SPECIAL REPORT

Computing as a Discipline

Final Report of the ACM Task Force on the Core of Computer Science, in cooperation with the IEEE Computer Society

Peter J. Denning (chair), Douglas E. Comer,
David Gries, Michael C. Mulder, Allen Tucker,
A. Joe Turner, and Paul R. Young

As ACM enters its 42nd year, an old debate continues. Is computer science a science? An engineering discipline? Or merely a technology, an inventor and purveyor of computing commodities? What is the intellectual substance of the discipline? Is it lasting or will it fade within a generation? Do core curricula in computer science and engineering accurately reflect the field? How can theory and lab work be integrated in a computing curriculum? Do core curricula foster competence in computing?

We in the field project an image of a technology-oriented discipline rooted in mathematics and engineering. For example, we represent algorithms as the most basic objects of concern, and programming and hardware design as the primary activities. The view that "computer science equals programming" is especially strong in most of our curricula: programming constitutes the introductory course, technology appears in the core courses, and science goes into the electives. This view blocks progress in reorganizing the curriculum and diverts the best students, who want a greater challenge. It denies a coherent approach to making experimental and theoretical computer science integral and harmonious parts of a curriculum.

Those in the discipline know that computer science encompasses far more than programming. For example, designing hardware, formulating system architectures, designing operating system layers, structuring a database for a specific application, and validating models belong to the discipline, but are not programming. The emphasis on programming arises from our long-standing belief that programming languages are excellent vehicles for gaining access to the rest of the field — a belief that limits our ability to speak about the discipline in terms that reveal its full breadth and richness.

The field has matured enough that we can now describe its intellectual substance in a new and compelling way. This realization arose in discussions among the heads of the PhD-granting departments of computer science and engineering at their meeting in Snowbird, Utah, in July 1984. The discussions inspired interest in the ACM and the IEEE Computer Society to form task forces chartered to create a new approach. In the spring of 1985, ACM President Adele Goldberg and ACM Education Board Chair Robert Aiken appointed this task force on the core of computer science with the enthusiastic cooperation of the IEEE Computer Society. At the same time, the Computer Society formed a task force on computing laboratories with the enthusiastic cooperation of the ACM.

We hope that the work of the ACM task force, embodied in this report, will produce benefits beyond the original charter. By identifying a common core of subject matter, it can catalyze the coalescing of the processes for developing curricula and model programs in the two societies. It can serve as the basis for future discussions of computer science and engineering as a profession. It can stimulate improvements in secondary school courses in computing. And, it can lead to a greater appreciation of computing as a discipline by those outside the field.

We set for ourselves the goal of creating a new way of thinking about the field. Hoping to inspire general inquiry into the nature of our discipline, we sought a new intellectual framework, not a prescription. We invite you to adopt the framework and adapt it for your own curricula.

Charter of the task force

The task force had three general charges:

(1) Describe computer science in a way that emphasizes fundamental questions and

0018-9162/89/0200-0063$01.00©1989 IEEE

significant accomplishments. The definition should recognize the constantly changing nature of the field and point out that what is said is merely a snapshot of an ongoing process of growth.

(2) Propose a teaching paradigm for computer science that conforms to traditional scientific standards, emphasizes the development of competence in the field, and harmoniously integrates theory, experimentation, and design.

(3) Give a detailed example of an introductory course sequence in computer science based on the curriculum model and the disciplinary description.

We immediately extended our task to encompass both computer science and computer engineering, for we concluded that the core material for the two fields shows no fundamental difference. The differences manifest in the way the two disciplines elaborate the core according to different standards of competence: computer science focuses on analysis and abstraction, computer engineering on abstraction and design. We use the phrase "discipline of computing" here to embrace all of computer science and engineering.

Two important issues lie outside the charter of this task force. First, the curriculum outline we created deals only with the introductory course sequence. It does not address the important, larger question of the design of the entire core curriculum. Nonetheless, the suggested introductory course would be meaningless without a new design for the rest of the core. Second, our specification of an introductory course is intended as an example approach to introduce students to the whole discipline in a rigorous and challenging way, an "existence proof" that our definition of computing can be put to work. We leave it to individual departments to apply the framework to develop their own introductory courses.

Paradigms for the discipline

As a context for our definition of the discipline of computing, it helps to make explicit the three major paradigms, or cultural styles, by which we approach our work.

The first paradigm, theory, is rooted in mathematics and consists of four steps followed in the development of a coherent, valid theory:

- characterize objects of study (definition),
- hypothesize possible relationships among them (theorem),
- determine whether the relationships are true (proof), and
- interpret results.

A mathematician expects to iterate these steps, for example, when errors or inconsistences are discovered.

The second paradigm, abstraction (modeling), is rooted in the experimental scientific method. It consists of four stages followed in the investigation of a phenomenon:

- form a hypothesis,
- construct a model and make a prediction,
- design an experiment and collect data, and
- analyze results.

A scientist expects to iterate these steps, for example, when a model's predictions disagree with experimental evidence. Even though "modeling" and "experimentation" might be appropriate substitutes, we have chosen the word "abstraction" for this paradigm because this usage is common in the discipline.

The third paradigm, design, is rooted in engineering. It consists of four steps followed in the construction of a system (or device) to solve a given problem:

- state requirements,
- state specifications,
- design and implement the system, and
- test the system.

An engineer expects to iterate these steps, for example, when tests reveal that the latest version of the system does not satisfactorily meet the requirements.

Theory is the bedrock of the mathematical sciences: applied mathematicians share the notion that science advances only on a foundation of sound mathematics. Abstraction (modeling) is the bedrock of the natural sciences: scientists share the notion that scientific progress is achieved primarily by formulating hypotheses and systematically following the modeling process to verify and validate them. Design is the bedrock of engineering: engineers share the notion that progress is achieved primarily by posing problems and systematically following the design process to construct systems that solve them.

Many debates about the relative merits of mathematics, science, and engineering implicitly rely on an assumption that one of the three processes (theory, abstraction, or design) is the most fundamental. However, a closer examination reveals that, in computing, the three processes so intricately intertwine that it is irrational to say that any one is the most fundamental. Instances of theory appear at every stage of abstraction and design, instances of modeling at every stage of theory and design, and instances of design at every stage of theory and abstraction.

Despite their inseparability, the three paradigms represent distinct areas of competence. Theory concerns the ability to describe and prove relationships among objects. Abstraction concerns the ability to use those relationships to make predictions that can be compared with the world. Design concerns the ability to implement specific instances of those relationships and use them to perform useful actions. Applied mathematicians, computational scientists, and design engineers generally do not have interchangeable skills.

What is more, in computing we tend to study computational aids that support people engaged in information-transforming processes. On the design side, for example, sophisticated VLSI design and simulation systems enable the efficient and correct design of microcircuitry, while programming environments enable the efficient design of software. On the modeling side, supercomputers evaluate mathematical models and make predictions about the world, while networks help disseminate

Complete ACM task force report available

An unabridged edition of the body and first appendix of this report was published in the January 1989 issue of *Communications of the ACM*.

The complete report includes two appendixes. The first consists of a full definition of computing as a discipline. A condensed version of Appendix I, based on the matrix framework proposed by the task force, appears on pages 68 and 69 of this issue of *Computer*. Appendix II of the complete report suggests a core course curriculum for computing majors.

The complete report and both appendixes, titled *Report of the ACM Task Force on the Core of Computer Science*, is available for $7 to ACM members and $12 to nonmembers by calling ACM at (800) 342-6626 or (301) 528-4261 (Alaska, Maryland, and outside the United States). Specify ACM order number 201880.

findings from scientific experiments. On the theory side, computers help prove theorems, check the consistency of specifications, check for counterexamples, and demonstrate test cases.

Computing sits at the crossroads among the central processes of applied mathematics, science, and engineering. The three processes have equal — and fundamental — importance in the discipline, which uniquely blends theory, abstraction, and design. The binding forces are a common interest in experimentation and design as information transformers, a common interest in computational support of the stages of those processes, and a common interest in efficiency.

Role of programming

We have noted that many activities in computing are not programming, such as hardware design, system architecture, operating system structure, database application design, and model validation. Hence, the notion that "computer science equals programming" is misleading. What role does programming play in the discipline? In the curriculum?

Clearly, programming is part of the standard practices of the discipline and hence every major should achieve competence in it. This does not, however, imply that the curriculum should be based on programming or that the introductory courses should be programming courses.

It is also clear that access to the distinctions of any domain comes through language, and that most of the distinctions of computing are embodied in programming notations. Hence, programming languages are useful tools for gaining access to the distinctions of the discipline.

We recommend, therefore, that programming be a part of the competence sought by the core curriculum, and that programming languages be treated as useful vehicles for gaining access to important distinctions of computing.

Description of computing

Our description of a computing as a discipline consists of four parts:

- requirements,
- short definition,
- division into subareas, and
- elaboration of subareas.

What we say here is merely a snapshot of a changing and dynamic field. We intend this to be a "living" definition, revised from time to time to reflect maturity and change in the field. We expect revisions to be most common in the details of the subareas, occasional in the list of subareas, and rare in the short definition.

Requirements. There are many possible ways to formulate a definition. We set five requirements for ours:

(1) It should be understandable by people outside the field.

(2) It should be a rallying point for people inside the field.

(3) It should be concrete and specific.

(4) It should elucidate the historical roots of the discipline in mathematics, logic, and engineering.

(5) It should set forth the fundamental questions and significant accomplishments in each area of the discipline.

In the process of formulating a description, we considered several previous definitions and concluded that a description meeting these requirements must have several levels of complexity. (See the complete version of this report for the previous definitions.)

Short definition. The discipline of computing is the systematic study of algorithmic processes that describe and transform information, their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, "What can be (efficiently) automated?"

Division into subareas. We grappled at some length with the question of dividing the discipline into subareas. We began with a preference for a small number of subareas, such as model versus implementation, or algorithm versus machine. However, the various candidates we devised were too abstract and the boundaries between divisions were too fuzzy. In addition, we felt that most people would not have identified comfortably with them.

Then we realized that the fundamentals of the discipline are contained in the three basic processes — theory, abstraction, and design — because they are used by the disciplinary subareas to accomplish their goals. Thus, a description of the discipline's subareas and their relation to these three basic processes would be useful. To qualify as a subarea, a segment of the discipline must satisfy four criteria:

- underlying unity of subject matter,
- substantial theoretical component,
- significant abstractions, and
- important design and implementation issues.

Theory includes the processes for developing the underlying mathematics of the subarea, supported by theory from other areas. For example, the subarea of algorithms and data structures contains complexity theory and is supported by graph theory.

Abstraction deals with modeling potential implementations. These models suppress detail while retaining essential features. They are amenable to analysis and provide means for calculating predictions of the modeled system's behavior.

Design deals with the process of specifying a problem; transforming the problem statement into a design specification; and repeatedly inventing and investigating alternative solutions until you achieve a reliable, maintainable, documented, tested design that meets your cost criteria.

We considered it desirable that each subarea be identified with a research community, or set of related communities, vital enough to create and maintain a body of literature in the area. We discerned nine subareas that cover the field:

- algorithms and data structures
- programming languages
- architecture
- numerical and symbolic computation
- operating systems
- software methodology and engineering
- database and information retrieval systems
- artificial intelligence and robotics
- human-computer communication

Elaboration of subareas. To present the content of the subareas, we found it useful to think of a 9×3 matrix. Each row is associated with a subarea, with one column each for theory, abstraction, and design. Each square of the matrix holds specific information about that subarea component, describing issues of concern and significant accomplishments.

Certain affinity groups providing scientific literature do not appear as subareas because they are basic concerns throughout the discipline. For example, parallelism surfaces in all subareas (there are parallel algorithms, parallel languages, parallel architectures, and so forth) and in theory, abstraction, and design. Similar conclusions hold for security, reliability, and performance evaluation.

Computer scientists will tend to associate with the first two columns of the matrix, computer engineers with the last two.

The accompanying matrix contains a full description of computing, as specified above.

Curriculum model

Competence in the discipline. The goal of education is to develop competence in a domain, in this case computing. Competence, the capability for effective action, measures individual performance against the standard practices of the field. The criteria are grounded in the history of the field. The educational process that leads to competence includes these elements (see F. Flores and M. Graves, "Education," working paper available from Logonet, 2000 Powell Street, 11th Floor, Emeryville, CA 94608):

- Motivate the domain.
- Demonstrate what can be accomplished in the domain.
- Expose the distinctions of the domain.
- Ground the distinctions in history.
- Practice the distinctions.

This model has interesting implications about the design of a curriculum. The first question it leads to is, "In what areas of computing must majors be competent?" There are two broad areas of competence:

(1) Discipline thinking: the ability to invent new distinctions in the field, leading to new modes of action and new tools that make those distinctions available for others to use.

(2) Tool use: the ability to use the tools of the field for effective action in other domains.

We suggest that discipline thinking is the primary goal of a curriculum for computing majors. Majors must also have a sufficient awareness of the tools so that they can work effectively with those in other disciplines to help design modes of effective action in those disciplines.

The inquiry into competence reveals a number of areas where current core curricula in computing are missing some elements. For example, the historical context of the computing field is often deemphasized, leading to many graduates who are ignorant of history and proceed to repeat its mistakes in their jobs. Many computing graduates wind up in business data processing, a domain in which most computing curricula do not seek to develop competence; it is an old controversy whether computing departments or business departments should develop that competence. Discipline thinking must be based on solid mathematical foundations, yet theory is not an integral part of most computing curricula. The standard practices of the computing field include setting up and conducting experiments, contributing to team projects, and interacting with other disciplines to support their interests in effective use of computing, yet most curricula neglect laboratory exercises, team projects, or interdisciplinary studies.

The question of the results to be achieved by computing curricula has not been explored thoroughly in past discussions, and we will not attempt a thorough analysis here. We do strongly recommend that this question be among the first taken up in the design of new core curricula for computing.

Lifelong learning. The curriculum should be designed to develop an appreciation for learning that graduates will carry with them throughout their careers. Many courses are oriented around a paradigm that presents "answers" in a lecture format, rather than focusing on the process of questioning that underlies all learning. We recommend that the follow-on committee consider other teaching paradigms that involve processes of inquiry, an orientation to using the computing literature, and the development of a commitment to a lifelong process of learning.

Introductory sequence

In the curriculum model cited above, the motivation and demonstration of the domain must precede instruction and practice in the domain. This is precisely the purpose of the introductory course sequence. The principal areas of computing — in which majors must develop competence — must be shown to the students, with sufficient depth and rigor that students can appreciate the power of the areas and the benefits from achieving competence in them. The remainder of the curriculum must be carefully designed to systematically explore those areas, exposing new concepts and distinctions and giving the students practice in them.

We therefore recommend that the introductory course consist of regular lectures and a closely coordinated weekly laboratory. The lectures should emphasize fundamentals; the laboratories should emphasize technology and know-how.

The recommended model is traditional in the physical sciences and in engineering; lectures emphasize enduring principles and concepts while laboratories emphasize the transient material and skills relating to the current technology. For example, lectures would discuss the design and analysis of algorithms or the organization of network protocols in functional layers. The corresponding laboratory sessions would require writing programs for algorithms analyzed in lecture and measuring their running times, or installing and testing network interfaces and measuring their packet throughputs.

Within this recommendation, the first courses in computer science would not only introduce programming, algorithms, and data structures, as is now commonly the case, but they would also introduce material from all the other subdisciplines. Mathematics and other theory would be well integrated into the lectures at appropriate points.

We recommend that the introductory course contain a rigorous, challenging survey of the whole discipline. The physics model, exemplified by the Feynman Lectures in Physics, is a paradigm for the introductory course we ultimately envisage.

We emphasize that simply redesigning the introductory course sequence following this recommendation, without redesigning the entire undergraduate curriculum, would be a serious mistake. The experience of physics departments contains many lessons for computing departments in this regard.

Prerequisites. We assume that students who aspire to become computing majors already have a modest background with programming in some language and with computer-based tools such word processors, spreadsheets, and databases. Given the widening use of computers in high schools and at home, it might seem that universities could assume that most incoming students have such a background and provide a remedial course in programming for the others. We have found, however, that the assumption of adequate high school preparation in programming provokes quite a bit of controversy. Evidence exists that adequate preparation is rare. We therefore recommend that computing departments offer an introduction to programming and computer tools that would be a prerequisite (or corequisite) for the introductory courses. We further recommend that departments provide advanced placement for students with adequate high school preparation.

Formal prerequisites and corequisites in mathematics are more difficult to state and will depend on local circumstances. However, accrediting boards in computing re-

quire considerable mathematics, including discrete mathematics, differential and integral calculus, and probability and statistics. These requirements are often exceeded in the better undergraduate programs. In our description of a beginning computing curriculum, we have spelled out in some detail what mathematics is applicable in each of the nine identified areas of computing. Where possible, we have displayed the required mathematical background for each of the teaching modules we describe. This will allow individual departments to synchronize their own local mathematical requirements and courses with the material in the modules. In some cases it might be appropriate to introduce relevant underlying mathematical topics as needed. In general, we recommend that students see applications of relevant mathematics as early as possible in their computing studies.

Modular organization. The introductory sequence should bring out the underlying unity of the field and should flow from topic to topic in a pedagogically natural way. It would therefore be inadequate to organize the course as a sequence of nine sections, one for each of the subareas; such a mapping would appear as a hodge-podge, with difficult transitions between sections. The following list is an ordering of topics that meets these requirements:

- fundamental algorithm concepts
- computer organization (von Neumann)
- mathematical programming
- data structures and abstraction
- limits of computability
- operating systems and security
- distributed computing and networks
- models in artificial intelligence
- file and database systems
- parallel computation
- a human interface

We have grouped the topics into 11 modules. Each module includes challenging material representative of the subject matter without becoming a superficial survey of every aspect or topic. Each module draws material from several squares of the definition matrix as appropriate. Many modules will therefore not correspond one-to-one with rows of the definition matrix. For example, the first module in our example course is entitled "Fundamental Algorithm Concepts." It covers the role of formalism and theory, method in programming, programming concepts, efficiency, and specific algorithms. It draws information from the first, second, fourth, and sixth rows of the definition matrix. It deals only with sequential algorithms. Later modules, "Distributed Computing and Networks" and "Parallel Computation," extend the material in the first module and draw new material from the third and fifth rows of the definition matrix.

As a general approach, each module contains lectures that cover the required theory and most abstractions. Theory is generally not introduced until needed. Each module is closely coupled with laboratory sessions, and the nature of the laboratory assignments is included with the module specifications.

The full specification of these modules appears in Appendix II of the complete report. Our specification is drawn up for a three-semester course sequence containing 42 lectures and 35 scheduled laboratory sessions per semester.

We reemphasize that this is intended as an example of a mapping from the disciplinary description to an introductory course sequence. We do not intend this as a prescription for all introductory courses. Other approaches are exemplified by existing introductory curricula at selected colleges and universities.

Laboratories

We have described a curriculum that separates principles from technology while maintaining coherence between the two. We have recommended that lectures deal with principles and laboratories with technology, with the two being closely coordinated. The laboratories serve three purposes:

(1) Laboratories should demonstrate how principles covered in the lectures apply to the design, implementation, and testing of practical software and hardware. They should provide concrete experiences that help students understand abstract concepts. These experiences are essential to sharpen students' intuition about practical computing and to emphasize the intellectual effort in building correct, efficient computer programs and systems.

(2) Laboratories should emphasize processes leading to good computing know-how. They should emphasize programming, not programs; laboratory techniques; understanding of hardware capabilities; correct use of software tools; correct use of documentation; and proper documentation of experiments and projects. Many software tools will be required on host computers to assist in constructing, controlling, and monitoring experiments on attached subsystems; the laboratory should teach proper use of these tools.

(3) Laboratories should provide an introduction to experimental methods, including use of measurement instruments, diagnostic aids, experiment design, software and hardware monitors, statistical analysis of results, and proper presentation of findings. Students should learn to distinguish careful experiments from casual observations.

To meet these goals, laboratory work should be carefully planned and supervised. It is desirable that students attend lab at specified times, nominally three hours weekly. Lab assignments should be preplanned, with written descriptions of the purposes and methodology of each experiment given to the students. The depth of description should be commensurate with students' prior lab experience: more detail is required in early laboratories. Lab assignments should be carried out under the guidance of a lab instructor who watches that each student follows correct methodology.

The labs associated with the introductory courses will require close supervision and will contain well-planned activities. This implies that more staff will be required per student for these laboratories than for those later in the curriculum.

Lab problems should be coordinated with material in the lecture parts of the course. Individual lab problems in general will deal with combinations of hardware and software. Some lab assignments emphasize technologies and tools that ease the software development process. Others emphasize analysis and measurement of existing software or comparison of known algorithms. Still others emphasize program development based on principles learned in class. (The descriptions of modules in Appendix II contain examples of associated laboratory assignments.)

Laboratory assignments should be self-contained in the sense that the average student should be able to complete the work in the time allocated. Laboratory assignments should encourage students to discover and learn things for themselves. Students should be required to maintain a proper lab book documenting experiments, observations, and data. Students should also be required to maintain their software and to build libraries for use in later lab projects.

We expect that, in labs as in lectures, students will be assigned homework that will require using computers outside the supervised realm of a laboratory. Hence, organized laboratory sessions will supplement, not replace, the usual programming

Definition matrix for computing as a discipline.

	Theory
Algorithms and data structures	Computability theory; computational complexity theory; algorithmic time and space bounds; levels of intractability; parallel computation, lower bounds, mappings from algorithms' dataflow requirements into machine communication paths; probabilistic algorithms; cryptography; supporting areas of graph theory, recursive functions, recurrence relations, combinatorics, calculus, induction, predicate and temporal logic, semantics, probability, and statistics
Programming languages	Formal languages and automata; Turing machines, Post Systems, λ-calculus; formal semantics; supporting areas of predicate logic, temporal logic, modern algebra, and mathematical induction
Architecture	Boolean algebra; switching theory; coding theory; finite state machine theory; supporting areas of statistics, probability, queueing, reliability theory, discrete mathematics, number theory, and arithmetic in different number systems
Numerical and symbolic computation	Number theory; linear algebra; numerical analysis; nonlinear dynamics; supporting areas of calculus, real analysis, complex analysis, and algebra
Operating systems	Concurrency theory; scheduling theory; program behavior and memory management theory; performance modeling and analysis; supporting areas of bin packing, probability, queueing theory, queueing networks, communication and information theory, temporal logic, and cryptography
Software methodology and engineering	Program verification and proof; temporal logic; reliability theory; supporting areas of predicate calculus, axiomatic semantics, and cognitive psychology
Databases and information retrieval	Relational algebra and relational calculus; dependency theory; concurrency theory; statistical inference; sorting and searching; performance analysis; supporting area of cryptography
Artificial intelligence and robotics	Logic; conceptual dependency; cognition; syntactic and semantic models for natural language understanding; kinematics and dynamics of robot motion and world models used by robots; supporting areas of structural mechanics, graph theory, formal grammars, linguistics, philosophy, and psychology
Human-computer communication	Geometry of two and higher dimensions; color theory; cognitive psychology; supporting areas of Fourier analysis, linear algebra, graph theory, automata, physics, and analysis

Abstraction	Design
Efficient, optimal algorithms and analyses for best, worst, and average performance; classifications of the effects of control and data structure on time and space requirements; important classes of techniques; parallel and distributed algorithms, methods of partitioning problems into tasks executable in separate processors	Selection, implementation, and testing of algorithms; implementation and testing of general methods, distributed algorithms, and storage managers; experimental testing of heuristic algorithms for cominatorial problems; cryptographic protocols
Classification of languages based on their syntactic and dynamic semantic models; classification of languages by intended application area; classification of major syntactic and semantic models for program structure; abstract implementation models for major types of language; methods for parsing, compiling, interpretation, and code optimization; methods for automatic generation of parsers, scanners, compiler components, and compilers	Specific languages that combine an abstract machine (semantics) and syntax to form a coherent, implementable, whole; specific implementation methods for particular classes of languages; programming environments; parser and scanner generators; programs for syntactic and semantic error checking, profiling, debugging, and tracing; applications of programming language methods to document-processing functions, other applications
Finite state machine and Boolean algebraic models of circuits relating function to behavior; other general methods of synthesizing systems from basic components; models of circuits and finite state machines for computing arithmetic functions over finite fields; models for data path and control structures; optimizing instruction sets for various models and workloads; hardware reliability; space, time, and organizational trade-offs in design of VLSI devices; organization of machines for various computational models; identification of design levels	Hardware units for fast computation; von Neumann machine; single-instruction sequence stored program computer, RISC and CISC implementations; efficient methods of storing and recording information and detecting and correcting errors; specific approaches for responding to errors; CAD systems and logic simulations for design of VLSI circuits, production programs for layout and fault diagnosis, silicon compilers; implementing machines in various computational models; supercomputers
Formulations of physical problems as models in continuous (and sometimes discrete) mathematics; discrete approximations to continous problems; finite element model for problems specifiable by regular meshes and boundary values, associated iterative methods and convergence theory, parallel solution methods, automatic grid refinement during numerical integration; symbolic integration and differentiation	High-level problem formulation systems; specific libraries and packages for linear algebra, ordinary differential equations, statistics, nonlinear equations, and optimizations; methods of mapping finite element algorithms to specific architectures; symbolic manipulators
Abstraction principles that permit user operation on idealized versions of resources; binding of objects perceived at the user interface to internal computational structures; models for important subproblems; models for distributed computation; models for secure computing; networking	Prototypes of time-sharing systems, automatic storage allocators, multi-level schedulers, memory managers, hierarchical file systems, and other important system components that have served as bases for commercial systems; techniques for building operating systems; techniques for libraries of utilities; files and file systems; queueing network modeling and simulation packages for evaluating performance of real systems; network architectures; protocol techniques embodied in TCP/IP, virtual circuit protocols, internet, real-time conferencing, and X.25
Specification methods; methodologies; methods for automating program development; methodologies for dependable computing; software tools and programming environments; measurement and evaluation of programs and systems; matching problem domains through software systems to particular machine architectures; life-cycle models of software projects	Specification languages, configuration management systems, revision control systems; syntax-directed editors, line editors, screen editors, word processing systems; specific methodologies advocated and used for software development; procedures and practices for testing, quality assurance, and project management; software tools for program development and debugging, profiling, text formatting, and database manipulation; specification of criteria levels and validation procedures for secure computing systems; design of user interfaces; methods for designing very large computer systems that are reliable, fault-tolerant, and dependable
Models for representing logical structure of data and relations among data elements; representations of files for fast retrieval; methods for assuring integrity (consistency) of database under updates; methods for preventing unauthorized disclosure or alteration and for minimizing statistical inference; languages for posing queries over databases of different kinds, similarly for information retrieval systems; models that allow documents to contain text at multiple levels, plus video, graphics, and voice; human factors and interface issues	Techniques for designing databases for relational, hierarchical, network, and distributed implementations; techniques for designing database systems; techniques for designing information retrieval systems; design of secure database systems; hypertext systems; techniques to map large databases to magnetic disk stores; techniques for mapping large, read-only databases onto optical storage media
Knowledge representation and methods of processing them; models of natural language understanding and representations, machine translation; speech recognition and synthesis, translation of text to speech; reasoning and learning models; heuristic search methods, branch and bound, control search; machine architectures that imitate biological systems; models of human memory, autonomous learning, and other elements of robot systems	Techniques for designing software systems for logic programming, theorem proving, and rule evaluation; techniques for expert systems in narrow domains and expert system shells programmable for new domains; implementations of logic programming; natural language understanding systems; implementations of neural networks and sparse distributed memories; programs that play games of strategy; working speech synthesizers, recognizers; working robotic machines, static and mobile
Algorithms for displaying pictures; models for CAD; computer representations of physical objects; image processing and enhancement methods; man-machine communication	Implementation of graphics algorithms on various graphics devices; design and implementation of experimental graphics algorithms; proper use of color graphics for displays, accurate reproduction of colors; graphics standards, languages, and graphics packages; implementation of various user interface techniques; implementation of various standard file interchange formats; working CAD systems; working image enhancement systems

and other written assignments.

In a substantial number of labs dealing with the development of programs, the assignment should be to modify or complete an existing program supplied by the instructor. This forces the student to read well-written programs, provides experience with integration of software, and results in a larger and more satisfying program for the student.

Computing technology constantly changes. It is difficult, therefore, to give a detailed specification of the hardware systems, software systems, instruments, and tools that ought to be in a laboratory. The choice of equipment and staffing in laboratories should be guided by the following principles:

(1) Laboratories should be equipped with up-to-date systems and languages. Programming languages have a significant effect on shaping a student's view of computing. Laboratories should deploy systems that encourage good habits in students; it is especially important to avoid outdated systems (hardware and software) in core courses.

(2) Hardware and software must be fully maintained. Malfunctioning equipment will frustrate students and interfere with learning. Appropriate staff must be available for maintaining the hardware and software used in the lab. The situation is analogous to laboratories in other disciplines.

(3) Full functionality is important. (This includes adequate response time on shared systems.) Restricting students to small subsets of a language or system might be useful in initial contacts, but the restrictions should be lifted as the students progress.

(4) Good programming tools are needed. Compilers get a lot of attention, but other programming tools are used as often. In Unix systems, for example, students should use editors like Emacs and learn to use tools like the shell, grep, awk, and make. Storage and processing facilities must be sufficient to make such tools available for use in the lab.

(5) Adequate support for hardware and instrumentation must be provided. Some projects might require students to connect hardware units together, take measurements of signals, monitor data paths, and the like. A sufficient supply of small parts, connectors, cables, monitoring devices, and test instruments must be available.

The IEEE Computer Society Task Force on Goal-Oriented Laboratory Development has studied this subject in depth. Its report includes a discussion of the resources — staff and facilities — needed for laboratories at all levels of the curriculum.

Accreditation

We conducted this work with the intent that example courses be consistent with current guidelines of the Computing Sciences Accreditation Board. Nonetheless, the details of the mapping of this content to CSAB guidelines does not come within the purview of this committee.

We designed this report to provoke new thinking about computing as a discipline by exhibiting the discipline's content in a way that emphasizes fundamental concepts, principles, and distinctions. The report also suggests a redesign of the core curriculum according to an education model used in other disciplines: a progression from demonstrating the existence of useful distinctions to practice that develops competence with them. We illustrated that by proposing a rigorous introductory course that puts the concepts and principles into the lectures and the use of technology into closely coordinated laboratories. A department cannot simply replace its current introductory sequence with the new one; it must redesign the curriculum so that the new introduction is part of a coherent whole. For this reason, we recommend that the ACM establish a follow-on committee to complete the redesign of the core curriculum.

We recognize that many practical problems must be dealt with before a new curriculum model can become part of the field. For example,

(1) Several years will be required for faculties to redesign their curricula based on a new conceptual formulation.

(2) Currently, no textbooks or educational materials are based on the framework proposed here.

(3) Most departments have inadequate laboratories, facilities, and materials for the educational task suggested here.

(4) Teaching assistants and faculty are inexperienced in the new approach.

(5) Good high school preparation in computing is rare.

We recognize that many of the recommendations given here are challenging and will require substantial work to implement. We are convinced that the improvements in computing education from the proposals here are worth the effort, and we invite you to join us in achieving them. □

Acknowledgments

A large number of people generously provided written comments in response to drafts of this report. Although it was not possible to accommodate every comment in detail, we did take every comment into account in finalizing this report. We are grateful to the following people for sending us their comments: Paul Abrahams, J. Mack Adams, Robert Aiken, Donald Bagert, Alan Biermann, Frank Boesch, Richard Botting, Albert Briggs, Jr., Judy Brown, Rick Carlson, Thomas Cheatham, Neal Coulter, Steve Cunningham, Verlynda Dobbs, Caroline Eastman, Richard Epstein, Frank Friedman, C.W. Gear, Robert Glass, Nico Habermann, Judy Hankins, Charles Kelemen, Elliot Koffma, Barry Kurtz, Doris Lidtke, Michael Loui, Paul Luker, Susan Merritt, John Motil, J. Paul Myers, Ken Nennedy, Bob Noonan, Alan Perlis, Jesse Poore, Terrence Pratt, Jean Sammet, Mary Shaw, J.W. Smith, Dennis Smolarski, Ed Upchurch, Garret White, Gio Wiederhold, and Stuart Zweben.