mathematics welcome to most students, and essential to many. Overlap with later courses should prove slight.
2. that considerable emphasis be placed on the translation of ideas and solutions between (both to and from) EAN and one or more programming languages chosen freely by individual students (according to their present knowledge or anticipated needs), with attention paid to the effects of language on the solutions developed. In particular, EAN may be used as an intermediate language in translating from existing treatments of topics in nonexecutable mathematical notation to execution in any language. Although translation should not be introduced at the very outset, it need not be deferred long; it was used very early in the AMFP course, but could well be deferred to a second or third term.
3. that topics beyond those already encountered in elementary mathematics also be introduced, provided that there is time to treat them and that they support, but do not unduly overlap, later courses. Information retrieval, the use of files, simple formal language translation, and parallel programming are obvious candidates.

Such a first-year course might also prove attractive to mathematics students as an introduction to the use of computers, although for such students it might be made both briefer and mathematically more advanced.

Language Considerations

Language comparisons tend to be distorted by the natural tendency to choose as illustrations those topics for which a given language is known to be convenient. Such bias should be avoided by choosing the topics first, although estimates of the amount that can be covered may have to be revised as assessments of languages progress.

Comparison should include the question of the difficulty of introducing the EAN. Ideally, it should be easy to introduce in context, with little or no explicit discussion of language; that is, in the manner long employed in mathematics.

It is essential to avoid confusion between the capabilities of a language and the style of its use appropriate to a given purpose (in this case, introductory programming). In other words, languages must not be dismissed because they *permit* constructions whose introduction can and should be avoided in a particular context. For example, a language that generalizes the simple power function from positive integer exponents to any exponent must not be dismissed as difficult to learn for people who have no need of the extension. Similarly, "functional" or "applicative" languages (which appear to be largely motivated by the desire to provide better analytic tools) may entirely proscribe the assignment of names to results; however, a language that permits name assignments must not be thereby dismissed, *provided* that it can serve the purpose without introducing name assignment. This "functional style" was largely adopted in the

AMFP course, although not as rigorously as a user of "functional" programming might wish.

Many examples of the treatment of various topics in various languages may be found in the literature. However, in making comparisons, credence should be placed only in examples provided by an acknowledged expert in the language. In particular, one should be wary of published comparisons that provide examples in several languages; the author may not be expert in all of them. Ideally, a set of example problems should be collected, agreed upon, and treated by proponents of various languages. Some attention has already been given to this matter at T.H. Twente.

Although availability of language and its present wide adoption in industry are not directly relevant to its use in teaching programming, these factors are important to the graduating student. They should therefore be given some weight, if only because of their potential impact on the students' enthusiasm for the language forced on them.

G. Blaauw (T.H. Twente)
L. Dickey (University of Waterloo)
A. Duijvestijn (T.H. Twente)
G. Helzer (University of Maryland)
K. Iverson (I.P. Sharp Associates Ltd.)
D. McIntyre (Pomona College)

REFERENCES

1. Hicks, J. Letter to *Micromath*. Journal of the Association of Teachers of Mathematics (Spring 1985), 6.
2. Ralston, A. The first course in computer science needs a mathematics corequisite. *Commun. ACM* 27, 10 (Oct. 1984), 1002–1005.

Core Understanding

I am 180 degrees out of phase with Matley's opinion that CS/CE majors need more "core" courses than the core majors themselves (Forum, June 1985, pp. 565–566; see also August 1985, p. 790).

In developing computer-based systems, the developer must achieve understanding of the user's needs to establish the requirements that lead to systems specifications and ultimately to the system. By being able to speak the user's "language," a developer can achieve this understanding without ambiguity, thus the reason for studying core courses.

When the developer has greater theoretical knowledge of the user's specialty than the user himself, however, the direction of the dialogue during requirements deviation tends to be reversed. The developer stops listening to the user and starts telling the user what is required. This leads to the development of systems that do not meet the user's needs and are not used. Both the developer and user would have been better served if the developer's education had expanded his or her CS/CE skills.

Having been a practitioner for 22 years, since receiving my undergraduate degree, I never knew the "time" when a computing education was limited to a postgraduate experience.

Robert C. Eggleston
10 Gedick Rd.
Burlington, MA 01803

. . . Entrenched (Fortran)

The companies or persons who pay Charles M. Strauss (Forum, July 1985 p. 670) his "well-into-six-figures income" for 100,000-line Fortran programs should recognize that, if he were to become just 10 percent more efficient by using modern programming languages and practices, they might enjoy well-into-five-figures savings!

David V. Moffat
North Carolina State University
Raleigh, NC 27650

Gene Zirkel wonders (Forum, March 1985, p. 239) if we should be teaching Fortran in CS departments, and the June Forum is full of letters on the subject (pp. 567–568). The two preceding pages of letters were all concerned with the "industry-university rift," with some breast-beating and wondering what we can do about it.

Hasn't it occurred to anyone that the two questions are related? If we don't teach Fortran (yes, and Cobol, and all those other venerable languages pioneered by ACMers like Backus and Bemer) how are we ever going to get industry to take us seriously?

I don't particularly *like* Fortran and Cobol; my language of preference is still Algol 60 and I fully concur with Dijkstra that it's a great improvement over all its successors. And I take seriously my membership in SIGPLAN, and hope to see continued advances in languages of all generations. But Fortran and Cobol pay the rent. I think I write clean, well-structured code, even if the vehicle is Fortran and I have to use GOTO's. Conversely, I have seen terrible Pascal code, and some of the Algol in the *Collected Algorithms* is downright incomprehensible, with call-by-name side effects and the like ("Jensen's device" used to be considered quite clever, and that is a technique you cannot possibly apply in Fortran).

Mention of the *Collected Algorithms* brings me to my final point. Why, if we're so hot to get rid of Fortran, has every algorithm this year published in our own TOMS and distributed in 80-character card image records on tape by our algorithms distribution service been written in Fortran? I'd love to submit an Algol procedure to TOMS (I remember when that was the only language acceptable for algorithms published by ACM) but I doubt that they could find any referees who had access to an Algol compiler. I could, of course, translate it into Pascal, but I imagine that if it was a very useful algorithm, everyone who really wanted to use it would grumble because they had to convert it to Fortran themselves. So let's be honest—whatever its drawbacks as a language, Fortran has a lot of practitioners who have "voted with their feet," and we would look extremely silly as an organization trying to shield students from learning the very language that has be-