Introducing Computer Science Fundamentals Before Programming

Russell L. Shackelford and Richard J. LeBlanc, Jr.
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

Abstract *The traditional approach to introducing students to Computer Science has been through a course built around the development of programming skills. While many textbooks intended for such courses include the words "Problem Solving" in their titles, the primary focus of most such courses is the skill of programming in a particular programming language.*

Computer scientists will quickly proclaim that the discipline is more than just programming, but our typical introductory courses fail to provide students with any conceptual foundation of computer science as a domain of study. In addition, the "programming-first" approach ignores the practical reality that increasing powerful application-oriented software packages have become the tools of choice for solving many computation problems.

In this paper, we describe a two course sequence that has been taught to majors in computer science and a variety of other disciplines for the last four years. The first course is called "Introduction to Computing"; the second course is "Introduction to Programming." The "Computing" course is a ten-week course that includes material on the following: the "computing perspective" (the algorithmic model as a way of thinking); foundational concepts underlying program design and implementation, including: algorithmic methods, static and dynamic data structures, and design using abstraction; fundamental notions of algorithm analysis and computability; and use of application tools such as e-mail, Web browsers, word processors, spreadsheets, databases, and equation solvers.

Building on this foundational pre-programming material, students in the "Programming" course are able to learn rapidly the skills in program design, implementation, and debugging necessary to solve computational problems in a high-level programming language. We emphasize effective use of abstraction and the acquisition of software development skills that are language independent, though obviously we use a particular programming language.

Our experience with these courses has convinced us that it is possible to introduce the conceptual foundations of Computer Science to beginning students in a way that both engages them and gives them a basis for learning advanced ways to solve problems using computing, either via programming or through use of modern, highly-programmable commercial applications.

Introduction

In 1992, Georgia Tech's College of Computing restructured its lower division curriculum to correct historical flaws and better respond to modern demands. The restructuring was motivated by the recognition that computing has ceased to be an arcane technical discipline of interest primarily to CS majors and instead has become a core element of a proper university-level education for a broad population of students. The restructuring acknowledged the need to provide an appropriate foundation in the conceptual and intellectual foundations of computing as well as relevant skills, to improve access for non-CS majors to the important ideas that computing offers, and to better exploit computing's numerous opportunities to provide students with rich and challenging experiences.

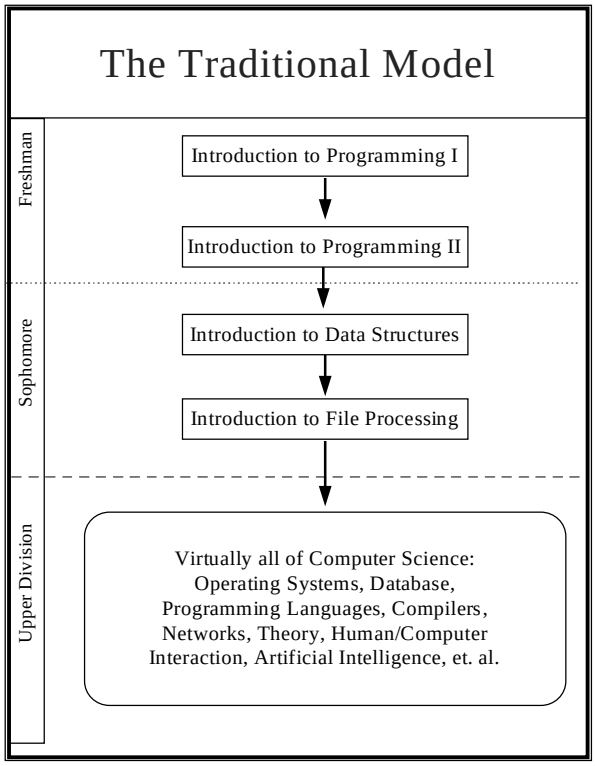
In the two years subsequent to the restructuring, enrollments in the first course skyrocketed from ~100 per year to ~1400, and computing became a de facto component of Georgia Tech's engineering curricula. This fact was recently formalized: computing is joining the more-traditional disciplines as part of the university's core curriculum. This change will occur concurrently with the University System of Georgia's shift from a quarter-based to a semester-based calendar, and we believe this calendar change provides us with yet another opportunity to improve the teaching-and-learning value of our lower division structure.

We summarize, below, the rationale and the specific goals of our restructured curriculum, and the lessons learned from our four years of experience with it.

Goals

Our curricular restructuring was informed by the observation that the traditional CS curriculum is an aberration that lacks a basic feature of time-honored curricular structure. While most curricula present fundamental ideas and principles early, CS curricula have traditionally presented two years of work focused on applied programming skills, effectively withholding key foundational material until upper division courses. We believe that this curricular tradition is residue of the fact that CS courses originated as "programming skills add-ons" to Mathematics and/or Electrical Engineering curricula. As

CS matured into a substantive discipline unto itself, it simply "grew upward" in the curriculum, adding more advanced courses on top of the traditional skills-oriented lower division. As a result, students who are not CS-majors are effectively denied access to foundational material. In recent years, numerous efforts have been made to repair the specific courses in the traditional structure without, in our view, adequate consideration of the accidental, historical design flaws of that course structure itself.



As computing plays an ever-growing role for students of other disciplines, the traditional structure has become increasingly problematic. Engineering students can be expected to confront multiple programming languages throughout their careers: the large body of existing FORTRAN programs, a more recent body of C programs, and a future that will include Java and other languages. No longer can we teach a particular language and believe that it will adequately prepare students for their future.

Each generation of programming languages is more complex than the last, with increased demands to understand and apply foundational algorithmic principles such as abstraction and design. Furthermore, the clear trend is for programmers to adapt and reuse libraries of modular components, with far less emphasis on coding from scratch. In total, these developments require that we provide students with a core set of fundamental principles and skills that will allow them to adapt to changing programming environments and practices.

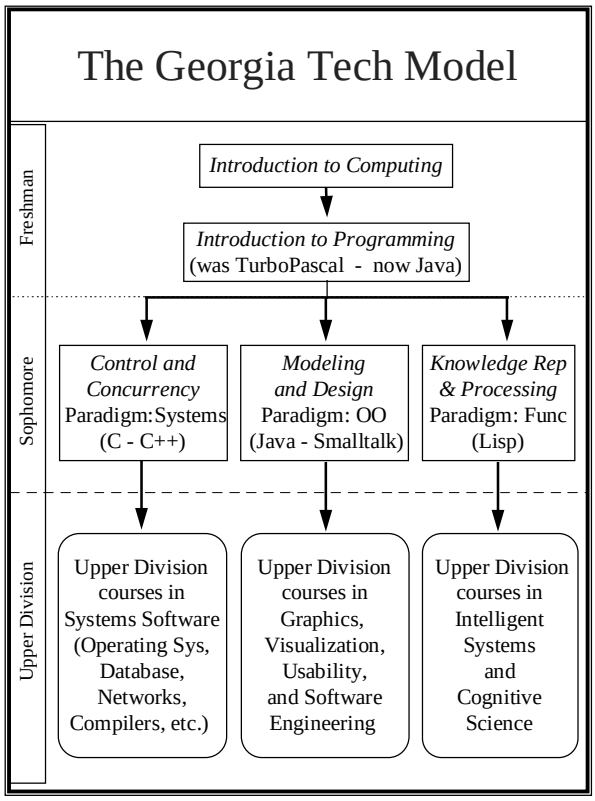
Introducing students to computing by way of a traditional programming course ignores the practical reality that standard application programs (such as spreadsheets, Matlab, Maple, etc.) have become the tools of choice for solving many computing-related problems. Engineers can bring to bear a tremendous amount of computing power on a wide range of problems without ever "writing a computer program" in the traditional sense of the phrase. Thus, introducing students to computing via a course that emphasizes writing programs effectively bypasses the most direct means of computer-supported problem solving.

As the power of off-the-shelf application programs has grown, so too has their complexity. The boundary between "programming" and "use of software packages" has become blurred, such that the very same principles that underlie effective program design (e.g., modularity, abstraction, reuse, etc.) also underlie the effective use of increasingly complex software tools. Thus, reliance on such tools increasingly requires an adequate foundation in algorithmic principles.

In light of these factors, we set out to create a curricular structure that would support not only CS-majors but also engineering students and others who require a foundation in computing. We also sought to provide non-CS majors with increased access to advanced computing courses that may be relevant to their professional goals. The result is a lower division structure that features:

1. A freshman level "Introduction to Computing" course emphasizing the design, construction, and analysis of algorithms, in concert with a software applications lab.
2. A freshman level "Introduction to Programming" course (which assumes the first course as a prerequisite) emphasizing the effective and efficient implementation of algorithms in a modern programming language.
3. A set of three sophomore courses, each one serving as a gateway to clusters of upper division courses. This structure allows non-CS students, who may be interested in such things as systems software, graphics, simulation, etc., to have elective access to such courses without facing a prohibitively long chain of prerequisites.

The resulting structure is shown below. It gives the faculty of various academic disciplines considerable latitude regarding how much of a computing background they wish their students to acquire. Some faculty may believe that students in their discipline require only application-use skills and an appropriate conceptual background. Other faculty may want their students to take a second course to obtain a significant foundation in computer programming. Still others may want their students to take a sophomore gateway course and additional upper division computing courses beyond it. Indeed, at Georgia Tech we presently have one or more engineering schools choosing each one of these options.



We report here on our four years of experience with the two freshman courses.

The "Introduction to Computing" Course

The first course has two related goals:

(1) provide a foundation in computing and algorithmic principles, and

(2) establish concrete skills in effective use of software packages.

We do this by way of a lecture-and-homework agenda that emphasizes the design, construction, and analysis of algorithms, coupled to a lab-and-project agenda focused on the application of those principles in the use of software packages.

The former is based on a pseudo-language for designing and constructing algorithms. We purposely use a pseudo-language to free students from the myriad of programming language details that annoy and distract attention from the core issue of algorithmic problem solving. Students do a great deal of algorithm construction focused on essential elements of abstraction, design, data structures and control, and reuse of code. Because students are liberated from non-essential syntax and compilation requirements, we can present them with a complete range of constructs and algorithmic methods.

In addition, we pull from the engineering tradition of effective "quick-and-dirty" estimation, and show students how to predict the cost and complexity of competing algorithmic approaches. As a result, students experience better integration of abstraction, design, implementation, and evaluation. We believe this to be far preferable to the traditional approach which emphasizes "button pushing" at the expense of any "big picture" perspective on what they are doing and why.

The lab component gives students applied experience with a range of computing resources and applications, including both the UNIX and Windows environments, communications facilities (such as e-mail, ftp, tools for report and document preparation, Web-based data search, and home page creation), and concrete problem solving (using spreadsheets, databases, and advanced equation solvers such as Matlab). Such experience is tied to core algorithmic principles from lecture, allowing students to apply those principles in concrete tasks of problem solving and communication.

Results

With any curricular enterprise, it is extremely difficult to perform adequately rigorous studies of teaching-and-learning effectiveness. The fact that we lack reasonable data about student competence under the old structure has lead us to develop prototype "teaching information systems" to help us better track student competence in the future. In the meantime, we are left with informal impressions and subjective questionnaire data to inform us about the results of our approach.

Faculty and instructional staff observe that students prove themselves quite capable of mastering and applying key algorithmic principles rapidly, given the opportunity. In fact, some of the learning problems that we had come to accept as normal have virtually disappeared. For example, it had been quite normal to find that students at every level of computing study preferred to manage repetition by iteration rather than recursion, and many preferred to use static, linear data structures rather than dynamic, non-linear structures. This is inconsistent with the desired learning outcome to produce students who do not have such arbitrary, fixed biases. We want to produce students who make informed design choices based on the properties of the problem at hand and on the cost and complexity implications of such choices, given the parameters of the problem.

By the end of the first course, objective measures show that students can and do routinely make reasonable, well-informed algorithmic design decisions. In retrospect, we now view the former "normal student bias" as having been an artifact of the traditional curricular structure in which one set of approaches is typically ingrained in the first

course, leaving important alternatives for the second course. It would appear that this has significant long-term effects on programmer preferences. We suspect that novice students were effectively imprinted on certain approaches such that many remain forever biased. While we cannot presently prove this, we do see quite clearly that students leave the first course more willing and more capable of making informed and unbiased design decisions than was the case under the old structure.

Students report far less confusion under the new structure. Under the old structure, it was a normal occurrence to have bright students informally report that, despite having earned good grades, they really did not know "what they were doing or why." Extensive use of weekly on-line surveys since the inception of the new curriculum shows that students sometimes report that they do not know "how" to do something. Remarkable by their absence are student complaints about not understanding "what" or "why." We simply do not hear those kinds of complaints nearly as often as we once did; we interpret this difference as confirmation that our new curriculum is on the right track..

Many students report that their experience in the first course helps them in practical ways that we did not foresee, i.e., helping them obtain preferred co-op jobs, because they now can present themselves to employers as having both applied software experience and a foundation in algorithmic principles. It is clear that the first course better prepares students for employment than did the first course of the old structure.

At present, the majority of engineering students have this course as a degree requirement. All undergraduates at Georgia Tech will soon have this course as part of their core degree requirements.

The "Introduction to Programming" Course

The second course has two related goals: (1) to introduce students to problem solving via a modern, full-featured programming language, and (2) to establish professional software development skills and practices that allow students to construct well-designed and tested programs quickly.

The first goal is enabled by first course. Students report that they enter the second course "knowing what they are trying to do." We observe that they are able to learn rapidly the skills in program design, implementation, and debugging necessary to solve computational problems in a complex programming language. We emphasize effective use of abstraction and the acquisition of software development skills that are language independent, though obviously we use a particular programming language.

(After several years of resisting market pressure to teach C or C++ at the introductory level, we are now transitioning from Turbo Pascal to Java. We believe Java presents reasonable pedagogical opportunities, and our initial experience with it has not dissuaded us from this belief. We hope to have more to say on this issue soon)

The second goal is a new one. We are no longer satisfied if students can implement reasonably designed programs and get them to work. Far too often, students do so by such a painful and laborious trail-and-error process that they learn to hate programming. We want students to achieve effective computational solutions in reasonable time and effort. Rather than expecting them to produce working programs, we now want them to evidence effective programmer practices. Thus, we emphasize the two core activities of design (for error prevention and avoidance) and debugging (strategies for diagnosis and repair).

Results

Again, our ability to assert significant results is inhibited by the difficulty of rigorously measuring curricular effectiveness. Nonetheless, it is abundantly clear to us that the fact that students now enter the programming course already familiar with the basics of algorithms and data structures has provided an unforeseen benefit. It has unmasked a problem that we now believe had been there all along.

Under the traditional curriculum, the first two courses were focused on introducing both a particular programming language and foundational constructs. In practice, students were left to develop their personal programming methods on their own in a very *ad hoc* way, generally while struggling with the compiler.

Under the new structure, we need not consume lecture in the programming course covering the ideas and mechanisms of parameters, modularity, static and dynamic data, etc., because students are well practiced in such things from the prerequisite course. Instead, we have the opportunity to focus explicitly on the myriad of important ideas and skills concerning the activity of programming that were either completely ignored or barely touched on in our old, traditional introductory programming course.

This has led us to recognize that our old curriculum mis-titled the introductory course. We called it "introduction to programming," but it would have been more accurately described as an "introduction to program components in the syntax of a specific language." By moving coverage of algorithmic concepts and skills to a non-programming course, we not only enable students to progress with programming at a much faster pace, we also allow ourselves to focus on key questions of programming skill that have traditionally been ignored.

We now have the opportunity to focus on effective software development skills, including error avoidance, prevention and debugging, which allow students to design and implement solutions in less time. In short, we can for the first time provide a course that is a true "introduction to programming." At present, we have made what we think are important strides in this direction, but are not yet at our goal state. Our current work is focused on developing effective means for best exploiting this new teaching-and-learning opportunity, and we hope to have a good deal more to say about this in the near future.

Course Management Issues

The fact that the new structure has been accepted across campus presents both good news and bad news. The good news is that we are having a much broader impact on many more students. We have succeeded in having computing incorporated into the university's core curriculum, and we believe that we are doing a far better job of providing a broad range of students with a good foundation.

The bad news is that we now have thousands of students to teach each year. This presents many logistical problems, including a heavy demand for student learning support. Rather than use a few graduate students as graders for the large numbers of students, we chose to use many undergraduate students as Teaching Assistants.

Because the freshman courses are intended for a broad audience, we do not wish our TA population to be limited to CS majors. Indeed, we recruit across the board and routinely employ numerous undergraduate engineering majors, including several who have risen to become "senior TAs" who help guide and manage the courses.

Results

Because undergraduates are less expensive to hire than are graduate students, we can achieve student/TA ratios that support individualized attention, not just massive grading. While this strategy was initially viewed as risky, we have found that undergraduates as a group are superior to graduate students in motivation, energy, initiative, and caring.

The relative affordability of undergraduates means that, despite the large numbers of students, we can and do mandate individual (i.e., one-to-one) student/TA meetings for every student each week (only those students who maintain an "A" average are exempt). In addition, underghraduate TAs teach weeklly small-group recitation sessions which provide the kind of human-scale classroom experience unattainable in huge lecture halls.

An unforeseen benefit is that our undergraduate TAs obtain valuable experience in teaching, public speaking, and working with people. These are precisely the skills that employers plead for. Many students report that their TA job is the only aspect of their entire undergraduate career that allows them to "do something that is real."

Summary

We undertook our curriculum revision with some trepidation, as we were trying to design curricular solutions without an adequate model or supporting textbooks. After four years of experience, we have created one textbook [1], are working on a second one, have learned a good deal about what works and what doesn't, and are pleased to see that our original goals have been largely achieved.

We now know that students from a variety of backgrounds can master key computing principles early, that students can quickly achieve competence with both software packages and computer programming, and that we can achieve much more of a tight coupling of computing theory and practice at the earliest levels.

In addition, we have been pleased to see that our efforts to better serve engineering students have resulted in what we believe is a better education for our own CS-majors. The early, integrated coverage of both principles and skills has resulted in: fewer misconceptions among our own students; less need for subsequent CS courses to correct problems which were rooted in the introductory courses; and a greater focus on explicitly teaching things (such as testing, verification, error avoidance and prevention, and debugging strategies) that have traditionally been given inadequate attention. In sum, by opening up our curriculum to better serve others, we have improved the undergraduate experience in computing for everybody.

We invite those interested in considering such an approach for their own curriculum to peruse the web pages for the two freshman courses. The pages for the Introduction to Computing course may be found at http://www.cc.gatech.edu/classes/cs1501. For the second course, Introduction to Programming, the home page may be found at http://www.cc.gatech.edu/classes/cs1502. We also welcome inquiries by e-mail. Address e-mail to: curriculum.support@cc.gatech.edu.

References

1) Shackelford, Russell L. *Introduction to Computing and Algorithms*. Addison-Wesley, Reading, Massachusetts, 1997.