Introduction to computing: a course in computer science fundamentals

Russell L. Shackelford
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, 30332, USA, e-mail: russ@cc.gatech.edu

Richard J. LeBlanc, Jr.
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, 30332, USA, e-mail: rich@cc.gatech.edu

Abstract
The traditional approach to introducing students to computer science has been through a course built around the development of programming skills, ignoring the practical reality of increasing powerful application-oriented software packages. In this paper we describe a two course sequence which has been taught to majors in computer science and a variety of other disciplines for the last four years. We emphasize effective use of abstraction and the acquisition of software development skills which are language independent. 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 which both engages them and gives them a basis for learning advanced ways to solve problems using computing.

Keywords
Informatics, university education, curriculum (start)

1 INTRODUCTION

In 1992 Georgia Tech's College of Computing restructured its lower division curriculum to correct historical flaws and to respond better to modern demands. The restructuring was motivated by recognition that computing is no longer an arcane technical discipline of interest primarily to computer science (CS) majors. It has become a core element of university-level education for a broad population. We sought to provide an introduction to the conceptual andintellectual foundations of computing, and to relevant computer use skills. We also sought to improve access for non-CS majors to the revolutionary ideas and capabilities which computing offers.

In the next two years introductory course enrollments skyrocketed from ~400 to ~1400 annually and computing became a de facto part of many 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. In this paper we summarize the rationale and specific goals of our restructured introductory curriculum and report the lessons learned in our four years of experience with it.

2 THE DEMANDS ON MODERN COMPUTING CURRICULA

Computing curricula have received frequent critical attention. About every ten years since the late 1960s, we have seen a new version of recommendations aimed at repairing curricular weaknesses. These weaknesses emerge in part because, over the thirty years of its curricular existence, computing has seen a multitude of changes which dramatically have increased the demands placed on introductory courses. Among these are the following.

Computer science as a discipline

Computer science is a discipline in itself with its own body of knowledge, its own intellectual and conceptual foundations, and its own effective practices. Thus, we would expect that introductory CS courses would introduce students to foundations of both the subject matter and its application.

Computing now affects everybody

Years ago computing affected very few people and 'computing' equalled 'programming'. Since then computing has impacted virtually every discipline and now means quite a bit more than 'just programming.' Thus we would expect introductory computing courses to introduce a wider audience to the ways that people 'do computing' which means both 'effective application use' and 'effective programming'.

Original requirements (mid-1960's)	*Current requirements (through Java)*
1. Assignment	1. Assignment
2. Numerical operations	2. Numerical operations
3. Data types: int, real, char, boolean	3. Data types: int, real, char, boolean
4. Data structures: arrays	4. Data structures: arrays
5. Control stmts: if's and loop's	5. Control stmts: if's and loop's
6. File operations	6. File operations
7. Formatting of I/O	7. Formatting of I/O
8. Procedures and parameters	8. Procedures and parameters
	9. Structured design
	10. Pointers
	11. Linked lists
	12. Trees
	13. Recursion
	14. Interactive debuggers
	15. Interactive programs
	16. Human-computer interface
	17. Graphics
	18. Larger programs
	19. SW Engineering fundamentals
	20. Complexity ('Big Oh')
	21. Fluency in multiple languages
	22. Application software skills
	23. OOP
	24. Applets

Figure 1 Requirements then and now.

Programming has changed in fundamental ways
No longer can we believe that students will need to know only one or two relatively simple languages. When the traditional curricular structure evolved, teaching someone to program usually meant covering the list of topics shown in the left column of Figure 1. Since that time evolution (in both programming languages and the problems which they are deployed to solve) has expanded the list as shown in the right column of Figure 1. Not only is this list longer by a factor of three, most items are more complex than those before them. It is not unreasonable to suggest that the complexity of 'teaching programming' has increased by an order of magnitude. Thus we would expect computing curricula to emphasize basic and universal algorithmic and programming constructs and skills, rather than to focus on 'programming in a given language'.

Everything about computing is more complex
Such things as 'effective abstraction', 'good design', 'effective debugging and verification' and 'good software engineering practices' are clearly more important than these used to be. Therefore we would expect curricula to explicitly focus on establishing a good foundation in precisely these fundamentals.

3 ARCHAIC ASSUMPTIONS

Over the years we find a history of curriculum revisions (most recently to complex languages such as C++ and Java) which imply 'downloading' more and more material to the introductory courses. In our view simple downloading of more material into an introductory sequence of programming courses is no longer an adequate response. Such an approach produces a number of negative consequences. We believe that each is a reflection of a specific assumption which is no longer valid, but which is implicit in the traditional curricular structure itself.

'Computing means programming'
It does not. Introducing students to computing via traditional programming courses ignores the practical reality that standard application programs have become the 'tools of choice' for solving many computing-related problems. One 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. Introducing students to computing via a course which emphasizes 'writing programs' effectively bypasses the most direct means of computer-supported problem solving.

'Software applications are not substantive enough to warrant our attention'
They are. As the power of off-the-shelf application programs has grown, so has their complexity. The boundary between 'programming' and 'use of software

packages' has become blurred, such that the very same principles which underlie effective program design (e.g. modularity, abstraction, reusability) also underlie the effective use of increasingly complex software tools. Thus reliance on such tools requires an adequate foundation in algorithmic principles. Treating powerful applications as trivial tools deprives students of guidance in their effective use and misses opportunities for teaching important computing principles.

'The best way to teach principles is in the context of a programming course'

We do not think it is. It may have been so years ago, but those days are long gone. The nature of 'programming courses' dictates that student grades are based largely on programs which students submit throughout the term. Regardless of the degree to which we emphasize things such as abstraction and design, as the deadline for each program approaches students quite naturally tend to forget all that and instead focus on getting their programs to 'somehow work.' As a result we find students manipulating an increasingly complex set of language constructs and features without adequate attention to the principles which govern their effective use. While faculty staff may intend that students obtain a foundation in important algorithmic principles, students often focus on the manipulation of the constructs and environments while ignoring as much as possible the principles of design and abstraction which underlie their effective use. The result is that students frequently leave such courses complaining that they 'managed to get their programs to work', but that they do not have the 'big picture' of what they were doing or why they were doing it.

'We have four years of study to get important ideas across'

We do not. Computing is rapidly becoming a mainstream discipline. As with any mainstream discipline we must face the reality that CS majors will constitute a minority of CS students. The majority will come from other disciplines and will be with us for only a course or two. This simple fact means that we must consider wisely the impact we have at the introductory level. To continue to deploy a curricular model which features two years of 'programming' prior to substantive CS course work means that we effectively deny 'most students' access to the important conceptual and intellectual contributions which computing makes to virtually all disciplines.

Summarizing

In short, we find that the traditional approach is obsolete because of changes we find at every level:

- the problems which we use computing to attack;
- the software applications which have evolved into powerful computing tools;
- the programming languages-and-environments which we use to create programs;
- the 'customer population' of students which computing education must serve.

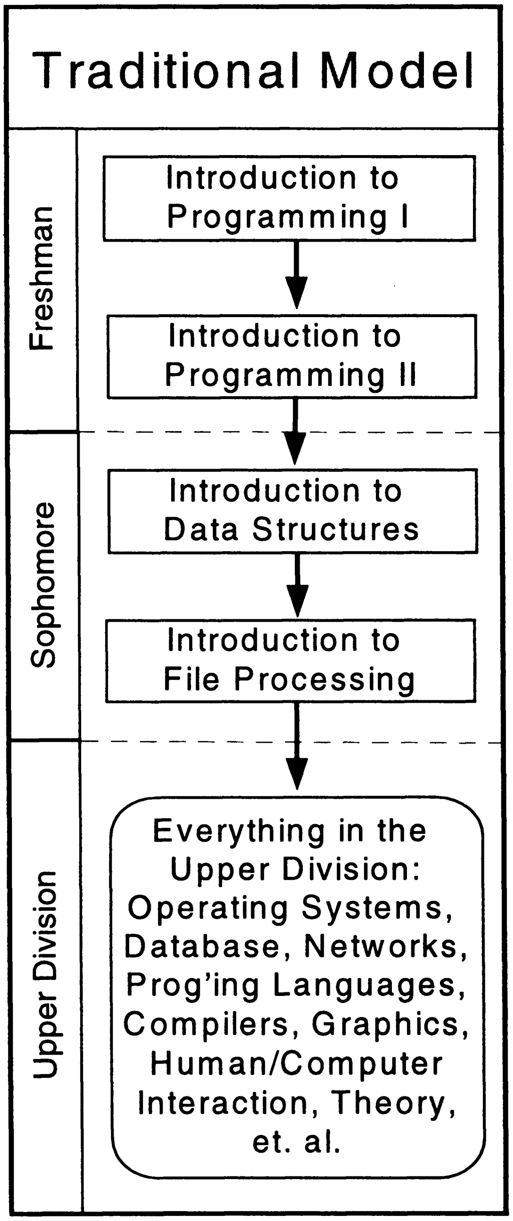
Traditional Model
Freshman
Sophomore
Upper Division
Introduction to Programming I
Introduction to Programming II
Introduction to Data Structures
Introduction to File Processing
Everything in the Upper Division: Operating Systems, Database, Networks, Prog'ing Languages, Compilers, Graphics, Human/Computer Interaction, Theory, et. al.

Figure 2 The tall, skinny curriculum tree.

4 THE GEORGIA TECH RESPONSE

Our curricular restructuring was based on the observation that traditional CS curricula are an aberration which lack a basic feature of time-honoured curricular structures. An informal survey of other disciplines reveals a structural pattern:

- introductory courses establish the subject domain and convey fundamental principles and methods;
- other lower division courses provide both breadth and depth in principles, methods and knowledge;
- upper division courses provide greater depth in specific areas of concentration.

In contrast to this pattern, CS curricula have traditionally presented two years of work focused on applied programming skills and constructs, effectively withholding fundamental concepts and knowledge until upper division courses.

This curricular tradition is an organic artefact of a brief history, not the result of explicit design. It is residue of the fact the CS courses originated as 'programming skills add-ons' to mathematics and electrical engineering curricula. As CS matured it simply 'grew upward' in the curriculum, adding more advanced courses 'on top of' the traditional skills-oriented lower division. From this perspective we can see traditional CS curricular structure as something inherited from our ad hoc past, not as the result of any conscious design decision. 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.

Before redesigning our lower division curriculum we surveyed what other institutions were doing in the way of non-traditional introductory courses. Most such offerings seemed to cluster in two categories which might be described as:

- 'Introduction to computing applications', i.e. instruction in using word processors, spreadsheets, etc.;
- 'Introduction to CS as a discipline', i.e. a survey of areas such as databases, networks, compilers.

We rejected both approaches. In our view the former lacks conceptual content and the latter 'misses the point' of what computing is and does. Furthermore neither approach prepares students for further computing study and thus must be followed by traditional programming courses.

We elected to design a common freshman sequence for both CS majors and others. In doing so we partitioned the teaching-and-learning agenda of the traditional two-term introduction to programming into chunks which are more focused and manageable:

- 'Introduction to Computing' which focuses on algorithmic concepts and skills and their application in both writing pseudo-code algorithms and using software tools;

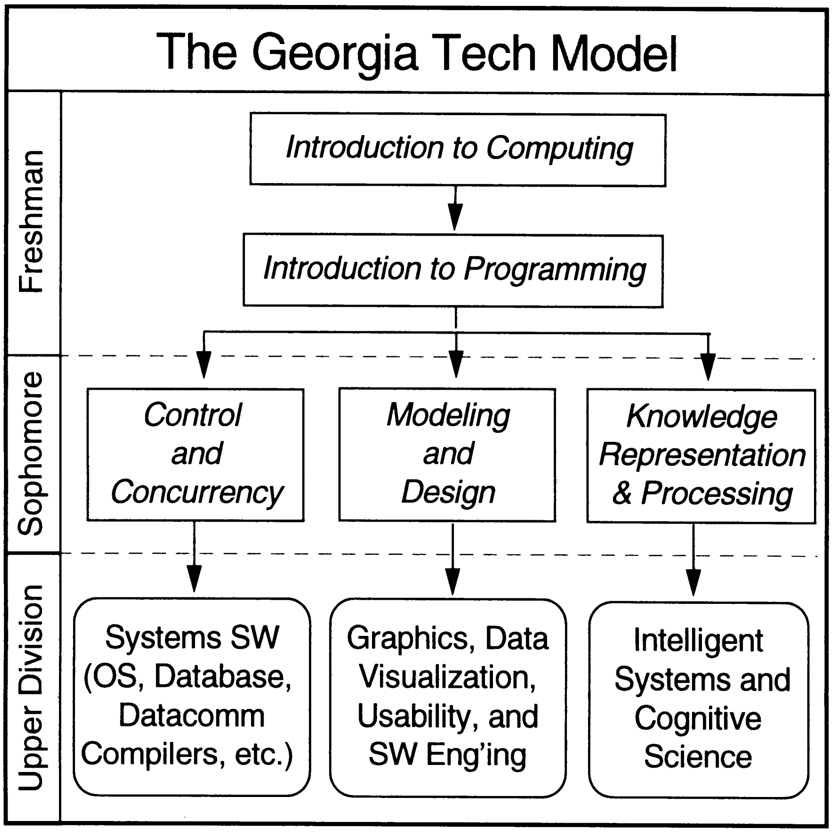
The Georgia Tech Model
Freshman
Sophomore
Upper Division
Introduction to Computing
Introduction to Programming
Control and Concurrency
Modeling and Design
Knowledge Representation & Processing
Systems SW (OS, Database, Datacomm Compilers, etc.)
Graphics, Data Visualization, Usability, and SW Eng'ing
Intelligent Systems and Cognitive Science

Figure 3 A short, fat curriculum tree.

- 'Introduction to Programming' which focuses on effective practices for algorithm implementation in a complex modern language (after students obtain a conceptual foundation in the previous course).

Following this two-course sequence we offer three sophomore courses, each of which assumes the first two courses as prerequisites and each of which introduces a different programming paradigm, language and set of computing problems. Each of the three serves as a gateway to a cluster of upper division courses, thus allowing non-CS majors access to substantive CS study without a prohibitively long chain of prerequisites. CS majors take all three, in whatever sequence they choose (plus a fourth sophomore course which introduces machine architecture and assembly language programming). We describe particulars of the freshman courses below.

Introduction to computing

The first course has two components, each with its own agenda:

- lecture-and-homework;
- laboratory.

The lecture-and-homework component focuses on basic principles and conceptual tools for creating algorithms, including a pseudo-code based introduction to the complete range of algorithm constructs (subroutines, control structures, data structures, etc.), as well as basic material in algorithm design, analysis and theory. The jury is still out on how successful one can be introducing the object-oriented paradigm without a foundation in the structured paradigm; various points of view have advocates. As a practical matter we must prepare students for a variety of possible futures and, for the present, we find it necessary to introduce both structured and object-oriented design.

With respect to analysis and theory we translate the agenda put forth by Harel (1992) into terms appropriate for freshmen. We introduce the issues of correctness, complexity and computability, with applied problem-solving in the first two. We also articulate the topics of concurrency and parallelism for a freshmen population, then build on them with applied work in determining the limits of parallelism for simple algorithms.

The laboratory component focuses on the use of campus computing resources and standard software applications, including UNIX and Windows, e-mail and news groups, word processors, spreadsheets, databases, equation solvers, graphics editors, use of the World Wide Web and the construction of web pages.

Experience has shown that liberating students from the many annoyances of programming and debugging allows them to focus more effectively on algorithm design and construction. We make sure that they understand and can apply basic concepts in a kinder pseudo-code environment which does not permit compilation. This facilitates both a better grasp and faster conceptual progress.

The laboratory focus on software tools allows us to reinforce conceptual lecture material by showing its relevance to problem solving in high-level software tools. For example, the difference between syntax and semantics can be illustrated using word processor style sheets, just as can the notion of abstraction. Similarly the distinction between procedures and functions is found in spreadsheets, embodied in the form of macros and cell formulae. Abstraction and reusability become key issues when using a graphics editor to construct composite images from graphical components. Modern applications are complex enough for most algorithmic points to be made using them. In addition we find that many students who are new to computing can achieve a sense of success and competence more rapidly than with traditional programming. And in addition it appears that the range of application experience the course provides makes students more immediately attractive to employers.

Introduction to programming

The second course also features a two-part agenda, namely a focus on effective:

- program design, including error prevention and avoidance, reusability, etc.;
- debugging, including skills and strategies.

For the first time we can count on students entering the programming course 'knowing what they are trying to do'. We therefore need not consume lecture time covering the ideas and mechanisms of parameters, modularity, static and dynamic data, recursion, etc., because students are well practised in such things from the prerequisite course. Instead we have the opportunity to focus explicitly on the myriad of important ideas and skills concerning programming. This has led us to recognize that our old curriculum used the wrong title for the introductory course: 'Introduction to programming'. It would have been more accurately described as an 'Introduction to program components in the syntax of language X'. By moving coverage of algorithmic concepts and skills to a non-programming course we enable students to progress with programming at a much faster, thereby for the first time providing a course which is a true 'introduction to programming'.

Course management issues

The fact that the new structure has been accepted across campus presents both good and bad news. The good news is that we are having a much broader impact on many more students, having succeeded in getting computing incorporated into the university's core curriculum. The bad news is that we have got 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 large numbers of students, we chose to use many undergraduate students as teaching assistants. Because undergraduates are less expensive, we can provide learning support to individual students, 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 with respect to a freshmen population. Our undergraduates obtain valuable experience in teaching, public speaking and working with people. Many students report that their teaching assistant job is the only aspect of their entire course of study which allows them to 'do something that's real'.

5 SUMMARY

Our revision of the lower division curriculum was motivated by four goals, i.e. to:

- provide students with an adequate foundation in the important concepts and skills necessary to master the complexities of modern computing;

- have students see algorithms as the focal point of problem solving, with programming language issues secondary;
- introduce students both to programming and to the tremendous power that modern applications provide;
- impact a broad population of students by positioning computing as a mainstream discipline, and by encouraging their participation in whatever computing courses might address their interests.

It appears that our design achieves each and all of these goals. In addition, it has proved to provide three benefits which we did not anticipate.

- The freshman sequence illuminated a 'hidden problem' of long standing: the absence of any course which was a true introduction to programming. Now we have the opportunity to explicitly teach programming knowledge and skills. We hope to have more to say on this in the near future.
- It gives valuable experience for our own undergraduates. Their work as teaching assistants not only enables us to give human-scale support to students in very large classes, it also provides our own students with experience in the very kind of people- and communication skills which have been a traditional weakness of CS graduates.
- Our efforts to open up our curriculum to non-CS majors has had a positive impact on our own undergraduates, as they now find themselves with a range of applied knowledge and skills (both in software applications and in multiple programming paradigms and languages) which make them much more attractive to employers.

As CS educators we have the responsibility of taking our students on a learning journey. We might think of ourselves as the driver, of students as passengers and of our curriculum as the vehicle. Viewed in this light we see that we had been transporting them in a tired old jalopy. The standard curricular structure, like an old worn-out car, did not allow us to travel as rapidly, as safely, as reliably or as far as modern complexities require. As with an old car the time had come when further repair was not sensible. But a new vehicle is costly, and we have been paying that cost, for example through the creation of textbooks (Shackelford, 1997), the invention of new course management practices (Canup and Shackelford, 1998; Schaffer, 1998) and thinking a bit differently about what we do (LeBlanc and Shackeford, 1998; Toothman and Shackelford, 1998). Despite the costs, we are well pleased with our new vehicle. We urge you to get one too.

6 REFERENCES

Canup, M. and Shackelford, R. (1998) Using Software to Solve Problems in Large Computing Classes, submitted to the *Twenty-ninth SIGCSE Symposium on Computer Science Education*, Atlanta, February 1998.

Harel, D. and Rosner, R. (1992) *Algorithmics: The Spirit of Computing,* 2nd edition. Addison-Wesley, Reading, Massachusetts.

LeBlanc, R. and Shackelford, R. (1998) Why Pseudocode Should Be Your Students' First Programming Language, submitted to the *Twenty-ninth SIGCSE Symposium on Computer Science Education*, Atlanta, February 1998.

Schaffer, K. (1998) Doing Something Real: Teaching as Part of the Undergraduate Experience, submitted to the *Twenty-ninth SIGCSE Symposium on Computer Science Education*, Atlanta, February 1998.

Shackelford, R. (1997) *Introduction to Computing and Algorithms.* Addison-Wesley, Reading, Massachusetts.

Toothman, B. and Shackelford, R. (1998) The Effects of Partially-Individualized Assignments on Subsequent Student Performance, submitted to the *Twenty-ninth SIGCSE Symposium on Computer Science Education*, Atlanta, February 1998.

7 BIOGRAPHY

Russell L. Shackelford received his Ph.D. degree in information and computer science in 1989 from the Georgia Institute of Technology, where he now is director of Lower Division Studies in the College of Computing. He came to computer science from a career in clinical psychology, and holds graduate degrees in psychology and in education as well. His research interests include computing curriculum development, the development of computing tools to support teachers and human-science researchers, and the impact of technology on human experience. He thinks computing is almost as interesting as baseball.

Richard J. LeBlanc, Jr. received his Ph.D. degree in computer sciences from the University of Wisconsin - Madison in 1977. He is a professor and the associate dean of the College of Computing at the Georgia Institute of Technology, where he has been a faculty member since January, 1978. His research interests include software engineering and programming language design and implementation. He has published a successful textbook on compiler construction ('Crafting a Compiler' and 'Crafting a Compiler with C') that has been adopted at over 100 colleges and universities. He is currently serving as chair of the ACM Education Board.