CMU-CS-83-101

Working Papers on an Undergraduate Computer Science Curriculum

Mary Shaw, editor

Computer Science Department
Carnegie-Mellon University
Pittsburgh PA 15213

1 February 1983

The Curriculum Design Project is supported by general operating funds of the Carnegie-Mellon University Computer Science Department

Table of Contents

Introduction

The Computer Science Department at CMU is in the process of reviewing and redesigning its undergraduate curriculum. As of January 1983, the members of the curriculum design project are Steve Brookes, Marc Donner, James Driscoll, Michael Mauldin, Randy Pausch, Bill Scherlis, Mary Shaw, and Alfred Spector.

Our initial efforts yielded some working papers that may be of interest to a wider community. These working papers are collected in this report. All bibliographic references refer to a single bibliography at the end of the report.

The first paper in the collection is a statement of goals and objectives for an undergraduate computer science curriculum. This paper is, in effect, the charge to the design project [27].

The second paper addresses the ways computer science relies on the mathematics curriculum [26]. It was presented at a conference on mathematics curriculum design.

The third paper discusses the need for mathematics in the undergraduate computer science curriculum and the shortcomings of one "standard" curriculum in this regard [22].

Finally, we present some outlines of the structure of computer science. We extracted these outlines from their sources and put them in a common format for our own use; we now hope that they may be of use to other curriculum designers.

We wish express our appreciation to the various authors, editors, and publishers for permission to reprint the papers that form Parts II and III and for permission to reprint material from the following in Parts III and IV:

> Alan Russell, Richard E. Fairley, and Julius A. Archibald, Jr. Letters to the editor in reply to "Curriculum '78 -- Is Computer Science Really that Unmathematical¿ [22]. *Communications of the ACM*, June 1980. Copyright © 1980, Association for Computing Machinery, Inc., reprinted by permission.
>
> ACM Curriculum Committee on Computer Science. "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science." *Communications of the ACM*, March 1979. Copyright © 1979, Association for Computing Machinery, Inc., reprinted by permission.
>
> ACM Curriculum Committee on Computer Science. "Recommendations for Master's Level Programs in Computer Science." *Communications of the ACM*, March 1981.

Part I: Plan for Developing an Undergraduate Computer Science Curriculum

Mary Shaw, Stephen Brookes, Bill Scherlis,
Alfred Spector, Guy Steele

The CMU Computer Science Department has periodically considered offering an undergraduate degree. Discussions of this subject in 1980-81 made it clear that substantial revision of the existing curriculum would be needed before we could make decisions about degrees. In response to this need we started a project to design a modern undergraduate computer science curriculum without preconceptions based on traditional course organizations. Part I presents the objectives and overall development plan that were formulated at the beginning of the curriculum design project [27].

In Spring of 1981, the Carnegie-Mellon University Computer Science Department expressed its willingness to consider developing a curriculum that could lead to a bachelor's or master's degree in computer science. A Curriculum Design Project has been established for the purpose of developing such a curriculum. This note describes the objectives and development plans of that project.

We know of no existing curriculum design that is suitable for undergraduate or masters'-level computer science programs of the next decade. Current designs neglect fundamental conceptual material in favor of programming techniques, facts about current technologies, and routine skills that are likely to become obsolete in a short time. By developing a strong curriculum that emphasizes underlying principles of the science and problem-solving skills with lasting value, CMU can influence the way computer science is taught throughout the country as well as at CMU.

We believe it is important to separate curriculum concerns from issues of degree programs, so we will begin with the curriculum design and consider degree programs after we understand the curriculum content. The first stage of this curriculum design will therefore address general questions of content, and a second stage will be concerned with organizing this content into individual courses. Assuming that a satisfactory curriculum emerges, we plan to take up the additional problems associated with running a degree program concurrently with the second stage of the curriculum design.

This note explains our view of the current status of undergraduate computer science education, describes the premises and goals of this project, and outlines our current plan.

1. Current Status

Computer Science is a rapidly-developing field, and the current national shortage of computer science personnel at all degree levels is expected to continue into the 1990's [19]. Professionals in a rapidly-growing field are particularly susceptible to technical obsolescence, so it is important that curricula in such fields emphasize fundamental conceptual material that transcends shifts of technology. Unfortunately, many existing computer science curricula fail to do this. Even the bachelor's and master's curricula that have been designed or endorsed by computer science's professional societies [2, 4, 5, 14] have serious deficiencies [22].

CMU offers computer science courses but no degree in computer science as such; the Mathematics department offers a computer science option. The CMU computer science course offerings, which de facto form a curriculum, have not been reviewed as a whole in about a decade. A number of new

courses have been developed during this period, but systematic review has been lacking and content coverage is spotty. Although the Computer Science Department offers nearly enough courses for a major, there is some reluctance within the department to commit other resources, especially human resources, to a degree program.

2. Premises

Certain assumptions about computer science, about education, and about CMU underlie this effort. It will be helpful to make them explicit:

- The major substance of an undergraduate computer science curriculum (as for any subject) should be fundamental conceptual material that transcends current technology and serves as a basis for future growth as well as for understanding current practice. This fundamental material should be reinforced by abundant examples drawn from the best of current practice.
- The CMU Computer Science Department should invest energy in a degree program only if that program is of very high quality -- ranking among the top programs in the country.
- Whether or not the CMU Computer Science Department offers an undergraduate degree, a complete review of the undergraduate curriculum is in order.
- An undergraduate computer science curriculum design should address the entire curriculum, not just the courses offered by the Computer Science Department proper or even just the technical courses related to computer science.

We take as a working hypothesis the proposition that computer science is now mature enough -- has enough intellectual substance -- to warrant an undergraduate or master's-level curriculum and degree program. In this context, the curriculum design process can be thought of as an experiment to test that hypothesis.

3. Goals

Our specific objective is a high-quality computer science curriculum for CMU. This curriculum should also merit national recognition, both for the quality of the students it educates and as an exemplar for curricula at other schools.

Following the Carnegie Plan for education [7, 8, 11, 20], we plan a curriculum through which a student can acquire:

- A thorough and integrated understanding of the fundamental conceptual material of computer science and the ability to apply this knowledge to the formulation and solution of real problems in computer science.
- A genuine competence in the orderly ways of thinking which scientists and engineers

have always used in reaching sound, creative conclusions; with this competence, the student will be able to make decisions in higher professional work and as a citizen.

- An ability to learn independently with scholarly orderliness, so that after graduation the student will be able to grow in wisdom and keep abreast of the changing knowledge and problems of his or her profession and the society in which he or she lives.
- A philosophical outlook, breadth of knowledge, and sense of values which will increase the student's understanding and enjoyment of life and enable each student to recognize and deal effectively with the human, economic, and social aspects of his or her professional problems.
- An ability to communicate ideas to others.

4. Plan

A complete curriculum redesign will proceed in two stages. As much as possible, administrative problems and degree programs will be dealt with separately.

Beginning in late 1981, a small working group began to lay out the overall content of the curriculum by attempting to identify the bodies of knowledge (theories, models, methods, etc.) that have sufficient substance and accessibility to justify their places in the curriculum. This group is also attempting to formulate a coherent view that shows the relations among this material. The content study is being conducted outside the traditional course framework in order to avoid the preconceptions about content and structure that are implicit in any established curriculum. Later this spring, we expect to begin detailed discussions with groups interested in particular aspects of the content.

When the content and its structure are under control, this working group or a successor will define a set of courses[1] that cover this content. The course plan will take into account the particular needs of CMU. In addition to courses, this group will deal with curriculum-related requirements such as prerequisite structure, concentrations, breadth, etc.

The focus of the design will be on a liberal professional education with emphasis on problem-solving skills. Some of the words in the previous sentence are subject to various interpretations. We intend all in a very positive sense. "Liberal" education is broad, including humanities and social science courses plus technical courses outside the student's specialty. Liberal education includes communication skills, both for understanding the work of others and for communicating one's own work. Describing the education as "professional" recognizes the legitimate motivations of many

[1] or other teaching units -- we are not irrevocably committed to the traditional course format

students who value education because they can apply it rather than for pure intellectual enjoyment. "Problem-solving skills" refers to the ability to apply general concepts and methods from a variety of disciplines to all kinds of problems, abstract as well as practical, whose solutions require thought, insight, and creativity. Thus "problems" can range from the proof of a theorem to the design and construction of a specialized computer program and "skills" means creative intellectual ability, not merely the ability to perform repetitive routine actions.

When this design is complete, it will be appropriate to consider what degree(s), if any, should be granted on the basis of the curriculum. For the time being, we believe that the commonality among bachelor's and master's programs and among terminal and nonterminal programs is very strong -- indeed, strong enough that the distinctions are not yet an issue. Administrative requirements such as program size, admission criteria, and resource requirements can be addressed at that time.

Part II: Mathematics Curriculum and the Needs of Computer Science

William L. Scherlis and Mary Shaw

In the summer of 1982, the Sloan Foundation conducted a workshop on the curriculum for the first two years of college mathematics. We were invited to contribute a paper on the relation between computer science and mathematics, especially the support that computer science needs from the mathematics curriculum. Scherlis presented that paper, which appears here as Part II as well as in the conference proceedings [26].

Abstract: Although computer science is not a proper part of mathematics, it nonetheless relies heavily on mathematics for its foundations and its methods. Computer science education must depend on the mathematics curriculum for specific ideas and techniques from discrete mathematics, for an understanding of mathematical modes of thought, and for a genuine appreciation for power of abstraction. This paper is a examination of these needs, intended to initiate discussion of the implementation of appropriate mathematics curriculum.

1. Some Words about Computer Science

Computer science is concerned with the phenomena surrounding computers and computation; it embraces the study of algorithms, the representation and organization of information, the management of complexity, and the relationship between computers and their users. Computer science is like engineering in that it is largely a problem-solving discipline, concerned with the design and construction of systems. But the computer scientist, like the mathematician, must be able to make deliberate use of the intellectual tools of abstraction and of analysis and synthesis. The relationship between computer science and mathematics is very close and has been discussed at length in the literature. Two very interesting examinations of this relationship are [9] and [15].

Computer science is a mathematical discipline --- so much so that the boundary between computer science and mathematics is often quite hard to pin down. While both disciplines are concerned primarily with abstract structures, computer science is not simply a branch of mathematics. It relies on skills, attitudes, and techniques derived from mathematics, but it is concerned not so much with proofs and the existence of structures as it is with algorithms and the design and organization of structures. In this sense computer science is an engineering discipline. Like engineering, it is pragmatic and empirical and is concerned with the selection, evaluation, and comparison of designs for implementation. But in computer science this study is focused on the behavior of systems such as algorithms, computer organizations, and data representations --- that is, on abstract rather than on concrete systems.

This paper addresses the mathematical component of a good undergraduate computer science curriculum. It begins by describing the general nature of the mathematical needs of computer science undergraduates and then discusses some specific mathematical topics that are particularly helpful in computer science education. These mathematical topics include not only traditional mathematical subjects that can be taught in self-contained courses, such as discrete mathematics, but also certain mathematical *modes of thought* that pervade computer science thinking and that

cannot be taught easily on their own. In the last sections we consider the impact of these needs on the curriculum.

2. Mathematical Aspects of Undergraduate Computer Science

There is a persistent misconception that computer science consists merely of writing computer programs and that, as a result, the education of a computer scientist consists merely of training in skills related to coding and debugging computer programs. On the contrary, the discipline embraces principles and techniques for the design, construction, and analysis of a wide variety of complex systems. Even programming, to be successful, requires the careful application of scientific principles.

Since the principles of computer science are largely mathematical, computer science curricula must necessarily rely on support from mathematics. The traditional mathematics and applied mathematics "service" curricula, steeped as they are in continuous mathematics, do not, however, provide adequate support for computer science. The demands of computer science on mathematics are in many respects quite different from the demands of traditional scientific or engineering disciplines. The most important difference is that, to a much greater extent than in other disciplines, *abstraction* is an essential tool of every computer scientist, not just of the theoretician. The computer scientist is not simply a user of mathematical *results*; he must use his mathematical tools in much the same way as a mathematician does.

A computer science undergraduate curriculum must attempt to develop in the student an appreciation of the power of abstraction and an ability to discover abstractions suitable to new situations. This ability is what mathematicians call *mathematical maturity* (see [29] for further discussion). Mathematical maturity will not be fostered if mathematics is taught to computer science students as a mere skill or as an unpleasant necessity.

Like other scientific and engineering disciplines, computer science must also teach certain specific attitudes, skills, and techniques from mathematics. In computer science, most of these come from *discrete mathematics* --- the mathematics dealing primarily with discrete objects. Discrete mathematics as an independent subject is a relatively new arrival, however, and present courses in this area often do not have the cohesion or intrinsic interest of the traditional calculus or algebra sequences. It is interesting, however, that many discrete mathematics courses use the notion of algorithm --- a concept from computer science --- as their unifying element [23, 28, 30].

2.1. Mathematical Modes of Thought Used by Computer Scientists

The most important contribution a mathematics curriculum can make to computer science is the one least likely to be encapsulated as an individual course: a deep appreciation of the modes of thought that characterize mathematics. We distinguish here two elements of mathematical thinking that are also crucial to computer science and speculate on how they might be integrated into a mathematics curriculum. These elements tend not to fall into identifiable courses, but are generally transmitted *culturally*, as part of the process of attaining that elusive quality of mathematical maturity. The two elements are the dual techniques of *abstraction and realization* and of *problem-solving*.

2.1.1. Abstraction and Realization

Computer scientists usually deal with situations that are too complicated to understand completely at one time. The chief tool for managing this complexity is *abstraction* --- a process of drawing away from detail or selectively ignoring structure. Conversely, complex real systems are built from abstract characterizations by the inverse process of *realization* or *representation* --- the selective introduction of underlying structure.

In mathematics, the deliberate use of abstraction is most noticeably manifest in the notion of *mathematical system*. The mathematical systems that are most useful to mathematicians, such as groups, fields, or categories, are those that best focus recurring problems. In computer science this kind of abstraction or encapsulation appears in many forms. Finite state automata, for example, permit study of control flow in programs without reference to variables or data.

Mathematics can be characterized by its search for gems of abstraction --- those abstractions that capture the essential qualities of a phenomenon and ignore the rest. Computer scientists carry on a similar search, but, because the structures they describe usually become manifest as real systems, they are concerned with the *performance* of systems as well as with their functional properties. Consequently, computer scientists find they are often fighting two sides of the same battle: Given a complex problem, they must develop abstractions that provide a way of managing the complexity, allowing for clear and effective reasoning about the problem. But they must also ensure that the representations or realizations that are hidden beneath their abstractions yield implementations with satisfactory performance.

The computer scientist who appreciates the variety of mathematical systems will be better able to evaluate structures and organizations for program and system design. A student who becomes comfortable thinking in terms of systems will be more likely to appreciate the full generality of the program or system structures he creates and less likely to think only in terms of the present specific application.

To strike the best balance between clarity and performance, the computer scientist needs a large and varied arsenal of abstraction and realization techniques. Some of these are rooted in conventional computer science and are therefore most appropriately taught in the context of computer science problems. Others, however, are best transmitted through a comprehensive study of mathematical reasoning.

One of the most powerful tools for abstraction is *language*. For example, programming languages are languages that allow the expression of algorithms without reference to particular realizations of algorithms in computer hardware. These languages also give us a way of describing data by means of its structure, not by its representation as "bits" in a computer memory. Like mathematical languages, computer languages are not designed in a purely *ad hoc* fashion; they are, rather, manifestations of carefully chosen lines of abstraction. If a computer science student is to appreciate the variety and universality of computer languages, he or she must have a mathematician's understanding of the nature and use of language. This includes, for example, understanding the nature of symbols and the essence of deduction --- carrying out worldly reasoning by means of symbol manipulation.

This discussion does not, alas, point to courses from "traditional" computer science curricula [4, 14] that will satisfy this need. (Indeed, the standard curriculum designs barely acknowledge the fact that exposure to mathematical reasoning is appropriate for computer science [22, 23].) There are courses in mathematics, however, that can foster the kind of understanding we seek. A good logic course, giving a kind of introspective view of mathematical reasoning, can be of great benefit to the computer scientist. Other mathematics courses, such as the analysis courses that are intended for mathematicians (as opposed to the ones intended for calculus "users"), can be of value simply because of the experience in mathematical definition and reasoning that the students obtain.

2.1.2. Problem-solving

Computer science is a problem-solving discipline, concerned with the development of cost-effective solutions (such as programs and machines) to computational problems. Computational problems do not in general have predictable structure and are almost always stated in abstract terms. As a consequence, the construction of programs (or even machine architectures) is analogous to the construction of mathematical proofs. While a proof (or program) has a well-defined structure, the process of obtaining it can be quite undisciplined, involving all sorts of peripheral and heuristic knowledge. Thus, the computer scientist, like the mathematician, must have command of a variety of problem-solving techniques, and must be able to apply them in a creative and yet disciplined fashion.

The designers of many graduate curricula in computer science have acknowledged the importance of

abstract problem-solving and have incorporated problem-solving workshops based on such texts as [21] into their programs. We suggest that this need should be directly addressed in undergraduate curriculum design [31]. It is very important for students to be aware of the problem-solving process and of the general techniques that they can apply to it. Courses on these topics have been offered in engineering and computer science departments using texts such as [25, 32], but they could be equally appropriate in mathematics departments.

2.2. Discrete Mathematics

In addition to the ability to think like a mathematician, a computer scientist requires fluency in some specific areas of mathematics. These are the areas usually (collectively) called discrete mathematics, and they include such topics as elementary set theory and logic, abstract algebra, and combinatorics. Since this material is well-understood, an outline should suffice:

- *Elementary Set Theory and Logic.* It is important that the treatment of logic go beyond the usual manipulative knowledge of the propositional connectives and quantifiers. Students should have an appreciation of the central issues of mathematical logic and in particular of the role of language in mathematical definition and reasoning. This appreciation can be brought out both in the subject material *per se* and in the way it is presented.
- *Induction and Recursion.* These are recurring themes in computer science and should be covered in depth; induction underlies nearly all techniques for reasoning about the correctness and performance of programs.
- *Relations, Graphs, Orderings, and Functions.* This is a part of basic mathematical fluency. Without this knowledge, it is hard to understand even the most basic algorithms.
- *Abstract Algebra.* Algebraic structures recur in computer science, particularly in automata theory, complexity, software specification, and coding theory. A good introduction to algebra will develop in the student an understanding of the notion of mathematical system and will give him experience in using several of the more common ones.
- *Combinatorial Mathematics.* Analysis of algorithms requires a wide variety of mathematical skills; these are drawn mostly from combinatorial mathematics and from probability and statistics.

Although we have not as yet found a completely satisfactory text for discrete mathematics in computer science, the books [17, 28, 30] can serve as a starting point.

2.3. Continuous Mathematics

Although our primary emphasis here has been on the role of discrete mathematics in the computer science curriculum, we believe that continuous mathematics is also important to the education of a computer scientist. A mathematician's calculus course can serve as an excellent introduction to mathematical thinking. We will need to consider the question of when calculus should appear in the curriculum. For the purposes of computer science courses, discrete mathematics should appear as early as possible, preferably in the freshman year, but it has also been argued that calculus should precede discrete mathematics in the mathematics curriculum.

3. Some Remarks about Computer Science and Mathematics Curricula

As we noted above, the undergraduate computer science curriculum designs currently endorsed by major professional organizations have very weak mathematics requirements [4, 14]. Perhaps this is only a side-effect of the recent rapid growth of undergraduate computer science, but in any case it is widely viewed as a shortcoming. (See [22] and reactions to that article [16].) It is interesting to note that *earlier* computer science curriculum designs [2] contained much stronger mathematical requirements. Comparisons of the early and recent curricula are given in [22, 23].

With respect to the mathematics curriculum, we believe that support for the ideas and topics listed here would not cause major disruption to most mathematics curricula. The most significant change would be the addition of a freshman- or sophomore-level course in discrete mathematics. We believe that this course would be beneficial to students in other departments as well as to computer scientists. (The case for teaching elementary discrete mathematics to all students is presented by Ralston in [23].) Beyond that, most of the material we propose is fairly standard, though perhaps different in emphasis from in the traditional mathematics service courses. We should note here that our list should in no way be construed as complete; we mention topics only to provide an indication of the kind of material that is relevant.

Although much of the material computer scientists need is already provided in standard courses, we believe that both computer science and mathematics curricula would be strengthened by recasting some of those courses a bit. Teachers of mathematics can take advantage of their students' knowledge of computers by showing how classical techniques are realized in computational systems and, where appropriate, by drawing on the rich collection of practical examples supplied by computer science. Linear algebra and numerical analysis courses already do this, teaching computational techniques along with abstract definitions. Discrete mathematics, combinatorics, and graph theory courses also often make extensive use of programming exercises. These programming exercises give students an unusual "hands-on" way of experimenting with abstract structures. Moreover,

Lochhead [18] argues that programming *per se* contributes to understanding mathematical ideas.

4. Conclusion

Computer science as a discipline has reached the point where there is enough intellectual substance for undergraduate degree programs to be meaningfully offered. Computer science courses are no longer simply programming "service" courses offered for the benefit of computer users; there is truly fundamental conceptual material to be imparted.

A successful undergraduate curriculum, in which basic principles are set forth and elucidated, can only come about after intensive self-examination in the field. Naturally enough, there is a certain lag between the time these principles first emerge and the time they can be effectively integrated into a curriculum, but we feel that there is now a consensus among computer science researchers and practitioners regarding the mathematical content of the field, as sketched in this paper. This consensus, unfortunately, does not extend to the methods for imparting the mathematical material; this remains one of the central challenges of computer science and mathematics curriculum design.

Acknowledgements

We thank Roy Ogawa and Dana Scott for their helpful comments on an earlier manuscript.

Part III: Curriculum '78 -- Is Computer Science Really that Unmathematical?

Anthony Ralston and Mary Shaw

In 1979, Ralston was investigating curricula for discrete mathematics [23] and Shaw was participating in evaluations of Curriculum '78 and the role of mathematics in undergraduate computer science. They combined their notes to form a criticism of the mathematical content of Curriculum '78 that appeared in Communications of the ACM [22]. Some comments on the paper appeared a few months later [16]. This paper and the correspondence appear here as Part III.

If computer science had not yet developed -- significantly -- as a science in the ten years between Curriculum '68 [2] and Curriculum '78 [4], then perhaps all those people who wondered if computer science was really a discipline would have been correct. In 1968 computer science was searching for but had not yet found much in the way of the principles and theoretical underpinnings which characterize a (mature) science. Ten years later, there is nothing laughable about calling computer science a science. This decade has seen major advances in the theory of computation and in the utility of theoretical results in practical settings. The rapid growth of the field of computational complexity has greatly increased our ability to analyze algorithms. And perhaps most significantly, we have finally started to make real progress in developing principles and theories for the design and verification of algorithms and programs.

Are these changes evident in Curriculum '78? Sadly, no. That curriculum only lends support to the equation

Computer Science = Programming

that is mistakenly believed by so many outside the discipline. In the "Objectives of the Core Curriculum" [4] only the second objective -- "be able to determine whether or not they have written a reasonably efficient and well-organized program" -- recognizes that good programming requires more than just mastery of the syntax and semantics of a programming language. And even here the reference to principles and theory is, to be charitable, vague.

The principles and theories of any science give it structure and make it systematic. They should set the shape of the curriculum for that science, for

- only in that way can they provide a framework for the mastery of facts, and
- only in that way will they become the tools of the practicing scientist.

This is as true for computer science as it is for mathematics, for the physical sciences, and for any engineering curriculum. Inevitably, for any science or any engineering discipline, the fundamental principles and theories can only be understood through the medium of mathematics. In the following sections we focus on the place of mathematics in the computer science curriculum and try to show how badly Curriculum '78 fails in this respect.

But first we note one matter of crucial importance which makes an emphasis on principles and theory even more important in computer science than in other disciplines. Computer science is an evolving field. Specific skills learned today will rapidly become obsolete. The principles that underlie these

skills, however, will continue to be relevant. Only by giving the student a firm grounding in these principles can he or she be protected from galloping obsolescence. Even a student who aspires only to be a programmer needs more than just programming skills. He or she needs to understand issues of design, of the capability and potential of software, hardware, and theory, and of algorithms and information organization in general.

(Curriculum '68)		(Curriculum '78)	
M1	Introductory calculus	MA1	Introductory calculus
M2	Mathematical analysis I	MA2	Mathematical analysis I
M2P	Probability	MA2A	Probability
M3	Linear algebra	MA3	Linear algebra
B3	Introduction to discrete structures	MA4	Discrete structures
B4	Numerical calculus		
	plus 2 of		(Required for some students)
M4	Mathematical analysis II		
M5	Advanced multivariate calculus	MA5	Mathematical analysis II
M6	Algebraic structures	MA6	Probability and statistics
M7	Probability and statistics		

Figure 1: Required Mathematics Courses.

1. Curriculum '78 and Mathematics

A comparison between the mathematics content of Curriculum '78 and that of Curriculum '68 is instructive. It reveals that

1. Whereas Curriculum '68 required the student to take eight (8) mathematics courses (see Figure 1), Curriculum '78 requires only five (5) mathematics courses.
2. The mathematics courses in Curriculum '68 formed an integral part of its prerequisite structure (see Figure 2). Note, in particular, for how many courses the discrete structures course (B3) is a prerequisite. In Curriculum '78, however, there is no mathematics prerequisite for any undergraduate computer science course with the exception of three advanced and clearly quite mathematical courses (only one of which has a computer science prerequisite). True, Curriculum '78 notes that the "mathematics requirements are integral to a computer science curriculum even though specific courses are not cited as prerequisites for most computer science courses." But this was clearly an afterthought, not present in the preliminary publication [3], and added only in response to

criticism of the preliminary version.[2] Moreover, if the mathematics courses are not prerequisite to the computer science courses, the latter cannot teach or use formal techniques that require mathematical literacy.

3. The mathematics emphasized in both curricula is traditional, calculus-based continuous mathematics. In both curricula the only course which is not a common part of the undergraduate mathematics curriculum is a single course in discrete structures.

More generally, the attitudes of Curriculums '68 and '78 toward mathematics are very different. Whereas the authors of C68 aver that "an academic program in computer science must be well based on mathematics since computer science draws so heavily upon mathematical ideas and methods," the authors of C78 say only that "An understanding of and the capability to use a number of mathematical concepts and techniques are vitally important for a computer scientist." The later, too, was an afterthought since the preliminary report stated that "it was recognized in the process of specifying this core material that no mathematical background beyond the ability to perform simple algebraic manipulation is a prerequisite to an understanding of the topics." And note that this "core material" consists of *eight* courses including one on Data Structures and Algorithm Analysis.

One would have to conclude that the authors of Curriculum '78 believe that

1. Mathematics is less important in the computer science undergraduate curriculum today than ten years ago.
2. Basic computer science courses have less need for mathematical prerequisites today than ten years ago.
3. The mathematics that is appropriate for computer science undergraduates has changed not at all in general flavor over the ten-year period between the two curricula.

We think all three of these propositions are wrong, and dangerously so. In the next section we will indicate why and how we would modify Curriculum '78.

2. Mathematics for Computer Scientists

A key sentence in C78, also not in the preliminary version, states that "Ideally computer science and mathematics departments should cooperate in developing courses on discrete mathematics which are appropriate to the needs of computer scientists." But, as if to emphasize that this recognition of the importance of discrete mathematics was only an attempt at a quick fix in response to criticism of the preliminary proposal, C78 goes on to say that "Until such time as suitable courses become readily available, it will be necessary to rely on the most commonly offered mathematical courses for the

---

[2] We think a comparison of the sections devoted to mathematics in the preliminary and final versions of Curriculum '78 clearly imply a "quick fix" which does not address the substantive issues.

mathematical background needed by computer science majors." And the report goes on to recommend the five courses listed in Table 1, four of which are standard

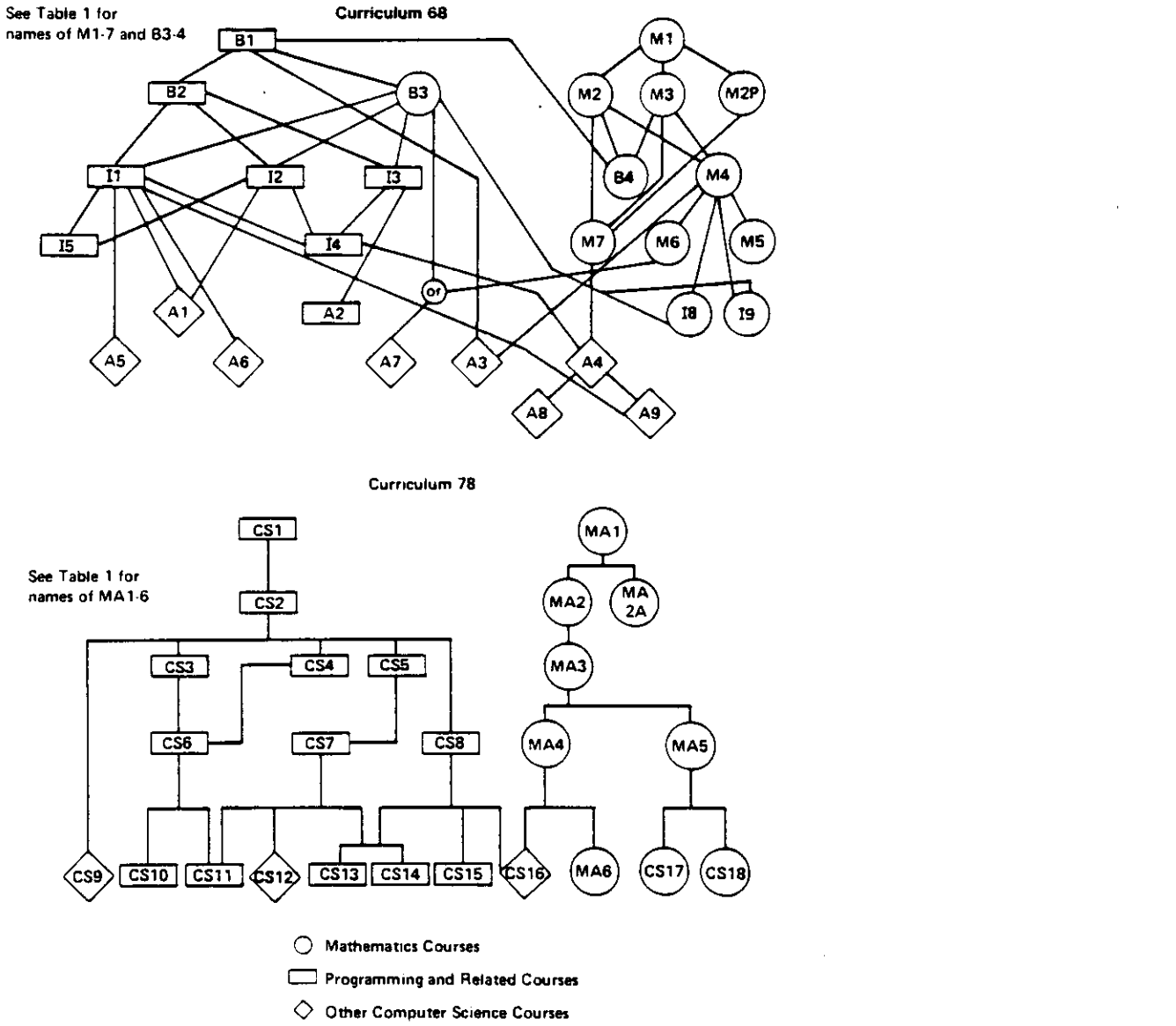

Figure 2: Prerequisite Structure

undergraduate mathematics courses from a 1965 report of the Committee on the Undergraduate Program in Mathematics (CUPM) [1] and the fifth is "a more advanced course in discrete structures than that given in C68." In other words, instead of going back to the drawing board and doing the mathematics portion of C78 properly, the authors elected to fudge the issue with pretty words and no substance.

For, of course, the quotation in the first sentence of the previous paragraph is correct and should have been the basic philosophy which informed the entire C78 report. In rather more detail this (and our) philosophy is:

1. Mathematical reasoning does play an essential role in all areas of computer science which have developed or are developing from an art to a science. Where such reasoning plays little or no role in an area of computer science, that portion of our discipline is still in its infancy and needs the support of mathematical thinking if it is to mature. Large portions of software design, development, and testing are still in this stage.
2. The student of computer science must be encouraged to use the tools and techniques of mathematics from the beginning of his or her computer science education. This means, for example, that even in the very first course in computer science (e.g., CS1 in C78 where, among other things, the student is to be introduced to "algorithm development") the basic ideas of the performance and correctness of algorithms and their associated mathematics need to be introduced or assumed from a parallel or prerequisite course.[3]
3. The mathematics curriculum for the computer science student must be designed to
   - provide, either in separate courses or within a computer science course, the mathematics prerequisites appropriate to the computer science curriculum. (Obvious, no? But signally missing from C78.)
   - more generally, develop mathematical reasoning ability and mathematical maturity so that students will be able to apply more and more sophisticated mathematics to their computer science courses as they progress through the computer science curriculum.

Some other, more pragmatic, points are worth making before we discuss the mathematics curriculum for a computer science major in more detail:

1. Only the quite basic courses can be required for all students. Depending upon the emphasis and areas of specialization in the last year or two, one set of mathematics courses rather than another may be most appropriate.
2. The needs of the practicing computer professional rather than those of the research computer scientist should be uppermost in consideration of appropriate mathematics for the undergraduate curriculum. To the extent that these needs are different -- it is not obvious that they are -- the future researcher will have to satisfy his/her needs through undergraduate electives or in graduate school.
3. Although we believe strongly that the values of a liberal education should infuse any undergraduate program, our focus here is on the professional needs of the computer scientist, not on the general education needs. Thus, it may be true that all educated men and women should be familiar with the essence of calculus but it does not *necessarily*

---

[3]The authors of C78 are, of course, quite correct in not making MA1, Introductory Calculus, a prerequisite for CS1; the problem is that MA1 is the wrong first mathematics course for computer science students.

follow that computer scientists have a significant professional need to know calculus.

What then is an appropriate sequence of mathematics courses for the computer science major?

1. *Discrete Mathematics.* The overwhelming mathematical needs in the courses which normally comprise the first two years of a computer science major are in areas broadly covered by the rubric discrete mathematics -- elementary logic, inductive proof, discrete number systems, basic combinatorics, difference equations, discrete probability, graph theory, some abstract and linear algebra, etc. We believe a two-year sequence can and must be developed (by mathematicians, if possible, but without them, if necessary) for computer science majors. This sequence should be *integrated* with the first two years of the computer science curriculum. Beyond the subject matter itself, we believe that such a sequence would be able to develop mathematical literacy and maturity at least as well as the classical two-year calculus sequence.
2. *Calculus.* A year -- but perhaps only a semester -- of calculus in the junior year would be appropriate for all or almost all computer science majors. The techniques of calculus have just enough application in standard undergraduate computer science courses to make this desirable. Note also that a year of calculus at the junior level could cover quite a bit more material than a year of freshman calculus.
3. *Statistics.* A basic knowledge of statistics is essential to almost all areas of professional work in computer science. It is not, however, entirely clear to us whether or not an adequate course in statistics can be taught to computer science students without a calculus prerequisite. If not, then at least a semester of calculus would be mandatory for computer science students.

Much more could be said about possible mathematics courses for computer science students but we shall not do so here. Rather our aim is to urge that the ACM Curriculum Committee on Computer Science go back to the drawing board, make a real study of the mathematics needs of a computer science curriculum, and emerge with recommendations which will have the respect and support of the computer science community.

The mathematics of central importance to computer science has changed drastically in the ten years from C68 to C78. The lack of recognition of this in C78 will undoubtedly lessen the impact of the entire report. Mathematics is at least as important to computer science today as in 1968. But the 1965 recommendations of CUPM are singularly inappropriate to the needs of computer science today.

Letters on the Mathematical Content of Curriculum '78

Comment from Alan Russell

I read with interest the Ralston-Shaw article [22] on the mathematical content of Curriculum '78 [4]. While I hesitate to overstress the mathematical principles of computer science for fear of keeping those who are not mathematically inclined away from the field, I still strongly agree with the arguments presented in this article. Historically part of the problem has been the inclusion of mathematics-based courses in the computer science curriculum

1. without sufficient emphasis on the integration of the mathematical content of these sometimes theory-based courses with the more practitioner-oriented computer science courses
2. without sufficient emphasis that these mathematical concepts are the principles upon which computer science is founded.

The end result of this situation has been that many computer science students are not able to relate their computer science and mathematical courses

1. because the courses have not been taught in a relatable fashion
2. because the student is not aware that the two areas are supposed to be related.

The Ralston-Shaw article focuses on the first of these two conditions and, as a long-term objective, spells out the guidelines for introducing the proper mathematical content into the curriculum. As a short-term objective, however, a solution to the second problem might be more useful. In particular, I think the following objective ought to be added to the objectives for course CS1:

(d) to foster an awareness of the mathematical principles behind computer science.

Upon completing this course the student should be able to recognize the relationship of mathematics to computer science both from a historical point of view and as regards current research and development efforts. More important, however, the student will be able to recognize the relationship of the mathematics courses in his/her curriculum to the computer science courses *regardless* of whether the course content is integrated or not. This overview of the "foundations" of computer science will also help to replace the equation

Computer Science = Programming

with a more balanced view of what computer science is all about.

A second concern I have regarding the curriculum is that it lacks a "real world" view from a career development standpoint. Too often a student completes a computer science curriculum

1. without any awareness of what he/she wants to do with the knowledge gained

2. without any awareness of the true nature of the available alternatives.

As an illustration, consider the student who had more fun in the operating system writing course (CS6) than he/she had writing a payroll check printing program (CS2) and on that basis applies for a job at several major companies as a "systems programmer," not willing to consider a position as a "programmer/analyst." While the solution to (1) requires career guidance which is beyond the scope of this curriculum, a solution to (2) can easily be constructed.

I would like to propose the following course as an addition to the curriculum:

CS2A. Roles of a Computer Scientist (1-0-1)

Prerequisite: CS2

The objectives of this course are: (a) to develop an understanding of the various roles that a person with a computer science education can take in society; and (b) to develop an understanding of the basic skills and requirements needed in each of these roles.

COURSE OUTLINE

After an initial overview of the subject, an in-depth look at some of the major segments of the computer science community should be undertaken. Guest speakers should definitely be considered. A partial list of topics which should be discussed are:

- Industry vs. Education vs. Government job segments
- Business vs. Scientific vs. Systems programming
- Research vs. Software/Hardware development vs. End-User programming
- Small shops vs. Large shops
- Outlook of demand in the various segments.

While it is not expected that such a course can be a substitute for personalized career counseling, there should be sufficient breadth and depth in the coverage of the various roles so that each student has an appreciation of the differences as well as the similarities of the requirements for each role. This course will not only give a better sense of direction to some students by giving them more definite goals, but will also give a better perspective of the integration of the total curriculum and its ultimate application to society.

As a final point, I think we can all be proud of the tremendous advances that have been made in the development of the science of computer science in the past decade and the role that ACM has played in helping to direct a corresponding development in the computer science curriculum. Curriculum

'78, and Curriculum '68 before it, have had a major role in shaping the direction of computer science education. I look forward to continuing developments in this area.

Alan Russell, CDP
RD1, Box 223C
Zionsville, PA 18092

Comment from Richard E. Fairley

The recent article by Ralston and Shaw concerning the undergraduate mathematics sequence in computer science is timely, appropriate, and absolutely correct. The February issue of *Communications* arrived as I was preparing the following in a memo to the computer science faculty here at Colorado State University:

1. Mathematics is a necessary and desirable component of computer science education. Mathematical problem solving ability is the primary skill that distinguishes a computer scientist from a programmer, and a strong foundation in mathematics is the best hedge against technical obsolescence of our graduates.
2. We are not requiring the correct math courses for our undergraduates. This conclusion is based on the following considerations:
    a. Our graduate curriculum has seven tracks: architecture, data structures and databases, graphics, languages and compilers, numerical methods, operating systems, and computing theory. Only one track (numerical methods) requires a strong calculus background. This is an indication that the undergraduate mathematics sequence is out of sync with the subject matter of computer science.
    b. I have revised the formal languages course to include a month of review of discrete structures and modern algebra. The students cannot handle the material without this review.
    c. Graduate students in my software engineering course complain that they are not equipped to read the literature in software specification techniques, proof of correctness, testing theory, etc. I believe this is also true in the graduate level compilers, data structures, database, graphics, and operating systems courses.
3. A better math sequence is:
    - two semesters of calculus (freshman level)
    - two semesters of discrete math (sophomore level)
    - one semester of probability and statistics (junior level)
    - one semester of math elective (junior or senior level)

    The math elective would be geared to the students' senior level elective courses in computer science. It could be used as follows:

- Numerical Methods -- Linear Algebra, Advanced Calculus, or Differential Equations
- Graphics -- Linear Algebra or Geometry
- All Others -- Applied Algebra

4. Possible topics in the discrete math sequence would include:
   - Elementary Logic
   - Proof Techniques (induction in particular)
   - Number Systems
   - Combinatorics
   - Difference Equations
   - Discrete Probability
   - Graph Theory
   - Matrix Algebra
   - Introduction to Modern Algebra
5. I suggest that we pursue the design of a two-semester, sophomore level sequence in discrete mathematics as a joint venture with the math department.

In a subsequent letter to Ralston and Shaw, I suggested that a national committee be formed to prepare a study of undergraduate mathematics in computer science. I would like to use this forum to express my appreciation to these two authors for initiating a dialogue on the appropriate mathematics sequence in undergraduate computer science.

Richard E. Fairley
Colorado State University
Fort Collins CO 80523

Comment from Julius A. Archibald, Jr.

I have read with great interest the article "Curriculum '78 -- Is Computer Science Really that Unmathematical?" by Ralston and Shaw, appearing in the February 1980 issue of *Communications*. It seems to me that these authors have identified a small (albeit important) issue in the very difficult task of computer science curriculum development, isolated it from its context, and arrived at conclusions which, in isolation, are very difficult to oppose. The difficulty is that, in isolation, the issue has become oversimplified.

The context of the curriculum development process can be set through the posing of a sequence of questions, many of which do not have answers agreed upon across the computing disciplines and professions.

The sequence is as follows:

1. The question of definitions.
    a. What is computer science?
    b. How does it relate to the other disciplines?
    c. How does it relate to the other computer professions?
2. The question of expectations.
    a. What does society, in general, expect of computer scientists?
    b. What does industry, in particular, that part of industry that is concerned in one way or another with computing, expect of computer scientists?
    c. What does academia expect of computer scientists?
3. The overall questions of preparation.
    a. How are practitioners to be prepared to meet the expectations of society, industry, and/or academia, as may be appropriate?
    b. What should be the division between formal training (i.e. training in academic institutions) and informal training (i.e. training through experience)?
4. The specific questions of academic training.
    a. What should be the division between the quantities and levels of training at training institutes, two-year undergraduate schools, four-year undergraduate schools, master's level graduate schools, and doctorate level graduate schools?
    b. What should be the division between theory and applications at each of these levels?
    c. What should be the level of specialization at the undergraduate level: liberal arts (at most one-third specialized), or professional (up to two-thirds specialized)?
    d. What are the priorities for the inclusion in the computer science program of material from other disciplines?

Part of the difficulty lies in the fact that, in the twelve years since Curriculum '68, the computing disciplines and professions themselves have become very greatly diversified. These disciplines and professions certainly include what are referred to by many as computer science, computer engineering, information science, software engineering, programming, systems design, systems analysis, data processing, etc. The questions are of identity, even of self-identity. Is there agreement as to the definitions of the foregoing fields by persons who identify themselves as practitioners of these respective fields? I think not! Before issues such as the one raised by Ralston and Shaw can be resolved, definitions of these respective subfields, and others, will have to be formulated and agreed to by a broad cross section of individuals in the computing professions. This may be a job for AFIPS. The problem is that Ralston and Shaw are using a traditional definition of computer science, one that

goes back to an almost classical period in the computing profession, and certainly to a period of infancy in computer education. This was a period in which, because of immaturity in the field, agreements were more easily reached.

The 1968 definition of computer science was highly mathematical, and, as a consequence, Curriculum '68 was also highly mathematical. In the interim, there have appeared in the literature any number of complaints from the industry which we serve that our graduates were of little benefit to them. Thus, highly theoretical programs which were inspired by Curriculum '68 were of benefit only to prepare students enter graduate school.

The bottom, line here again, is one of definitions. Industry's definition of what it wanted was different from academia's definition what it was producing. Whether or not either is to be called a computer scientist is irrelevant. The point is that the academic institutions were producing a product of little benefit to industry. I maintain that any undergraduate curriculum which does little more than prepare students for graduate school is of little benefit to society. There is also the question of the student's expectations from their college educations. What do they see themselves wanting to be or do? It must be assumed that the majority of undergraduate students, regardless of major, at a majority of the undergraduate colleges on this continent, are not going to graduate school and therefore must be prepared for useful employment in the industrial community. I realize that this statement strikes at the heart of the concept of "liberal education," but one must realize that the students whom we serve have become extremely practical in their outlook. We must also recognize that the present high enrollments in computer science are due to the high level of employment opportunities. Accordingly, we must respond to the expectations of industry.

The specific problem, as it pertained to Curriculum '68, was that an urgent need had developed for greater applied content. There has also developed a need for greater liberal arts content, communication skills in particular. Given these new demands, and given the time limitations inherent in a four-year academic program, the only solution is a reduction in the theoretical content of the program. Indeed, it must be argued, independently, that heavy theoretical content is much more appropriate in graduate school than it is in undergraduate school. Curriculum '78 may not be perfect, but it is a step in the right direction.

Let me again return to the diversification of the last twelve years. It seems to me that this diversification is a key to future developments. The ACM, the IEEE Computer Society and other concerned agencies have, from time to time, published suggested curricula. That is all that they are, suggested curricula, or guides. Each department, in each institution, must be responsible for its own curriculum development. Curriculum development must be an ongoing activity; curricula are not

static. They must be based upon several factors:

1. the department's review of societal needs, both recognized and unrecognized;
2. the department's perceived strengths and capabilities;
3. the exchange of ideas through the professional societies and the printed media; and
4. other considerations deemed appropriate by the department concerned.

Let us view the published professional differences of judgment as a positive testimonial to the maturing process taking place within our profession. The proper response to both Curriculum '78 and the Ralston-Shaw article is for each department to review its own curriculum in terms of the published material and its own local situation, and to take whatever actions seem to be appropriate to it, in a professional and collegial manner.

Julius A. Archibald, Jr.
SUNY at Plattsburgh
Plattsburgh, NY 12901

Authors' Response

We appreciate the support of the essential theses of our article in the letters of Fairley and Russell, and note only that there are various possible different sequences of mathematics courses which would support a computer science curriculum better than what is proposed in Curriculum '78.

As to Archibald's letter, it raises some important issues, but two things in it disturb us:

1. There is an implication -- admittedly no more than this -- that "mathematical" should be equated with "theoretical." We reject this. The argument in our article was addressed to all undergraduate computer science programs whether or not students in them are likely to go to graduate school. Mathematics is -- or should be -- a practical tool for working programmers and should be as important in their education as in that of the research computer scientist.
2. The argument that academe did not or is not producing a "product" of "benefit to industry" is a hoary one. We doubt this was ever true although it is true that some segments of industry did and do and always will complain about the education of computer science majors. And it is probably true that computer science departments are less sensitive to the current needs of prospective employers than they might be. But almost all complaints about the education of computer science majors have been short-sighted and oriented to the very short-term concerns which motivate the "expectations" of most of industry. To respond to them would be to guarantee the early obsolescence of our "products." Moreover, we don't believe that Archibald's characterization of industry's concerns is accurate. We see a significant and growing trend among industrial leaders to place a high value on the mastery of mathematical fundamentals. This often

takes the form today of extensive in-house training programs.

One last point. Archibald mentions the need for the formulation of definitions of the fields and subfields encompassed by what we call computer science. In this connection we direct the attention of readers of *Communications* to the *Taxonomy of Computer Science and Engineering* [6] which has just been published by AFIPS Press.

Anthony Ralston
SUNY at Buffalo
Amherst, NY 14226

Mary Shaw
Carnegie-Mellon University
Pittsburgh, PA 15213

Part IV: Some Organizations of Computer Science

In order to develop a comprehensive undergraduate curriculum, the Curriculum Design Project needs to have an overall view of computer science. Although we have not found an entirely satisfactory structure or curriculum, we have examined quite a few. We found that comparison was simplified if we extracted the outlines from their context and presented them in a consistent format. The results are presented in this Part.

An organization's view of computer science is often incorporated in an outline or a curriculum. The top few levels of such an outline can capture that view in just a few pages. It is true that this sort of summary can sometimes be misleading -- the critical point of view may pervade the organization rather than driving the explicit outline. Nevertheless, pedagogic decompositions can be revealing. The excerpts selected here present the first two or three levels of organization from a variety of curricula or other presentations of computer science. Some of these are undergraduate or graduate curriculum designs; others were developed for very different reasons.

1. ACM Curriculum '78

Curriculum '78 [4] was designed under auspices of the ACM to replace Curriculum '68 as the standard guideline for undergraduate computer science education. The committee that prepared these guidelines established a "core" of "elementary material" that should be included in any undergraduate major. Advanced material is described in the report in terms of specific course outlines.

From Curriculum '78 we extract the statement of objectives, the outline of "elementary material", and the topic lists for the designated courses.

1.1. Objectives

The core material is required as a prerequisite for advanced courses in the field and thus it is essential that the material be presented early in the program. In learning this material, the computer science student should be provided with the foundation for achieving at least the objectives of an undergraduate degree program that are listed below.

Computer science majors should:

1. be able to write programs in a reasonable amount of time that work correctly, are well documented, and are readable;
2. be able to determine whether or not they have written a reasonably efficient and well organized program;
3. know what general types of problems are amenable to computer solution, and the various tools necessary for solving such problems;
4. be able to assess the implications of work performed either as an individual or as a member of a team;
5. understand basic computer architectures;
6. be prepared to pursue in-depth training in one or more application areas or further education in computer science.

It should be recognized that these alone do not represent the total objectives of an undergraduate program, but only those directly related to the computer science component.

1.2. Elementary Material

In order to facilitate the attainment of the objectives above, computer science majors must be given a thorough grounding in the study of the implementation of algorithms in programming languages which operate on data structures in the environment of hardware. Emphasis at the elementary level then should be placed on algorithms, programming, and data structures, but with a good understanding of the hardware capabilities involved in their implementation.

Specifically, the following topics are considered elementary. They should be common to all undergraduate programs in computer science.

Programming Topics

P1. Algorithms: includes the concept and properties of algorithms; the role of algorithms in the problem solving process; constructs and languages to facilitate the expression of algorithms.

P2. Programming Languages: includes basic syntax and semantics of a higher level (problem oriented) language; subprograms; I/O; recursion.

P3. Programming Style: includes the preparation of readable, understandable, modifiable, and more easily verifiable programs through the application of concepts and techniques of structured programming; program documentation; some practical aspects of proving programs correct. (Note: Programming style should pervade the entire curriculum rather than be considered as a separate topic.)

P4. Debugging and Verification: includes the use of debugging software, selection of text data; techniques for error detection; relation of good programming style to the use of error detection; and program verification.

P5. Applications: includes an introduction to uses of selected topics in areas such as information retrieval, file management, lexical analysis, string processing and numeric computation; need for and examples of different types of programming languages; social, philosophical, and ethical considerations.

Software Organization

S1. Computer Structure and Machine Language: includes organization of computers in terms of I/O, storage, control and processing units; register and storage structures, instruction format and execution; principal instruction types; machine arithmetic; program control; I/O operations; interrupts.

S2. Data Representation; includes bits, bytes, words and other information structures; number representation; representation of elementary data structures; data transmission, error detection and correction; fixed versus variable word lengths.

S3. Symbolic Coding and Assembly Systems: includes mnemonic operation codes; labels; symbolic addresses and address expressions; literals; extended machine operations and pseudo operations; error flags and messages; scanning of symbolic instructions and symbol table construction; overall design and operation of assemblers,

compilers, and interpreters.

S4. Addressing Techniques: includes absolute, relative, base associative, indirect, and immediate addressing; indexing; memory mapping functions; storage allocation, paging and machine organization to facilitate modes of addressing.

S5. Macros: includes definition, call, expansion of macros; parameter handling; conditional assembly and assembly time computation.

S6. Program Segmentation and Linkage: includes subroutines, coroutines and functions; subprogram loading and linkage; common data linkage transfer vectors; parameter passing and binding; overlays; re-entrant sub-programs; stacking techniques; linkage using page and segment tables.

S7. Linkers and Loaders: separate compilation of sub-routines; incoming and outgoing symbols; relocation; resolving intersegment references by direct and indirect linking.

S8. Systems and Utility Programs: includes basic concepts of loaders, I/O systems, human interface with operating systems; program libraries.

Hardware Organization

H1. Computer Systems Organization: includes characteristics of, and relationships between I/O devices, processors, control units, main and auxiliary storage devices; organization of modules into a system; multiple processor configurations and computer networks: relationship between computer organization and software.

H2. Logic Design: includes basic digital circuits; AND, OR, and NOT elements; half-adder, addder, storage and delay elements; encoding-decoding logic; basic concepts of microprogramming; logical equivalence between hardware and software; elements of switching algebra; combinatorial and sequential networks.

H3. Data Representation and Transfer: includes codes, number representation; flipflops, registers, gates.

H4. Digital Arithmetic: includes serial versus parallel adders, subtraction and signed magnitude versus complemented arithmetic; multiply/divide algorithms; elementary speed-up techniques for arithmetic.

H5. Digital Storage and Accessing: includes memory control; data and address buses; addressing and accessing methods; memory segmentation; data flow in multimemory and hierarchical systems.

H6. Control and I/O: includes synchronous and asynchronous control; interrupts; modes of communication with processors.

H7. Reliability: includes error detection and correction, diagnostics.

Data Structures and File Processing

D1. Data Structures: includes arrays, strings, stacks, queues, linked lists; representation in memory; algorithms for manipulating data within these structures.

D2. Sorting and Searching: includes algorithms for in-core sorting and searching

methods; comparative efficiency of methods; table lookup techniques; hash coding.

D3. Trees: includes basic terminology and types; representation as binary trees; traversal schemes; representation in memory; breadth-first and depth-first search techniques; threading.

D4. File Terminology: includes record, file, blocking, database; overall idea of database management systems.

D5. Sequential Access: includes physical characteristics of appropriate storage media; sort/merge algorithms; file manipulation techniques for updating, deleting, and inserting records.

D6. Random Access: includes physical characteristics of appropriate storage media; physical representation of data structures on storage devices; algorithms and techniques for implementing inverted lists, multi-lists, indexed sequential, hierarchical structures.

D7. File I/O: includes file control systems (directory, allocation, file control table, file security); I/O specification statements for allocating space and cataloging files; file utility routines; data handling (format definition, block buffering, buffer pools, compaction).

1.3. Topic Lists for Courses

The full course descriptions include objectives, narrative description, and a detailed list of topics. The major topic headings are summarized here.

1.3.1. Elementary Level Courses

CS 1. Computer Programming I

Computer Organization (5%)
Programming Language and Programming (45%)
Algorithm Development (45%)
Examinations (5%)

CS 2. Computer Programming II

[Prerequisite: CS 1]
Review (15%)
Structured Programming Concepts (40%)
Debugging and Testing (10%)
String Processing (5%)
Internal Searching and Sorting (10%)
Data Structures (10%)
Recursion (5%)
Examinations (5%)

CS 3. Introduction to Computer Systems

[Prerequisite: CS 2]

Computer Structure and Machine Language (15%)
Assembly Language (30%)
Addressing Techniques (5%)
Macros (10%)
File I/O (5%)
Program Segmentation and Linkage (20%)
Assembler Construction (5%)
Interpretive Routines (5%)
Examinations (5%)

CS 4. Introduction to Computer Organization

[Prerequisite: CS 2]

Basic Logic Design (35%)
Coding (5%)
Number Representation and Arithmetic (10%)
Computer Architecture (35%)
Example (20%)
Examinations (5%)

CS 5. Introduction to File Processing

[Prerequisite: CS 2]

File Processing Environment (5%)
Sequential Access (30%)
Data Structures (20%)
Random Access (35%)
File I/O (5%)
Examinations (5%)

1.3.2. Sample Intermediate Level Courses

CS 6. Operating Systems and Computer Architecture I

[Prerequisites: CS 3 and CS 4 (CS 5 recommended)]

Review (10%)
Dynamic Procedure Activation (15%)
System Structure (10%)
Evaluation (15%)
Memory Management (20%)
Process Management (20%)
Recovery Procedures (5%)
Examinations (5%)

CS 7. Data Structures and Algorithm Analysis

[Prerequisite: CS 5]

Review (10%)
Graphs(15%)
Algorithms Design and Analysis (30%)
Memory Management (15%)
System Design (25%)
Examinations (5%)

CS 8. Organization of Programming Languages

[Prerequisite: CS 2 (CS 3 and CS 5 highly recommended)]

Language Definition Structure (15%)
Data Types and Structures (10%)
Control Structures and Data Flow (15%)
Run-time Consideration (25%)
Interpretive Languages (20%)
Lexical Analysis and Parsing (10%)
Examinations (5%)

1.3.3. Advanced Level Electives

CS 9. Computers and Society

[Prerequisite: elementary core material]

(The following list is suggestive, but not exhaustive:)

History of computing and technology
The place of the computer in modern society
The computer and the individual
Survey of computer applications
Legal issues
Computers in decision-making processes
The computer scientist as a professional
Futurists' views of computing
Public perception of computers and computer scientists

CS 10. Operating Systems and Computer Architecture II

[Prerequisite: CS 6; corequisite, a statistics course]

Review (10%)
Concurrent Processes (15%)
Name Management (15%)
Resource Allocation (25%)
Protection (15%)
Advanced Architecture and Operating Systems Implementations (15%)
Examinations (5%)

CS 11. Database Management Systems Design

[Prerequisites: CS 6 and CS 7]

Introduction to Database Concepts (5%)
Data Models (15%)
Data Normalization (5%)
Data Description Languages (10%)
Query Facilities (15%)
File Organization (25%)
Index Organization (5%)
File Security (10%)
Data Integrity and Reliability (5%)
Examinations (5%)

CS 12. Artificial Intelligence

[Prerequisite: CS 7]

Representation (40%)
Search Strategies (15%)
Control (20%)
Communication and Perception (10%)
Applications (10%)
Examinations (5%)

CS 13. Algorithms

[Prerequisites: CS 7 and CS 8]

Combinatorics (10-25%)
Numerical Analysis (10-25%)
Systems Programming (10-25%)
Artificial Intelligence (10-25%)
Domain Independent Techniques (15%)
Examinations (5%)

CS 14. Software Design and Development

[Prerequisites: CS 7 and CS 8]

Design Techniques (50%)
Organization and Management (15%)
Team Project (30%)
Examinations (5%)

CS 15. Theory of Programming Languages

[Prerequisite: CS 8]

Review (15%)
Scanners (20%)
Parsers (40%)
Translation (20%)
Examinations (5%)

CS 16. Automata, Computability, and Formal Languages

[Prerequisites: CS 8 and MA 4]

Finite State Concepts (30%)
Formal Grammars (35%)
Computability and Turing Machines (30%)
Examinations (5%)

CS 17. Numerical Mathematics: Analysis

[Prerequisites: CS 1 and MA 5]

Floating Point Arithmetic (15%)
Use of Mathematical Subroutine Packages (5%)
Interpolation (15%)
Approximation (10%)
Numerical Integration and Differentiation (15%)
Solution of Nonlinear Equations (15%)
Solution of Ordinary Differential Equations (20%)
Examinations (5%)

CS 18. Numerical Mathematics: Linear Algebra

[Prerequisites: CS 1 and MA 5]

Floating Point Arithmetic (15%)
Use of Mathematical Subroutine Packages (5%)
Direct Methods for Linear Systems of Equations (20%)
Error Analysis and Norms (15%)
Iterative Methods (15%)
Computation of Eigenvalues and Eigenvectors (15%)
Related Topics (10%)
Examinations (5%)

1.3.4. Mathematics Courses

The titles of CUPM mathematics courses assumed by Curriculum '78 are:

MA 1 *Introductory Calculus*
MA 2 *Mathematical Analysis I*
MA 2A *Probability*
MA 3 *Linear Algebra*
MA 4 *Discrete Structures*
MA 5 *Mathematical Analysis II*
MA 6 *Probability and Statistics*

2. ACM Recommendations for Master's Level Programs

The ACM Curriculum Committee on Computer Science extended its recommendations to cover masters-level programs [5]. They were not able to arrive at agreement on a specific model curriculum, but they recommended a list of possible courses. The courses are listed below, and the brief course descriptions are provided for courses not included in Curriculum '78.

These courses are presented as representative of courses offered in established master's programs. Some pairs are redundant, such as [CS 19, CS 21] and [CS 22, CS 23]. A master's program should include at least two courses from group A, two courses from group B, and one course from each of groups C, D, and E.

A. Programming Languages

CS 14 *Software Design and Development*

CS 15 *Theory of Programming Languages*

CS 19 *Compiler Construction:* An introduction to the major methods used in compiler implementation. The parsing methods of LL(k) and LR(k) are covered as well as finite state methods for lexical analysis, symbol table construction, internal forms for a program, run time storage management for block structured languages, and an introduction to code optimization. [Prerequisite: CS 8]

CS 20 *Formal Methods in Programming Languages:* Data and control abstractions are considered. Advanced control constructs including backtracking and nondeterminism are covered. The effects of formal methods for program description are explained. The major methods for proving programs correct are described. [Prerequisite: CS 8]

CS 21 *Architecture of Assemblers:* Anatomy of an assembler: source program analysis, relocatable code generation, and related topics. Organization and machine language of two or three architecturally different machines; survey and comparison of these machines in various programming environments. [Prerequisite: CS 6]

CS 25 *High Level Language Computer Architecture:* An introduction of architectures of computer systems which have been developed to make processing of programs in high level languages easier. Example systems will include SYMBOL and the Burroughs B1700. [Prerequisite: CS 6]

B. Operating Systems and Computer Architecture

CS 10 *Operating Systems and Computer Architecture II:*

CS 22 *Performance Evaluation:* A survey of techniques of modeling concurrent processes and the resources they share. Includes levels and types of system simulation, performance prediction, benchmarking and synthetic loading, hardware and software monitors. [Prerequisite: CS 6]

CS 23 *Analytical Models for Operating Systems:* An examination of the major models that have been used to study operating systems and the computer systems which they manage. Petri nets, dataflow diagrams, and other models of parallel behavior will be studied. An introduction to the fundamentals of queuing theory is included. [Prerequisite: CS 6]

CS 24 *Computer Communication Networks and Distributed Processing:* A study of networks of interacting computers. The problems, rationales, and possible solutions for both distributed processing and distributed databases will be examined. Major national and international protocols including SNA, X.21, and X.25 will be presented. [Prerequisite: CS 6]

CS 26 *Large Computer Architecture:* A study of large computer systems which have been developed to make special types of processing more efficient or reliable. Examples include pipelined machines and array processing. Tightly coupled multiprocessors will be covered. [Prerequisite: CS 6]

CS 27 *Real-Time Systems:* An introduction to the problems, concepts, and techniques involved in computer systems which must interface with external devices. These include process control systems, computer systems embedded within aircraft or automobiles, and graphics systems. The course concentrates on operating system software for these systems. [Prerequisite: CS 6]

CS 28 *Microcomputer Systems and Local Networks:* A consideration of the uses and organization of microcomputers. Typical eight or sixteen bit microprocessors will be described. Microcomputer software will be discussed and contrasted with that available for larger computers. Each student will gain hands-on experience with a microcomputer. [Prerequisite: CS 6]

C. Theoretical Computer Science

CS 13 *Algorithms*

CS 16 *Automata, Computability, and Formal Languages*

CS 29 *Applied Combinatorics and Graph Theory:* A study of combinatorial and graphical techniques for complexity analysis including generating functions, recurrence relations, Polya's theory of counting, planar directed and undirected graphs, and NP complete problems. Applications of the techniques to analysis of algorithms in graph theory and sorting and searching. [Prerequisites: CS 7 and CS 13]

CS 30 *Theory of Computation:* A survey of formal models for computation. Includes Turing Machines, partial recursive functions, recursive and recursively enumerable sets, the recursive theorem, abstract complexity theory, program schemes, and concrete complexity. [Prerequisites: CS 7 and CS 16]

D. Data and File Structures

CS 11 *Database Management Systems Design*

CS 31 *Information Systems Design:* A practical guide to Information System Programming

and Design. Theories relating to module design, module coupling, and module strength are discussed. Techniques for reducing a system's complexity are emphasized. The topics are oriented toward the experienced programmer or systems analyst. [Prerequisites: CS 6 and CS 11]

CS 32 *Information Storage and Access:* Advanced data structures, file structures, databases, and processing systems for access and maintenance. For explicitly structured data, interactions among these structures, accessing patterns, and design of processing/access systems. Data administration, processing system life cycle, system security. [Prerequisites: CS 6 and CS 11]

CS 33 *Distributed Database Systems:* A consideration of the problems and opportunities inherent in distributed databases on a network computer system. Includes file allocation, directory systems, deadlock detection and prevention, synchronization, query optimization, and fault tolerance. [Prerequisites: CS 11 and CS 24]

E. Other Topics

CS 9 *Computers and Society*

CS 12 *Artificial Intelligence*

CS 34 *Pattern Recognition:* An introduction to the problems, potential, and methods of pattern recognition through a comparative presentation of different methodologies and practical examples. Covers feature extraction methods, similarity measures, statistical classification, minimax procedures, maximum likelihood decisions, and the structure of data to ease recognition. Applications are presented in image and character recognition, chemical analysis, speech recognition, and automated medical diagnosis. [Prerequisites: CS 6 and CS 7]

CS 35 *Computer Graphics:* An overview of the hardware, software, and techniques used in computer graphics. The three types of graphics hardware: refresh, storage, and raster scan are covered as well as two-dimensional transformations, clipping windowing, display files, and input devices. If a raster scan device is available, solid area display, painting and shading are also covered. If time allows, three-dimensional graphics can be included. [Prerequisites: CS 6 and CS 7]

CS 36 *Modeling and Simulation:* A study of the construction of models which simulate real systems. The methodology of solution should include probability and distribution theory, statistical estimation and inference, the use of random variates, and validation procedures. A simulation language should be used for the solution of typical problems. [Prerequisites: CS 6 and CS 7]

CS 17 *Numerical Mathematics: Analysis*

CS 18 *Numerical Mathematics: Linear Algebra*

CS 37 *Legal and Economic Issues in Computing:* A presentation of the interactions between users of computers and the law and a consideration of the economic impacts of computers. Includes discussion of whether or not software is patentable, as well as discussion of computer crime, privacy, electronic fund transfer, and automation.

[Prerequisites: CS 9 and CS 12]

CS 38 *Introduction to Symbolic and Algebraic Manipulation:* A survey of techniques for using the computer to do algebraic manipulation. Includes techniques for symbolic differentiation and integration, extended precision arithmetic, polynomial manipulation, and an introduction to one or more symbolic manipulation systems. Automatic theorem provers are considered. [Prerequisite: CS 7]

3. IEEE Model Curriculum for Computer Science and Engineering

Within the IEEE Computer Society, the Education Committee is the body chiefly concerned with curriculum issues. This body formed a Model Curriculum Subcommittee to extend earlier curriculum efforts and to bridge the gap between software- and hardware-oriented programs. From the result fo their effort, the IEEE Model curriculum [14], we extract the statement of objectives, the top two levels of the core content outline, and short course descriptions. Substantially more detail is contined in the report proper.

3.1. Objectives

The primary objective of the effort culminated by this document is to provide model curricula for four year bachelor level degree programs in computer science and engineering (CSE). The contents of this document may be valuable both to institutions initiating new CSE programs and to those updating or evolving established CSE programs.

The second objective of this document is to provide detailed course outlines, instructional objectives, and lists of references for each recommended course.

The third objective is to identify and define a core curriculum in CSE. Accreditation requirements and institutional resources may make it impossible for many institutions to implement the entire set of courses. Therefore, a core has been identified. The core is the minimum essential set of concepts and subject material that a CSE student should master before graduation with a bachelor's degree.

The model curricula presented in this document were designed according to the following criteria:

1. *Provide breadth and depth.* To help reduce the change of technical obsolescence of the graduates in a fast changing technology, the core program provides the necessary breadth. Intensive work in one or more of the identified areas of specialization can supply depth.
2. *Bridge the gap between hardware and software.* The curricula have been designed to integrate hardware and software as well as theory and practice. This criterion is the principal objective of the recommended laboratory sequence.
3. *Be implementable.* A specific requirement was that the curricula be flexible to allow each institution to implement a CSE program after considering departmental facilities, faculty interests, regional needs, and accreditation requirements. The Regional Help Subcommittee of the Computer Society is charged with assisting implementation.

3.2. Core Curriculum Concepts

The core concepts represent the minimal set needed in a curriculum to provide a minimal background for a career in computer science and engineering.

The student who has completed the core curriculum will be professionally prepared to perform tasks spanning logic design, assembly language programming, and system analysis and will be able to apply theoretical techniques to support computer applications. Academically, the student wil lhave the necessary breadth, depth, and adaptability to pursue any specialized area of computer science and engineering. Furthermore, this core computer science and engineering curriculum has been designed to include fundamental concepts to give the student a base from which to remain current.

Digital Logic Area

Basic digital concepts and terminology
Digital fundamentals
Fundamentals of minimization (Karnaugh maps)
Combination circuits
Traditional approaches to sequential circuits
Microprocessors, microcomputers and other LSI components
Typical microprocessor instruction sets
Interfacing devices
Microcomputer system concepts
System evaluation and development aids

Computer Organization and Architecture Area

A stored program computer
Data representations
Algorithm treatment
Instruction formats
Computer units
System structure
System examples
Hardware description methodologies
Interfacing
Interrupt structures for I/O
I/O Structures
Memory hierarchy

Software Engineering Area

The computing system
Input preparations
"Guided design"/analysis of problem solving and computer programming
Practice of software engineering principles and guided design of programs
Introduction to data structures

Linear structures and list structures
Arrays
Tree structures
File systems
Data base management systems (DBMS)
Structure of simple statements
Structure of algorithmic languages
Translators
Program history
Review of batch process system program
Processor organization, multiprogramming, and multiprocessor systems
Addressing techniques
Memory organization
Parallelism in operating systems
Mutual exclusion
Synchronization
Basic functions
Techniques
Communications with peripherals--the I/O supervisor
Queue management
Memory management
Multiprocessing systems
Virtual memory and virtual machines
Batch vs. time-sharing systems
Protection
Memory management
File management
System accounting
The multiprogramming executive (MPX) operating system
Process control
Reliability
Generality
Efficiency
Complexity
Compatibility
Implementation
Modularity
Sharing

Theory of Computing Area

Propositional logic and proofs
Set theory
Algebraic structures
Groups and semigroups

Graphs
Lattices and Boolean algebra
Finite fields
Analysis of algorithms
Upper bounds analysis

3.3. Course Descriptions

Specific course descriptions are provided to cover both core and advanced courses in each of the four areas.

3.3.1. Digital Logic Subject Area

DL-1 *Switching Theory and Digital Logic I:* Basics of digital systems, languages, inter-domain conversion, information, codes, problem statements, documentation and formulation procedures, gates, axiomatic systems, minimization, combination circuits, Flip-Flops and introductory sequential circuits.

DL-2 *Switching Theory and Digital Logic II:* System controller definition, phases leading to system design, flow diagraming, specification, functional partitions, timing diagrams, interface considerations, subfunction identification, map-entered-variable techniques, MSI and LSI device utilization, introduction to microprogrammable controllers, asynchronous circuits and high speed asynchronous controller.

DL-3 *Microprocessor Systems:* A characterization of microprocessors and their use in microcomputer systems. Typical instruction sets, I/O interfacing adaptors and memory devices. Interrupts: their identification, handling and selection for servicing. System development aids: resident and cross-software. High level languages for microprocessors.

DL-4 *Digital Logic Devices:* Switching waveforms, device models, switching characteristics of diodes, bipolar and field effect transistors, saturated logic gates, DTL and TTL devices, non-saturating logic gates, memory devices, one shots, Schmitt triggers, digital application of the OP-AMP, and transmission line concepts.

DL-5 *Digital Design Automation:* The developmental considerations of a CAD system: CAD software, CAD hardware, simulation methods, emulation test analysis, evaluation study of existing CAD capabilities.

3.3.2. Computer Organization and Architecture Subject Area

CO-1 *Introduction to Computer Organization:* Stored program concept; main-line computer organization; data representation; instruction formats and instruction sets; common arithmetic and logic algorithms and their hardware implementation; addressing; timing and machine cycles; interrupts; memory and I/O devices; direct access;

hardware description methodologies and simulation.

CO-2 *I/O and Memory Systems:* Random access, semi-random access and sequential access; magnetic and solid state technologies and new developments; memory hierarchy and methods of direct access; I/O devices and their characteristics and limitations; interfacing and buffering problems; channels and I/O programming.

CO-3 *Computer Architecture:* Information representation and its impact on architectural parameters; interpretation and control structures; sequencing and execution; choice of instruction sets and addressing schemes; named and associative addressing; addressing large spaces with short addresses; formats and frequency distrubution of instructions; stack processing; memory hierarchies and I/O; protection and performance classification and examples as to processor concurrency; case studies of selected computer systems.

CO-4 *Microprogramming:* Function and implementation of the control unit; technologies supporting microprogramming; typical instruction sets and their microcode implementation; optimization and iteration of data dependent cycles; interpretation and emulation through microprogramming; measurements for performance imporvements; design trade-offs and case study of some microprogrammable machines (the B1700 or the HP2100 for example).

CO-5 *Distributing Processing and Networks:* Multiprocessors and distributed multiprocessing; concurrency and cooperation of dispersed processors, network topologies; switching, routing and control; communicaton software and protocol; examples of commercial, academic and experimental networks.

3.3.3. Software Engineering Subject Area

SE-1 *Introduction to Computing:* The primary objective of this course is to provide the student with a fundamental yet elementary background in computer science and engineering. In this course the student will (1) learn to identify and interrelate the basic functional units and components of a computer system, (2) learn basics of software engineering and applications programming through problem analysis, design, documentation, implementation, and evaluation, and (3) master a standard subset of a general purpose language. Emphasis in the last of the course is on design, algorithm development, coding, debugging, and documenting programs using techniques of good programming style.

SE-2 *Data Structures I:* This is the core course in data structures. The objective is to introduce the common data structures, operations, applications and alternate methods of data representation. Emphasis should be placed on the analysis of data structures, file organizations, and algorithms in terms of space and time, that is performance, requirements.

SE-3 *Data Structures II:* This is the second semester, or term, on data structures. The course is intended to cover a number of system applications of data structures which

will help the student to design practical language translators, operating systems, and data base management systems later.

SE-4 *Data Base Systems:* An important feature of this course lies in the evaluating of overall performances of several data base systems, the designing of a prototype data base management system, and the choosing of peripherals. The course covers the traditional approach to programs and data, and the integrated data base approach to programs and data. The student will study the important interfaces between users, data base management sysatem, access methods, and data base. The student will evaluate the basic problem of designs for fast query response versus easy updating.

SE-5 *Programming Languages:* The basic sections of this course are the structure of statements, the structure of algorithmic languages, list processing, string manipulation and text editing, array manipulation languages, types of translators, and translator writing systems.

SE-6 *Operating Systems and Computer Architecture I:* This integrated course on operating systems and computer architecture covers the structure, function and management of processors and processes, memory, files, and I/O devices. The implementation studies, in the interfacing systems design lab, cover hardware and software processes; hardware features include an interrupt mechanism, storage protection, privileged mode, and hardware relocation. A major task of an operating system is job scheduling.

SE-7 *Operating Systems and Computer Architecture II:* Topics in this course will include multiprocessor systems, stack processors, networks, file systems, and protection mechanisms. The student will study various operating systems and the techniques used within these operating systems.

SE-8 *Translators and Translator Writing Systems:* Translator writing systems are of commercial importance, and this course covers the specifications, design, writing and implementation of compilers, interpreters, and translator writing systems. Topics covered include compiler-compilers, syntax-oriented symbol processors, extendible (extensible) languages, and syntax directed translators.

3.3.4. Theory of Computing Subject Area

TC-1 *Discrete Structures:* Propositional logic and proofs, logical connectives, induction, sets, relations, unions and intersections, functions, isomorphisms and homomorphisms, groups, rings, fields, graphs, sequential machines, error correcting, codes, introduction to computabilities.

TC-2 *Design and Analysis of Algorithms:* Models of computation, Turing machines, computational techniques, upper bounds, data structure, algorithms, sorting, searching, graph isomorphism, matrix multiplications, fast Fourier transforms, Lauer bounds, NP-complete problems.

TC-3 *Automata and Formal Languages:* Formal grammars and automata, production

systems and languages, regular, context free, context sensitive and recursive grammars, deterministic and non-deterministic finite automata, context free languages, LR(k) grammars, complexity of recognition, time and tape bounded Turing machines, abstract complexity Blum measures, abstract families of languages.

TC-4 *Theory of Computation:* Algorithms, effective procedures, programming languages, algorithmic equivalence of various programming languages, Church's thesis, diagonalization, halting problem, recursive functions, recursively enumerable sets, post production systems, undecidability.

4. GRE Computer Science Test

The Educational Testing Service provides a description of the content of the Computer Science Test in the Graduate Record Examination program. The content outline as revised in 1982 [12] is given below.

I. Software Systems and Methodology 35%

- A. Data organization
  1. Abstract data types (e.g. stacks, queues, lists, strings, trees, sets)
  2. Implementations of data types (e.g. pointers, hashing, encoding, packing, address, arithmetic)
  3. File organization (e.g. sequential, indexed, multilevel)
  4. Data models (e.g. hierarchical, relational, network)
- B. Organization of program control
  1. Iteration and recursion
  2. Functions, procedures, and exception handlers
  3. Concurrent processes, interprocess communication, and synchronization
- C. Programming languages and notation
  1. Applicative versus procedural languages
  2. Control and data structure
  3. Scope, extent, and binding
  4. Parameter passing
  5. Expression evaluation
- D. Design and development
  1. Program specification
  2. Development methodologies
  3. Development tools
- E. Systems
  1. Examples (e.g. compilers, operating systems)
  2. Performance models
  3. Resource management (e.g. scheduling, storage allocation)
  4. Protection and security

II. Computer Organization and Architecture 20%

- A. Logic design
  1. Implementation of combinational and sequential circuits
  2. Functional properties of digital integrated circuits
- B. Processors and control units
  1. Instruction sets, register and ALU organization
  2. Control sequencing, register transfers, microprogramming, pipelining

C. Memories and their hierarchies
   1. Speed, capacity, cost
   2. Cache, main, secondary storage
   3. Virtual memory, paging, segmentation devices

D. I/O devices and interfaces
   1. Functional characterization, data rate, synchronization
   2. Access mechanism, interrupts

E. Interconnection
   1. Bus and switch structures
   2. Network principles and protocols
   3. Distributed resources

III. Theory 20%

A. Automata and language theory
   1. Regular languages (e.g. finite automata, nondeterministic finite automata, regular expressions)
   2. Context-free languages (e.g. notations for grammars, properties such as emptiness, ambiguity)
   3. Special classes of context-free grammars (e.g. LL, LR, precedence)
   4. Turing machines and decidability
   5. Processors for formal languages, (e.g. parsers, parser generators)

B. Correctness of programs
   1. Formal specifications and assertions (e.g. pre- and post-assertions, loop invariants, invariant relations of a data structure)
   2. Verification techniques (e.g. predicate transformers, Hoare axioms).

C. Analysis of Algorithms
   1. Exact or asymptotic analysis of the best, worst, or average case of the time and space complexity of specific algorithms
   2. Upper and lower bounds on the complexity of specific problems
   3. NP - completeness

IV. Computational mathematics 20%

A. Discrete structures: Basic elements of
   1. Abstract algebra
   2. Mathematical logic, including Boolean algebra
   3. Combinatorics
   4. Graph theory
   5. Set theory
   6. Discrete probability
   7. Recurrence relations

B. Numerical mathematics
   1. Computer arithmetic

2. Classical numerical algorithms
3. Linear algebra

V. Special topics 5%

A. Modeling and simulation
B. Information retrieval
C. Artificial intelligence
D. Computer graphics
E. Data communications

5. What Can Be Automated? (The COSERS Report)

During the period 1975-1979, the National Science Foundation sponsored the Computer Science and Engineering Research Study (COSERS), whose goal was to report on the nature and status of computer science research. The chapger and section outline of the resulting report [9] is given below. Bear in mind that the chief purpose of the study was to assess the organization of computer science research, not the established pedagogical core of the discipline.

1. COSERS Overview
2. COSERS Statistics
    - Education
    - Employment
    - Funding
    - Publication
3. Numerical Computation
    - The nature of the area
    - Why numerical computation is difficult
    - Background concepts in numerical analysis
    - Matrix computations
    - Optimization and nonlinear equations
    - Ordinary differential equations
    - Partial differential equations
    - Mathematical software
    - Curves, surfaces, and graphics
    - Research highlights of other areas
    - The research environment in numerical computation
4. Theory of Computation
    - What is theoretical computer science?
    - The complexity of numerical computations
    - Data structures and search algorithms
    - Computational complexity and computer models
    - Language and automata theory
    - The logic of computer programming
    - Mathematical semantics
    - The theoretical computer science community in the United States: status and prospects
5. Hardware Systems
    - History
    - Computer operation and construction
    - Significant advances in computer system design
    - Device technology and its impact on computers

- Calculators, microprocessors, and microcomputers
- Minicomputers and mainframe computers
- Large-scale computers
- Storage technology and memory structures
- Peripherals and terminals
- Interactive computer graphics
- Computer networks
- Computer system reliability
- Performance modeling and measurement
- Design automation
- Summary

6. Artificial Intelligence
   - Working artificial intelligence systems
   - Searching alternatives
   - Contemporary approaches to problem-solving
   - Automatic theorem proving
   - Understanding natural languages
   - Speech and visual perception
   - Speech perception
   - Vision systems
   - Productivity technology
   - Applying AI methods
   - Conclusions

7. Special Topics
   - Algebraic manipulation
   - Applying algebraic computation programs
   - Computational linguistics
   - Pattern-recognition and image processing

8. Programming Languages
   - The general goals of programming language design
   - Major general-purpose languages
   - Very high level languages
   - Systems implementation languages
   - Special purpose languages
   - Operating system languages
   - Language description theory
   - Language implementation techniques
   - Global program analysis and optimization
   - Program verification
   - Programming language extensibility

9. Operating Systems
   - Processes

6. Encyclopedia of Computer Science

The classification of articles in the *Encyclopedia of Computer Science* [24] embodies a taxonomy that should be helpful to the reader in grasping the scope of material contained in the encyclopedia. The major classifications from that taxonomy are presented here.

Articles in the encyclopedia are classified under nine categories:

1. Hardware
2. Computer Systems
3. Information and Data
4. Software
5. Mathematics of Computing
6. Theory of Computing
7. Methodologies
8. Applications
9. Computing Milieux

Except for a minor variation in the name of category 3 -- "Information and Data" rather than just "Data" -- these are the categories used in the *Taxonomy of Computer Science and Engineering* [6]. Articles in the encyclopedia are listed in this classification in a way patterned after the *Taxonomy*.

I. Hardware
- Types of Computers
- Computer Architecture
- Computer Circuitry
- Digital Computer Subsystems
- Hardware Description Languages
- Maintenance of Computers
- Reliability, Hardware

II. Computer Systems
- Structure-Based Systems
- Access-Based Systems
- Special Purpose Computers

III. Information and Data
- Codes
- Data Bank
- Data Communications Systems
- Data Communications: Principles
- Data Communications: Software

- Data Compression and Compaction
- Data Definition Languages (DDL)
- Data Encryption
- Data Management
- Data Security
- Data Structures

IV. Software
- Applications Programming
- Machine and Assembly Language Programming
- Operating Systems
- Procedure-Oriented Languages: Programming In
- Program Architecture
- Programming Languages
- Programming Language Semantics
- Programming Linguistics
- Software Complexity
- Software Engineering
- Software Flexibility
- Software History
- Software Maintenance
- Software Management
- Software Packages
- Software Reliability
- Software Science
- Systems Programming

V. Mathematics of Computing
- Discrete Mathematics
- Numerical Analysis

VI. Theory of Computation
- Algorithm
- Algorithms, Theory of
- Automata Theory
- Formal Languages
- Lambda Calculus
- Logics of Programs
- Petri Nets

VII. Methodologies
- Algebraic Manipulation
- Artificial Intelligence
- Computer Graphics
- Database Management
- Image Processing

Information Retrieval
Information Systems
Mathematical Software
Operations Research
Pattern Recognition
Simulation
Sorting

VIII. Applications
Administrative Applications
Computer-Assisted Learning and Teaching
Engineering Applications
Humanities Applications
Library Automation
Medical Applications
Publishing, Computers in
Scientific Applications
Social Science Applications
Word Processing

IX. Computing Milieux
The Computer Industry
Computer Science and Technology
Computing and Society
The Computing Profession
Education in Computer Science and Technology
History
Legal Aspects of Computing
Literature in Computing
Management of Computing

7. IBM Systems Research Institute Curriculum

The mission of the IBM Systems Research Institute is to prepare IBM systems professionals involved in design development, and marketing of information systems for the future. One of its major programs is a 10-week instructional program designed to provide education in depth in certain subjects. The resulting program resembles a university curriculum in many respects; the main organization of that program is given here. These descriptions are taken from the SRI catalog of September 1982 [13].

- *Data Communications and Networking:* These courses deal with the components, technologies and characteristics of communication based systems and the techniques and tools used to plan, design, implement, and manage them. Also covered are some current systems, problem areas and future directions.
- *Data Base and Information Systems:* These courses focus on the design and use of systems that manage data as a shared resource of an organization. They include concepts and techniques for data representation and data manipulation as well as tools and techniques used to design systems that process data efficiently.
- *Disciplines and Techniques of Systems Science:* These courses present some of the formal disciplines and theoretical foundations upon which the field of computer and information systems is based.
- *Systems Development and Management:* These courses deal with the analysis, design, implementation and management of information systems. They cover concepts, tools and techniques, as well as new issues and problem areas.
- *Systems Architecture and Technology:* These courses deal with the complementary aspects of hardware, microcode, and programming architectures. This includes the evolution of present architectures as well as the interaction of programming technology, hardware technology, application requirements and manufacturing capabilities with future architectural development.
- *Human Factors and Interfaces:* These courses deal with the man-machine interface, and with the nature and psychology of man as they affect interpersonal communication and the design of systems which are easy to learn and use.
- *The Business Environment:* These courses address the basic areas of economics, accounting, finance, and quantitative methods; survey the IBM development, financial, international, and legal environment; and examine the past, present and future of the computer industry.

8. Computing Reviews Classifications

A revision of the standard categories for Computing Reviews classifications was prepared in 1981 [10]. The outline of topics is given here.

A. General Literature
- A.0 General
- A.1 Introduction and Survey
- A.2 Reference

B. Hardware
- B.0 General
- B.1 Control Structures and Microprogramming
  - B.1.0 General
  - B.1.1 Control Design Styles
  - B.1.2 Control Structure Performance Analysis and Design Aids
  - B.1.3 Control Structure Reliability, Testing and Fault-Tolerance
  - B.1.4 Microprogram Design Aids
  - B.1.5 Microcode Applications
- B.2 Arithmetic and Logic Structures
  - B.2.0 General
  - B.2.1 Design Styles
  - B.2.2 Performance Analysis and Design Aids
  - B.2.3 Reliability, Testing and Fault-Tolerance
- B.3 Memory Structures
  - B.3.1 General
  - B.3.2 Design Styles
  - B.3.3 Performance Analysis and Design Aids
  - B.3.4 Reliability, Testing and Fault-Tolerance
- B.4 Input/Output and Data Communications
  - B.4.0 General
  - B.4.1 Data Communications Devices
  - B.4.2 Input/Output Devices
  - B.4.3 Interconnections (subsystems)
  - B.4.4 Performance Analysis and Design Aids
  - B.4.5 Reliability, Testing and Fault-Tolerance
- B.5 Register-Transfer-Level Implementation
  - B.5.0 General
  - B.5.1 Design
  - B.5.2 Design Aids
  - B.5.3 Reliability and Testing
- B.6 Logic Design
  - B.6.0 General

- B.6.1 Design Styles
- B.6.2 Reliability and Testing
- B.6.3 Design Aids
- B.7 Integrated Circuits
  - B.7.0 General
  - B.7.1 Types and Design Styles
  - B.7.2 Design Aids
  - B.7.3 Reliability and Testing

C. Computer Systems Organization
- C.0 General
- C.1 Processor Architectures
  - C.1.0 General
  - C.1.1 Single Data Stream Architectures
  - C.1.2 Multiple Data Stream Architectures (Multiprocessors)
  - C.1.3 Other Architecture Styles
- C.2 Computer-Communication Networks
  - C.2.0 General
  - C.2.1 Network Architecture and Design
  - C.2.2 Network Protocols
  - C.2.3 Network Operations
  - C.2.4 Distributed Systems
  - C.2.5 Local Networks
- C.3 Special-Purpose and Application-Based System
- C.4 Performance of Systems

D. Software
- D.0 General
- D.1 Programming Techniques
  - D.1.0 General
  - D.1.1 Applicative (Functional) Programming
  - D.1.2 Automatic Programming
  - D.1.3 Concurrent Programming
  - D.1.4 Sequential Programming
- D.2 Software Engineering
  - D.2.0 General
  - D.2.1 Requirements/Specifications
  - D.2.2 Tools and Techniques
  - D.2.3 Coding
  - D.2.4 Program Verification
  - D.2.5 Testing and Debugging
  - D.2.6 Programming Environments
  - D.2.7 Distribution and Maintenance
  - D.2.8 Metrics

- D.2.9 Management
- D.3 Programming Languages
  - D.3.0 General
  - D.3.1 Formal Definitions and Theory
  - D.3.2 Language Classifications
  - D.3.3 Language Constructs
  - D.3.4 Processors
- D.4 Operating Systems
  - D.4.0 General
  - D.4.1 Process Management
  - D.4.2 Storage Management
  - D.4.3 File Systems Management
  - D.4.4 Communications Management
  - D.4.5 Reliability
  - D.4.6 Security and Protection
  - D.4.7 Organization and Design
  - D.4.8 Performance
  - D.4.9 Systems Programs and Utilities

E. Data
- E.0 General
- E.1 Data Structures
- E.2 Data Storage Representations
- E.3 Data Encryption
- E.4 Coding and Information Theory

F. Theory of Computation
- F.0 General
- F.1 Computation by Abstract Devices
  - F.1.0 General
  - F.1.1 Models of Computation
  - F.1.2 Modes of Computation
  - F.1.3 Complexity Class
- F.2 Analysis of Algorithms and Problem Complexity
  - F.2.0 General
  - F.2.1 Numerical Algorithms and Problems
  - F.2.2 Nonnumerical Algorithms and Problems
  - F.2.3 Tradeoffs Among Complexity Measures
- F.3 Logics and Meanings of Programs
  - F.3.0 General
  - F.3.1 Specifying and Verifying and Reasoning about Programs
  - F.3.2 Semantics of Programming Languages
  - F.3.3 Studies of Program Constructs
- F.4 Mathematical Logic and Formal Languages

- F.4.0 General
- F.4.1 Mathematical Logic
- F.4.2 Grammars and Other Rewriting Systems
- F.4.3 Formal Languages

G. Mathematics of Computing
- G.0 General
- G.1 Numerical Analysis
  - G.1.0 General
  - G.1.1 Interpolation
  - G.1.2 Approximation
  - G.1.3 Numerical Linear Algebra
  - G.1.4 Quadrature and Numerical Differentiation
  - G.1.5 Roots of Nonlinear Equations
  - G.1.6 Optimization
  - G.1.7 Ordinary Differential Equations
  - G.1.8 Partial Differential Equations
  - G.1.9 Integral Equations
- G.2 Discrete Mathematics
  - G.2.0 General
  - G.2.1 Combinatorics
  - G.2.2 Graph Theory
- G.3 Probability and Statistics
- G.4 Mathematical Software

H. Information Systems
- H.0 General
- H.1 Models and Principles
  - H.1.0 General
  - H.1.1 Systems and Information Theory
  - H.1.2 User/Machine Systems
- H.2 Database Management
  - H.2.0 General
  - H.2.1 Logical Design
  - H.2.2 Physical Design
  - H.2.3 Languages
  - H.2.4 Systems
  - H.2.5 Heterogeneous Databases
  - H.2.6 Database Machines
  - H.2.7 Database Administration
- H.3 Information Storage and Retrieval
  - H.3.0 General
  - H.3.1 Content Analysis and Indexing
  - H.3.2 Information Storage

- H.3.3 Information Search and Retrieval
- H.3.4 Systems and Software
- H.3.5 On-Line Information Services
- H.3.6 Library Automation

H.4 Information Systems Applications

- H.4.0 General
- H.4.1 Office Automation
- H.4.2 Types of Systems
- H.4.3 Communications Applications

I. Computing Methodologies

I.0 General

I.1 Algebraic Manipulation

- I.1.0 General
- I.1.1 Expressions and Their Representation
- I.1.2 Algorithms
- I.1.3 Languages and Systems
- I.1.4 Applications

I.2 Artificial Intelligence

- I.2.0 General
- I.2.1 Applications and Expert Systems
- I.2.2 Automatic Programming
- I.2.3 Deduction and Theorem Proving
- I.2.4 Knowledge Representation Formalisms and Methods
- I.2.5 Programming Languages and Software
- I.2.6 Learning
- I.2.7 Natural Language Processing
- I.2.8 Problem Solving, Control Methods and Search
- I.2.9 Robotics
- I.2.10 Vision and Scene Understanding

I.3 Computer Graphics

- I.3.0 General
- I.3.1 Hardware Architecture
- I.3.2 Graphics Systems
- I.3.3 Picture/Image Generation
- I.3.4 Graphics Utilities
- I.3.5 Computational Geometry and Object Modeling
- I.3.6 Methodology and Techniques
- I.3.7 Three-Dimensional Graphics and Realism

I.4 Image Processing

- I.4.0 General
- I.4.1 Digitization
- I.4.2 Compression (coding)
- I.4.3 Enhancement

- I.4.4 Restoration
- I.4.5 Reconstruction
- I.4.6 Segmentation
- I.4.7 Feature Measurement
- I.4.8 Scene Analysis
- I.4.9 Applications

I.5 Pattern Recognition

- I.5.0 General
- I.5.1 Models
- I.5.2 Design Methodology
- I.5.3 Clustering
- I.5.4 Applications
- I.5.5 Implementation

I.6 Simulation and Modeling

- I.6.0 General
- I.6.1 Simulation Theory
- I.6.2 Simulation Languages
- I.6.3 Applications
- I.6.4 Model Validation and Analysis

I.7 Text Processing

- I.7.0 General
- I.7.1 Text Editing
- I.7.2 Document Preparation
- I.7.3 Index Generation

J. Computer Applications

- J.0 General
- J.1 Administrative Data Processing
- J.2 Physical Sciences and Engineering
- J.3 Life and Medical Sciences
- J.4 Social and Behavioral Sciences
- J.5 Arts and Humanities
- J.6 Computer-Aided Engineering
- J.7 Computers in Other Systems

K. Computing Milieux

- K.0 General
- K.1 The Computer Industry
- K.2 History of Computing
- K.3 Computers and Education
  - K.3.0 General
  - K.3.1 Computer Uses in Education
  - K.3.2 Computer and Information Science Education
- K.4 Computers and Society

- K.4.0 General
- K.4.1 Public Policy Issues
- K.4.2 Social Issues
- K.4.3 Organizational Impacts

K.5 Legal Aspects of Computing

- K.5.0 General
- K.5.1 Software Protection
- K.5.2 Governmental Issues

K.6 Management of Computing and Information Systems

- K.6.0 General
- K.6.1 Project and People Management
- K.6.2 Installation Management
- K.6.3 Software Management
- K.6.4 System Management

K.7 The Computing Profession

- K.7.0 General
- K.7.1 Occupations
- K.7.2 Organizations
- K.7.3 Testing, Certification, and Licensing

K.8 Personal Computing

Bibliography

1. ACM Committee on the Undergraduate Program in Mathematics. A General Curriculum in Mathematics for Colleges. Rep. to Math. Assoc. of America, CUPM.

2. ACM Curriculum Committee on Computer Science. "Curriculum 68: Recommendations for Academic Programs in Computer Science." *Communications of the ACM 11*, 3 (March 1968), 151-197.

3. ACM Curriculum Committee on Computer Science. "Curriculum Recommendations for the Undergraduate Program in Computer Science." *SIGCSE Bulletin (ACM) 9*, 2 (June 1977), 1-16.

4. ACM Curriculum Committee on Computer Science. "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science." *Communications of the ACM 22*, 3 (March 1979), 147-166.

5. ACM Curriculum Committee on Computer Science. "Recommendations for Master's Level Programs in Computer Science." *Communications of the ACM 24*, 3 (March 1981), 115-123.

6. AFIPS Taxonomy Committee. *Taxonomy of Computer Science & Engineering.* American Federation of Information Processing Societies, Inc., 1980.

7. CMU Graduate School of IndustrialAdministration. Announcements for 1954-1956. Pittsburgh, PA, 1954.

8. Carnegie-Mellon University. Carnegie-Mellon University Undergraduate Catalogue 1981-1983. Pittsburgh, PA, 1980.

9. Bruce W. Arden (ed.). *What Can Be Automated? The Computer Science and Engineering Research Study (COSERS).* MIT Press, 1981.

10. Jean E. Sammet and Anthony Ralston. "The New (1982) *Computing Reviews* Classification System - Final Version." *Communications of the ACM 25*, 1 (January 1982).

11. Robert E. Doherty. The Development of Professional Education. CMU, Carnegie Press.

12. Educational Testing Service. A Description of the Computer Science Test, 1982-84. Princeton, NJ, 1982. Information Booklet for Graduate Record Examination Computer Science Test.

13. IBM Systems Research Institute. SRI Class 69 Catalog. New York, NY, 1982.

14. Education Committee (Model Curriculum Subcommittee) of the IEEE Computer Society. A Curriculum in Computer Science and Engineering. IEEE Computer Society, November, 1976. Committee Report

15. Donald E. Knuth. "Computer Science and Its Relation to Mathematics." *American Mathematical Monthly 81*, 4 (April 1974).

16. letters to the editor. "Comments on the Mathematical Content of Curriculum '78." *Communications of the ACM 23*, 6 (June 1960), 356-359.

17. C.L. Liu. *Elements of Discrete Mathematics.* McGraw-Hill, 1977.

18. Jack Lochhead. Math for Physics. In *The Future of College Mathematics*, Anthony Ralston and Gail Young, Eds., Springer-Verlag, to appear 1983.

19. National Science Foundation and the Department of Education. Science and Engineering: Education for the 1980's and Beyond. U.S. Government Printing Office, Washington, D.C.

20. Frank W. Paul, Donald L. Feucht, B.R. Teare, Jr., Charles P. Neuman and David Tuma. Analysis, Synthesis and Evaluation -- Adventures in Professional Engineering Problem Solving. Proceedings of the Fifth Annual Frontiers in Education Conference, IEEE and the Amer. Soc. for Engr. Ed., October, 1975, pp. 244-251.

21. George Polya. *How to Solve It.* Princeton University Press, 1973.

22. Anthony Ralston and Mary Shaw. "Curriculum '78 -- Is Computer Science Really that Unmathematical?" *Communications of the ACM 23*, 2 (February 1980), 67-70.

23. Anthony Ralston. "Computer Science, Mathematics, and the Undergraduate Curricula in Both." *American Mathematical Monthly 88*, 7 (1981).

24. Anthony Ralston and Edwin D. Reilly, Jr.. *Encyclopedia of Computer Science and Engineering.* Van Nostrand Reinhold, 135 W. 50th Street, New York, NY, 1983. Second Edition

25. Moshe F. Rubinstein. *Patterns of Problem Solving.* Prentice-Hall, Inc., 1975.

26. W.L. Scherlis and M.Shaw. Mathematics Curriculum and the Needs of Computer Science. In *The Future of College Mathematics*, Anthony Ralston and Gail Young, Eds., Springer-Verlag, to appear 1983.

27. Mary Shaw, Stephen Brookes, Bill Scherlis, Alfred Spector, and Guy Steele. Plan for Developing an Undergraduate Computer Science Curriculum. CMU CS Curriculum Design Note 82-02.

28. D.F. Stanat and D.F. McAlister. *Discrete Mathematics in Computer Science.* Prentice-Hall, Inc., 1977.

29. Lynn Arthur Steen. Developing Mathematical Maturity. In *The Future of College Mathematics*, Anthony Ralston and Gail Young, Eds., Springer-Verlag, to appear 1983.

30. J.P. Tremblay and R.P. Manohar. *Discrete Mathematical Structures With Applications to Computer Science.* McGraw-Hill, 1975.

31. D.T. Tuma and F. Reif. *Problem Solving and Education: Issues in Teaching and Research.* Lawrence Erlbaum Associates, 1980.

32. Wayne A. Wickelgren. *How to Solve Problems.* W.H. Freeman and Company, 1974.