An Overview Course in Academic Computer Science: A New Approach for Teaching Nonmajors

Alan W. Biermann
Department of Computer Science
Duke University
Durham, North Carolina 27706

Introduction

An introductory course for nonmajors that overviews academic computer science has been under development at Duke University since the spring semester of 1986. This course introduces students to programming, teaches them the fundamentals of hardware and software and covers advanced topics such as program time complexity, parallel architectures, noncomputability, and artificial intelligence. *The course emphasizes deep academic experiences for students including nontrivial programming assignments, circuit design problems, assembly language programming, hand simulation of a compiler to see its functioning, the study of execution time of programs, programming for parallel machines, proofs of noncomputability, and the hand simulation of some artificial intelligence systems.*

The purpose of the course is to prepare people in numerous disciplines to live with computers in the long term by understanding the fundamentals of the field. The course attempts to present most of the great ideas of computer science that have developed over the past several decades. The claim is that, with such preparation, students will be maximally prepared to understand the capabilities and limitations of machines and to have perspective on their appropriate role in society.

This course is in contrast to the introductory courses that are usually given to such students at most colleges and universities. The typical choice is to teach either *programming* or what is usually called *computer literacy* [4,9,11]. Courses in programming are, of course, excellent educational experiences in clear thinking and problem solving and are almost always valuable for students. However, such courses do not address the broader concerns of a liberal arts student who needs to know the ultimate capabilities and limitations of machines. Computer literacy courses emphasize the development of useful skills such as the operation of software packages and their application to real world situations. They teach many useful facts about computers, but they do not necessarily engage the students in actually doing computer science. Again, the deeper understanding that comes from a detailed study of academic computer science is not achieved.

This course attempts to provide the alternative to computer literacy asked for by J. Paul Myers at last year's conference [8]. In his paper, he objected to the current trend in computer literacy courses, pointed to the increased maturity of current students, and campaigned for more genuinely academic studies. The general view of computer science given in the Duke course is in agreement with the Denning *et al.* report [5], "Computing as a Discipline". The following sections describe the contents of the course as it has evolved and give the reactions of students who have taken it.

The Course

The presentation of computer topics to general liberal arts students must be greatly modified from traditional methods because of their lack of mathematical sophistication and scientific vocabulary. It is not possible to simply condense standard coverages of programming, hardware and software topics, and advanced material that appear in texts and breeze through them at double speed. The material must be examined carefully to determine what portion of it is worthwhile to teach, and then the total coverage must be designed to get to the main points directly without burdening the students with unnecessary concepts or vocabulary. It is not possible to teach all of computer science in one course. Careful decisions must be made to omit what is not of greatest importance so that time is available to cover the major topics.

The following paragraphs describe all of the topics in the course, giving in each case the purpose of the study and the type of coverage used to achieve that purpose. In many cases, the nature of the material covered differs dramatically from traditional presentations because of the need to make the ideas understandable by nonmathematical and scientifically inexperienced people. The interested reader may wish to examine *Great Ideas in Computer Science: A Gentle Introduction* [1] which is a textbook based on the classroom notes and which gives the essential coverage of the course. (There are other surveys of computer science [3,6,7,10,12] but most of them are too difficult for this population of students.)

Programming. The purpose of the programming portion of the course is to teach students the lessons in clear thinking that go with any programming experience and to give them the tremendous intuition for machines that programming makes possible. Students are taught the traditional lessons related to finding proper representations for their problems and "divide and conquer" methodologies for code development. This series of studies encompasses about one third of the course, but weekly laboratories continue to the end of the semester so that considerable programming is eventually accomplished. The major student exercises include programming a simple interactive text editor, several numerical computation problems, and a relational database program.

However, some shortcuts must be found to make it possible to teach all this programming and still have time for the rest of the topics. The technique is to limit the coverage of the programming language, Pascal in this case, to a relatively few constructions. In fact, students are taught a kind of micro-Pascal that includes only integer, real, and string data types (with arrays), assignment statements, *if-then* and *if-then-else* statements, *while* loops, *read* and *write* statements, and a single subroutine feature. It was found that syntactic variety is one of the main sources of confusion for such students and that tremendous simplification results from the elimination of the variety of features available in the full

language. This subset of the language is taught in a relatively short time at the beginning of the course and is used repetitively throughout the rest of the semester. The programs are often somewhat pedantic in their appearance because of the syntactic limitations. But the gains in simplicity and learnability for these students are well worth the compromise.

The programming section of the course is concluded with a study of software engineering where students learn that experiences gained in university laboratories do not always scale up well to large projects. The major lessons from such standard sources as *The Mythical Man-Month* [2] are taught. These include descriptions of software engineering calamities involving slipped deadlines, overrun budgets, and specifications not met. They also include descriptions of modern methodologies for systematic organization of software projects and the typical software life cycle.

Computer Hardware-Software. The purpose of the hardware-software section of the course is to teach students to think of computing as a mechanistic process that they can understand. They have no need to stand in awe of "thinking machines" and they can read articles about new architectures, breakthroughs in chip technologies, or new computer languages with understanding and perspective.

This portion of the course begins with the design of switching circuits and their assembly into elementary components of machines. Then the course examines the transistor and VLSI technologies that make it possible to construct large circuits on tiny chips. Next, students are shown the architecture of a typical machine, how it functions, and how to program it. Finally, they are shown a small compiler for Pascal assignment statements and how it translates Pascal into a machine executable language. Some of the details of these studies will now be explained.

The trick to teaching circuit design is to realize that little or no Boolean algebra is needed to convey the most important lessons. Students can be taught to write down a functional table defining the target behavior and then to construct directly a nonminimal switching circuit to realize that behavior. They can have all of the satisfaction that goes with creating circuits to do any task, and they can avoid having to learn to manipulate and minimize algebraic expressions. While Boolean algebra is one of the great discoveries of computer science, a course that tries to cover the whole field must find ways to cut all but the most central ideas. Boolean algebra was removed from the course after its first presentation because of student difficulties. Students, even with these few tools, can design many computation circuits such an adder or an elevator controller. The study of switching circuits anticipates the later study of machine architecture by showing how one designs circuits for elementary machine operations and a circuit to decode machine instructions to activate such operations.

Next the course undertakes a short introduction to solid state physics and the principles of transistor design. Students are taught the concept of positive and negative carriers in semiconductors and the mechanisms for controlling the carrier supply to build a switch. Then VLSI methods are explained including CMOS technologies and the ways that microcircuits are etched into silicon chips. Part of this study includes a discussion of the speed of computations and the importance of miniaturization for achieving fast computation.

The level above circuit design is machine architecture where students see how the switching circuits from the previous study are assembled to build a machine. Here the concepts of the control counter, the instruction register, the computation register, memory, and the "fetch-execute" cycle are introduced. Students solidify these lessons by learning to both read and write simple programs in assembly language.

The highest level of the hardware-software sequence is the study of the compilation process. Here students are shown a set of translation rules for compiling Pascal assignment statements (restricted to nestings of addition and multiplication operators) and are taught how to use them to compile any such statement into the assembly language of the previous section. The exercise is meant to make translation seem like a straightforward and understandable process. It shows students how machines can be programmed to process any formal language and provides an avenue to understanding a variety of languages. In fact, students are given examples of programs from eight different languages with the point being that many other languages such as FORTRAN, PL/I, etc. are similar in structure to the Pascal that they have already learned and that some other languages such as LISP and Prolog are rather different.

The final exercise of the hardware-software study is to trace the execution of a sample Pascal assignment statement through all of the stages from compilation, to execution on the architecture, through the operation of the computational circuits to the migration of electrical carriers in the semiconductor fabrication.

Advanced Topics. The two-fold purpose of this portion of the course is to introduce a number of very exciting topics from current research and to make students aware of the limitations of computer science. The topics include computer program time complexity studies, parallel architectures, noncomputability, and artificial intelligence.

The program timing studies begin with the development of some experimentally derived formulas for certain example programs. A study of these formulas gives a measure of how large the problems can be and still be computed in "reasonable" lengths of time. Continued examination of these formulas leads to the discovery of the "tractable" and "intractable" classes. Then a number of examples of problems of both types are presented with the hope that students will gain an intuition for them. The test of success in this part of the course is whether students can properly classify a variety of such problems into appropriate categories.

The study of parallel architectures begins by showing how simple problems from previous parts of the course can be reprogrammed for execution on a parallel machine. Then execution time studies are used to show what can be gained by the use of parallelism. A variety of architectures are introduced with indications of their strengths and weaknesses. The last portion of the study examines a sample connectionist architecture and shows how learning rather than programming can be used to achieve the desired behavior.

The problem with teaching noncomputability is enabling students to understand what the word means. The teaching methodology here is to introduce the idea of having programs read other programs and compute something. Thus a number of programs are given that do such simple tasks as measure the length of an input program or check whether it contains an *if* statement. After a series of such examples, the question arises as to whether a program could be written to check the halting property for the input program. This becomes the prototypical example of a noncomputable task, and a number of other examples are presented. The conclusion of this study is that programs that read other programs can usually be written if they are to check syntactic features of the input. But if they are to check any feature related to the execution of the input program, one is probably attempting a noncomputable computation. Again the test of whether students understand

this part of the course is to check whether they can properly classify a variety of given problems as either computable or noncomputable.

The artificial intelligence study is divided into two parts, a presentation of representation methods and a study of search. The representation methods include formal languages, semantic networks, computer programs, and others. The search part of the study includes detailed examples of systems for natural language processing, game playing and expert reasoning. Students are taught to understand these systems by doing hand simulations of them on sample inputs. The main purpose of the study is to give students an understanding of the state of the art so that they can reasonably judge what can and what cannot be done in artificial intelligence.

(Note: The description given here is generic. Individual presentations of the course may emphasize some topics heavily while giving others only short mention. The time allotments for the topics depend on the interests of the instructor.)

Student Response

This course has been taught each semester since the beginning of 1986 and has attracted about 100 to 200 students per year. The students come from a variety of majors most of which are nonmathematical as shown in the following table. (Data was gathered from a single class of 65 students in the Fall of 1988.)

Major	Percent
Art History	3
Comparative Area Studies	3
Comparative Literature	1
Economics	11
English	8
History	10
Mathematics	3
Philosophy	3
Political Science	15
Psychology	6
Public Policy Studies	3
Religion	1
Zoology	3
(undecided)	30

These students came from all levels of undergraduate work at the following percentages: freshman 9, sophmore 48, junior 26, and senior 17. When asked why they were taking the course, most students indicated that they simply wanted to learn something about computers. A few mentioned the fact that they were taking the course to fulfill a requirement in quantitative reasoning. Most had no previous experience with computers and no more than one college level course in mathematics.

Before this course was taught, many educators believed that such a student body could not and would not succeed in academic computer studies at the depth described here. The image of such students designing switching circuits, hand simulating a compiler, or studying noncomputability seemed farfetched. The success described here may provide incentive to other institutions to try similar courses.

During the 1988 Fall semester class, students were repeatedly surveyed to determine their response to the material and their level of accomplishment. Specifically, after each of the significant thirds of the course, programming, hardware-software, and advanced topics, students were queried concerning their response to that particuliar material. The following table summarizes the results of these questionaires. Each student was classified on each part of the course as either "interested and doing well", "neutral", or "unhappy with the material". Considering the facts that this was a course for nonmajors which many students were taking because they were required to and that the contents of the course were unusually technical for this type of student, the results should be considered to be very positive.

	Programming	Hardware Software	Advanced Topics
Interested and doing well.	83%	81%	63%
Neutral.	5%	12%	13%
Unhappy with the material.	12%	7%	24%

In one particular question, students were asked to rate their level of interest in each of the fifteen different subjects in the course. Ranking the topics on a scale from 0 to 10, *every topic in the course* was ranked by the students with an average score above 5. The most popular topics were those related to programming and artificial intelligence.

In summary, experience with this course provides ample evidence that liberal arts students with little mathematical sophistication can be taught successfully, in a single course, an overview of much of academic computer science including some very technical material.

The Product: Students Who Understand the Fundamental Principles of Computer Science

Students who complete this course should have a reasonable grasp of what computers are, how they work, what they can do, and what they cannot do. In their personal, academic, and professional lives, they should be able to face new situations where computing may be proposed as a solution and have good judgment as to what realistically can and cannot be accomplished. When they read the news regarding new developments in computing, they should have the proper background to understand them whether they concern the announcement of new architectures, advances in artificial intelligence, or other significant discoveries. They will have had substantial exposure to most of the important paradigms in computing and will be as well prepared as we can make them, in a single course, for dealing with future eventualities.

References

[1] A. W. Biermann, *Great Ideas in Computer Science, A Gentle Introduction*, The MIT Press, Cambridge, Massachusetts, 1990.

[2] F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1975.

[3] J. G. Brookshear, *Computer Science, An Overview*, Second Edition, Benjamin/Cummings Publishing Company, Menlo Park, California, 1988.

[4] J. S. Burstein, *Computers and Information Systems*, Holt, Rinehart & Winston, New York, 1986.

[5] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a Discipline", *Communications of the ACM, Vol. 32, No. 1, 1989.*

[6] L. Goldschlager and A. Lister, *Computer Science, A Modern Introduction*, Prentice Hall, New York, 1988.

[7] D. Harel, *Algorithmics, The Spirit of Computing*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1987.

[8] J. P. Myers, Jr., "The New Generation of Computer Literacy", *The Proceedings of the Twentieth SIGCSE Technical Symposuim on Computer Science Education*, Louisville, Kentucky, Feb. 23-25, 1989.

[9] T. Owens and P. Edwards, *Information Processing Today with Applications and BASIC*, Burgess Communications, Edina, Minnesota, 1986.

[10] I. Pohl and A. Shaw, *The Nature of Computation, An Introduction to Computer Science*, Computer Science Press, Rockville, Maryland, 1981.

[11] D. H. Sanders, *Computers Today*, McGraw-Hill, New York, 1985.

[12] C. Schaffer, *Principles of Computer Science*, Prentice Hall, New York, 1988.