COMPUTING AS A DISCIPLINE

The final report of the Task Force on the Core of Computer Science presents a new intellectual framework for the discipline of computing and a new basis for computing curricula. This report has been endorsed and approved for release by the ACM Education Board.

PETER J. DENNING (CHAIRMAN), DOUGLAS E. COMER, DAVID GRIES, MICHAEL C. MULDER, ALLEN TUCKER, A. JOE TURNER, and PAUL R. YOUNG

It is ACM's 42nd year and 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 project an image of a technology-oriented discipline whose fundamentals are 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 current curricula: the introductory course is programming, the technology is in our core courses, and the science is in our electives. This view blocks progress in reorganizing the curriculum and turns away 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, hardware design, system architecture, designing operating system layers, structuring a database for a specific application, and validating models are all part of 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 it is now possible to describe its intellectual substance in a new and compelling way. This realization arose in discussions among the heads of the Ph.D.-granting departments of computer science and engineering in their meeting in Snowbird, Utah, in July 1984. These and other similar discussions prompted ACM and the IEEE Computer Society to form task forces to create a new approach. In the spring of 1985, ACM President Adele Goldberg and ACM Education Board Chairman 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 core task force, embodied in this report, will produce benefits beyond the original charter. By identifying a common core of subject matter, we hope to streamline the processes of developing curricula and model programs in the two societies. The report can be the basis for future discussions of computer science and engineering as a profession, stimulate improvements in secondary school courses in computing, and can lead to a greater widespread appreciation of computing as a discipline.

Our goal has been to create a new way of thinking about the field. Hoping to inspire general inquiry into

This article has been condensed from the *Report of the ACM Task Force on the Core of Computer Science*. Copies of the report in its entirety may be ordered, prepaid, from

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

Please specify order #201880. Prices are $7.00 for ACM members, and $12.00 for nonmembers.

the nature of our discipline, we sought a framework, not a prescription; a guideline, not an instruction. We invite you to adopt this framework and adapt it to your own situation.

We are pleased to present a new intellectual framework for our discipline and a new basis for our curricula.

CHARTER OF THE TASK FORCE

The task force was given three general charges:

1. Present a description of computer science that emphasizes fundamental questions and significant accomplishments. The definition should recognize that the field is constantly changing and 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, because we concluded that no fundamental difference exists between the two fields in the core material. The differences are manifested in the way the two disciplines elaborate the core: computer science focuses on analysis and abstraction; computer engineering on abstraction and design. The phrase *discipline of computing* is used here to embrace all of computer science and engineering.

Two important issues are outside the charter of this task force. First, the curriculum recommendations in this report deal only with the introductory course sequence. It does not address the important, larger question of the design of the entire core curriculum, and indeed 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 to be an example of an 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 that meet local needs.

PARADIGMS FOR THE DISCIPLINE

The three major paradigms, or cultural styles, by which we approach our work provide a context for our definition of the discipline of computing. The first paradigm, *theory*, is rooted in mathematics and consists of four steps followed in the development of a coherent, valid theory:

(1) characterize objects of study (definition);
(2) hypothesize possible relationships among them (theorem);
(3) determine whether the relationships are true (*proof*);
(4) interpret results.

A mathematician expects to iterate these steps (e.g., when errors or inconsistencies are discovered.

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

(1) form a hypothesis;
(2) construct a model and make a prediction;
(3) design an experiment and collect data;
(4) analyze results.

A scientist expects to iterate these steps (e.g., 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 and consists of four steps followed in the construction of a system (or device) to solve a given problem:

(1) state requirements;
(2) state specifications;
(3) design and implement the system;
(4) test the system.

An engineer expects to iterate these steps (e.g., 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. Likewise, 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 are implicitly based on an assumption that one of the three processes (theory, abstraction, or design) is the most fundamental.

Closer examination, however, reveals that in computing the three processes are so intricately intertwined that it is irrational to say that any one is 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 are distinct from one another because they represent separate areas of competence. Theory is concerned with the ability to describe and prove relationships among objects. Abstraction is concerned with the ability to use those relationships to make predictions that can be

compared with the world. Design is concerned with 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.

Moreover, 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, and programming environments enable the efficient design of software. On the modeling side, supercomputers evaluate mathematical models and make predictions about the world, and networks help disseminate 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 are of equal—and fundamental—importance in the discipline, which is a unique blend of interaction among 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.

THE ROLE OF PROGRAMMING

Many activities in computing are not programming—for example, hardware design, system architecture, operating system structure, designing a database application, and validating models—therefore the notion that "computer science equals programming" is misleading. What is the role of programming in the discipline? In the curriculum?

Clearly programming is part of the standard practices of the discipline and every computing 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 is given through language, and that most of the distinctions of computing are embodied in programming notations. 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.

A DESCRIPTION OF COMPUTING

Our description of computing as a discipline consists of four parts: (1) requirements; (2) short definition; (3) division into subareas; and (4) elaboration of subareas. Our presentation consists of four passes, each going to a greater level of detail.

What we say here is merely a snapshot of a changing and dynamic field. We intend this to be a "living definition," that can be revised from time to time to reflect maturity and change in the field. We expect revisions to occur most frequently in the details of the subareas, occasionally in the list of subareas, and rarely 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 other previous definitions and concluded that a description meeting these requirements must have several levels of complexity. The other definitions are briefly summarized here.

In 1967, Newell, Perlis, and Simon [5] argued that computer science is the study of computers and the major phenomena that surround them, and that all the common objections to this definition could just as well be used to demonstrate that other sciences are not science. Despite their eloquence, too many people view this as a circular definition that seems flippant to outsiders. It is, however, a good starting point because the definition we present later can be viewed as an enumeration of the major phenomena surrounding computers.

A slightly more elaborate version of this idea was recently used by the Computing Sciences Accreditation Board (CSAB), which said, "Computer science is the body of knowledge concerned with computers and computation. It has theoretical, experimental, and design components and includes (1) theories for understanding computing devices, programs, and systems; (2) experimentation for the development and testing of concepts; (3) design methodology, algorithms, and tools for practical realization; and (4) methods of analysis for verifying that these realizations meet requirements."

A third definition is, "Computer science is the study of knowledge representations and their implementations." This definition suffers from excessive abstraction and few people would agree on the meaning of knowledge representation. A related example that suffers the same fate is, "Computer science is the study of abstraction and the mastering of complexity," a statement that also applies to physics, mathematics, or philosophy.

A final observation comes from Abelson and Sussman, who say, "The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emer-

gence of what might best be called procedural espistemology—the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects. Mathematics provides a framework for dealing precisely with notions of 'what is.' Computation provides a framework for dealing precisely with notions of 'how to' [1]."

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, the boundaries between divisions were too fuzzy, and most people would not have identified comfortably with them.

Then we realized that the fundamentals of the discipline are contained in three basic processes—theory, abstraction, and design—that 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:

(1) underlying unity of subject matter;
(2) substantial theoretical component;
(3) significant abstractions;
(4) important design and implementation issues.

Moreover, we felt that each subarea should be identified with a research community, or set of related communities, that sustains its own literature.

Theory includes the processes for developing the underlying mathematics of the subarea. These processes are 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 a reliable, maintainable, documented, and tested design that meets cost criteria is achieved.

We discerned nine subareas that cover the field:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation
5. Operating systems
6. Software methodology and engineering
7. Database and information retrieval systems
8. Artificial intelligence and robotics
9. Human–computer communication

Elaboration of Subareas

To present the content of the subareas, we found it useful to think of a 9 × 3 matrix, as shown in Figure 1. Each row is associated with a subarea, and theory, abstraction, and design each define a column.

Each square of the matrix will be filled in with specific statements about that subarea component; these statements will describe issues of concern and significant accomplishments.

Certain affinity groups in which there is scientific literature are not shown 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, etc.) 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, and computer engineers with the last two. The full description of computing, as specified here, is given in the appendix.

CURRICULUM MODEL

Competence in the Discipline

The goal of education is to develop competence in a domain. Competence, the capability for effective action,

	Theory	Abstraction	Design
1 Algorithms and data structures			
2 Programming languages			
3 Architecture			
4 Numerical and symbolic computation			
5 Operating systems			
6 Software methodology and engineering			
7 Databases and information retrieval			
8 Artificial intelligence and robotics			
9 Human–computer communication			

FIGURE 1. Definition Matrix for the Computing Discipline

is an assessment of individual performance against the standard practices of the field. The criteria for assessment are grounded in the history of the field. The educational process that leads to competence has five steps: (1) motivate the domain; (2) demonstrate what can be accomplished in the domain; (3) expose the distinctions of the domain; (4) ground the distinctions in history; and (5) practice the distinctions [4].

This model has interesting implications for curriculum design. 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-Oriented 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-oriented thinking is the primary goal of a curriculum for computing majors, and that majors must be familiar enough with the tools to work effectively with people 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 is inadequate. For example, the historical context of the computing field is often deemphasized, leaving many graduates ignorant of computing history and destined to repeat its mistakes. Many computing graduates wind up in business data processing, a domain in which most computing curricula do not seek to develop competence; whether computing departments or business departments should develop that competence is an old controversy. Discipline-oriented 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, but most curricula neglect laboratory exercises, team projects, or interdisciplinary studies.

The question of what results should 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 considered in the design of new core curricula for computing.

Lifelong Learning

The curriculum should be designed to develop an appreciation for learning which graduates will carry with them throughout their careers. Many courses are designed with 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 which 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 this curriculum model, the motivation and demonstration of the domain must precede instruction and practice in the domain. The purpose of the introductory course sequence is precisely this. The principal areas of computing—in which majors must develop competence—must be presented to students with sufficient depth and rigor that they 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 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 technology and know-how.

This model is traditional in the physical sciences and 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. In the corresponding laboratory sessions, students would write programs for algorithms analyzed in lecture and measure their running times, or instal and test network interfaces and measure their packet throughputs.

Within this recommendation, the first courses in computer science would not only introduce programming, algorithms, and data structures, but introduce material from all the other subdisciplines as well. 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 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 computing majors have a modest background in programming in some language and some experience with computer-based tools such as 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 program-

ming is quite controversial and there is evidence 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 an advanced placement procedure so that students with adequate high school preparation can bypass this course.

Formal prerequisites and corequisites in mathematics are more difficult to state and will depend on local circumstances. However, accrediting boards in computing require 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 mathematical requirements and courses with the material in the modules. In some cases it may be appropriate to introduce appropriate underlying mathematical topics as needed for the development of particular topics in computing. 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 to be a hodge-podge, with difficult transitions between sections. An ordering of topics that meet these requirements is:

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
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. As a result, many modules will 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, methods in programming, programming concepts, efficiency, and specific algorithms, draws information from the first, second, fourth, and sixth rows of the definition matrix and deals only with sequential algorithms. Later modules, on Distributed Computing and Networks, and on 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 it is needed. Each module is closely coupled with laboratory sessions, and the nature of the laboratory assignments is included with the module specifications. Our specification is drawn up for a three-semester course sequence containing 42 lectures and 35 scheduled laboratory sessions per semester. Our specification is not included here, but is in the full report.

We reemphasize that this specification is intended only to be an example of a mapping from the disciplinary description to an introductory course sequence, not 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 introduce experimental methods, including use and design of experiments, 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. Students should attend labs at specified times, nominally three hours per week. Lab assignments should be planned, and written descriptions of the purposes and methodology of each experiment should be 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 ensures that each student follows correct methodology.

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

The 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 analyzing and measuring existing software or comparing known algorithms. Others emphasize program development based on principles learned in class.

Laboratory assignments should be self-contained in the sense that an 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 that can be used 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. In other words, organized laboratory sessions will supplement, not replace, the usual programming and other written assignments.

In a substantial number of labs dealing with program development, 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 to maintain 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 may 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 may 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. Their report includes a discussion of the resources (i.e., staff and facilities) needed for laboratories at all levels of the curriculum.

ACCREDITATION

This work has been conducted with the intent that example courses be consistent with current guidelines of the Computing Sciences Accreditation Board (CSAB). The details of the mapping of this content to CSAB guidelines does not fall within the scope of this committee.

CONCLUSION

This report has been designed to provoke new thinking about computing as a discipline by exhibiting the discipline's content in a way that emphasizes the fundamental concepts, principles, and distinctions. It has also suggested a redesign of the core curriculum according to an education model used in other disciplines: demonstrating the existence of useful distinctions followed by practice that develops competence. The method is illustrated by a rigorous introductory course that puts the concepts and principles into the lectures and 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.

Many practical problems must be dealt with before a new curriculum model can become part of the field.

For example,

1. Faculties will need to redesign their curricula based on a new conceptual formulation.
2. No textbooks or educational materials based on the framework proposed here are currently available.
3. Most departments have inadequate laboratories, facilities, and materials for the educational task suggested here.
4. Teaching assistants and faculty are not familiar with the new view.
5. Good high school preparation in computing is rare.

We recognize that many of our recommendations 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 invite you to join us in achieving them.

APPENDIX

A DEFINITION OF COMPUTING AS A DISCIPLINE

Computer science and engineering is the systematic study of algorithmic processes—their theory, analysis, design, efficiency, implementation, and application—that describe and transform information. The fundamental question underlying all of computing is, What can be (efficiently) automated [2, 3]. This discipline was born in the early 1940s with the joining together of algorithm theory, mathematical logic, and the invention of the stored-program electronic computer.

The roots of computing extend deeply into mathematics and engineering. Mathematics imparts analysis to the field; engineering imparts design. The discipline embraces its own theory, experimental method, and engineering, in contrast with most physical sciences, which are separate from the engineering disciplines that apply their findings (e.g., chemistry and chemical engineering principles). The science and engineering are inseparable because of the fundamental interplay between the scientific and engineering paradigms within the discipline.

For several thousand years, calculation has been a principal concern of mathematics. Many models of physical phenomena have been used to derive equations whose solutions yield predictions of those phenomena—for example, calculations of orbital trajectories, weather forecasts, and fluid flows. Many general methods for solving such equations have been devised—for example, algorithms for systems of linear equations, differential equations, and integrating functions. For almost the same period, calculations that aid in the design of mechanical systems have been a principal concern of engineering. Examples include algorithms for evaluating stresses in static objects, calculating momenta of moving objects, and measuring distances much larger or smaller than our immediate perception.

One product of the long interaction between engineering and mathematics has been mechanical aids for calculating. Some surveyors' and navigators' instruments date back a thousand years. Pascal and Leibniz built arithmetic calculators in the middle 1600s. In the 1830s, Babbage conceived of an "analytical engine" that could mechanically and without error evaluate logarithms, trigonometric functions, and other general arithmetic functions. His machine, never completed, served as an inspiration for later work. In the 1920s, Bush constructed an electronic analog computer for solving general systems of differential equations. In the same period, electromechanical calculating machines capable of addition, subtraction, multiplication, division, and square root computation became available. The electronic flip-flop provided a natural bridge from these machines to digital versions with no moving parts.

Logic is a branch of mathematics concerned with criteria of validity of inference and formal principles of reasoning. Since the days of Euclid, it has been a tool for rigorous mathematical and scientific argument. In the 19th century a search began for a universal system of logic that would be free of the incompletenesses observed in known deductive systems. In a complete system, it would be possible to determine mechanically whether any given statement is either true or false. In 1931, Gödel published his "incompleteness theorem," showing that there is no such system. In the late 1930s, Turing explored the idea of a universal computer that could simulate any step-by-step procedure of any other computing machine. His findings were similar to Gödel's: some well-defined problems cannot be solved by any mechanical procedure. Logic is important not only because of its deep insight into the limits of automatic calculation, but also because of its insight that strings of symbols, perhaps encoded as numbers, can be interpreted both as data and as programs.

This insight is the key idea that distinguishes the stored program computer from calculating machines. The steps of the algorithm are encoded in a machine representation and stored in the memory for later decoding and execution by the processor. The machine code can be derived mechanically from a higher-level symbolic form, the programming language.

It is the explicit and intricate intertwining of the ancient threads of calculation and logical symbol manipulation, together with the modern threads of electronics and electronic representation of information, that gave birth to the discipline of computing.

We identified nine subareas of computing:

1. Algorithms and data structures
2. Programming languages
3. Architecture
4. Numerical and symbolic computation

5. Operating systems
6. Software methodology and engineering
7. Databases and information retrieval
8. Artificial intelligence and robotics
9. Human-Computer communication

Each has an underlying unity of subject matter, a substantial theoretical component, significant abstractions, and substantial design and implementation issues. Theory deals with the underlying mathematical development of the subarea and includes supporting theory such as graph theory, combinatorics, or formal languages. Abstraction (or modeling) deals with models of potential implementations; the models suppress detail, while retaining essential features, and provide means for predicting future behavior. Design deals with the process of specifying a problem, deriving requirements and specifications, iterating and testing prototypes, and implementing a system. Design includes the experimental method, which in computing comes in several styles: measuring programs and systems, validating hypotheses, and prototyping to extend abstractions to practice.

Although software methodology is essentially concerned with design, it also contains substantial elements of theory and abstraction. For this reason, we have identified it as a subarea. On the other hand, parallel and distributed computation are issues that pervade all the subareas and all their components (theory, abstraction, and design); they have been identified neither as subareas nor as subarea components.

The subsequent numbered sections provide the details of each subarea in three parts—theory, abstraction, and design. The boundaries between theory and abstraction, and between abstraction and design, are necessarily fuzzy; it is a matter of personal taste where some of the items go.

Our intention is to provide a guide to the discipline by showing its main features, not a detailed map. It is important to remember that this guide to the discipline is not a plan for a course or a curriculum; it is merely a framework in which a curriculum can be designed. It is also important to remember that this guide to the discipline is a snapshot of an organism undergoing constant change. It will require reevaluation and revision at regular intervals.

1. ALGORITHMS AND DATA STRUCTURES

This area deals with specific classes of problems and their efficient solutions. Fundamental questions include: For given classes of problems, what are the best algorithms? How much storage and time do they require? What is the tradeoff between space and time? What is the best way to access the data? What is the worst case of the best algorithms? How well do algorithms behave on average? How general are algorithms—i.e., what classes of problems can be dealt with by similar methods?

1.1 Theory

Major elements of theory in the area of algorithms and data structures are:

1. Computability theory, which defines what machines can and cannot do.
2. Computational complexity theory, which tells how to measure the time and space requirements of computable functions and relates a problem's size with the best- or worst-case performance of algorithms that solve that problem, and provides methods for proving lower bounds on any possible solution to a problem.
3. Time and space bounds for algorithms and classes of algorithms.
4. Levels of intractability: for example, classes of problems solvable deterministically in polynomially bounded time (P-problems); those solvable nondeterministically in polynomially bounded time (NP-problems); and those solvable efficiently by parallel machines (NC-problems).
5. Parallel computation, lower bounds, and mappings from dataflow requirements of algorithms into communication paths of machines.
6. Probabilistic algorithms, which give results correct with sufficiently high probabilities much more efficiently (in time and space) than determinate algorithms that guarantee their results. Monte Carlo methods.
7. Cryptography.
8. The supporting areas of graph theory, recursive functions, recurrence relations, combinatorics, calculus, induction, predicate and temporal logic, semantics, probability, and statistics.

1.2 Abstraction

Major elements of abstraction in the area of algorithms and data structures are

1. Efficient, optimal algorithms for important classes of problems and analyses for best, worst, and average performance.
2. Classifications of the effects of control and data structure on time and space requirements for various classes of problems.
3. Important classes of techniques such as divide-and-conquer, Greedy algorithms, dynamic programming, finite state machine interpreters, and stack machine interpreters.
4. Parallel and distributed algorithms; methods of partitioning problems into tasks that can be executed in separate processors.

1.3 Design

Major elements of design and experimentation in the area of algorithms and data structures are:

1. Selection, implementation, and testing of algorithms for important classes of problems such as searching,

sorting, random-number generation, and textual pattern matching.
2. Implementation and testing of general methods applicable across many classes of problems, such as hashing, graphs, and trees.
3. Implementation and testing of distributed algorithms such as network protocols, distributed data updates, semaphores, deadlock detectors, and synchronization methods.
4. Implementation and testing of storage managers such as garbage collection, buddy system, lists, tables, and paging.
5. Extensive experimental testing of heuristic algorithms for combinatorial problems.
6. Cryptographic protocols that permit secure authentication and secret communication.

2. PROGRAMMING LANGUAGES

This area deals with notations for virtual machines that execute algorithms, with notations for algorithms and data, and with efficient translations from high-level languages into machine codes. Fundamental questions include: What are possible organizations of the virtual machine presented by the language (data types, operations, control structures, mechanisms for introducing new types and operations)? How are these abstractions implemented on computers? What notation (syntax) can be used effectively and efficiently to specify what the computer should do?

2.1 Theory

Major elements of theory in the area of programming languages are:

1. Formal languages and automata, including theories of parsing and language translation.
2. Turing machines (base for procedural languages), Post Systems (base for string processing languages), λ-calculus (base for functional languages).
3. Formal semantics: methods for defining mathematical models of computers and the relationships among the models, language syntax, and implementation. Primary methods include denotational, algebraic, operational, and axiomatic semantics.
4. As supporting areas: predicate logic, temporal logic, modern algebra and mathematical induction.

2.2 Abstraction

Major elements of abstraction in the area of programming languages include:

1. Classification of languages based on their syntactic and dynamic semantic models; e.g., static typing, dynamic typing, functional, procedural, object-oriented, logic, specification, message passing, and dataflow.
2. Classification of languages according to intended application area; e.g., business data processing, simulation, list processing, and graphics.
3. Classification of major syntactic and semantic models for program structure; e.g., procedure hierarchies, functional composition, abstract data types, and communicating parallel processes.
4. Abstract implementation models for each major type of language.
5. Methods for parsing, compiling, interpretation, and code optimization.
6. Methods for automatic generation of parsers, scanners, compiler components, and compilers.

2.3 Design

Major elements of design and experimentation in the area of programming languages are:

1. Specific languages that bring together a particular abstract machine (semantics) and syntax to form a coherent implementable whole. Examples: procedural (COBOL, FORTRAN, ALGOL, Pascal, Ada, C), functional (LISP), dataflow (SISAL, VAL), object-oriented (Smalltalk, CLU), logic (Prolog), strings (SNOBOL), and concurrency (CSP, Occam, Concurrent Pascal, Modula 2).
2. Specific implementation methods for particular classes of languages: run-time models, static and dynamic execution methods, typing checking, storage and register allocation, compilers, cross compilers, and interpreters, systems for finding parallelism in programs.
3. Programming environments.
4. Parser and scanner generators (e.g., YACC, LEX), compiler generators.
5. Programs for syntactic and semantic error checking, profiling, debugging, and tracing.
6. Applications of programming-language methods to document-processing functions such as creating tables, graphs, chemical formulas, spreadsheets equations, input and output, and data handling. Other applications such as statistical processing.

3. ARCHITECTURE

This area deals with methods of organizing hardware (and associated software) into efficient, reliable systems. Fundamental questions include: What are good methods of implementing processors, memory, and communication in a machine? How do we design and control large computational systems and convincingly demonstrate that they work as intended despite errors and failures? What types of architectures can efficiently incorporate many processing elements that can work concurrently on a computation? How do we measure performance?

3.1 Theory

Major elements of theory in the area of architecture are:

1. Boolean algebra.
2. Switching theory.

3. Coding theory.
4. Finite state machine theory.
5. The supporting areas of statistics, probability, queueing, reliability theory, discrete mathematics, number theory, and arithmetic in different number systems.

3.2 Abstraction

Major elements of abstraction in the area of architecture are:

1. Finite state machine and Boolean algebraic models of circuits that relate function to behavior.
2. Other general methods of synthesizing systems from basic components.
3. Models of circuits and finite state machines for computing arithmetic functions over finite fields.
4. Models for data path and control structures.
5. Optimizing instruction sets for various models and workloads.
6. Hardware reliability: redundancy, error detection, recovery, and testing.
7. Space, time, and organizational tradeoffs in the design of VLSI devices.
8. Organization of machines for various computational models: sequential, dataflow, list processing, array processing, vector processing, and message-passing.
9. Identification of design levels; e.g., configuration, program, instruction set, register, and gate.

3.3 Design

Major elements of design and experimentation in the area of architecture are:

1. Hardware units for fast computation; e.g., arithmetic function units, cache.
2. The so-called von Neumann machine (the single-instruction sequence stored program computer); RISC and CISC implementations.
3. Efficient methods of storing and recording information, and detecting and correcting errors.
4. Specific approaches to responding to errors: recovery, diagnostics, reconfiguration, and backup procedures.
5. Computer aided design (CAD) systems and logic simulations for the design of VLSI circuits. Production programs for layout, fault diagnosis. Silicon compilers.
6. Implementing machines in various computational models; e.g., dataflow, tree, LISP, hypercube, vector, and multiprocessor.
7. Supercomputers, such as the Cray and Cyber machines.

4. NUMERICAL AND SYMBOLIC COMPUTATION

This area deals with general methods of efficiently and accurately solving equations resulting from mathematical models of systems. Fundamental questions include: How can we accurately approximate continuous or infinite processes by finite discrete processes? How do we cope with the errors arising from these approximations? How rapidly can a given class of equations be solved for a given level of accuracy? How can symbolic manipulations on equations, such as integration, differentiation, and reduction to minimal terms, be carried out? How can the answers to these questions be incorporated into efficient, reliable, high-quality mathematical software packages?

4.1 Theory

Major elements of theory in the area of numerical and symbolic computation are:

1. Number theory.
2. Linear algebra.
3. Numerical analysis.
4. Nonlinear dynamics.
5. The supporting areas of calculus, real analysis, complex analysis, and algebra.

4.2 Abstraction

Major elements of abstraction in the area of numerical and symbolic computation are:

1. Formulations of physical problems as models in continuous (and sometimes discrete) mathematics.
2. Discrete approximations to continuous problems. In this context, backward error analysis, error propagation and stability in the solution of linear and nonlinear systems. Special methods in special cases, such as Fast Fourier Transform and Poisson solvers.
3. The finite element model for a large class of problems specifiable by regular meshes and boundary values. Associated iterative methods and convergence theory: direct, implicit, multigrids, rates of convergence. Parallel solution methods. Automatic grid refinement during numerical integration.
4. Symbolic integration and differentiation.

4.3 Design

Major elements of design and experimentation in the area of numerical and symbolic computation are:

1. High-level problem formulation systems such as CHEM and WEB.
2. Specific libraries and packages for linear algebra, ordinary differential equations, statistics, nonlinear equations, and optimizations; e.g., LINPACK, EISPACK, ELLPACK.
3. Methods of mapping finite element algorithms to specific architectures—e.g., multigrids on hypercubes.
4. Symbolic manipulators, such as MACSYMA and REDUCE, capable of powerful and nonobvious manipulations, notably differentiations, integrations, and reductions of expressions to minimal terms.

5. OPERATING SYSTEMS

This area deals with control mechanisms that allow multiple resources to be efficiently coordinated in the execution of programs. Fundamental questions include: What are the visible objects and permissible operations at each level in the operation of a computer system? For each class of resource (objects visible at some level), what is a minimal set of operations that permit their effective use? How can interfaces be organized so that users deal only with abstract versions of resources and not with physical details of hardware? What are effective control strategies for job scheduling, memory management, communications, access to software resources, communication among concurrent tasks, reliability, and security? What are the principles by which systems can be extended in function by repeated application of a small number of construction rules? How should distributed computations be organized so that many autonomous machines connected by a communication network can participate in a computation, with the details of network protocols, host locations, bandwidths, and resource naming being mostly invisible?

5.1 Theory

Major elements of theory in the area of operating systems are:

1. Concurrency theory: synchronization, determinacy, and deadlocks.
2. Scheduling theory, especially processor scheduling.
3. Program behavior and memory management theory, including optimal policies for storage allocation.
4. Performance modeling and analysis.
5. The supporting areas of bin packing, probability, queueing theory, queueing networks, communication and information theory, temporal logic, and cryptography.

5.2 Abstraction

Major elements of abstraction in the area of operating systems are:

1. Abstraction principles that permit users to operate on idealized versions of resources without concern for physical details (e.g., process rather than processor, virtual memory rather than main-secondary hierarchy, files rather than disks).
2. Binding of objects perceived at the user interface to internal computational structures.
3. Models for important subproblems such as process management, memory management, job scheduling, secondary storage management, and performance analysis.
4. Models for distributed computation; e.g., clients and servers, cooperating sequential processes, message-passing, and remote procedure calls.
5. Models for secure computing; e.g., access controls, authentication, and communication.
6. Networking, including layered protocols, naming, remote resource usage, help services, and local network protocols such as token-passing and shared buses.

5.3 Design

Major elements of design and experimentation in the area of operating systems are:

1. Prototypes of time sharing systems, automatic storage allocators, multilevel schedulers, memory managers, hierarchical file systems and other important system components that have served as bases for commercial systems.
2. Techniques for building operating systems such as UNIX, Multics, Mach, VMS, and MS-DOS.
3. Techniques for building libraries of utilities; e.g., editors, document formatters, compilers, linkers, and device drivers.
4. Files and file systems.
5. Queueing network modeling and simulation packages to evaluate performance of real systems.
6. Network architectures such as ethernet, FDDI, token ring nets, SNA, and DECNET.
7. Protocol techniques embodied in the Department of Defense protocol suite (TCP/IP), virtual circuit protocols, internet, real time conferencing, and X.25.

6. SOFTWARE METHODOLOGY AND ENGINEERING

This area deals with the design of programs and large software systems that meet specifications and are safe, secure, reliable, and dependable. Fundamental questions include: What are the principles behind the development of programs and programming systems? How does one prove that a program or system meets its specifications? How does one develop specifications that do not omit important cases and can be analyzed for safety? How do software systems evolve through different generations? How can software be designed for understandability and modifiability?

6.1 Theory

Major elements of theory in the area of software methodology and tools are:

1. Program verification and proof.
2. Temporal logic.
3. Reliability theory.
4. The supporting areas of predicate calculus, axiomatic semantics, and cognitive psychology.

6.2 Abstraction

Major elements of abstraction in the area of software methodology and tools are:

1. Specification methods, such as predicate transformers, programming calculi, abstract data types, and Floyd–Hoare axiomatic notations.
2. Methodologies such as stepwise refinement, modular

design, modules, separate compilation, information-hiding, dataflow, and layers of abstraction.
3. Methods for automating program development; e.g., text editors, syntax-directed editors, and screen editors.
4. Methodologies for dependable computing; e.g., fault tolerance, security, reliability, recovery, *N*-version programming, multiple-way redundancy, and checkpointing.
5. Software tools and programming environments.
6. Measurement and evaluation of programs and systems.
7. Matching problem domains through software systems to particular machine architectures.
8. Life cycle models of software projects.

6.3 Design

Major elements of design and experimentation in the area of software methodology and tools are:

1. Specification languages (e.g., PSL 2, IMA JO), configuration management systems (e.g., in Ada APSE), and revision control systems (e.g., RCS, SCCS).
2. Syntax directed editors, line editors, screen editors, and word processing systems.
3. Specific methodologies advocated and used in practice for software development; e.g., HDM and those advocated by Dijkstra, Jackson, Mills, or Yourdon.
4. Procedures and practices for testing (e.g., walkthrough, hand simulation, checking of interfaces between modules, program path enumerations for test sets, and event tracing), quality assurance, and project management.
5. Software tools for program development and debugging, profiling, text formatting, and database manipulation.
6. Specification of criteria levels and validation procedures for secure computing systems, e.g., Department of Defense.
7. Design of user interfaces.
8. Methods for designing very large systems that are reliable, fault tolerant, and dependable.

7. DATABASE AND INFORMATION RETRIEVAL SYSTEMS

This area deals with the organization of large sets of persistent, shared data for efficient query and update. Fundamental questions include: What modeling concepts should be used to represent data elements and their relationships? How can basic operations such as store, locate, match, and retrieve be combined into effective transactions? How can these transactions interact effectively with the user? How can high-level queries be translated into high-performance programs? What machine architectures lead to efficient retrieval and update? How can data be protected against unauthorized access, disclosure, or destruction? How can large databases be protected from inconsistencies due to simultaneous update? How can protection and performance be achieved when the data are distributed among many machines? How can text be indexed and classified for efficient retrieval?

7.1 Theory

Major elements of theory in the area of databases and information retrieval systems are:

1. Relational algebra and relational calculus.
2. Dependency theory.
3. Concurrency theory, especially serializable transactions, deadlocks, and synchronized updates of multiple copies.
4. Statistical inference.
5. Sorting and searching.
6. Performance analysis
7. As supporting theory: cryptography.

7.2 Abstraction

Major elements of abstraction in the area of databases and information retrieval systems are:

1. Models for representing the logical structure of data and relations among the data elements, including the relational and entity-relationship models.
2. Representations of files for fast retrieval, such as indexes, trees, inversions, and associative stores.
3. Methods for assuring integrity (consistency) of the database under updates, including concurrent updates of multiple copies.
4. Methods for preventing unauthorized disclosure or alteration and for minimizing statistical inference.
5. Languages for posing queries over databases of different kinds (e.g., hypertext, text, spatial, pictures, images, rule-sets). Similarly for information retrieval systems.
6. Models, such as hypertext, which allow documents to contain text at multiple levels and to include video, graphics, and voice.
7. Human factors and interface issues.

7.3 Design

Major elements of design in the area of database and information retrieval systems are:

1. Techniques for designing databases for relational, hierarchical, network, and distributed implementations.
2. Techniques for designing database systems such as INGRES, System R, dBase III, and DB-2.
3. Techniques for designing information retrieval systems such as LEXIS, Osiris, and Medline.
4. Design of secure database systems.
5. Hypertext systems such as NLS, NoteCards, Intermedia, and Xanadu.
6. Techniques to map large databases to magnetic disk stores.
7. Techniques for mapping large, read-only databases onto optical storage media—e.g., CD/ROM and WORMS.

8. ARTIFICIAL INTELLIGENCE AND ROBOTICS

This area deals with the modeling of animal and human (intelligent) behavior. Fundamental questions include: What are basic models of behavior and how do we build machines that simulate them? To what extent is intelligence described by rule evaluation, inference, deduction, and pattern computation? What is the ultimate performance of machines that simulate behavior by these methods? How are sensory data encoded so that similar patterns have similar codes? How are motor codes associated with sensory codes? What are architectures for learning systems, and how do those systems represent their knowledge of the world?

8.1 Theory

Major elements of theory in the area of artificial intelligence and robotics are:

1. Logic; e.g., monotonic, nonmonotonic, and fuzzy.
2. Conceptual dependency.
3. Cognition.
4. Syntactic and semantic models for natural language understanding.
5. Kinematics and dynamics of robot motion and world models used by robots.
6. The supporting areas of structural mechanics, graph theory, formal grammars, linguistics, philosophy, and psychology.

8.2 Abstraction

Major elements of abstraction in the area of artificial intelligence and robotics are:

1. Knowledge representation (e.g., rules, frames, logic) and methods of processing them (e.g., deduction, inference).
2. Models of natural language understanding and natural language representations, including phoneme representations; machine translation.
3. Speech recognition and synthesis, translation of text to speech.
4. Reasoning and learning models; e.g., uncertainty, nonmonotonic logic, Bayesian inference, beliefs.
5. Heuristic search methods, branch and bound, control search.
6. Machine architectures that imitate biological systems, e.g., neural networks, connectionism, sparse distributed memory.
7. Models of human memory, autonomous learning, and other elements of robot systems.

8.3 Design

Major elements of design and experimentation in artificial intelligence and robotics include:

1. Techniques for designing software systems for logic programming, theorem proving, and rule evaluation.
2. Techniques for expert systems in narrow domains (e.g., Mycin, Xcon) and expert system shells that can be programmed for new domains.
3. Implementations of logic programming (e.g, PROLOG).
4. Natural language understanding systems (e.g., Margie, SHRDLU, and preference semantics).
5. Implementations of neural networks and sparse distributed memories.
6. Programs that play checkers, chess, and other games of strategy.
7. Working speech synthesizers, recognizers.
8. Working robotic machines, static and mobile.

9. HUMAN–COMPUTER COMMUNICATION

This area deals with the efficient transfer of information between humans and machines via various human-like sensors and motors, and with information structures that reflect human conceptualizations. Fundamental questions include: What are efficient methods of representing objects and automatically creating pictures for viewing? What are effective methods for receiving input or presenting output? How can the risk of misperception and subsequent human error be minimized? How can graphics and other tools be used to understand physical phenomena through information stored in data sets?

9.1 Theory

Major elements of theory in human–computer communication are:

1. Geometry of two and higher dimensions including analytic, projective, affine, and computational geometries.
2. Color theory.
3. Cognitive psychology.
4. The supporting areas of Fourier analysis, linear algebra, graph theory, automata, physics, and analysis.

9.2 Abstraction

Major elements of abstraction in the area of human–computer communication are:

1. Algorithms for displaying pictures including methods for smoothing, shading, hidden lines, ray tracing, hidden surfaces, transparent surfaces, shadows, lighting, edges, color maps, representations by splines, rendering, texturing, antialiasing, coherence, fractals, animation, representing pictures as hierarchies of objects.
2. Models for computer-aided design (CAD).
3. Computer representations of physical objects.
4. Image processing and enhancement methods.
5. Man–machine communication, including psychological studies of modes of interaction that reduce human error and increase human productivity.

9.3 Design

Major elements of design and experimentation in the area of human–computer communication are:

1. Implementation of graphics algorithms on various graphics devices, including vector and raster displays and a range of hardcopy devices.
2. Design and implementation of experimental graphics algorithms for a growing range of models and phenomena.
3. Proper use of color graphics for displays; accurate reproduction of colors on displays and hardcopy devices.
4. Graphics standards (e.g., GKS, PHIGS, VDI), graphics languages (e.g., PostScript), and special graphics packages (e.g., MOGLI for chemistry).
5. Implementation of various user interface techniques including direct manipulation on bitmapped devices and screen techniques for character devices.
6. Implementation of various standard file interchange formats for information transfer between differing systems and machines.
7. Working CAD systems.
8. Working image enhancement systems (e.g., at JPL for pictures received from space probes).

ACKNOWLEDGMENTS

Many 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 revising 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
Ken Kennedy
Elliot Koffman
Barry Kurtz
Doris Lidtke
Michael Loui
Paul Luker
Susan Merritt
John Motil
J. Paul Myers
Bob Noonan
Alan Perlis
Jesse Poore
Terrence Pratt
Jean Rogers
Jean Sammet
Mary Shaw
J. W. Smith
Dennis Smolarski
Ed Upchurch
Garret White
Gio Wiederhold
Stuart Zweben

REFERENCES

1. Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.
2. Arden, B., ed. *See What Can Be Automated?* Report of the NSF Computer Science and Engineering Research Study (COSERS). MIT Press, Cambridge, Mass., 1980.
3. Denning, P. What is computer science? *Am. Sci.* 73 (Jan.–Feb. 1985), 16–19.
4. Flores, F., and Graves, M. Education. (working paper available from Logonet, Inc., 2200 Powell Street, 11th Floor, Emeryville, Calif. 94608.)
5. Newell, A., Perlis, A., and Simon, H. What is computer science? *Sci.* 157 (1967), 1373–1374. (reprinted in *Abacus* 4, 4 (Summer 1987), 32.)