A MODEL CURRICULUM FOR A LIBERAL ARTS DEGREE IN COMPUTER SCIENCE

This report proposes developing a rigorous undergraduate curriculum for a B.A.-degree program in computer science. The curriculum is intended as a model not only for high-quality undergraduate colleges and universities, but also for larger universities with strong computer science programs in a liberal arts setting.

NORMAN E. GIBBS and ALLEN B. TUCKER

Traditionally, the B.A. degree is a strong degree, distinguished by its emphasis on preparation for a lifetime of learning and encouragement of academic breadth without regard to traditional academic divisions. In lieu of focusing mostly on courses in a single scientific or engineering discipline, students are required to study a variety of subjects in the humanities and social sciences. They are expected to develop substantial analytical and communication skills, to gain a sense of social and ethical values, and to fully appreciate the application of these values in contemporary and classical societies.

Institutions that offer the B.A. degree typically limit the number of courses required for any major, including computer science, to slightly more than one year in order to allow students to pursue diverse four-year programs. Through a combination of distribution requirements and advising, this usually amounts to a minimum of two years of course work in the humanities and social sciences for a science major. Within the liberal arts setting, students normally declare majors at the end of the sophomore year, but it is not unusual to see changes as late as the junior year, or for students to combine a computer science major with a minor in a very different field such as economics, history, or English.

Recent efforts to revise the definition of computer science as an academic major have not fully addressed the curricular needs of liberal arts programs. Although this article does not pretend to fully explicate the rationale and substance of a liberal arts degree, the Model Curriculum for a B.A. degree in computer science is presented in the context of its most typical setting: liberal arts institutions that maintain high standards of academic quality in their curricula.

This work was supported by a grant from the Alfred P. Sloan Foundation.

Norman E. Gibbs is currently on leave at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

CURRICULA IN COMPUTER SCIENCE: A BRIEF REVIEW

Academic programs in computer science evolved in the 1950s from two directions: On the one hand, college and university computer centers offered short noncredit courses in programming for students in the scientific disciplines. On the other hand, a small number of graduate programs in computer science emerged in selected large universities with significant research interests in computing. For various reasons, the Ph.D. and M.S. programs in computer science evolved more quickly than did the undergraduate programs.

In 1968, ACM published its first undergraduate curriculum standard in computer science, known as "Curriculum 68" [1]. Subsequently, the growth and diversity of academic programs in computer science have spread rapidly and at all levels. A second ACM curriculum standard, "Curriculum 78" [2], was developed and published a decade later. Several other, more specialized curriculum standards have been drafted in the past 15 years [3, 6]; indeed, the frequency of curriculum revision in computer science is a reflection of its newness and rapid evolution as an academic discipline.

Recently, the "Curriculum 78" standard came under serious scrutiny. Many feel that it is time to reform the introductory course sequence CS1–CS2 [11, 12, 16], while others have seriously challenged the standard's

mathematics requirements for computer science majors [18]. The core curriculum in computer science is frequently questioned because it seems to be composed of a collection of different programming and applications courses and fails to explicate adequately the principles that underlie the discipline. The less generous among our colleagues in mathematics and the natural sciences have chided computer scientists by observing that computer science is the only science whose applications comprise the core curriculum, and whose principles and theories are left for the senior electives. Although this is a clear overstatement of the case, a legitimate criticism of "Curriculum 78" is that it does not prevent such degree programs from being sanctioned.

In our view, these concerns reflect a fundamental problem with the current definition of computer science as an academic discipline. In short, the standard set by "Curriculum 78" has become obsolete as a guiding light for maintaining contemporary high-quality undergraduate degree programs and cannot serve as a basis for developing a new degree program in computer science within a liberal arts setting.

Recent efforts have been made to shore up this problem indirectly. The Computer Science Accreditation Board (CSAB) has attempted, in a move unrelated to "Curriculum 78," to modify the existing standard by specifying minimal numbers of courses, computer science faculty, research and laboratory equipment, and so forth [14]. Carnegie-Mellon University, on the other hand, has drafted a 198-page document that describes an undergraduate computer science curriculum design in exhaustive detail [20]. This curriculum includes nearly all courses that could reasonably appear in an undergraduate catalog under the heading of computer science.

Although they represent substantial efforts on the part of their creators, neither of these approaches seems to adequately or appropriately address deficiencies in the current ACM curriculum standard as it applies to liberal arts institutions. The CSAB accreditation standard has been widely criticized since its inception, both for its inflexibility and for its strong bias toward a professional engineering education [8]. The Carnegie-Mellon document speaks to a "liberal professional education" with a significant engineering point of view. Neither effort addresses the curricular needs of liberal arts programs whose purpose is to prepare students for a lifelong career of learning. These needs prompted several concerned computer scientists from liberal arts colleges to seek funding in 1984 from the Alfred P. Sloan Foundation to convene a series of workshops to address the problem (see sidebar).

COMPUTER SCIENCE *IS* SCIENCE

We seek to define computer science as a coherent body of scientific principles that will continue to guide the

Model Curriculum Workshops

The first workshop dealing with a "Model Curriculum" was held at Bowdoin College on October 19–20, 1984, and was attended by the following persons:

Kim Bruce, Williams College;
Robert Cupper, Allegheny College;
Norman Gibbs (convener), Bowdoin College;
Stuart Hirshfield, Hamilton College;
Nancy Ide, Vassar College;
Charles Kelemen, Swarthmore College;
Jeffrey Parker, Amherst College;
Ted Sjoerdsma, Washington and Lee University;
Allen Tucker, Colgate University.

This session was unusually productive due to the active and thoughtful participation of these participants, and the strong common feeling that a comprehensive review of computer science curriculum standards was severely needed.

The result of this session was a "Draft Curriculum for High-Quality B.A. Programs in Computer Science," an unpublished manuscript that was nevertheless widely distributed and discussed during the spring of 1985. It was presented at an ACM SIGCSE panel session in New Orleans in March, and written suggestions from 25 colleagues were received in response.

The second workshop convened at Colgate University on June 28–29, 1985, to revise this draft and consider several future activities in which the group might work in consortium. The original nine participants were joined by the following persons:

James Bradley, Nazareth College;
Joyce Brennan, University of Texas at Austin;
James Cameron, Denison University;
Henry Walker, Grinnell College.

These representatives were included to add geographic and institutional diversity to the list of participating colleges and universities while maintaining high standards of academic quality. Thanks to the collective efforts of the workshop participants, combined with numerous constructive written suggestions, the Draft Curriculum has been fully revised and appears herein as a Model Curriculum. The 13 institutions represented here are in different stages ranging from initial design to full implementation of a computer science major consistent with this model.

Although this Model Curriculum addresses the computer science major, it explicitly does not deal with computer literacy, the computer science minor, and interdisciplinary programs involving computer science as a component. While these are important subjects, they are outside the scope of our work at the present time. It also does not attempt to speak for the interests of institutions or programs offering professional training in computer science.

discipline for the next decade or two, rather than allow it to be driven by the needs and priorities of particular technologies. Such a definition should naturally result in a curriculum standard that mandates that the teaching of these principles takes precedence over the teaching of specific applications and technologies.

In defining *computer science*, we should be able to distinguish it from *computer engineering*, just as *chemistry* is distinguished from *chemical engineering*, and *physics* from *mechanical and electrical engineering*. The following definition of computer science allows for this distinction:

Computer science is the systematic study of algorithms and data structures, specifically

(1) their formal properties,

(2) their mechanical and linguistic realizations, and

(3) their applications.

Before describing the curriculum that follows from this definition, it is important to reaffirm the essential ordering of emphasis among the three components listed above. The *formal properties* (1) of algorithms and data structures must be emphasized over their specific machines and languages (2), as well as their applications (3), in order for a program to be legitimately called *computer science*. If (2) takes precedence over (1), then the program might be called *computer engineering*; if (3) takes precedence, the program might be called *information systems*. (Neither of these alternatives is further considered in this report.)

Thus, we support the recent ruminations of Peter Denning, who observes that the current model of the computer science core curriculum has "come to the end of its useful life" [7]. Denning points out that computer scientists project to colleagues in other scientific disciplines the "illusion that we are mostly technicians and that our field has nowhere the same intellectual depth as the physical sciences or engineering" [7]. In the following sections, we will discuss the core curriculum and other courses that would constitute an undergraduate major in computer science, guided by the principles and priorities of the above definition.

AN UNDERGRADUATE CURRICULUM IN COMPUTER SCIENCE: OVERVIEW

This curriculum is designed principally as a guideline for liberal arts institutions that are already offering or anticipate developing a B.A.-degree program in computer science. The institutions are assumed to offer this program both because they view computer science as an essential discipline within their general academic mission, and because they see the major as preparation for a variety of career paths and graduate programs (inside and outside the computer science community). This notion is shared by many different institutions: large and small, public and private, and regionally diverse. It is not the exclusive view of small, private, liberal arts colleges in the northeast. Its essential ingredients are the liberal arts philosophy and the expectation that students will meet high standards.

The curriculum is divided into three major parts shown in Figure 1. In this article, computer science degree requirements, sample concentration programs, faculty, staff, and laboratory requirements are discussed in the context of the Model Curriculum depicted in Figure 1. The "Introductory" and "Core" parts of this design emphasize the teaching of principles and theory, while the "Electives" part emphasizes applications. Thus, the scientific dimensions of computer science should be well understood by students as they complete the core curriculum.

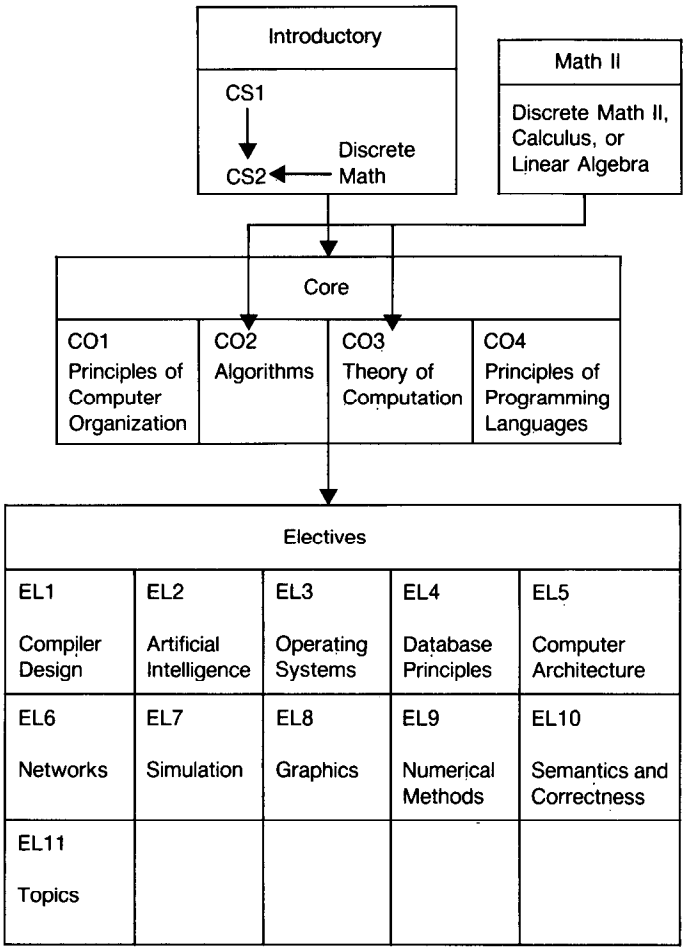
It is important to emphasize here that mathematics requirements *are not separate* from this curriculum—many of the central issues in computer science are best understood from a theoretical perspective. Such a perspective requires facility with appropriate mathematical tools as well as a certain level of general mathematical maturity. The concepts, principles, and methods of mathematics are thus integrated throughout the introductory and core courses.

Specifically, a minimum of two mathematics courses is required at the introductory level: a first course in discrete mathematics and a second course in discrete mathematics, calculus, or linear algebra. The first discrete mathematics course is a corequisite for CS2 and a prerequisite for all courses in the Core. The second course provides additional mathematical maturity and is a strict prerequisite for courses CO2 and CO3 in the Core. The first discrete mathematics course may be taught by the mathematics department or the computer science department depending on the institution. What is important is that a strong working relationship exists between the mathematics and computer science faculties in order for this curriculum to be properly supported. In smaller institutions, this relationship is essential.

THE INTRODUCTORY COURSES: CS1 AND CS2

The success of a computer science curriculum greatly depends on the quality of its introductory sequence. Recently, much interest has been paid to a student's initial exposure to computer science. Our proposals for CS1 closely follow the proposals of the ACM Curriculum Committee Task Force for CS1 [11], whereas those for CS2 differ somewhat from the report of the ACM Curriculum Committee Task Force for CS2 [12].

The content and approach to CS1 and CS2 tend to shape the rest of the curriculum. Ralston [17] discusses two models for a first course in computer science. We agree with his preference for the second model in which ". . . the programming languages are a vehicle for introducing the student to the discipline of computer science; the emphasis is on algorithms (and their analysis and even verification), programming methodology, data structures (particularly in the two-semester sequence), and so on." It is important to point out here that CS1 need not be the introductory course for all



The arrows connecting the boxes in Figure 1 indicate necessary prerequisites, and by implication, the absence of arrows indicates the absence of prerequisites. Thus, CS1, CS2, and Discrete Math must be completed before the Core is begun. The second mathematics course is not prerequisite for CO1 and CO4, although it is to CO2 and CO3. The entire Core should be completed before any electives are taken. There is no prerequisite structure within the Core or the electives.

Although this curriculum is a collection of one-semester courses, certain courses have significant laboratory requirements and therefore should be considered as four-credit rather than three-credit courses. This laboratory component is considered essential for CS1, CS2, and CO1. It may also be desirable for CO4 and many of the electives as well, depending on how these courses are taught. However, the two mathematics courses and the core courses CO2 and CO3 are essentially theoretically oriented and in many implementations may not have a laboratory component.

FIGURE 1. Overview of the Model Curriculum

students who wish to learn programming; the goals of a service or computer literacy course are often quite different from those of a course designed for computer science majors and minors.

The ACM Curriculum Committee Task Force for CS1 recommends the following objectives:

- to introduce a disciplined approach to problem-solving methods and algorithm development;
- to introduce procedural and data abstraction;
- to teach program design, coding, debugging, testing, and documentation using good programming style;
- to teach a block-structured high-level programming language;
- to provide a familiarity with the evolution of computer hardware and software technology;
- to provide a foundation for further studies in computer science.

We agree wholeheartedly with this approach and the detailed course coverage in the committee's report [11]. We do, however, postpone the hardware and software technology element until CS2.

The fine line between CS1 and CS2 is difficult to

determine and will no doubt vary among institutions. Language features covered in CS1 should include all control structures and built-in data structures, with the possible exception of pointer variables if time does not permit. Recursion should also be given at least cursory treatment even though full understanding might not take place until CS2. A great deal of emphasis should be placed on the ideas of procedural abstraction, top-down design, and correct programming style—students' programming styles are formed in this introductory sequence, and it is far better for them to learn a rigid style in these first courses than a sloppy one that would later have to be reformed.

It is important that some ideas (like recursion) be at least introduced in CS1 to give students some indication of what will come in later courses. This introduction should include such ideas as divide-and-conquer algorithms, algorithm complexity, program invariants, and program verification. It is far more useful, for instance, to talk about binary search as a form of a divide-and-conquer algorithm and to explore its complexity even briefly than to leave the impression that binary search is simply a clever programming trick in isolation. Moreover, the idea that mathematical induction can serve as a basis for understanding recursion (or vice versa) is something that students will find helpful in their subsequent studies. Although these concepts do not play a central role in CS1, they prepare students for more detailed study in CS2 and later courses.

We believe that CS2 should have four principal themes:

- to consolidate the knowledge of algorithm design and programming gained in CS1, especially emphasizing the design and implementation of large programs;
- to begin a detailed study of data structures and data abstraction as exemplified by packages or modules;
- to introduce the uses of mathematical tools such as complexity (O-notation) and program verification;
- to provide an overview of computer science.

The first three themes should be interweaved throughout the course, but it may be necessary to cover the last one separately. For example, parallels should continuously be drawn between procedural abstraction and data abstraction, and between recursive procedures and recursive data structures (trees, stacks, etc.). Students should be introduced early on to the regular simple analysis of algorithms that will then increase in sophistication as the course progresses. Students should therefore begin to evaluate their programs in terms of good style, design, and efficiency. Program verification should be introduced, especially as a way of ensuring the correctness of recursive and iterative solutions.

The fourth item listed above reflects our belief that computer scientists have neglected to give an overview of their field in introductory courses. Including such an overview in CS1 and CS2 will provide a solid foundation in computer science and give a better idea of the nature of the field to both majors and nonmajors.

Recommended topics for CS2 are outlined as follows:

- data structures and their representations; data abstraction, internal representation, lists, stacks, queues, trees, graphs, and their applications;
- searching and sorting; algorithms and data structures for internal and external searching and sorting, with emphasis on correctness and efficiency;
- unsolvable problems; brief treatment of the halting problem (in a high-level language);
- systems software; a brief survey of simple architecture and assembly language; discussion of assemblers, compilers, operating systems, etc., with emphasis on important algorithms and data structures (e.g., parse trees and symbol tables for compilers).

Since the last recommendation is not usually a part of CS2, it is important to clarify its purpose: This is designed to give a brief survey (one to two weeks) of the principles behind simple computer architecture and systems software. This should not be a series of lectures on the architecture and particular systems software of the local computer; instead, the idea is to use this survey to pull together many of the data structures (including abstraction) used in CS2. As such it could either be covered in the last few lectures or spread throughout the semester as the appropriate data structures are covered. Unfortunately, many current texts for CS2 that emphasize data structures do not cover this material at the level envisioned here. Possible sources for this survey material include *Computer Science: A Modern Introduction* by Goldschlager and Lister (Chapters 4 and 5) [9], and *Fundamental Concepts of Programming Systems* by Ullman (Chapters 2, 4, and 9) [23].

The coverage of unsolvability in CS2 should require only a single class. The above combination of topics should give students a solid introduction to data structures (including abstraction), as well as a taste for the material in courses like architecture, algorithms (or advanced data structures), compilers, operating systems, and the theory of computation.

Finally, to do justice to the concept of procedural and data abstraction, it will be helpful for students to study a language that directly supports this concept (e.g., Modula-2, Ada,® or CLU). Thus, it may be necessary here to introduce a different language from the one learned in CS1. (This also reinforces the notion that it is the algorithm that is important, and that it can be coded in almost any programming language.) Another approach would be to introduce CS2 students to a functional programming language (such as Lisp) if they have had no previous exposure to it in CS1. In either shift, however, care must be taken not to divert the main purpose of CS2; it must not be construed as simply a course for learning a second language.

THE INTRODUCTORY MATHEMATICS COURSES

Mathematics plays an essential role in the development of computer science—not only in the particular knowl-

Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

edge that is required to understand computer science, but also in the reasoning skills associated with mathematical maturity. Thus, the mathematics requirement is designed to provide both of these aspects through a one-semester discrete mathematics course plus an additional one-semester course in either discrete mathematics, calculus, or linear algebra.

As previously mentioned, the discrete mathematics course plays an important role in the computer science curriculum. (Ralston [17] and others have argued this point at length, and so these arguments will not be repeated here.) We recommend that discrete mathematics be either a prerequisite or corequisite for CS2. This early positioning of discrete mathematics reinforces the fact that computer science is not just programming—there is substantial mathematical content throughout the discipline. Moreover, this course should have significant theoretical content and be taught at a level appropriate for freshman mathematics majors. Proofs will be an essential part of the course. The following topics are as relevant to a first discrete mathematics course in a computer science curriculum:

1. required:

- introduction to logical reasoning, including such topics as truth tables and methods of proof; quantifiers should be included and proofs by induction should be emphasized; simple diagonalization proofs should be presented;
- counting and simple finite probability theory—these are essential to understanding algorithm complexity and certain material among the electives;
- the rudiments of sets, functions, and relations—these are essential to a substantial amount of material in the Core;

2. recommended:

- recurrence relations and difference equations—these are useful in the analysis of algorithms;
- graph theory and trees; formalizations for important classes of data structures, and good laboratory material for proofs;
- matrices.

The second mathematics course ensures that students reach the level of mathematical maturity necessary for the core courses—it must be completed before CO2 and CO3—and covers desirable topics not treated in the first course. Mathematical topics that are particularly relevant to computer science include

1. discrete mathematics:

- any topics among those recommended for, but not covered in, the first discrete mathematics course;
- finite state machines and languages;
- partially ordered sets and lattices;
- coding theory;
- modular arithmetic and arithmetic in other bases;
- Boolean algebra;
- algebraic structures; semigroups and groups;

2. calculus:

- limits;
- function manipulation;
- derivatives;
- max-min problems;
- simple integration;
- infinite series;

3. linear algebra:

- vectors and linear transformations;
- bases;
- matrix manipulation;
- eigenvalues and eigenvectors.

Since it is unlikely that a single course will cover all topics, a special course may be designed to contain subsets of these topics—as can often be found in discrete mathematics, calculus, or linear algebra courses. The particular courses that satisfy this requirement at any given institution will depend on local availability.

Students who anticipate pursuing graduate study in computer science, as well as certain other career paths, will generally need more mathematics preparation than the two courses required in this model. Such preparation is offered in additional courses such as calculus, logic, algebraic structures, graph theory, probability and statistics, coding theory, and differential equations.

THE CORE

The core curriculum for a B.A.-degree program in computer science is based on the working definition of computer science given previously. Because the B.A.-degree program presupposes that the components of that definition appear in decreasing order of importance in the curriculum, the formal properties or theoretical foundations of the discipline should constitute the Core. In identifying the Core, it is expected that these principles will have more of a lasting presence in the field of computer science than the topics that constitute the various computer applications, particular architectures, and specific language designs and programming styles.

The topics that constitute the Core are divided into four categories, each roughly a one-semester course:

CO1. Principles of Computer Organization,
CO2. Algorithms,
CO3. Theory of Computation, and
CO4. Principles of Programming Languages.

Every B.A.-degree program for computer science should have at least a semester course in each of these subjects, covering them in the fashion outlined below. Note that there is no strict prerequisite structure among them (see Figure 1, p. 205), but that the introductory courses CS1, CS2, and the first discrete mathematics course must be completed before the Core is begun. (The second mathematics course is a strict prerequisite for CO2 and CO3.) Thus, these four core courses may be scheduled in a variety of ways, depending on the needs of each institution and student.

CO1. Principles of Computer Organization
This course stresses the hierarchical structure of computer architecture, beginning at the lowest level with the simple Turing machine or a similar model. Its topics include

- levels of computer organization; digital logic, microprogramming, machine language, macro language, and operating systems;
- processors; instruction execution, memory, registers, addressing, input/output, control, and synchronization;
- instruction sets, addressing, data flow, control flow, interrupts, and multitasking;
- assembly language programming; macros, linkers, and loaders.

Here, a laboratory must accompany the course so that the student gains experience with a particular example of computer organization at one or more of these levels. A text in the style of Tanenbaum's *Structured Computer Organization*, 2nd ed. [21], plus a companion text on an assembler language would be appropriate for this course.

CO2. Algorithms
This course is a systematic study of algorithms and their complexity. The limitations of algorithms are also studied in the context of NP-completeness. Topics include

- measuring algorithm complexity; O-notation;
- searching and sorting algorithms and their complexity—those aspects not covered in CS2;
- mathematical algorithms—matrices, polynomials, and algebra—and their complexity;
- tree and graph traversal algorithms and their complexity;
- the classes P and NP; NP-complete problems and intractable problems.

CO2 may be accompanied by a laboratory in which the complexity of algorithms can be measured experimentally. A suitable text for this course would be Aho, Hopcroft, and Ullman's *Data Structures and Algorithms* [4], Horowitz and Sahni's *Fundamentals of Computer Algorithms* [10], Reingold, Nievergelt, and Deo's *Combinatorial Algorithms* [19], or Baase's *Computer Algorithms* [5].

CO3. Theory of Computation
This course covers basic theoretical principles embodied in formal languages, automata, computability, and computational complexity. Its topics are as follows:

- finite automata, pushdown automata, nondeterminism, regular expressions, and context-free grammars;
- Turing machines, Church's thesis, Gödel numbering, and universal Turing machines;
- the halting problem, unsolvability, and computational complexity.

CO3 assumes the two-semester mathematics prerequisite, particularly including the topics of sets, relations, and induction and diagonalization proofs. A recommended text would be Lewis and Papadimitriou's *Elements of the Theory of Computation* [13]. Because this course emphasizes theoretical issues, students would be expected to prove theorems and demonstrate fundamental results.

CO4. Principles of Programming Languages
This course emphasizes the principles and programming styles that govern the design and implementation of contemporary programming languages. Topics include

- language syntax; lexical properties, BNF, and parsing examples;
- language processors; compilers, interpreters, and direct execution;
- language representations; data structures, control structures, binding, the run-time environment, and formal semantic models;
- language styles; procedural, functional programming, object programming, logic programming, modular programming, data flow, and their uses.

CO4 should be supplemented by a laboratory in which one or more contemporary programming languages or processors are studied as examples of the principles covered above. At the moment, such languages include Ada, C, Icon, Lisp (and other functional languages), Modula-2, Pascal, Prolog, and Smalltalk. (Recent developments in compilers for microcomputers permit such a laboratory to be established at a relatively low cost.) Textbooks appropriate for this course are Pratt's *Programming Languages, Design and Implementation*, 2nd ed. [15], and Tucker's *Programming Languages*, 2nd ed. [22].

It is expected that a student making normal progress through a computer science B.A. program will complete the Core by the end of the junior year; more intensive concentration will permit completion by the end of the sophomore year. Individual options, such as a junior year of study abroad, might make the second alternative preferable. In any case, the student will have ample time to complete the general education requirements that distinguish the B.A. degree from the B.S. degree. Opportunities for developing a double concentration, such as computer science and Japanese, are also facilitated by this presentation of the Core. However, in no case does the Core diminish or diffuse the quality or completeness of the fundamental topics considered essential for *all* computer science majors.

THE ELECTIVES

A quality undergraduate program offers a variety of upper level courses that build on the intellectual foundations of the core curriculum. Our intent is not to elaborate on the exact content of these courses, but to

convey that there must be *capstone* courses beyond this step that cause the student to synthesize several ideas seen in the Core and blend them with new ideas that are specific to the elective. A variety of mutually independent courses (see Figure 1, p. 205) is recommended for two reasons: Further specialization should be reserved for graduate school, and not all of the courses listed here will be offered by any one program.

For example, a course in compiler design (EL1 in Figure 1) brings together the concepts of strings, finite state and pushdown store machines, context-free grammars, computer organization, tree traversal, and so forth that are introduced in the Core. The student should therefore appreciate the fundamental importance of abstract principles brought together to construct a useful product that is both complex and elegant. This approach stands in contrast to the *cookbook* approach to compiler construction. It is essential for the student to understand that a lexical analyzer, for example, is a special application of a finite state machine and not just a clever piece of programming.

The elective courses listed in Figure 1 are intentionally divided into two tiers (EL1–EL5 and EL6–EL11), to reflect two distinct levels of importance to the computer science major. Specifically, we recommend that the student be required to complete *two* of the electives EL1–EL5, and one additional elective from EL1–EL11 to complete the major. These requirements dictate that an institution offer at least three electives per year, two of which must be selected from among EL1–EL5. Moreover, electives other than these may be added as faculty research interests and institutional variations dictate.

Finally, we recommend that each student be required to complete a substantial software project, including a significant writing (documentation) component in one of these electives. This experience will further unify the student's appreciation of the interaction between theory and applications, one of the guiding themes of this model.

DEGREE REQUIREMENTS AND SAMPLE PROGRAMS

The B.A.-degree program in computer science, by our definition, can therefore be summarized as follows:

1. completion of the four Introductory courses,
2. completion of the four Core courses, and
3. completion of three electives (at least two from EL1–EL5).

To illustrate how this concentration may be completed by an undergraduate in a liberal arts institution, two sample programs (Tables I and II) are presented. The first sample shows a typical progression through the major, while the second shows a "worst-case" scenario in which the student does not begin CS1 until the sophomore year.

Even in the worst case, we are convinced that students who wish to pursue graduate study in computer science will be well prepared by this program. Moreover, this program provides strong preparation for students who wish to follow a variety of lifelong careers, both inside and outside the computer field.

TABLE I. Sample Program (typical case)

	Fall	Spring
Freshman	CS1	CS2 Discrete Math
Sophomore	CO1 Math II	CO2
Junior	CO3	CO4
Senior	EL EL	EL

TABLE II. Sample Program (worst case)

	Fall	Spring
Freshman	—	—
Sophomore	CS1	CS2 Discrete Math
Junior	CO1 Math II	CO2 CO4
Senior	CO3 EL	EL EL

FACULTY, STAFF, AND LABORATORY

Although this workshop did not set out to discuss the logistics of implementing a B.A.-degree program based on these recommendations, some points are obvious and should be made here.

First, a high-quality B.A. degree in computer science can be offered by a faculty of three or four full-time computer scientists, either as a small, separate department or as a subgroup within a department of "mathematics and computer science" or the like. Faculty would be expected to be active scholars as well as teachers, doing research and publishing in the same style as their colleagues in mathematics and the other sciences.

Second, this minimal level of staffing permits three sections of CS1, two sections of CS2, the four Core courses, and three electives to be offered each year, assuming a four-course annual teaching load for each faculty member and support from the mathematics department for staffing Discrete Math and Math II (see Table III, p. 210). Note that CS1 and EL for Professor 3 in Table III could be reversed to satisfy the "typical-case" sample program in Table I. Each of the electives listed in an institution's catalog should be taught at least once over a two-year period.

Third, the laboratory requirements that follow from our curriculum recommendations are quite different

TABLE III. Sample Staffing of Three Faculty Members

	Fall	Spring
Professor 1	CS1	CS2
	EL	CO2
Professor 2	CO1	CS1
	CS2	EL
Professor 3	CO3	CO4
	CS1	EL

Two-course/semester teaching load.

from those now recommended by the CSAB guidelines and "Curriculum 78." In fact, laboratory support may be adequately provided for the lower level courses in a variety of ways, from a shared collection of terminals supplied by an academic computer center to a cluster of micros running Pascal and an assembler. Many of the electives, on the other hand, will require specialized laboratory hardware and software. Minimally, a cluster of micros running UNIX® could be adequate. The specific research interests and elective courses offered by the faculty will also govern the particular choices of equipment available at any particular institution. Consideration should be given to joining CSNET so that a small-sized faculty can interact with the computer science research community at large. However, because of the strong theoretical flavor of this curriculum, the need for large amounts of expensive laboratory equipment and software is less severe than it would be in either a computer engineering or an information systems curriculum.

CONCLUSIONS

This discussion of computer science and its curriculum has evolved from an intense effort by concerned computer scientists in liberal arts colleges. We are primarily interested in enhancing the fundamental integrity of computer science education and believe that this can only result from widespread and careful review of the definition of computer science itself. We hope that this recommendation will receive serious consideration as a contribution toward a complete revision of "Curriculum 78."

Acknowledgments. The authors wish to thank those people who contributed their time and energies to the two workshop meetings held at Bowdoin College and Colgate University and to the two panel sessions held at SIGCSE 85 and ACM 86. We especially wish to thank Kim Bruce of Williams College for his significant contributions to the sections on introductory computing and mathematics courses. Further, we thank all who read earlier drafts of this article and the preliminary report and cared enough to send us so many fruitful comments. Final thanks are to Steve Maurer who was kind enough to listen to new ideas and then help obtain Sloan funding for the workshops.

UNIX is a trademark of AT&T Bell Laboratories.

REFERENCES

1. ACM Curriculum Committee on Computer Science. Curriculum 68: Recommendations for academic programs in computer science. *Commun. ACM 11*, 3 (Mar. 1968), 151–197.
2. ACM Curriculum Committee on Computer Science. Curriculum 78: Recommendations for the undergraduate program in computer science. *Commun. ACM 22*, 3 (Mar. 1979), 147–166.
3. ACM Curriculum Committee on Computer Science. Recommendations for master's level programs in computer science. *Commun. ACM 24*, 3 (Mar. 1981), 115–123.
4. Aho, A.V., Hopcroft, J.E., and Ullman, J.D. *Data Structures and Algorithms.* Addison-Wesley, Reading, Mass., 1983.
5. Baase, S. *Computer Algorithms: Introduction to Design and Analysis.* Addison-Wesley, Reading, Mass., 1978.
6. Berztiss, A.T. Towards a rigorous curriculum for computer science. Tech. Rep. 83-5, Dept. of Computer Science, Univ. of Pittsburgh, Pa., Sept. 1983.
7. Denning, P. Educational ruminations. *Commun. ACM 27*, 10 (Oct. 1984), 979–983.
8. Gibbs, N., and Tucker, A. On accreditation. *Commun. ACM 27*, 5 (May 1984), 411–412.
9. Goldschlager, L., and Lister, A. *Computer Science: A Modern Introduction.* Prentice-Hall, Englewood Cliffs, N.J., 1982.
10. Horowitz, E., and Sahni, S. *Fundamentals of Computer Algorithms.* Computer Science Press, Rockville, Md., 1978.
11. Koffman, E.B., Miller, P.L., and Wardle, C.E. Recommended curriculum for CS1, 1984: A report of the ACM Curriculum Committee Task Force for CS1. *Commun. ACM 27*, 10 (Oct. 1984), 998–1001.
12. Koffman, E.B., Stemple, D., and Wardle, C.E. Recommended curriculum for CS2, 1984: A report of the ACM Curriculum Committee Task Force for CS2. *Commun. ACM 28*, 8 (Aug. 1985), 815–818.
13. Lewis, H.R., and Papadimitriou, C.H. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
14. Mulder, M.C., and Dalphin, J. Computer science program requirements and accreditation: An interim report of the ACM/IEEE Computer Society Joint Task Force. *Commun. ACM 27*, 4 (Apr. 1984), 330–335.
15. Pratt, T.W. *Programming Languages, Design and Implementation.* 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1984.
16. Ralston, A. Computer science, mathematics, and the undergraduate curricula in both. *Am. Math. Monthly 88*, 7 (Aug.–Sept. 1981), 472–485.
17. Ralston, A. The first course in computer science needs a mathematical corequisite. *Commun. ACM 27*, 10 (Oct. 1984), 1002–1005.
18. Ralston, A., and Shaw, M. Curriculum 78—Is computer science really that unmathematical? *Commun. ACM 23*, 2 (Feb. 1980), 67–70.
19. Reingold, E.M., Nievergelt, J., and Deo, N. *Combinatorial Algorithms: Theory and Practice.* Prentice-Hall, Englewood Cliffs, N.J., 1977.
20. Shaw, M., Ed. *The Carnegie-Mellon Curriculum for Undergraduate Computer Science.* Springer-Verlag, New York, 1984.
21. Tanenbaum, A.S. *Structured Computer Organization.* 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1984.
22. Tucker, A.B. *Programming Languages.* 2nd ed. McGraw-Hill, New York, 1986.
23. Ullman, J.D. *Fundamental Concepts of Programming Systems.* Addison-Wesley, Reading, Mass., 1976.

CR Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education
Additional Key Words and Phrases: B.A. degree, computer science, curriculum, education, liberal arts

Authors' Present Addresses: Norman E. Gibbs, Computer Science and Information Studies, Bowdoin College, Brunswick, ME 04011; Allen B. Tucker, Dept. of Computer Science, Colgate University, Hamilton, NY 13346.