Proposal for an Undergraduate Computer Science Curriculum for the 1980s

Part I: Discussion

Mary Shaw, Steve Brookes, Marc Donner,
James Driscoll, Michael Mauldin, Randy Pausch,
Bill Scherlis, Alfred Spector

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213
20 October 1983

Abstract

The authors propose to the Carnegie-Mellon Computer Science Department a curriculum for undergraduate computer science. This report sets forth objectives for computer science education, presents an overview of the content of a curriculum, defines the course structure for a degreee program, and outlines a set of courses. The curriculum design is intended to anticipate the content that will be appropriate at the end of this decade. We have tried to avoid being unduly prejudiced by traditional courses and organizations.

In addition to proposing a curriculum design for computer science majors, the authors recommend the development of new curricula to serve other groups of students.

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

Executive Summary

The Carnegie-Mellon Computer Science Department's Curriculum Design Project has examined the current state of computer science and computer science curricula, has projected the requirements for undergraduate education in computer science, and has developed a curriculum suitable for a computer science major. This report presents our curriculum design.

Carnegie-Mellon currently has a computer science curriculum (a body of courses), but it does not have a computer science major (a formal degree program).

We recommend that the Computer Science Department adopt a curriculum based on this proposal.

We recognize that resource limitations may prevent a complete implementation of the curriculum. We believe that some reasonable subset of the curriculum could form the basis for a computer science major. We also recognize that a curriculum is a necessary, but not a sufficient, condition for a major.

We make no recommendation on the question of offering a computer science major at Carnegie-Mellon.

Goals

Computer science is opening new specialties in many fields, and the pattern of student involvement in computing is changing. As a result, four different undergraduate populations within the university will require distinct kinds of education about computer science. These groups are: computer science majors, students in computational specializations within other disciplines, students who will write programs for personal use, and students who will make only casual use of computers.

We took as our goal the design of a curriculum for the first group of students: those interested primarily in computer science. We have formulated a unified view of the discipline, identified a suitable collection of courses, and defined the content requirement for a major. We chose not to address the university resources required to support such a curriculum.

The University must also provide for the computer science education of non-majors. This report discusses the needs of these students and some suitable responses, but it does not go into depth. New curriculum designs will be required for two of the three groups of non-majors.

We recommend that studies of computer science specializations in other disciplines and of education for students only casually involved with computing be undertaken as separate projects.

Educational Philosophy

We set out to develop a curriculum that would support a computer science degree of the highest quality. Such a curriculum requires a balanced blend of fundamental conceptual material and examples drawn from the best of current practice. In many ways, our educational philosophy is based directly on the Carnegie Plan for education, which emphasizes an integrated understanding of basic concepts and the application of those concepts to practical problems. We believe that a curriculum with a small common core and a broad selection of advanced courses supports a variety of computer science specializations including both terminal and nonterminal programs.

Results

We have designed a computer science curriculum consistent with this educational philosophy. The curriculum includes a unified overview of computer science, the content requirements for a computer science major, and detailed descriptions of a number of computer science courses. The interactions of this curriculum with offerings in other departments are not yet completely specified.

The design recognizes that computer science is a maturing field with a growing set of increasingly comprehensive models and theories. As such, it relies very heavily on mathematics, and it has close ties to several other disciplines. Because the field is changing rapidly, students need fundamental knowledge that they can adapt to new situations. In addition, students must be able to apply their knowledge to real problems, and they must be able to generate tasteful and cost-effective solutions to these problems. In this curriculum, virtually every course emphasizes the integration of theoretical results and practical applications.

We have sketched outlines for twenty-nine computer science courses. They include seven courses in systems and design, three courses in programming languages, two courses in algorithms and analysis, three courses in computer systems, one course in elementary discrete mathematics, four courses in theory and mathematical foundations, three courses in artificial intelligence, one course in graphics, and five independent study, project, or seminar courses. Many of these courses are completely new, and the rest are revised from their present form. As a result, a major effort will be required to implement the individual course designs.

In addition to the courses we define here, we have identified a number of courses in other departments that present material relevant to computer science. Though such material is often conceptually part of a computer science education, we did not develop new descriptions for such courses.

We have also proposed requirements for a computer science major based on this curriculum. These requirements are the basis for a liberal professional education. The required core is small (five specific courses plus three courses constrained to specific areas), thereby allowing a variety of specializations within the major. Additional requirements assure breadth, both by requiring substantial exposure to humanities, social sciences, and fine arts and by requiring a concentration of study outside the major.

Innovations

Because the design was carried out without prior commitment to course organizations, the resulting organization is based on the structure of modern computer science rather than on traditional course divisions. The major innovative characteristics of the resulting curriculum include the following:

- ► *Organization around a core.* The curriculum comprises a core of courses that present the basis of the field and a set of more advanced courses that provide depth of knowledge. The core courses emphasize the mathematical foundations of the field in practical settings.
- ► *Curriculum integration.* The courses are carefully integrated with each other, and strong prerequisite relations ensure that the material will be presented in a coherent order. Subareas often have one course that provides a broad introduction and a sequence of courses that provide for greater depth of knowledge.
- ► *Courses designed around topics rather than artifacts.* Topics based on common ideas often are in a single course, even if the topics are not traditionally taught together. This often entails rearrangement of traditional course boundaries; it allows integration of theory and practice.

- *Use of proper computer support.* Many courses require extensive access to computers and to software that illustrates points being made in the course. Though the forthcoming campus-wide personal computer effort will aid in this, we have presented the functional requirements for computer support rather than discussing specific ways to use personal computers.

After developing an overview of the curriculum, we derived specific courses from the overview. We have re-established the need for an elementary sequence much like the one developed at Carnegie-Mellon in the late 1970s (FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212]). This provides a solid foundation for sequences of advanced courses. In many cases, the initial course of an advanced sequence is eminently suited for a student who wants to use the techniques of an area without specializing in it. The major new courses and course sequences include

- A sophomore course that provides a concrete appreciation of the nature of computation through a unified blend of hardware, software, and theory (240).
- A reorganization of the traditional operating systems course that integrates the hardware, software, and theoretical views of concurrency, generalizes the resource management aspect of operating systems, and deals with complex, long-lived data (310, 412, 413).
- A new course that presents module-level program organizations and software development techniques; this course fills a gap between the courses that teach data structures or program fragments and the courses that deal with constructing systems from modules (311).
- A reorganization of the traditional comparative programming languages and compiler construction courses that focuses first on programming languages and user interfaces, progresses to the use of advanced software tools for system (especially compiler) development, and culminates in language design and compiler construction techniques (320, 420, 421).
- A set of courses that present algorithms and the mathematical foundations of Computer Science with emphasis on integrating the practical uses and consequences of the material with the presentation of the theory. The courses cover algorithms, logic, formal languages, automata, computability, complexity, and theory of programming languages (330, 350, 351, 430, 450, 451).
- An artificial intelligence sequence with a first course that covers the fundamentals of both the psychological and practical aspects of AI and independent follow-on courses providing depth in cognition and robotics (360, 460, 461).

In addition, we plan joint development of a course for advanced students that establishes a basis for responsible evaluation of the consequences of computing and for interpreting the technology to laymen (380).

Organization of the Report

The setting for this design is discussed in Chapter 1. Roles for the university to play in the education of both majors and non-majors are examined in Chapter 2. Our general educational philosophy is defined in more detail in Chapter 3. Chapter 4 presents our unified view of the content of computer science. Chapter 5 shows how majors could be created from the courses of this curriculum. Chapter 6 discuss the rationale for our organization. Summary descriptions of courses appear in Chapter 7. Outlines for the computer science courses we propose are presented in Chapter 8. Chapter 9 lists courses from other departments that cover material in the area we broadly view as computer science. Since the latter two chapters are of somewhat more limited interest than the earlier chapters, they are bound as a separate volume.

Preface

Carnegie-Mellon has had a Computer Science Department and a PhD degree program since 1965, but an undergraduate major leading to a Bachelor's degree in Computer Science has never been offered. Although a formal degree is not offered, a set of undergraduate courses is taught, and the Mathematics Department offers an option that relies heavily on these courses. Undergraduate students who wish to study computer science usually take mathematics degrees in the computer science option.

On a number of occasions over the past decade, the Computer Science Department has considered offering an undergraduate Computer Science major. Until recently, the decision has always been negative. In the Spring of 1981, however, the Department agreed to consider taking steps toward offering a major.

We decided that the first step should be a thorough review of the existing curriculum. The content of the present courses has evolved through the years, and a complete review has not been done in quite some time. Because Computer Science is evolving rapidly, we felt that the changing nature of computers and computing was not adequately reflected by the existing curriculum. As a result, we decided to reconsider the entire curriculum, including both computer science courses and courses that will probably be offered by other departments. Our goals are described in a previous report [12] (reprinted in [13]) and briefly reviewed in Chapter 3.

To date, the interaction between this project and the university personal computer network project has been minimal. However, we have tried to identify ways to take advantage of advanced computing technology as we have developed courses. In addition, we expect to coordinate our plans with those of the university, which has a growing need to use computers in support of undergraduate education and to develop courses that deal with computation in fields other than computer science. We hope that by systematically including software support requirements in course designs we can influence the development of the campus network and justify software development as an ordinary part of course development.

Acknowledgments

A curriculum necessarily spans its discipline; the designers of any such document need all the help they can get. We have received a great deal. Though it is impossible to acknowledge all of it, we want to express our appreciation to some of the people who have affected our thinking most significantly. Thanks, then:

- ► To the Carnegie-Mellon Computer Science Department for support and encouragement in this project.
- ► To the following people, who contributed significantly to the design of individual courses: Jon Bentley, Ellen Borison, Jaime Carbonell, Wes Clark, Geoff Hinton, Elaine Kant, Dan Leivant, Matt Mason, Dana Scott, Chris Stephenson, Bob Wedig, and Bill Wulf.
- ► To Jill Fain, Cynthia Hibbard, Allen Newell, and Steve Shafer, who provided extensive critical comments on drafts of the proposal.
- ► To previous members of this project team, Jon Bentley and Guy Steele. Their participation in the early stages of the design contributed enormously to the final philosophy and structure.
- ► To members of the IEEE/ACM Software Engineering Planning Group. Section 1.2 was prepared while its author was working on both reports, and many of the ideas were developed or refined during the discussions at the planning meeting in September 1982.
- ► To the participants of the Sloan Conference/Workshop on the first two years of College Mathematics, from which the first version of our discrete mathematics syllabus emerged.

Table of Contents

1. Setting

This report presents a curriculum that addresses the needs of computer science education in the 1980s. It is intended to provide the basis of a computer science education of the highest quality. The curriculum is based upon our evaluation of the structure of computer science and the educational needs generated by the maturation of the field. In this chapter, we discuss the nature of computer science and the prospects for enormous growth in the near future. The second chapter describes university roles in computer science education. Next we present a succinct description of our overall premisses and goals and a general description of the content of our curriculum. The remaining chapters describe the curriculum in much greater detail, including a description of an undergraduate degree program and outlines for courses.

We begin in Section 1.1, which describes the scope of the field we consider to be computer science. In Section 1.2 we make some projections about the kind of computing we may be doing ten years hence. On the basis of these projections, we predict some of the issues the field must face over the next decade and some of the changes we must anticipate.

1.1 Working Definition of Computer Science

There is no generally accepted definition of the field of computer science, and we do not expect to remedy that deficiency here. Nevertheless, we need a characterization of the discipline in order to focus our discussions. The curriculum design presented in this report is based on the following working definition.

Computer science is concerned with the study of computers and of the phenomena connected with computing, notably algorithms, programs, and programming. A major objective of the discipline is the formulation of a systematic body of knowledge, theories, and models to explain the properties of computers and related phenomena. It is often the case that computer (or computational) systems exhibit extremely complex structure and behavior; techniques for identifying, quantifying and managing complexity are therefore central to computer science. The discipline is also concerned with producing solutions to technological (real-world) problems using a detailed knowledge of the properties and the applicability of current computing technology. Since there are usually many different ways to solve a problem, an important engineering activity is the evaluation, comparison, and selection of alternatives on the basis of criteria such as cost or efficiency. Unlike the natural sciences, computer science studies objects and systems that are artificial; since both the rules and the artifacts can be modified by the scientist, this can be both a problem and an advantage.

A description of computer science should include not only its subject matter, but also its characteristic paradigms and modes of analysis, reasoning, and problem solving. Computer science borrows heavily from mathematics, using analytic and synthetic techniques such as inductive definitions and case analysis. But it is not exclusively a formal, quantitative field, because the need for practical systems suitable for human use leads the field to rely, for example, on design and modeling techniques from engineering and on techniques from psychology for the study of human performance and behavior. In addition, the leading edge of computer science is moving rapidly. As a result, particular examples or techniques become obsolete and research results move rapidly into the body of pragmatic knowledge.

Using this working definition as a starting point, we conclude that the curriculum must deal with:

- ► Computers and related phenomena: machines and computations, both real and abstract
- ► Algorithms, programs, and programming
- ► Complex structure and behavior of information: how to identify, quantify, and manage it
- ► Engineering concerns: how to find cost-effective solutions to technological problems and to apply current technology
- ► Design tradeoffs: how to compare and select alternatives with respect to given criteria, and some appropriate criteria for such decisions
- ► Human performance: the nature of the people who use computers and the ways that humans manage complex problems

Computer science is a changing field, and the curriculum must be acknowledge this fact. It must be flexible enough to allow allow adaptation to changes in both technology and current thinking, and it must provide students with an education of lasting value despite this change. It must be broad enough to train computer scientists who can interpret the evolution of computer science to laymen. Further, it must make students aware of the roles of computers in society, because as professionals in a field that will so change society they must be able to make informed, responsible decisions that will affect the lives of many.

1.2 A View of Future Computing

The nature of computing, and hence of computer science, is changing rapidly. Many topics that now seem interesting will be obsolete or irrelevant within ten years. If the curriculum we design now is to remain effective through 1990 or beyond, we must try to understand the forces that are shaping the field and to anticipate the roles that computing and computer science will play in the future. This section points out some of the trends that will affect the field over the next decade and describes some of the new phenomena and issues that may arise.

Computers are becoming smaller and cheaper, and they are being distributed across a wider and more varied population. Important current trends include:

- ► Decreasing hardware costs
- ► Increasing share of computing costs attributable to software
- ► Increasing expectations about the power and reliability of applications
- ► Increasing range of applications, particularly those on which lives will depend
- ► Increasing development of distributed computing and convenient network access
- ► Increasing availability of computing power, especially in homes
- ► Widening view of computers as an information utility
- ► Increasing quality of interfaces to humans (voice, high-performance graphics)
- ► Increasing exposure of naive people to computers, both at home and in the work place
- ► Increasing general reliance on computers for day-to-day operations.
- ► Continuing or increasing shortage of qualified professionals
- ► Continuing lack of appreciation for the nature of software
- ► Increasing importance of "intelligent" systems

On the basis of these trends, we can extrapolate some future developments:

- *Pervasive Consumer Computing:* Computers will be extremely widespread, both as multiple-purpose machines in homes and offices and as dedicated (embedded) machines for applications such as household environment control. Most of the users of these machines will be naive — certainly the majority of them will not be programmers. As a result, most of the *users* of programs will not be *creators* of programs.
- *Information Utility:* We will come to think of computers primarily as tools for communicating and for accessing information, rather than primarily as calculating machines. Networks will provide a medium for making available numerous public data bases, both passive (catalogs, library facilities, newspapers) and active (newsletters, individualized entertainment). Real-time control problems will become more visible.
- *Broad Range of Applications:* The range of applications will continue to broaden, and an increasing number will be applications in which unreliable computation could lead to risk of human life. As a result of this and widespread use by nonprogrammers, much of the software will provide packaged services that require little, if any, programming. There will be substantial economic incentives for producing general systems that can be tuned to individual, possibly idiosyncratic, requirements.
- *Changes in the Workplace:* Distributed systems and networks will facilitate a distributed workplace, but we doubt that the norm for office workers will be to work at home instead of in an office — computers will not replace human interaction for decision-making. Electronic work stations will change the nature of work that now depends on paper flow, and robotics will substantially change manufacturing.
- *Massively More Complex Computers:* Some computer networks and large computers will be replaced by or evolve into massive computer systems with orders of magnitude greater capacity than any systems now available. These systems may include enormous databases (nationwide banking records, interactive consumer catalogs, the location and velocity of all ocean-going vessels, etc.), and be used by millions of people simultaneously. The first steps have already been taken by airline and hotel reservation systems. Embedded systems will proliferate to a point where the ordering of software and processors is as important as ordering nuts and bolts when designing any machine (witness the automobile industry today).
- *Intelligent Systems:* Intelligent software systems will provide intellectual multipliers that substantially increase professional productivity in some areas. Intelligent robots will take over an increasing percentage of the industrial workload, and perhaps even make a dent in the household chores. Increasingly sophisticated systems will lessen the need for programmers, and perhaps increase everyone's need for a basic understanding of computers. Otherwise, today's expert systems may be tomorrow's oracles.
- *Impact on GNP:* Computing and information will represent a major component of the GNP, heralding the arrival of a society as dependent on information as on wheat or metal.

Even if this projection is inaccurate, we can expect a substantial qualitative shift in the role of computers in the world at large. The nature of education will surely be affected: we can already see the effects of pocket calculators on the teaching of mathematics. Further, entertainment technology (e.g., Sesame Street and video games) has raised students' expectations about the educational process.

This view of the future raises a number of issues.

- ► *Consumer Concerns:* The use of computers by large numbers of nontechnical people, together with the increasing number of sensitive applications that involve computers, will raise issues about the responsibilities of vendors towards their products. These will certainly include analogs of the familiar problems associated with product and professional liability, merchantability and warrantability (guarantees), usability and reliability, licensing, copyrights, and product safety (e.g., development of an analog to the certification that Underwriters Laboratories provides for electrical products). Other problems, such as security and privacy concerns, will undoubtedly arise from the special nature of computers.
- ► *Production and Distribution:* An expanding role for computers and computer-related products and services in the retail marketplace will introduce new problems in manufacturing, sales and service, equitable methods of charging for shared resources, and industry compatibility standards. Another class of problems will center on how to create software for a mass market, perhaps including some notion of mass production of software (e.g., by tailoring packages rather than by writing code).
- ► *Safety and Security:* In addition to the consumer-safety issues, we can expect questions concerning licensing, product and professional liability, and the trustworthiness or integrity of data provided via public databases. Existing concerns about security and privacy will increase. These concerns will be particularly acute where life-critical applications are involved.
- ► *Economic Impact:* The economic impact of these major innovations must be widespread. Of particular concern for the computing industry will be the interplay between technological development and limiting factors, such as productivity, on the growth of the information sector. Accurate software cost estimates and well considered marketing policies will be vital as the computer industry matures. One of the most important economic changes will be in personnel, especially elimination of unskilled positions by automation, or the replacement of unskilled jobs with positions requiring a high level of technical expertise.
- ► *Human Issues:* Currently, humans deal directly with computers primarily by choice. As computers become pervasive, humans will interact with them through necessity. There will be a variety of sociological consequences, including the necessity of systems designed for naive users, personnel dislocation caused by technical change, and major shifts in the content and style of education.
- ► *Social Issues:* The computer age could bring about a new underprivileged class of the computer illiterate. Women and minorities might make up the majority of this new class by virtue of insufficient technical education. Preventing this scenario will require computer scientists to be aware of the social implications of their work, and a society made aware of the implications this new technology holds.

In response to these issues, universities must broaden the scope of their computer-related offerings in order to prepare students to use the new electronic tools and to adapt these tools to a variety of new uses. We believe that this is best accomplished by teaching students the principles that support current tools; current practices will rapidly become obsolete, and students must be prepared to adapt.

2. Roles for Universities

Computer science has grown rapidly through its short lifetime. Universities have been major contributors to that growth, and they bear a major responsibility for dissemination of knowledge about computer science. Professional education in computer science is growing more rigorous, and we expect an increasing need for students to master a growing set of fundamental concepts. Mere programming skill will no longer suffice for most computing professionals. The field will require solid technical expertise comparable to that expected of engineers, and most development work will require genuine competence in both the application field and computer science. In addition, many people will need to use computers in sophisticated ways and need to understand the implications of the spreading computer technology. Universities must begin now to respond to these emerging needs.

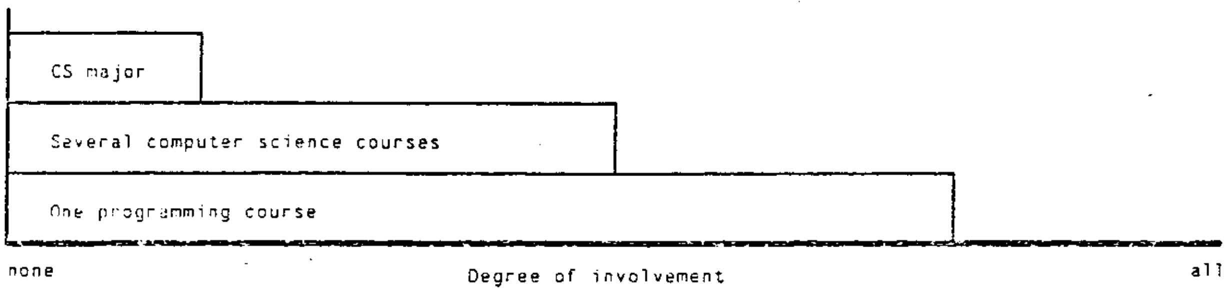
The widespread availability of inexpensive computation will also affect the process of education. Applications will range from direct implementation of routine exercises to innovative systems that present material in fundamentally different ways. Courses that make extensive use of computation will take on the character of laboratory courses; development of computer support, especially of programs suitable for student use, will be at least as difficult as development of new textbooks.

This chapter discusses roles for computing in universities. It begins with an analysis of the audiences for computer science education — the groups of students who need some kind of computer science education. Next, it describes the potential for exploiting computing technology in the educational process. Finally, it assesses the current state of affairs in the computer science curriculum establishment and argues that existing curriculum designs are not adequate.

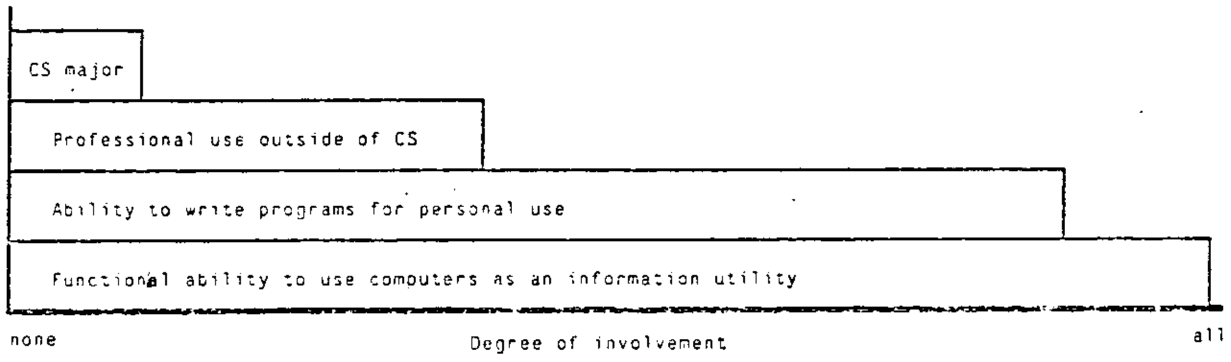
2.1 The Audience

Because of the growing importance of computers in many fields, universities now have a responsibility for teaching several distinct groups of students about computers and about computer science. In this section, we examine the pattern of student involvement with computing and suggest that a significant change in that pattern is taking place. We describe several distinct groups of students and discuss the kind of computer science education each group needs. We identify one group, the computer science majors, as the focus of this report, and we recommend that curriculum design efforts be undertaken for two other groups. In the early 1970s many technical students and a few nontechnical students took some kind of introductory programming course. Perhaps half of these students went on to take a few more computer science courses; these courses usually emphasized programming languages or programming techniques. Only a few students pursued computer science to the depth required of a major.

This pattern of student involvement in computing can be illustrated by the histogram of Figure 2-1(a), whose vertical axis represents increased technical depth in computing and whose horizontal axis represents the fraction of the student body involved. In this figure we see three groups: a modest number of computer science majors, a significant number of students with extensive programming experience and some exposure to the ideas of computer science, and a large number of students with enough programming ability to use computers in their own work. It is important to note that most introductory courses and many of the more advanced courses emphasized computer programming. As a result, the conceptual basis of computer science and the potential of computers for personal information processing were often slighted.



(a) Conventional pattern in the 1970s



(b) Cross-disciplinary pattern expected in the 1980s

Figure 2-1: Profiles of Student Involvement in Computing in the 1970s and1980s

Students graduating in the 1970s with degrees in computer science or with degrees in other disciplines coupled with some computing experience were often employed as programmers. Usually, it was expected that the students would grow into increasingly responsible technical or managerial positions. In most cases their professional growth either lay within computer science or involved an explicit change to, for example, management.

In the 1980s, as computing becomes part of general competence and computer science moves from a narrow specialty to a component of the basic education of every student, this pattern will surely change. There is currently a documented shortage of computer professionals at all levels, from technicians to researchers [10]; this manpower shortage is projected to continue through the 1980s. We believe that a major component of the demand for bachelor's and master's level computer professionals will soon be for students with advanced technical competence in computing as an integral component of computing specializations within other disciplines. Such joint education in computer science and another discipline is now seriously neglected; indeed, many of the students who currently select computer science majors might be better served by computing specializations in other departments. In addition, the criteria for general literacy in the university at large will require education for large numbers of students who will *use* a variety of sophisticated programs and packages but who will do very little *creation* of programs.

From these predictions and the sketch of future trends in Section 1.2 we can project a very different profile for student involvement in computing in the 1980s than we saw in the 1970s. Figure 2-1(b) shows the pattern of student computing experience that we expect to see in this decade. We expect, as before, a modest number of computer science majors. These should be students planning on graduate work in computer science or students who want to work directly within the computer industry. We project the number of computer science majors to be smaller than the demand currently seen in many schools because we expect much of that demand to be redirected to the second group — joint majors who master the fundamentals of both computer science and a second discipline, then specialize in a computational branch of the second discipline. A third group of students, those who want computing expertise for more casual use, may need opportunities (e.g., short introductions to the use of special packages or particular languages) that are not properly provided within an academic department. Finally, we expect that virtually all students will need an introduction to the use of computers and the role of computers in modern society.

The primary objective of this project has been to design a curriculum for computer science majors. We believe that a single curriculum can suffice for both terminal and nonterminal students. Modern computer science requires a core of fundamental material for both groups of students, and a program with flexibility in the use of electives can be tailored to a variety of individual needs.

We also see a need for intermediate-to-advanced computer science training for students who will become computing specialists within some discipline other than computer science. Professional specialization of this sort requires genuine competence in both fields — unlike present applications programming. We see a need for joint majors with closer cooperation between departments than is usually implied by a double major. Such joint programs might include scientific computation (in cooperation with a mathematics or physics department), human-computer interaction (with a psychology department), music synthesis (with a music department), computer-aided design (with a design or architecture department), or information systems (with an economics or management department). Students pursuing these joint majors should take at least the fundamental courses in computer science, the fundamental courses in a second discipline, and additional courses that deal with computing specialization within the second discipline. Carnegie-Mellon has an opportunity to make a bold move by declaring computing specializations to be important and by backing up this declaration with appropriate curricula in a variety of departments.

Many students will want to use computers in personal projects. This will often involve writing programs. These students can be served by the same basic and elementary courses as students who will become more deeply involved in computer science. In addition, there will be a need for education about certain computing topics that do not fall within a computer science curriculum. Many of these are so narrowly directed at specific programs or programming languages that they should not carry academic credit; for these, a program of tutorials in a nonacademic arm of the university should be considered.

An educational program must also be developed for students who will make only casual personal use of computing. The major need is for a fundamentally new introductory course. This course should introduce the nature of computing, show the social implications of widespread computing, make students comfortable accessing an information utility, and develop fluency in the use of packaged software. It should not be a programming course as such, though it should provide some elementary programming experience. This course could also provide an opportunity to introduce nontechnical students to problem solving, deductive reasoning and analytic thinking, in a setting where they could get direct experience and immediate feedback.

We conclude that there are four significant roles for universities to play in computer-related education. These include

- ► Educating future computer scientists at all degree levels,
- ► Educating non-computer scientists who will bring computing expertise to their own fields of specialization,
- ► Educating people who need modest programming skills, and
- ► Educating the entire university population about the potential and use of computers.

This curriculum design addresses only the first group: the computer science majors. We recommend that separate studies be undertaken to study computational specializations in other disciplines and computing education for the University population as a whole. We do not see a major problem with educating students who need only modest programming skills.

2.2 Use of Computing Technology in Education

In addition to organizing the content of computer science, a modern curriculum design must consider ways to use computing technology in the educational process. It is no longer unusual to rely on computers to support courses; applications range from ordinary bookkeeping for course administration through novel interactive teaching systems. It is important, however, to avoid confusing education about the substance of computer science with the use of computers in education. In this section, we are primarily concerned with the use of computers in computer science education. We consider both the use of computers in the general educational process and the specific needs of computer science courses to use computing facilities as course laboratories.

At Carnegie-Mellon and a number of other institutions, computer resources are already commonly used for many of the mundane activities that are part of every course. These include distributing information to students; electronic mail between students, instructors, and teaching assistants; electronic collection of assignments; text processing; and record keeping. Further progress in this area requires a substantial commitment to application software development. This development must involve the users heavily, since the interface to the users is what makes the computer a worthwhile tool in this area. Carnegie-Mellon's new University Center for Design in Educational Computing provides a centralized base for innovative uses of computers in education.

At Carnegie-Mellon, computer support for the students in the courses dates back at least to the 1950s, when the Graduate School of Industrial Administration (GSIA) introduced the "Management Game" (a computer-based management simulation of a detergent industry in which student teams competed for simulated profits). This course (much updated) is still required for second-year Master's students in GSIA. Other current computer-based instruction at Carnegie-Mellon includes microprocessor support for Physics laboratory courses, advanced graphics support for fluid mechanics, color graphics tools for art students, instructional programs for elementary concepts for formal logic, and a data bank of the French Revolution used in a freshman history course. In addition, plans are being made for an advanced computerized engineering laboratory in the engineering college, addition of animation and three-dimensional graphics in the art program, a Computer Music Center for computer-assisted composition and music analysis, and incorporation of computer support in a campus-wide writing program.

Applications such as these are becoming more common in computer science as well. At Carnegie-Mellon, interest dates back at least to a proposal for a laboratory of interchangeable components [6]. More recent activity in computer science includes simulators for abstract machines such as finite-state automata or Turing Machines and difficult-to-understand algorithms (e.g., van Dam's work at Brown [3]); program development environments (e.g., Teitelbaum's program synthesizer [14] at Cornell or Miller's Gnome [9], an adaptation of Gandalf, at Carnegie-Mellon). We can envision other automated tools, and libraries of programs to read, modify, or use as elements of assignments.

There are many more opportunities for imaginative uses here. The more speculative ones include the transfer of research systems to the classroom, including expert systems, theorem-provers, transformation systems, and intelligent advice-givers with user models and tracking of student performance. We anticipate a steady increase in the demand for innovative "course-ware." These applications can support a variety of courses. Some of the course-specific uses of computing technology for educational purposes are indicated in the individual descriptions of courses in Chapter 8.

In computer science, computers are not only pedagogical tools, they are the subject matter of many of the courses. Thus, computer science is an experimental science; computer science courses need undergraduate laboratories, just as physics, chemistry, and biology courses do. Unlike those disciplines, however, computer science laboratories can often share the physical facilities — the costs of establishing and maintaining the laboratories will be largely software costs. Undergraduate computer science courses require software support for the same reasons that more traditional laboratories require construction, maintenance, staffing, and supplies.

2.3 The Establishment

The ACM [1, 2] and the IEEE [8] have made a variety of recommendations on undergraduate curricula. None of these provides a suitable foundation for a curriculum that meets the emerging needs described above.

The major shortcoming of the ACM and IEEE designs is that they seem to be merely summaries of existing curricula rather than projections designed to last for the next decade. We believe, however, that a curriculum design should play a leadership role. Any actual implementation will involve compromise and dilution, so a design should provide a level of aspiration and a direction of development rather than simply an inventory of current practice.

There are two main reasons why we did not simply develop a curriculum directly based on the ACM recommendation. First, the ACM proposal is based more on the status quo in computer education than on any attempt to unify the intellectual content of computer science. Second, the ACM curriculum relegates mathematics to a totally inadequate position — an attitude perhaps appropriate for a data processing curriculum, but not for a computer science curriculum.

The IEEE computer engineering curriculum also lacks unity. That design fails to expose the important common fundamentals joining hardware and software. In addition, its balance of hardware, software, and theory is heavily biased to hardware, and the result is more suitable to computer engineering than to computer science.

In addition to these well-known curriculum designs, we are beginning to see proposals about "software engineering" undergraduate programs as distinct from "computer science" programs. While many undergraduate programs are currently centered on the activity of programming, we believe that software engineering is clearly a subset of what a well-trained computer scientist should know, and we believe that software engineering alone is too narrow a program for an undergraduate degree.

3. Objectives

The curriculum described here was developed in response to objectives set forth in Spring 1982 [12]. The premises and goals from that project plan are reproduced in this section.

3.1 Premises

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

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

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

3.2 Goals

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

Following the Carnegie Plan for education [4, 5, 7, 11], we want to design a curriculum through which a student can acquire:

- ► A thorough and integrated understanding of the fundamental conceptual material of computer science and the ability to apply this knowledge to the formulation and solution of real problems in computer science.
- ► A genuine competence in the orderly ways of thinking which scientists and engineers have always used in reaching sound, creative conclusions; with this competence, the student will be able to make decisions in higher professional work and as a citizen.
- ► An ability to learn independently with scholarly orderliness, so that after graduation the student will be able to grow in wisdom and keep abreast of the changing knowledge and problems of his or her profession and the society in which he or she lives.
- ► A philosophical outlook, breadth of knowledge, and sense of values which will increase the

student's understanding and enjoyment of life and enable each student to recognize and deal effectively with the human, economic, and social aspects of his or her professional problems.

- ► An ability to communicate ideas to others.

The focus of the design will be on a liberal professional education with emphasis on problem-solving skills. Some of the words in the previous sentence are subject to various interpretations. We intend all in a very positive sense. "Liberal education" is broad, including humanities and social science courses plus technical courses outside the student's specialty. Liberal education includes communication skills, both for understanding the work of others and for communicating one's own work. Describing the education as "professional" recognizes the legitimate motivations of many students who value education because they can apply it rather than for pure intellectual enjoyment. "Problem-solving skills" refers to the ability to apply general concepts and methods from a variety of disciplines to all kinds of problems, abstract as well as practical, whose solutions require thought, insight, and creativity. Thus "problems" can range from the proof of a theorem to the design and construction of a specialized computer program and "skills" means creative intellectual ability, not merely the ability to perform repetitive routine actions.

4. Content

This chapter surveys the content of computer science. The objective is to present a coherent view of the conceptual structure of the field and to indicate the scope of our concerns, while indicating connections with other fields. Chapter 8 shows how the material described here is organized into specific courses.

We realize that a unified discussion of ideas and concepts may not lead directly to a good organization for courses. That is, the conceptual structure provided here does not necessarily correspond with the pedagogical organization that forms the foundation of a curriculum design. It is impossible, for example, to convey certain ideas without a background of methods and conventions. Some issues (e.g., reliability, optimization, performance, adaptive design) appear in different forms and a variety of subject areas. Further, courses often focus on some kind of system (e.g., compilers) in order to use a single rich example to bring out a variety of related topics.

Computer science embraces a variety of ideas and modes of scientific thought that must be presented throughout a curriculum for their significance to be conveyed, even though they may appear as a single element of a conceptual organization. It is important for everyone who teaches the courses to present not only the concepts themselves but also an understanding of why these concepts are necessary for a wider understanding of the science. Because of the importance of these distributed ideas, we mention them explicitly here.

- ► *Abstraction and representation.* In a field such as computer science, in which the essential notions are quite abstract, it is important that certain modes of thought be presented explicitly. Perhaps the most important of these is the management of complexity through abstraction and representation. Computer science deals with systems of human design that can appear to be extremely malleable, particularly when realized in software. This malleability belies the problem of handling complexity in such systems. The student must be given a firm grasp of how abstraction is used to control complexity. In this style of thought computer science bears a strong similarity to mathematics. Because mathematicians also deal with systems built upon human imagination, they have developed conceptual tools to manage complexity in the mathematical systems they create. Mathematical maturity and an understanding of how mathematics deals with complexity are essential for computer science students. Other important modes of thought for computer science are discussed later in this chapter.
- ► *Recurrent notions.* Certain particular ideas, such as naming and addressing, binding, state, resource management and allocation, and concurrency, recur in different contexts throughout computer science. It is important for them to be identified as recurrent ideas so students can consolidate their understanding. In our organization, there will usually be a single course charged with presenting an overview of a given idea in varied settings. This course should be sufficiently advanced that the student will have already encountered the topic in several forms; it should be elementary enough that many students will take it.
- ► *Theory and the practice of computing.* A good curriculm must be based on sound theories and models, and it must also teach these foundations in the context of good engineering practice. Most ideas in computer science can be presented in both theoretical and practical settings; these ideas are brought out most effectively when introduced at varying levels of abstraction. This is

true not only because students easily grasp ideas in a concrete setting, but also because the varying range of presentations illustrates how abstract notions are realized in the concrete forms of programs or machines (or, conversely, how practical experience is expressed in abstract terms). It is for this reason that nearly all the courses in this curriculum make connections between theory and practice.

- *Cumulative experience with ideas.* Certain essential ideas must be developed over several courses for them to be completely assimilated. Students are exposed to the ideas early, but they may not be expected to articulate them or synthesize them until much later. For example, students first encounter data types when they learn to program. At this stage they are exposed to the idea, but they do not deal with it as a distinguishable concept. When the first programming language is re-examined in more depth, students perceive types as an identifiable programming concept. Later, students gain experience operating with types, for example by developing realizations for given type specifications. Only after this experience are students able to create new abstract type specifications. Since the development of such ideas must be distributed over several courses, it is necessary that each course instructor understand this progression.

The remainder of this chapter surveys the content of computer science, but without assigning topics to specific courses. The material is organized into four levels of sophistication in computer science, corresponding to the four levels of student involvement suggested in Section 2.1.

4.1 Basics

This basic material provides fluency in the use of computers and familiarity with their capabilities that will be of interest to the broad population of students, not just to computer scientists. Within the next five to ten years, this material will likely be regarded as an essential part of a good liberal education in any discipline. We expect that what is now considered fluency will change from our present notion of programming skill to a very different style of computer usage as the technology improves. The following basic material is expected to follow that change.

4.1.1 Content

Carnegie-Mellon is committed to requiring all students to use computers effectively. The introduction to computing for all these students should provide not merely the clerical skills required to use the computing resources, but also facility in logical and algorithmic thinking and an understanding of the notions of deductive reasoning, cause and effect, time and sequentiality, and state transition. This introduction should also provide students with an understanding of the role of computers in a technological society and of the responsibilities of a professional in the field.

These subjects constitute minimal literacy for a computer-based society:

- Basic computer literacy (i.e., as a naive user): what computers can and cannot do, dealing with an information utility (creating and using files), using existing programs and packages.
  - The office/file cabinet model of computation
  - Gaining access to simple useful facilities such as
    - Computer mail
    - Simple files and campus data bases (the library card catalog, class schedules, etc.)
    - Personal data bases (calendars etc.)
    - Text formatting and writing tools (spelling checking etc.)

- Using interactive programs and packages (interactive spreadsheets etc.)
  - Non-textual interaction, drawing packages, painting and layout
- Networking and Communication
  - Using information services (bulletin boards, mailing lists, etc.)
  - Using data facilities (central file servers etc.)
  - The implications of information sharing

► Elementary facts about computers: organization, architecture (processors, primary and secondary memories, communication), concept of stored program.

- Processors and memory
- The fetch/execute cycle
- Storage devices (especially personal dismountable storage: disks, tapes, etc.)
- Representation of information with binary devices
  - Binary numbers, encoding (e.g., character sets), instructions as data
- Networking, how information is shared between computers

► Elementary facts about programs: concept of algorithm, simple program structures, including control structures and procedures, declaration and use of data, use of libraries.

- Computation and sequential execution (following directions)
- Simple programming in a high level language
  - Using and writing procedures (but not recursion)
  - Using integers, reals, strings, vectors, records (but not pointers)
  - File input and output

► Thinking about computers and programs: problem solving ability, programming technique, and concepts of correctness and performance.

- Elementary problem-solving
- Mechanics and discipline of writing programs
  - Simple program forms (filter on text file, summarization of data, etc.)
  - Documentation
  - Incremental coding strategy
  - Debugging strategies, including data selection
- Correctness and the fact that programs can be reasoned about precisely
- Costs and the existence of time/space tradeoffs

► Role of computers in society: range of potential applications, appropriate and inappropriate use of computers.

- History of computers and their use
- View of computers as providing an information utility
- Potential future applications (current AI research provides examples)
- Social issues (computer crime, security and privacy, consumer issues, etc.)
- Ethics of computer use in an electronic community
- Careers in computers, technological displacement
- Impact of computers and robots on industry and employment
- Psychological and social aspects of computation
- Examples of applications

4.1.2 Skills

Skills developed in the basic curriculum include problem solving and simple deductive reasoning, the ability to carry on simple interactions with a computer such as using selected programs, and sufficient familiarity with computing to learn more as required.

4.2 Elementary Computer Science

These topics provide the foundation for a computer science degree, but they are relevant to others interested in computer science as well as to computer scientists. Thus this material should be considered for joint programs with other disciplines.

Generally, this is material for sophomore and junior courses. Many of the topics are fundamental, in the sense that they form the basis for development of more advanced material. The material also fosters functional fluency with contemporary systems. These skills are immediately useful; they provide experience in the use of the fundamental ideas, and the resulting experience makes richer examples accessible.

We organize the material into three rough categories, corresponding to the body of material itself, common modes of thought that students should be aware of, and skills associated with the material covered.

4.2.1 Content

This section outlines our view of the body of elementary computer science. It should be clear that some of these topics are drawn from other disciplines (such as mathematics and electrical engineering). Both theoretical and practical topics are included throughout the categories listed here.

- ➤ The nature of computation. Concept of algorithm; relation between algorithm and program. Elementary automata theory. Supporting material from discrete mathematics.
    - Time, sequentiality, and concurrency
    - Algorithm, state
    - Finite-state automata as model of computation (introduce additional power of Turing Machine)
    - Relations between algorithms, programs that express them, and machines that execute them
    - Probabilistic algorithms and heuristics
    - Discrete mathematics:
        - Inductive definitions and proofs
        - Linear algebra, graphs, functions, relations
        - Propositional logic and proofs; set theory; boolean algebra
    - Elementary notions of calculus and numerical analysis
- ➤ Computer organizations. The von Neumann model and machine/assembly language. ISP-level and PMS-level organizations, elementary network issues. (At this level, the study is fairly superficial — the objective is understanding the structures in order to use them, rather than to be able to design new ones. Supporting material from electrical engineering, including electricity, circuit design, and device characteristics.)
    - Digital Logic level
        - Basic digital concepts and terminology
        - Combinatorial circuits
        - Discrete time abstraction (clocks)
        - Circuit family abstraction
    - Register Transfer Level
    - Program level
        - Instruction formats and how they get interpreted
        - Concept of microcode
        - Architecture as specification of instruction set
    - Processor - Memory - Switch level
        - Properties of processors
        - Classes of switches (busses, crosspoints, etc.)
        - Memory technology
            - Characteristics of disks, tapes, drums, etc.
            - Memory hierarchy

- Addressing techniques, data representation and register transfer
- Physical memory and addressing techniques
- Virtual memory and address mapping

I/O and bus structures

Examples

- Calculators, microprocessors, and microcomputers
- Minicomputers and mainframe computers
- Multiprocessors, supercomputers
- Control synchronization

▶ Program organizations. Organization of simple programs and elementary modular composition. Data structures and some common program forms. Elementary concurrency issues. Reasoning about correctness and performance. (The objective is design as well as use; the rudimentary programming skill from the basic material is now refined to a useful level.)

- Program development methods
  - Structured programming
  - Use of specification and verification
  - Documentation
  - Debugging and testing
- Program organizations
  - Data organization primitives (pointers, hashing, encoding, packing, etc.)
  - Implementations of data types
  - Abstract data types and their specification
  - Some classical program organizations (filters, abstract data types, pattern-matching systems, table-driven interpreters, etc.)
  - Imperative and applicative programming
  - Recursion
  - Matching data with control
- Some classical algorithms (sorting and searching, numerical algorithms from linear algebra, etc.)

▶ Languages and notations. Programming languages. An appreciation of the power of good notation. Syntax and language description. Examples such as BNF and regular expressions. (Students should appreciate the language component of any interface design, and be aware of the influence on design of pragmatic issues. Supporting material from discrete mathematics.)

- Language as communication, interface medium, means of shaping ideas
- Syntax and semantics
- Formal language issues.
  - Syntax: regular and context-free languages; hierarchies of languages
  - Semantics: denotational; operational; axiomatic (Hoare-axiom or predicate-transformer)
  - Formal specification techniques: axioms and models
- Classical programming language matters
  - Organization of program control: iteration and recursion
  - Functions, procedures, and exception handlers
  - Data structures and declarations
  - Scope, extent, and binding, including parameters
  - Expression evaluation
  - Abstraction facilities (procedures, types)
  - Specification
- Kinds of languages: applicative and procedural
- Specialized languages
  - Production systems
  - Query languages
  - Graphical interaction
  - Semi-languages such as RPG, Visicalc, Makefile, editors

▶ Design techniques. Advanced programming and specification techniques. Also, relevant ideas from hardware design and other design disciplines.

- Advanced programming techniques
- Specification methods and languages
- Decomposing programs into modules
- Design tools
- Documentation
- Contemporary approaches to problem-solving
- Devising and evaluating alternatives
- Evaluation criteria

► Evaluation and analysis. Analysis of algorithms, elementary models for performance. Criteria used for evaluation (correctness, speed, space, reliability, generality, complexity, etc.) and tradeoffs among them. Supporting material from discrete mathematics.

- Correctness
  - Specification and verification
- Performance
  - Formal models
  - Bottleneck identification and elimination
- Analysis of algorithms
- Models and modelling
  - What models are and how to use/construct them
  - Empirical vs analytic models
  - Validation
  - Specific models (at this level, introduction only)
    - Queueing-theoretic models for operating systems and hardware
    - Productivity and life-cycle models and their limitations
- Human factors

► Advanced personal use of computers. Text manipulation, personal data bases. Using an operating system (command files, Unix pipes, etc.). Graphical interaction. Access to libraries. Appreciation of what makes computers easy and hard to use. Supporting material from design and psychology. Reasonable and unreasonable social behavior; computing as a valuable commodity.

- Small examples of program development
- Practical matters:
  - Program segmentation and linkage
  - Linkers and loaders
  - Error recovery techniques
  - Systems and utility programs
- Text retrieval and processing (editing and document preparation)
- Introduction to operating systems concepts
- Batch, timesharing, and personal (dedicated) systems
- Elementary software engineering
  - Debugging, preventing debugging, test data selection
  - Organization of programming teams
  - Program organization for maintainability
  - Verification
  - Software libraries
- Ethics, privacy, implications of having a user community

► Some larger systems as examples. Study of systems large enough for complexity to become an issue. (The point here is to generate some elementary familiarity with the systems and some of the issues — the hard problems of design and analysis come later.)

- Compilers (relatively small complex system, but well-understood)
- Cognitive models (the human as a complex system)
- Data bases (complexity of both size and interaction over time)
- Large software systems, such as operating systems (concurrency issues)
- Distributed systems

4.2.2 Modes of Thought

The following paradigms of computer science thinking are illustrated in the topics listed above. Students will be better able to assimilate the technical material if they perceive the role of these paradigms.

- ► *Hypothesis and test.* That is, the classical scientific method. Models and their validation. Generalization as a technique for refining hypotheses.
- ► *Problem solving.* Finding and exploiting structure. Tradeoffs between generality and efficiency. Heuristic exploration of problem spaces.
- ► *Analysis and synthesis.* Managing complex systems by decomposition into parts. Development of systems on the basis of structural organization. Quantitative techniques.
- ► *Abstraction and realization.* Abstraction as control of complexity and detail. Realization as a process of binding underlying structure to implement an abstraction.
- ► *Inductive reasoning.* Drawing conclusions from limited observations.

4.2.3 Skills

There are certain skills that are useful and important for students to have when they interact with the ideas described above. These skills include simple programming skills such as coding and debugging, basic hardware logic design techniques, and the various mathematical skills related to discrete mathematics, such as inductive proofs and an ability to manipulate propositional calculus formulas.

4.3 Liberal Professional Education

These topics serve to make the computer scientist a well-rounded professional, able to appreciate the significance of work in other disciplines and able to relate his computer science expertise to problems outside computer science.

This section also provides a taste of the interdependencies between computer science and other areas and suggests areas in which joint degrees might be appropriate.

4.3.1 General Scope

The curriculum for a liberal professional education must define the general coverage as well as the core material in the field of specialization. For us, that means a set of inclusive statements about the total scope and some more specific statements about areas that are related to computer science.

Note that a liberal professional education in other disciplines may require joint majors with or computing specializations in those fields. In this section we are dealing solely with the problem of a liberal professional education in computer science itself.

4.3.2 Liberal Education

We believe that students should be broadly educated. Our definition of a broad education includes mathematics, science, and engineering as well as humanities, social sciences, and the arts. A broad education is possibly more important in computer science than in other disciplines for two major reasons: First, computer science is strongly tied to many other disciplines. As computers become more prevalent, the range of related disciplines can be expected to increase further. Second, there is an overwhelming need for literate citizens to interpret the field to others.

4.3.3 Areas Related to Computer Science

The boundaries of computer science overlap with several other areas. Some of the material at the boundary should be developed (and possibly taught) with other departments. Specific offerings at the boundary should arise from close cooperation between the interested departments.

The following sections list topics where there is likely to be overlap between computer science and related fields. However, the list is by no means definitive. If curricula are established to educate non-computer scientists to specialize in computing aspects of their fields, it is likely that there would be many new courses that explore highly specific computing problems of various disciplines. Few such courses are included in this section.

4.3.3.1 Mathematics and Statistics

Related courses in mathematics and statistics might cover such topics as:

- Probability and statistics
- Combinatorics
- Modern algebra
- Linear algebra
- Numerical analysis
- Scientific computation, especially applications of linear algebra and numerical analysis

4.3.3.2 Electrical Engineering

Related courses in electrical engineering might cover such topics as:

- Circuit theory
- Solid state electronics and semiconductor devices
- Communications
- Control theory
- Information theory

4.3.3.3 Physics

Related courses in physics might cover such topics as:

- Electricity and magnetism
- Solid state physics
- Computational physics

4.3.3.4 Psychology

Related courses in psychology might cover such topics as:

- Cognitive psychology and information processing
- Problem solving
- Artificial intelligence
- Human factors

- ► Psychological linguistics
- ► Perception

4.3.3.5 Mechanical Engineering

Related courses in mechanical engineering might cover such topics as:

- ► Mechanical linkages, particularly as they relate to robotics
- ► Computer-assisted manufacturing

4.3.3.6 Management and Information Science

Related courses and topics in management and information science might cover such topics as:

- ► Operations research, particularly optimization
- ► Economics, especially project scheduling and estimation
- ► Management, especially relating to automation and to high-technology development
- ► Role of computers in organizations; how organization structures interact with information flow

4.3.3.7 Public Policy

Related courses in public policy might cover such topics as:

- ► Social implications of large-scale computing
- ► Consumer issues in personal computers
- ► Policy issues arising from computing and communications
- ► Computer models for policy analysis
- ► Legal issues such as ownership of software, liability, and security

4.4 Advanced Computer Science

This material is of interest to specialists. Whereas all computer science students would be expected to master the previous material, specialization begins here. The depth intended here is at or just below first-year graduate level; as a result this material might form part of a master's degree curriculum. This is not to say that we see an undergraduate education as covering the graduate curriculum, but rather that we believe that a senior undergraduate should master that depth in *selected* areas.

The organization given here has been driven by the content of the material; it is not to be misconstrued as a course organization. For example, a course in software engineering might cover much of the material listed under "Systems" and "Process" and also reenforce previous topics in the other areas. Likewise, a course similar to the traditional compiler course might be retained — not to teach compiler building, but to exhibit a medium-sized system with a well-understood structure and to take advantage of the time invested in the example by using it to cover advanced material in data structures, application of formal methods (parsing theory), and interface construction.

Within each of the areas listed below we would expect to find contributions from the traditional areas of theory, software, and hardware. We hope that this organization will avoid inappropriate compartmentalizations. We also hope that it will stimulate thought about the interactions among historically disparate areas and simplify the inclusion of individual topics that haven't grown into course-sized entities.

4.4.1 Control

This area includes scientific and engineering aspects of algorithms, especially as expressed in programs running on computers. Topics such as the function of CPU's, sequentiality and concurrency of processing, use and analysis of algorithms, correctness of algorithms, fault tolerance, probabilistic and heuristic algorithms are all aspects of control.

Many of these concepts appear repeatedly: for example, the notion of concurrent processing appears in various guises in numerous contexts. These include hardware circuits, interrupts, communication protocols, software synchronization mechanisms such as semaphores and monitors, software process constructs such as coroutines and tasks, data base transactions, and operating systems policy for scheduling and allocation.

4.4.2 Data

This area encompasses the manipulation and representation of information, in computation and especially in computer programs. Thus notions of state, physical storage devices, addressing and accessing methods, types, representation, specification, encryption, and "quantity" of information are all included here.

These topics also appear throughout computer science. For example, the notion of naming or addressing appears in hardware addressing (direct, indirect, virtual), memory hierarchies, program variables (scope, extent, binding), operating system storage policies (working sets, overlays, virtual memory management), database models, file directories, and file access methods.

4.4.3 Systems

A system is a regularly interacting or interdependent group of software or hardware modules which form a unified whole. The study of systems includes the identification, quantification, and management of complexity in systems, the design and construction of large systems, the evaluation of performance, reliability, and security of systems, and how systems are distributed and how communication is performed.

4.4.4 Language

The representation of programs (as opposed to the representation of data) characterizes the study of language. Thus the language area includes the ideas of notation, syntax, semantics and the study of traditional programming languages and their implementation and specification. This area also includes issues of user interfaces, human factors, and technologies such as speech and graphics.

4.4.5 Foundations

These topics are predominantly scientific; the results are applied in engineering contexts. At the moment, they are mostly mathematical and theoretical; there are perhaps many non-theoretical topics which should be placed here. Also some portions of these topics are included in the elementary computer science section; inclusion here indicates study at a more advanced level. Sample topics are the study of computability and complexity, queuing theory, graph theory and information theory. The study of modeling as a tool for analysis is also included here. The ability to formulate and analyze empirical and theoretical models is essential.

4.4.6 Process/Design

This area covers the management of complexity, especially when human behavior or performance is involved. It includes engineering considerations pertinent to development (e.g. readability and maintainability of code) and techniques for managing the design and development of large systems (e.g. instrumentation of programs to aid in debugging and performance evaluation). It also addresses the economics of software including creation costs, maintenance costs, and life cycles.

4.4.7 Communication

This area covers topics related to the transmission of information. As computer architectures become more distributed, the transmission of information between them becomes a key feature. Personal and home computers are being used more and more for storing, processing, and sharing information and less for computational purposes. This area includes both the methodologies used to achieve information transmission and the implications of widespread access to information. It also covers methods for preventing such access, i.e. information security.

4.4.8 Applications

These include both applications that depend mainly on ideas from computer science and applications that are important because of their use of computer science material, but that are not computer science in and of themselves. The first category includes applications that are taught to synthesize knowledge from various parts of computer science and present these ideas as parts of a working system. Examples are compilers, operating systems, graphics, and some artificial intelligence programs. The second category includes systems which are rich in their use of computer science techniques. Examples are large financial systems, airline reservation systems, commercial database applications, CAD/CAM systems, remote sensing, and CAI.

5. Program Organization

This chapter presents a set of requirements and a suggested curriculum for a complete undergraduate computer science major. In keeping with our prediction of the responsibilities of future computer scientists, considerable flexibility is provided to allow for professional breadth and the computer science component places heavy emphasis on fundamental concepts. To this end, a number of electives are specifically constrained to mathematics courses, a number of electives are specifically constrained to be nontechnical, and a concentration in an area outside of computer science is required. Even if the curriculum is adopted without creating a major, this brief discussion of the sort of program that could be based on the curriculum provides a good global perspective.

Section 5.1 tabulates the requirements for the program. For both philosophical and practical reasons we have allowed considerable flexibility in the choice of electives. We have done this to provide students and their advisors with the opportunity to construct focussed programs tailored to students' interests, not as a sort of permissiveness. Section 5.2 describes our intentions about the use of electives; it takes the form of advice to advisors. To show that the program is actually achievable, Section 5.3 shows how several versions of the program could be scheduled in four years.

5.1 Requirements

We assume that a normal load is five courses per semester for eight semesters and that college requirements dominate the first year. The distribution of courses in the last three years is roughly 35% to computer science and mathematics, 15% to other technical electives, and 30% to humanities, social sciences, and arts. There is considerable flexibility in the remaining 20%.

We expect that the electives will be used to form focussed, coherent programs; in many cases computer science courses will be coupled to specializations in a non-computer science area. Wise use of electives depends critically on individual advising; electives should be chosen in keeping with an overall plan rather than as isolated decisions each semester. Good use of the electives may be encouraged by publishing examples of approved specializations and providing a review and approval mechanism for individual programs.

A total of 40 courses are required. The distribution is

Freshman year (controlled by college)		10	
Computer Science and Mathematics		10	
Specific required courses	4		
Constrained Computer Science 3xx courses	2		
Advanced Computer Science (4xx)	2		
Constrained Mathematics courses	2		
Technical courses		5	
Nontechnical courses (Humanities, Social Sciences, and Fine Arts)		9	
Electives outside the Computer Science Department		6	
Non-CS Concentration [constructed from other electives]			
Total			**40**

These course counts can be converted (approximately) to Carnegie-Mellon "units" by multiplying by 9. The specific course requirements are somewhat more liberal than the requirements of other departments. The elective structure provides considerable flexibility for adapting the program to joint degrees.

The specific requirements for the program are as follows:

► *Freshman Requirements (10 courses):* We believe that college requirements will dominate the freshman year. This design therefore reserves space for a year's worth of courses required by the college (including freshman writing and history). Although this delays the student's entry into computer science courses, it preserves flexibility for selection of a major at the end of the freshman year. We expect that for students likely to enter computer science the freshman courses will include at least

PROGRAMMING AND PROBLEM SOLVING [110],
DISCRETE MATHEMATICS [150],
CALCULUS I [MATH 121],
PHYSICS I [PHYS 121],
two other courses in natural sciences or mathematics
one course each in writing and in history or social science
two other courses

We believe that the six courses not named should be broadly distributed, so we make no additional constraints.

► *Computer Science and Mathematics (10 courses):* Four specific courses are required in addition to those in the college core:

FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211],
FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212],
REAL AND ABSTRACT MACHINES [240],
ALGORITHMS AND PROGRAMS [330].

Four more courses are constrained within the computer science department, and two courses must be taken within the mathematics department. These include:

> One 300-level system or software course (from TIME, CONCURRENCY, AND SYNCHRONIZATION [310], COMPARATIVE PROGRAM STRUCTURES [311], LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]).

> One 300-level theory or approved mathematics course (from FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350], LOGIC FOR COMPUTER SCIENCE [351], NUMERICAL METHODS [MATH 369].)

> Two 400-level computer science courses.

> Two mathematics courses (from COMBINATORIAL ANALYSIS [MATH 301 / CS 251], OPERATIONS RESEARCH I [MATH 292], LINEAR ALGEBRA [MATH 341], NUMERICAL METHODS [MATH 369], MODERN ALGEBRA [MATH 473], LARGE-SCALE SCIENTIFIC COMPUTING [MATH 712 / CS 453], PROBABILITY AND APPLIED STATISTICS [STAT 211 / CS 250]).

► *Technical Courses (5 courses):* These may be selected from courses in science and engineering departments plus selected technical courses in other departments.

► *Nontechnical courses - Humanities, Social Sciences, and Fine Arts (9 courses in addition to freshman requirements):* Some of these courses are constrained by college and university requirements. In addition to those requirements, three courses are constrained as follows:

> POLICY ISSUES IN COMPUTING [EPP 380 / CS 380],

> A writing course in addition to the freshman writing requirement,
> Another course with a substantial writing component.

► *Electives (6 courses):* Electives are to be selected to support objectives established jointly by the student and his or her advisor. These six electives must be chosen from outside the Computer Science Department.

► *Non-CS Concentration:* A concentration of at least three related nonintroductory courses in an area other than computer science is required. Some possibilities will be recommended; students may propose others for approval. Although the concentration may be in the mathematics department, mathematics courses taken as part of the Computer Science and Mathematics requirement may not be used to satisfy the concentration requirement.

Although most of the discussion here has dealt with bachelor's degrees, the curriculum would also support a master's degree. The 4xx courses provide master's-level depth; we believe that undergraduate students should achieve this depth in one or two areas. A master's program would require more breadth at that level; it would also include a master's thesis. We have not addressed the question of whether specialized topics such as software engineering are suitable programs for a master's degree; certainly such explicit specialization is more appropriate at the master's level than at the bachelor's level.

5.2 Advice on the Use of Electives

The design of a degree program might take either of two forms. The course sequence and requirements could be so tightly constrained and highly specified that a strong program is guaranteed. The disadvantage of this style is that only the variations anticipated by the designers are likely to be accommodated by the program in any reasonable way. As an alternative, the requirements could be left sufficiently flexible that many different strong programs can be constructed. The weakness of such a flexible approach is that weak programs also become easy to construct, either by design or by error.

We have decided in favor of flexibility in this design because we feel that the field of computer science is still so fluid that we cannot accurately predict what it will look like in a decade. The price of this decision is that the responsibility of the advisor in helping design the individual course of study is increased. The faculty advisor must spend considerable time with each advisee, understanding his strengths, weaknesses, and interests and providing firm guidance to ensure that each computer science major receives an excellent education. This will only work if the number of students is small in comparison to the advising faculty. If individual advisors are responsible for too many students, the flexible alternative may not be feasible. In that case it would be necessary to specify allowable programs more rigidly.

Because of the cross-disciplinary nature of computer science, every computer science major ought to have significant exposure to advanced material in some field other than computer science. For that reason we have introduced the requirement for a non-CS concentration, a sequence of at least three non-introductory courses in any other field. This concentration may be in a technical or non-technical field. We imagine that the main areas that will be selected are electrical engineering, mathematics, and psychology, but we also want to encourage students to consider concentrations in fine arts, humanities, social science, the physical sciences, business, or any other area offered at an advanced level by the university. As a result of our commitment to this breadth, our specification of a computer science major has substantially fewer specific requirements than most other majors. However our intention is to provide a more rigorous, not a less rigorous, overall program.

We intend that most students take more mathematics than is required in the proposed program. Combinatorial analysis, linear algebra, operations research, numerical methods, graph theory, probability, and statistics are all extremely valuable in many areas of computer science and are commended to the attention of students and advisors.

The typical undergraduate program, as we envision it, draws approximately two-fifths of its content from computer science and mathematics, one-fifth from other technical areas, and two-fifths from humanities, social sciences, and fine arts. This appears to be considerably broader than many of the existing technical and engineering majors at Carnegie-Mellon; we feel that this is appropriate. We expect the number of introductory courses taken to be fairly small. We strongly discourage the kind of "breadth" that comes from intellectual dilettantism, particularly when the symptom is a plethora of introductory survey courses.

5.3 Example Programs

To show the feasibility of this program and some of the ways it can be adapted to the needs of individual students, we show plans for a few particular instantiations. These plans should be interpreted as one way, but certainly not the only way, to schedule courses into semesters and to satisfy requirements. Naturally, courses may be reordered as long as prerequisite requirements are satisfied. In particular, electives shown in the senior year may be exercised earlier, courses restricted to particular topics may be taken in the senior year, and the non-CS concentration may be taken at some time other than that shown here.

The first example shows how a reasonably balanced program could be organized. The remaining examples are extreme cases, designed to illustrate the flexibility of the program: they are not suggested programs. These examples are a mathematics concentration, an electrical engineering concentration for a student who wishes to take as many technical courses as possible, and a psychology concentration for a student who enters the computer science program late, after pursuing the social sciences for three semesters.

5.3.1 Balanced Program

The sample program shown here might be designed by a student seeking a balanced program. In this example we have assumed an interest in systems, but not one that is so overriding as to produce a skewed program.

Freshman		Sophomore		Junior		Senior	
Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
CS 110 Program'g & Prob. Solving	CS 150 Discrete Math	CS 211 Fundamental Struc I	CS 212 Fundamental Struc II	CS 330 Algorithms & Programs	CS 320 Lang. Int & Processors	CS 412 Resource Management	CS 420 Transducers of Programs
Math 121 Calculus I	Math 122 Calculus II	CS 240 Real&Abstr Machines	Elective	CS 310 Time, Synch Concurrency	CS 350 Formal Langs Aut, Cmplxty	Math 341 Linear Algebra	CS 400 Senior Project
Phys 121 Physics I	Freshman Year Requirement	Stat 211 Probability & Statistics	Math 301 Combin-atorics	<--------- Non CS Concentration --------->			Elective
Freshman Year Requirement	Freshman Year Requirement	Writing Course	H&SS or CFA Elective	Another Writing Course	CS/EPP 380 Policy Issues in Computing	H&SS or CFA Elective	H&SS or CFA Elective
Freshman Year Requirement	Freshman Year Requirement	<------- Humanities, Social Sciences, and Fine Arts (subject to college restrictions) ------->					

5.3.2 Mathematics Concentration

The example program below might be designed by a student with a strong interest in computer science and mathematics. The concentration requirement is fulfilled by either the sequence of physics or chemistry courses or by a selection of the mathematics electives. Many students currently at Carnegie-Mellon major in mathematics with a concentration in computer science. For many of these students, this would be an appropriate course of study.

Freshman		Sophomore		Junior		Senior	
Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
CS 110 Program'g & Prob. Solving	CS 150 Discrete Math	CS 211 Fundamental Struc I	CS 212 Fundamental Struc II	CS 330 Algorithms & Programs	CS 351 Logic for CS	CS 450 Theory of PLs	CS 451 Complexity
Math 121 Calculus I	Freshman Year Requirement	CS 240 Real&Abstr Machines	Math 301 Combin-atorics	CS 350 FLAC	CS 3xx Restricted (310,311,320)	Math 341 Linear Algebra	Math 473 Algebra
Phys 121 Physics I	Freshman Year Requirement	Stat 211 Probability & Statistics	Math 259 Ordinary Diff Eqns	<------ Physics or Chemistry Electives ------>			Math 369 Numerical Methods
Freshman Year Requirement	Freshman Year Requirement	Writing Course	H&SS or CFA Elective	Another Writing Course	CS/EPP 380 Policy Issues in Computing	H&SS or CFA Elective	H&SS or CFA Elective
Freshman Year Requirement	Freshman Year Requirement	<-------- Humanities, Social Sciences, and Fine Arts (subject to college restrictions) -------->					

5.3.3 Electrical Engineering Concentration

In this case will will assume a student with a strong interest in electrical engineering that wishes to take the maximum number of technical courses as soon as possible. This example is quite rigorous, involving one of the more difficult sequences from Electrical Engineering. This is only one example, many others could be constructed.

Freshman		Sophomore		Junior		Senior	
Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
CS 110 Program'g & Prob. Solving	CS 150 Discrete Math	CS 211 Fundamental Struc I	CS 212 Fundamental Struc II	CS 330 Algorithms & Programs	CS 3xx Comp Sci Elective	CS 440 Computer Architecture	CS 441 VLSI
Math 121 Calculus I	Freshman Year Requirement	CS 240 Real&Abstr Machines	Electrical Engr	CS 3xx Restricted (310,311,320)	CS 3xx Restricted (350,351,369)	Elective	CS/EPP 380 Policy Issues in Computing
Phys 121 Physics I	Freshman Year Requirement	Stat 211 Probability & Statistics	Math 259 Ord Diff Eqns	Elective	Writing Writing	Another Course	H&SS or CFA Elective
Freshman Year Requirement	Freshman Year Requirement	EE 101 Linear Circuits	EE 102 Electronic Circuits	EE 221 Electronic Circuits	EE 222 Digital Circuits	H&SS or CFA Elective	H&SS or CFA Elective
Freshman Year Requirement	Freshman Year Requirement	Physics	Physics	<-------- Humanities, Social Sciences, and Fine Arts -------->			

5.3.4 Psychology Concentration

Here we will assume a student with interests in cognitive psychology and artificial intelligence. To illustrate the flexibility of the program we will assume that the student has interests in social science and public policy, and decides to enter the computer science program after taking only Calculus I but no other relevant courses in the first three semesters.

Freshman		Sophomore		Junior		Senior	
Fall	Spring	Fall	Spring	Fall	Spring	Fall	Spring
Freshman Year Requirement	Freshman Year Requirement	H&SS Course	CS 110 Program'g & Prob. Solving	CS 211 Fundamental Struc I	CS 212 Fundamental Struc II	CS 330 Algorithms & Programs	CS 460 Cognitive Processes
Freshman Year Requirement	Freshman Year Requirement	H&SS Course	CS/M 150 Discrete Math	CS/EE 240 Real & Abstr Machines	CS Restricted (350,351,369)	CS 3xx Restricted (310,311,320)	CS 461 Robotics
Freshman Year Requirement	Freshman Year Requirement	Math 121 Calc I	Math 122 Calc II	Stat 211 Probability & Statistics	Math 301 Combin-atorics	CS 360 Artificial Intelligence	CS 3xx Comp Sci Elective
Freshman Year Requirement	Freshman Year Requirement	PSY 113 Problem Solving	PSY 213 Inf Proc and AI	<-------- Technical Electives -------->			CS/EPP 380 Policy Issues in Computing
Freshman Year Requirement	Freshman Year Requirement	<-------- Humanities, Social Sciences, and Fine Arts (subject to college restrictions) -------->					

6. Remarks

The curriculum presented here departs from traditional curricula in a number of ways. Further, some of our objectives do not appear explicitly in the design. This chapter presents remarks about the curriculum that may help the reader to interpret and evaluate it.

6.1 General Philosophy

We believe that previous computer science curricula have been too compartmentalized. They contain many courses focused on specific areas, but offer little or no overview of the material. We attempt to organize courses by coherence of the content, not necessarily following traditional boundaries. We also attempt to blend theory and practice in all courses. We believe that a strong emphasis on recurring themes (e.g. abstraction and reliability) will help to bridge the gaps between topics that, on the surface, may seem unrelated.

Though we stress "foundations" and "unifying concepts", we also intend to teach students to appreciate and produce specific solutions to specific problems. We realize that in many respects this is best achieved by exposing students to a wealth of examples of good "engineering solutions" and by providing large amounts of supervised "hands on" experience of the sort that lectures and examinations simply can not provide.

We also feel that the "liberal professional" goal is best served by breadth. An undergraduate who overspecializes can graduate unable to learn on his own, and unable to communicate his knowledge to laymen. We believe that specialization can and should happen after graduation, whether the student enters the work force or graduate school. Our goal is to produce an individual with a broad base on which to build additional knowledge, not a worker with skills that may be obsolete shortly after graduation.

Certain "themes" run throughout the curriculum, and are discussed in Chapter 4. Although these topics are not mentioned explicitly in most courses, all carry the responsibility to convey those ideas through their tone and examples.

We strongly feel that a curriculum for computer science must have support from current technology. Software support for undergraduate courses is essential. Section 2.2 discusses this in more detail. Students must have convenient and substantial access to computer resources. This access is vital. Computer technology can also be applied in more enterprising applications than those common today. In addition to routine applications, computers can be used in creative settings; examples include teaching via intelligent, advice-giving programs, and other kinds of computer-aided instruction. Extensive experience with current technology at this early stage will prepare the student for the real programming world.

6.2 Relation to Traditional Courses

Many of the concepts of computer science appear in nontraditional contexts in this curriculum. This is often because older courses were organized around artifacts such as computers or software systems, whereas our design tends to be organized around ideas.

As a particular instance, there are no courses specifically about operating systems or compilers. Both of these traditional courses take as an integrating theme a complex system such as a compiler or operating system from which most of the main ideas are naturally motivated. This curriculum, however, organizes these topics by grouping similar abstract ideas. As a result, the major concepts from an operating systems course appear in TIME, CONCURRENCY, AND SYNCHRONIZATION [310], COMPUTER ARCHITECTURE [440], and RESOURCE MANAGEMENT [412]. The topics from traditional compiler and comparative languages courses are distributed through LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320], TRANSDUCERS OF PROGRAMS [420], and ADVANCED PROGRAMMING LANGUAGES AND COMPILERS [421].

FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I AND II [211/212] is the introductory sequence for computer science students. It replaces what is usually a second programming course with one that introduces many important concepts (abstraction, representation, correctness, performance analysis) as early as possible. It also has a significant programming component. This sequence has been taught at Carnegie-Mellon for several years, and we are very satisfied with it as an introduction to computer science and a foundation for further study.

Early courses usually deal with building programs from individual statements, while a software engineering course (e.g. SOFTWARE ENGINEERING [410]) deals with the Interaction of whole modules in a complete system. A new course, COMPARATIVE PROGRAM STRUCTURES [311], is intended to cover the intermediate stage. Its emphasis is on the common frameworks for building modules from code fragments.

Two courses in this curriculum are descendants from traditional courses, but with a significant shift in emphasis. REAL AND ABSTRACT MACHINES [240] is an introduction to hardware that includes relevant material from programming systems and automata theory with the explicit intention to bridge the normal distances among those areas. ALGORITHMS AND PROGRAMS [330] is an algorithms course with a highly pragmatic bent — blending the more usual "Abstract Algorithms" and "Advanced Programming" courses.

Several of the intermediate courses are derived from traditional courses with modest changes in emphasis. These include FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350] and INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360].

6.3 Course Organization and Style

By longstanding tradition a "course" is a series of lessons long enough to fill a semester and containing a reasonable amount of intellectual content. The honor accorded by tradition should not be permitted to mask inadequacies of the course format; nor should skepticism be permitted to destroy the well-reasoned product of our predecessors. This curriculum focuses on the course as the atomic unit of instruction, not because of a blind devotion to tradition, but because we found no compelling alternative. Our commitment is to excellence in instruction of the next generation of computer scientists, not to the provision of particular courses or degrees.

We are not certain that every aspect of computer science education is best served by the traditional course structure. Alternatives to the traditional emphasis on the "course" include the Oxford/Cambridge tutorial model, various work/study programs, and the competency examination model. This curriculum should be seen as an interim solution, realizing that some concepts may be best taught via structures not yet imagined.

Computer science courses often resemble physics laboratory courses in their need for direct observation and manipulation of the phenomena of computer science. Consequently, computer science courses depend on laboratory facilities and staff to create and operate those facilities; including machines, working space, and programmers. They also require numerous development tools, libraries of examples, and instructional software.

Parallels can also be drawn to literary criticism courses, where students learn to write by reading the writing of others. This example supports the belief that people can best learn to program by reading programs. Thus courses involving reading good example programs might be very productive.

More difficult is the challenge of teaching those lessons that by their very nature span more than the semester duration of a traditional course. The true value of good programming practice and detailed documentation are learned only when a programmer has to modify a program months or years after he has written it, a situation that most students never face. It may be possible to design a sequence of courses requiring the student to examine or use his previous programs to motivate the need for these habits.

Two brief notes on mechanics: Firstly, our prerequisite structure is intended to be strictly observed. We recommend that a letter grade of "C" or better be required to satisfy a prerequisite. Secondly, we feel classes should be restricted to a manageable size. The report of the 1980 CMU Computer Science Undergraduate Program Committee [15] recommended the following limits on the size of course sections: 50-60 students in second year courses (2xx) and 20-30 students in upper division courses (3xx and 4xx). These limits were independent of issues related to degrees. We strongly endorse these limits.

6.4 Course Numbering Scheme

A rational system of course numbers provides a quick hint about the level and content of each course. Our scheme is derived from the Carnegie-Mellon system, and is similar to those in use at many universities. The level of a course is the year of the average student taking the course (1xx for freshman courses, 2xx for sophmores, and so on), except that 4xx courses are for suitable for both seniors and graduate students. There are no 5xx courses.

A three-digit course number *JKL* is interpreted as follows:

- The first digit *J* indicates the level of the course:

1xx	Basic, introductory, or general literacy
2xx	Elementary computer science
3xx	Intermediate topics focussing on individual ideas
4xx	Advanced or specialized topics integrating individual ideas

- The second digit *K* indicates the general subject matter of the course:

x0x	General
x1x	Systems
x2x	Programming Languages
x3x	Algorithms and Analysis
x4x	Computer Systems; Hardware
x5x	Theory and Mathematics
x6x	Artificial Intelligence and Psychology
x7x	Design, Graphics, and Computer-Aided Activities
x8x	Management, Economics, Policy
x9x	Applications

- ▶ The third digit is assigned arbitrarily to distinguish different courses within a category.

In the rest of this document we will often refer to courses offered by other departments. For this, we will generally use the Carnegie-Mellon course numbers, which do not necessarily adhere to this convention. Such course numbers are either prefixed with a department name (e.g. CIT 300), or by a two-digit department code (e.g. 39-300).

7. Abbreviated Course Descriptions

This chapter contains brief descriptions of the courses we have designed. These descriptions contain the same level of detail as a college catalog. More complete course outlines are given in Chapter 8, in a separate volume. Courses in other departments that are of interest to computer scientists are listed in Chapter 9. An overview of the course structure, including course names and prerequisites[1], is given in Figure 7-1.

7.1. Fundamental Structures of Computer Science I [211]

Prerequisites:	PROGRAMMING AND PROBLEM SOLVING [110]
	DISCRETE MATHEMATICS [150]

This course introduces students to the fundamental scientific concepts that underlie computer science and computer programming. Software concepts such as abstraction, representation, correctness, and performance analysis are developed and are related to underlying mathematical concepts. Students are asked to apply these concepts to programming problems throughout the course.

7.2. Fundamental Structures of Computer Science II [212]

Prerequisites:	FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]

The course is a continuation of FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]. It comprises five major parts: data abstraction, implementation of data types and corresponding algorithms, models of computation, topics in computer implementations, and a brief introduction to LISP. In addition to lectures on these areas, students are asked to complete a number of programming assignments that are an integral part of the course. They are often the first programs that are large enough to force the student to deal with abstraction (by necessity), and they give the student an opportunity to apply algorithms and abstraction techniques that are presented in class. Students are asked to program and think about programming during the entire course.

7.3. Real and Abstract Machines [240]

Prerequisites:	PROGRAMMING AND PROBLEM SOLVING [110]
	DISCRETE MATHEMATICS [150]

In this course the student is introduced simultaneously to the theoretical models and the hardware instances of machines that compute. The notion of layers of virtual machines and their realization in various combinations of hardware and software are major themes. Beginning with primitive computations, the mathematical concept of function is used to capture the capabilities of combinatorial digital logic circuits. From that base, finite automata are introduced as tools for understanding, analyzing, and designing finite state machines. After that, Turing Machines and, more appropriately, register machines are introduced and related to the architectures of real computers. Finally, microcode, machine/assembly language, and general-purpose programming languages are positioned in this hierarchy. The laboratory component of this course will require about three hours of lab work per week and will expose the student to simple instances of some of the machine types. Students will simulate several classes of machine and will design and construct simple combinatorial circuits and a simple finite state machine.

---

[1]The prerequisite structure is complete only for computer science courses.

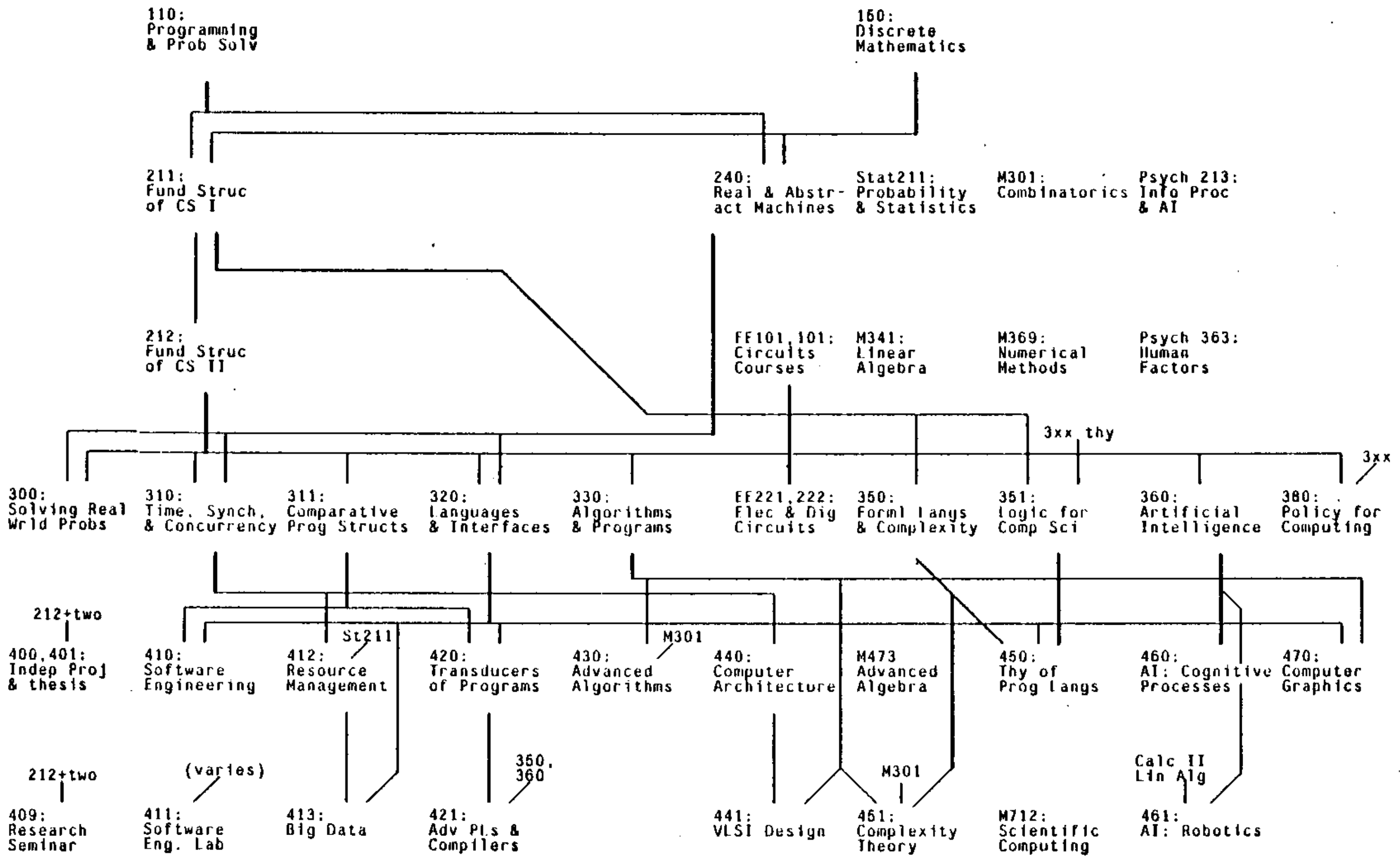

110: Programming & Prob Solv
150: Discrete Mathematics
211: Fund Struc of CS I
240: Real & Abstract Machines
Stat211: Probability & Statistics
M301: Combinatorics
Psych 213: Info Proc & AI
212: Fund Struc of CS II
EE101,101: Circuits Courses
M341: Linear Algebra
M369: Numerical Methods
Psych 363: Human Factors
3xx thy
3xx
300: Solving Real Wrld Probs
310: Time, Synch, & Concurrency
311: Comparative Prog Structs
320: Languages & Interfaces
330: Algorithms & Programs
EE221,222: Elec & Dig Circuits
350: Forml Langs & Complexity
351: Logic for Comp Sci
360: Artificial Intelligence
380: Policy for Computing
212+two
St211
M301
400,401: Indep Proj & thesis
410: Software Engineering
412: Resource Management
420: Transducers of Programs
430: Advanced Algorithms
440: Computer Architecture
M473 Advanced Algebra
450: Thy of Prog Langs
460: AI: Cognitive Processes
470: Computer Graphics
212+two
(varies)
350, 360
M301
Calc II Lin Alg
409: Research Seminar
411: Software Eng. Lab
413: Big Data
421: Adv PLs & Compilers
441: VLSI Design
451: Complexity Theory
M712: Scientific Computing
461: AI: Robotics

Figure 7-1: Course and Prerequisite Structure

7.4. Solving Real Problems [300]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]

This problem-oriented course provides students with an opportunity to solve real-world problems under the guidance of an instructor. Skills from a variety of areas both within and outside of computer science will need to be brought to bear on class examples and assignments posed as problems by the instructor. The emphasis is on the techniques used in obtaining the solution, rather than the solution per se. While proper software engineering techniques will, of course, be expected for all solutions involving software, it should be noted that the emphasis in the course is problem solving, not software engineering.

7.5. Time, Concurrency, and Synchronization [310]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]
PHYSICS I [PHYS 121] (MECHANICS)

This course conveys the fundamental notions of flow of time and control of temporal behavior in computer systems, both at the hardware and the software level. The fundamental issues of synchronization, deadlock, contention, metastable states in otherwise multistable devices and related problems are described. Solutions that have been evolved, like handshaking, synchronization with semaphores, and others are described and analyzed so that the fundamental similarities between the software and hardware techniques are exposed. This course has a significant laboratory component.

7.6. Comparative Program Structures [311]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]

This course covers a variety of common program organizations and program development techniques that should be in the vocabulary of a competent software engineer. The student learns advanced methods for programming-in-the-small including implementation of modules to given specifications and some common program organizations. The course also covers techniques for reusing previous work (e.g., transformation techniques and generic definitions) and elementary design and specification.

7.7. Languages, Interfaces, and their Processors [320]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
REAL AND ABSTRACT MACHINES [240]

This course examines the nature of programming languages and the programs that implement them. It covers the basic elements of programming language organization and implementation; it also touches on the design of interactive interfaces. The emphasis is on the elements of general-purpose programming languages that are common to many programming languages and on ideas that are also applicable to specialized systems. Implementation techniques covered include lexical analysis, simple parsing, semantic analysis including symbol tables and types, and interpretation for elementary arithmetic expressions. Programming projects include a simple interpreter and an interactive program.

7.8. Algorithms and Programs [330]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]

An introduction to abstract algorithms and to their design, analysis, and realization. The goal of the course is to develop skill with practical algorithm design and analysis techniques and to develop the ability to apply these techniques to the construction of real systems. The student is presented with a collection of useful algorithms and with design and analysis techniques. Like all models, abstract algorithms do not always match real problems exactly, and some skill is required to use them well.

7.9. Formal Languages, Automata, and Complexity [350]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]

An introduction to the fundamental material on formal languages, automata, computability, and complexity theory. Practical applications and implications of the material are emphasized.

7.10. Logic for Computer Science [351]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE I [211]
One 300-level mathematics or theoretical computer science course

The basic results and techniques of Logic are presented and related to fundamental issues in computer science.

7.11. Introduction to Artificial Intelligence [360]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
or INFORMATION PROCESSING PSYCHOLOGY AND AI [PSY 213]

This course teaches the fundamentals of artificial intelligence, including problem solving techniques, search, heuristic methods, and knowledge representation. Ideas are illustrated by sample programs and systems drawn from various branches of AI. Small programming projects will also be used to convey the central ideas of the course.

7.12. Independent Project [400]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with Bs or better
Instructor's permission, based on acceptance of project proposal

This is an independent project laboratory for the most advanced students. The student will design and construct a substantial software or hardware system under the supervision of the Project Lab faculty. Before construction of the project may proceed, a detailed design proposal must be submitted to and accepted by the faculty member running the course. A design review with the lab faculty and TAs will be held at mid-term time. A final review of the functioning system and its supporting documentation will be held at the end of the semester. The intent is to permit the best students to exercise their design skills in the construction of a real system, so good design practice and good documentation are mandatory. The production of a functioning but undocumented system will not be sufficient. The instructor may accept projects intended to last two semesters, in which case the review at the end of the first semester will be another major design review.

7.13. Undergraduate Thesis [401]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with B's or better
Instructor's permission, base on acceptance of proposal

This is an independent study and research course for the most advanced students. The student will write an undergraduate thesis or carry out a program of directed reading. Objectives for the course of study will be established by the student and a faculty advisor. With concurrence of a faculty advisor, an undergraduate thesis project may be planned for two semesters

7.14. Research Seminar [409]

Prerequisites: FUNDAMENTAL STRUCTURES OF COMPUTER SCIENCE II [212]
Two more courses (beyond 212) with Bs or better

Students attend the regular research seminars of the Computer Science Department and submit short written summaries. The Computer Science Department conducts a rich and varied set of public seminar series throughout the academic year. Undergraduates with sufficient maturity and experience in the field can benefit from attending, even if they do not completely understand the material presented. Attending these seminars is a good way to learn about very current ideas and to appreciate the scope and excitement of the field.

7.15. Software Engineering [410]

Prerequisites: COMPARATIVE PROGRAM STRUCTURES [311]
LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]

The student studies the nature of the program development task when many people, many modules, many versions, or many years are involved in designing, developing, and maintaining the system. The issues are both technical (e.g.. design, specification, version control) and administrative (e.g., cost estimation and elementary management). The course will consist primarily of working in small teams on the cooperative creation and modification of software systems.

7.16. Software Engineering Lab [411]

Prerequisites: vary with the individual arrangement
SOFTWARE ENGINEERING [410]

This course is intended to provide a vehicle for real-world software engineering experience. Students will work on existing software that is or will soon be in service. In a work environment, a student will experience first-hand the pragmatic arguments for proper design, documentation, and other software practices that often seem to have hollow rationalizations when applied to code that a student writes for an assignment and then never uses again. Projects and supervision will be individually arranged.

7.17. Resource Management [412]

Prerequisites: TIME, CONCURRENCY, AND SYNCHRONIZATION [310]
PROBABILITY AND APPLIED STATISTICS [STAT 211 / CS 250]

This course provides a synthesis of many of the ideas that students have learned in earlier courses. The vehicle for this synthesis is the exploration of at least one instance of a real operating system in great detail. Taking the view that an operating system is a resource manager, we will explore some resource issues and how they may be handled. The primary issues are resource classes, properties, and management policies. This course has a substantial programming laboratory component in which an existing operating system will serve as an experimental testbed.

7.18. Big Data [413]

Prerequisites:	LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
	RESOURCE MANAGEMENT [412]

The central theme of this course is the storage and retrieval of large amounts of data. Topics include user data models, underlying data storage techniques, data representations, algorithms for data retrieval, specialized data manipulation languages, and techniques for providing reliability and security. Systems that permit the storage and retrieval of large amounts of data are exemplified.

7.19. Transducers of Programs [420]

Prerequisites:	COMPARATIVE PROGRAM STRUCTURES [311]
	LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]

This course studies ways to gain leverage on the software development process by using programs to create or modify other programs, by reusing previously-created software, and by using automated tools to manage the software development process. Examples are drawn from the tools locally available. Students use these tools in projects that lead to useful software components. Special emphasis is placed on the use of integrated systems of compatible tools.

7.20. Advanced Programming Languages and Compilers [421]

Prerequisites:	FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
	INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
	TRANSDUCERS OF PROGRAMS [420]

This course is intended for students seriously interested in the construction of compilers for general-purpose programming languages. The student studies an optimizing compiler as an example of a well-organized system program, studies algorithms and data structures appropriate to the optimization process, examines code generators, optimizers, and their interactions. The student also studies comparative programming languages with emphasis on the interaction between language design and implementation considerations. Compiler-generator technology is used to build a compiler, thereby demonstrating the use of system-building tools.

7.21. Advanced Algorithms [430]

Prerequisites:	ALGORITHMS AND PROGRAMS [330]
	COMBINATORIAL ANALYSIS [MATH 301 / CS 251]

A second course in the design and analysis of algorithms, this course is intended to familiarize the student with the unifying principles and underlying concepts of algorithm design and analysis. It extends and refines the algorithmic concepts introduced in ALGORITHMS AND PROGRAMS [330]. Here a more abstract view is taken, with emphasis on the fundamental ideas of problem diagnosis, design of algorithms, and analysis. The course assumes familiarity with material on combinatorial analysis.

7.22. Computer Architecture [440]

Prerequisites:	REAL AND ABSTRACT MACHINES [240]
	or INTRODUCTION TO DIGITAL SYSTEMS [EE 133]

This course teaches the important concepts in computer system hardware design. System architecture is the focus of this course, so the technological details of the components from which such systems are constructed are avoided except where they are crucial to design goals like capacity and performance. The topics that are taught include design models including the Register Transfer Level model, Instruction Set Processor model, and PMS model. Analytic tools taught include notions of quantity of data based on Information Theory, Queueing Theory concepts, and Performance Evaluation techniques.

7.23. VLSI Systems [441]

Prerequisites:	COMPUTER ARCHITECTURE [440]
	ALGORITHMS AND PROGRAMS [330]

This course introduces the technology of VLSI and its use in system design. A broad survey of current technologies and simple design methodologies is given. The emphasis throughout is on practical issues, and the student will learn how to design projects and implement them on a chip. Some ideas of the potentials and limitations of VLSI design will be given, and special-purpose VLSI designs for a number of application areas will illustrate these points.

7.24. Theory of Programming Languages [450]

Prerequisites:	LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
	FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
	LOGIC FOR COMPUTER SCIENCE [351]

This course brings together fundamental material on the theory of programming languages. Techniques for assigning mathematical meanings to programs and for reasoning precisely about program functionality and behavior are described. Some indication is given of the influence of formal methods on programming methodology and programming language design.

7.25. Complexity Theory [451]

Prerequisites:	ALGORITHMS AND PROGRAMS [330]
	FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]
	COMBINATORIAL ANALYSIS [MATH 301 / CS 251]

This course extends in much more detail the material first introduced in FORMAL LANGUAGES, AUTOMATA, AND COMPLEXITY [350]. After a quick review of the basic ideas of complexity theory, the course introduces some of the advanced results and open questions of abstract complexity theory, and the techniques used in proving these results. Emphasis is made on relating these results and open questions to their theoretical and practical implications for Computer Science; the study of computability leads to theoretical limitations on what a computer can in principle (given enough time and space) do, while the study of complexity yields limitations on what is feasibly computable: if we are restricted to using only a limited amount of time or space, the class of problems solvable by computer is restricted.

7.26. Artificial Intelligence — Cognitive Processes [460]

Prerequisites:	INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
	or INFORMATION PROCESSING PSYCHOLOGY AND AI [PSY 213]

Covers more advanced aspects of the cognitive side of AI, including natural language processing, use of knowledge sources, and learning and discovery. The use of computer programs as psychological models will also be discussed. Students will implement a large AI system as a semester project.

7.27. Artificial Intelligence — Robotics [461]

Prerequisites:	INTRODUCTION TO ARTIFICIAL INTELLIGENCE [360]
	LINEAR ALGEBRA [MATH 341]
	CALCULUS II [MATH 122] (MULTIVARIATE CALCULUS)

Covers Artificial Intelligence systems that deal in some way with the physical world, either through visual, acoustic, or tactile means. Topics include vision, speech recognition, manipulation, and robotics.

7.28. Interactive Graphics Techniques [470]

Prerequisites:	LANGUAGES, INTERFACES, AND THEIR PROCESSORS [320]
	ALGORITHMS AND PROGRAMS [330]

A course in the creation and use of graphical information and user-interfaces.

References

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

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

3. Marc H. Brown, Norman Meyrowitz, and Andries van Dam. Personal Computers Networks and Graphical Animation: Rationale and Practice for Education. ACM SIGCSE 14th Annual Technical Symposium, Association for Computing Machinery, February, 1983.

4. CMU Graduate School of Industrial Administration. Announcements for 1954-1956. CMU Catalog. Pittsburgh PA, ,1954.

5. Carnegie-Mellon University. Carnegie-Mellon University Undergraduate Catalogue 1981-1983. CMU Catalog. Pittsburgh PA, ,1980.

6. W. Corwin and W. Wulf. SL230 - A Software Laboratory Intermediate Report. Carnegie-Mellon University Computer Science Department, May, 1972.

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

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

9. Philip Miller. GNOME: An Introductory Programming Environment. in preparation.

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

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

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

13. Mary Shaw. Working Papers on an Undergraduate Computer Science Curriculum. Tech. Rept. CMU-CS-83-101, Carnegie-Mellon University, Computer Science Department, February, 1983.

14. Tim Teitelbaum, Thomas Reps, Susan Horwitz. The Why and Wherefore of the Cornell Program Synthesizer. Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Cornell University, June, 1981, pp. 8-16.

15. The CSD Undergraduate Program Committee. Initial Report on an Undergraduate CS Program. CMU internal memorandum.