Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective

Randal E. Bryant
Klaus Sutner
Mark J. Stehlik

August, 2010
CMU-CS-10-140

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

The School of Computer Science at Carnegie Mellon University is planning major revisions to its introductory course sequence in ways that will affect not just our own students, but also the many students from across campus who take computer science courses. Major changes include: 1) revising our introductory courses to promote the principles of computational thinking, for both majors and nonmajors, 2) increasing our emphasis on the need to make software systems highly reliable and the means to achieve this, and 3) preparing students for a future in which programs will achieve high performance by exploiting parallel execution.

Keywords: computer science education, introductory computer science, computational thinking, parallel computing, Java

1 Introduction

The School of Computer Science at Carnegie Mellon University is planning major revisions to its introductory course sequence in ways that will affect not just our own students, but also the many students from across campus who take computer science courses. A committee of computer science faculty, chaired by Bob Harper, has done a careful analysis of our current course structure and proposed important changes ([1], included here as Appendix A.) We are indebted to the efforts of these faculty members in thinking about both the intellectual underpinnings of our courses as well as the many practical issues faced when making changes to course structure or content. In this document, we provide the perspective of a Dean, an Associate Dean, and Assistant Dean regarding the introductory portions of our program and the motivations for changing it. We collectively have more than 70 years of experience in undergraduate computer science education at Carnegie Mellon. We have taught many of the courses in our curriculum, we have interacted extensively with our students, and we have had many conversations with our alumni and their employers. Over the years, undergraduate education has taken an increasingly higher priority among our faculty, and we take great pride in the quality of our program.

Given the success of our current program, a natural question to ask is "Why change it?" Both our students and their employers are very pleased with the outcomes of our program. In making any changes, we must take care not to unintentionally cause harm.

Three main factors motivate our desire for change—one concerning how we communicate an intellectual vision for computer science, one concerning the changing ways that computers get used, and one concerning how computer systems are constructed.

- *Promoting Computational Thinking*. Prof. Jeannette Wing introduced this term [9] to capture the idea that computer scientists have developed unique ways of formulating and solving computational problems, yielding a rigorous discipline with a well-defined intellectual core. The principles embodied in this core can inform other disciplines, including math, science, engineering, as well as aspects of humanities, arts, and business. We would like to pursue her vision by bringing elements of computational thinking into our introductory computer science courses, including those targeting nonmajors. We believe that even for nonmajors, our introductory courses can serve the dual roles of providing a useful set of computer science skills while also providing a rigorous grounding in computational thinking, enabling students to acquire new skills throughout their careers.

- *Increasing software reliability.* There is a growing sense that we must inject greater discipline into the software development process. As computer programs control safety critical systems, the consequences of bugs become much more serious. In addition, the rising number and sophistication of malicious hackers create an environment in which seemingly minor bugs lead to serious security breaches. Inspired by the work of Prof. Edmund Clarke, we believe it is important to introduce students to the tools and techniques by which they can reason about and evaluate their programs right from the start.

- *Preparing for a world of parallel computation.* Maintaining the ever-increasing computer

performance we have experienced since the 1950s will soon require that we write programs that can exploit parallel computation. With the exception of several upper-level and graduate courses, our current presentations of how computers operate, how they are programmed, and what constitutes an efficient algorithm are based on a purely sequential model. As championed by Prof. Guy Blelloch [3], we want our students to think about how to decompose a problem such that many parts of the problem can be solved in parallel.

This document focuses on the introductory computer science courses and two of our five core courses. We continually revise and refine our more advanced courses, as well. These revisions tend to be less disruptive, however, since they involve far smaller student populations and do not require as much coordination with other parts of the university.

2 Whom We Serve

Unlike many of our peer institutions, the School of Computer Science provides computer science education for almost the entire campus. Every semester, we have around 900 students taking our introductory courses, including students from all six colleges offering undergraduate degrees. Considering that the university has around 1,450 entering undergraduate students every year, and even accounting for the fact that some students take two introductory courses, we can see that our courses engage a large fraction of the undergraduate students at some point in their time here.

We can partition the students taking our courses into three general categories

CS Majors Our undergraduate program enrolls 130–140 students per year. They are admitted directly into our program as entering freshmen. Their backgrounds range from students who have never written a line of code in their lives to ones who have developed commercial software.

Allied Nonmajors A number of other students take multiple courses in computer science. These include all of the electrical and computer engineering majors (around 150 per year), plus some students in math, science, other parts of engineering, and even some in such diverse majors as architecture, psychology, and philosophy. These students can be distinguished from the other nonmajors by the fact that they take at least one of our "core" courses, typically 15-211 (algorithms and data structures) or 15-213 (computer systems). Some of the Electrical and Computer Engineering (ECE)students take many more CS courses, covering a large fraction of the undergraduate program. Around 30 students each year fulfill the requirements to receive a minor in computer science. All told, there are around 200 students per year whom we would classify as "allied nonmajors."

Other Nonmajors Over 600 students per year take one or two computer science courses without any plan to go beyond an introductory level. Many of them are required to take at least one CS course as part of their degree requirements. This includes around 50 students per year enrolled in the Information Systems program in the School of Humanities and Social Science.

Implications

The diversity of the students we encounter has significant repercussions for how we structure our introductory courses:

- The sheer number of students taking introductory courses stresses our teaching resources.
- Over half of our students never go beyond an introductory level. What we teach them in their first course will define their entire concept of the field of computer science.
- We must satisfy the educational needs of a wide variety of students, ranging from those wishing to pursue computer science for the rest of their lives to those hoping to get by with minimal exposure.
- We must structure our course sequences to be compatible with a number of different educational programs.

Providing an educational model for the world

In addition to educating our own students, Carnegie Mellon has the opportunity, and perhaps the duty, to inform the world with a vision for computer science education. Jeannette Wing's writings on computational thinking have inspired educators worldwide. We historically played a major role in designing the advanced placement exams for computer science. Our Alice Project has demonstrated that young people can learn the concepts of programming in the context of computer game construction and storytelling. Our success in female participation in computer science has gained widespread recognition.

How we teach introductory computer science is especially important in this role as an educational model. When students learn computer science in high school, one primary objective is to gain advanced placement, skipping over one or more college-level courses. The advanced placement exam is designed to test mastery of freshman-level material. Right now, that exam is largely a test of elementary programming in Java. High school teachers are highly motivated to "teach to the test," and consequently students are given a very skills-oriented perspective on computer science. This conveys a message to many high school students—the very ones we would like to recruit—that computer science has little in the way of intellectual content and that their career opportunities will be limited to sitting in cubicles all day long churning out code that will have little impact on the world. Students also see this path as providing weak career prospects, as they appear to be jobs that can readily be outsourced to lower wage countries. Thus, how we teach introductory computer science at Carnegie Mellon has a ripple effect on how society views computer science as an intellectual discipline.

Overall, universities in the United States saw steep drops in enrollments in computer science programs, after peaking during the "dot-com boom." At Carnegie Mellon, we saw this as a drop in the number of students applying to the computer science program from an all-time high of 3,237 in 2001 to a low of 1,732 in 2005. (Since we have the luxury of selecting only the top students

from our applicant pool, we have been able to maintain a steady rate of 130–140 entering students per year without compromising at all on quality.) Fortunately, our applicant pool has returned to a healthy level, with 3,026 students—the second highest number ever—applying to enter in 2010. Nationally, the number of students deciding to major in computer science has risen only slightly in the past few years. Contrary to this trend, and to the perceptions of many, the U. S. Bureau of Labor Statistics forecasts that almost 75% of the nation's new science and engineering jobs will be in computing, while just 16% will be in other engineering disciplines. They forecast that the number of job openings in computing through 2018 will average 140,000 per year, while the number of students receiving degrees in computing during that time will average less than 50,000 per year.

One unfortunate trend is that computer science courses are rapidly disappearing from our nation's high schools due to budget pressures, the need to devote all resources to achieving the metrics imposed by the No Child Left Behind Act, a lack of qualified teachers, and a lack of interest on the part of students. One consequence of this is that, although our enrollment numbers are very strong, an increasing number of our students have had little opportunity to program computers. Thus, we must continue to accommodate a wide range of prior experience among students taking our introductory computer science classes.

Major efforts are underway by the Association for Computing Machinery (ACM—a computer science professional society), the Computing Research Association, the National Science Foundation, and even the Department of Defense to find ways to infuse an appreciation and excitement for computer science in middle and high school students in the United States, through experiences both within and outside of the classroom. As an institution that wants to be viewed as one of the thought leaders for the field, we would like to play a role in these efforts. A first step is to make sure our introductory courses provide a more inviting picture of the meaning of "computer science."

3 Proposed Changes

Figure 1 highlights the proposed introductory courses and several of the follow-on core courses. The content of the courses are summarized as follows. We provide more extensive descriptions later in this document and in the report of Appendix A

15-110: Principles of Computer Science. An introduction to computer science, based on the principles of computational thinking. This course is primarily for nonmajors, although we also anticipate that any entering student with limited programming experience will be strongly encouraged to take it (as indicated by the dashed line from 15-110 to 15-122.)

15-122: Principles of Imperative Computation. Introduces students to methods for writing and reasoning about programs written in an imperative style, where each step of computation updates some portion of the program state. The course will go beyond our current Java-based introductory programming course (15-121) to include elementary algorithms and data structures and how to systematically reason about program behavior, for example by expressing invariant properties of loops.

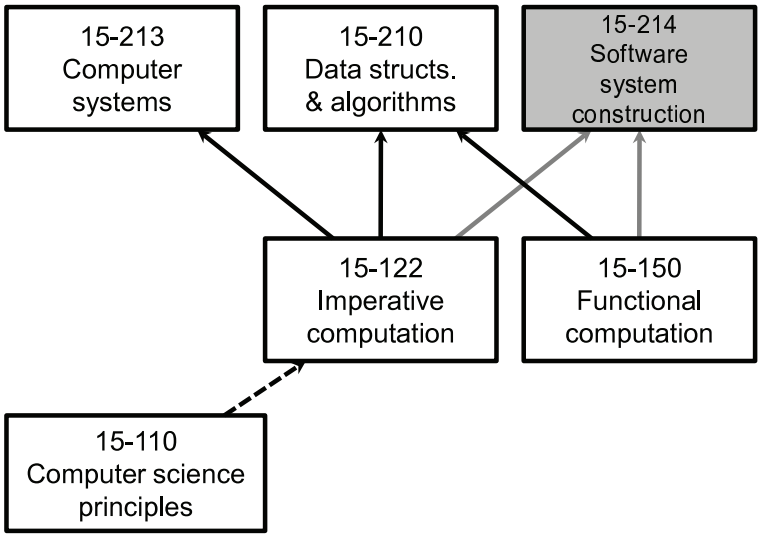

Figure 1: **Proposed introductory and follow-on courses.** Arrows indicate prerequisite dependencies.

15-150: Principles of Functional Computation. Introduces students to methods for writing and reasoning about programs written in a functional style, where a computation is realized as a nested sequence of function calls, each call defining a map from inputs to outputs, but not altering any state. The course covers methods to define and reason about data types, infinite data structures, and higher-order control constructs. The material will be based on the current 15-212 course, which will be discontinued.

15-210: Fundamental algorithms and data structures, along with supporting theoretical foundations and practical applications. Our existing data structures and algorithms course (15-211), but totally revamped to include ways to reason about and optimize their performance on both sequential and parallel machines.

15-213: Introduction to computer systems. Our existing course introducing students to how computer systems execute programs, store information, and communicate, presented from a programmer's perspective. This course will remain largely unchanged.

15-214: Principles of software system construction. A new course that will cover the methodology for designing large-scale software systems, including object-oriented programming, concurrency, and component-based software reuse. The box for this course is shaded in gray, since much about this class has not been resolved, including its name, course number, prerequisites, and content.

Students wanting to take 15-210 would need to take both the imperative and functional programming courses. Students just wishing to take 15-213 (including most ECEs) would only need to take 15-122. Most nonmajors would just take 15-110, and this course could also be used to help stu-

dents who need more familiarity with programming before they are ready for the two introductory programming courses.

The two major changes to the overall course structure are 1) moving our coverage of functional computation to an introductory course that precedes the data structures and algorithms course, and 2) creating a new course on software system construction. The first change is based on our belief that a functional perspective should be a key part of expressing and reasoning about parallel computation. The second is in recognition of the need to better prepare students for the complex software systems they will encounter in their careers.

Let us now review the motivation for this course structure.

Computational thinking from the start

Computational thinking, a term coined by Prof. Jeannette Wing [9], refers to the set of concepts and strategies used by computer scientists to formulate and solve problems. As she has described it, computational thinking involves two primary aspects:

- *Creating Abstractions.* Every field of engineering or science involves creating and working with abstracted models, such as those found in Newtonian mechanics and Keynesian economics. In computer science, however, we can often be more creative with our abstractions, since these abstractions serve more as principled ways to structure a system, rather than as models of real-world processes. Consider, for example, an online retailer's computer system. It incorporates many abstract models: client/server, relational database, distributed transactions, program objects, etc., that enable the designers to create a system that provides the necessary functionality in a reliable and secure way. These models are purely organizing principles and not abstracted views of real-world systems. In constructing such models, system designers are constrained more by their imaginations and the need to manage complexity than by the laws of physics.

  Important forms of abstraction in computer science include:

  Algorithmic formulations: Many problems in computer science can be formulated as abstract processes or classes of operations on data that can be solved by systematic computational methods. There is a wide range of possible solution methods that can then be analyzed for both correctness and efficiency. Often, a single category of algorithm can solve problems across many application domains. So, for example, the idea of hashing—creating a mapping where the output appears to be randomly distributed even if the input data are not—finds applications in information retrieval, digital signatures, and distributed file sharing. The study of algorithms is an intellectually deep and compelling subject that we cover from many different perspectives in our courses, with some requiring an extensive mathematical background.

  Modularity: Modular design involves partitioning a system into components, such that each component can have a succinct description of its behavior or properties, hiding the

details of its implementation. Some common forms of modularity in computer science include procedures, data abstraction, and object-oriented programming. Of course, modular design is also used in many other domains, ranging from building construction to product supply chains, but the use of modularity in computer science can be much more complete and pervasive.

- *Realizing Abstractions.* Computer science offers a wide range of different ways to implement particular abstractions. Compared to other fields of engineering, our abstractions are less coupled to a particular physical realization. Most of the traditional focus has been on mapping a computation onto a single, sequential processor. As we will describe shortly, however, there are increasing opportunities to divide up a task and map it onto multiple computing elements, either through parallelism or distribution.

  A second dimension concerns iteration versus recursion, two ways of scaling a computation to multiple elements. Both approaches can be applied to sequential, parallel, and distributed computations.

Our introductory courses should present the core concepts of computer science, conveying the principles of computational thinking. Even though we cannot delve into them deeply, we can provide enough coverage to demonstrate the intellectual and practical value of the principles of computer science. We will still teach students how to write programs, since programming provides the tools to actually try out computer science concepts and make them more concrete, but we must not let the mastery of programming skills be the principle focus. We also recognize the desire of other departments to have students be able to write programs to solve problems in their domains.

One other aspect of computational thinking is to think about how the power of computation can be applied to tasks that we encounter in our courses, professions, and day-to-day lives. When a mathematician has a conjecture, it often makes sense to write a small program that will test the conjecture for millions of cases before attempting a formal proof. When computer architects design microprocessors, they routinely write programs to generate large collections of tests that exercise different combinations of instructions. When one of us wanted to appeal his property assessment, he wrote programs that extracted the records for over 100 comparable properties from the Allegheny County website. One of the most powerful productivity tools we could provide students in all disciplines is instruction and experience in writing programs that automate tasks such as these.

The desire to introduce the concepts of computer science even to nonmajors motivates our recasting 15-110, which currently focuses on programming in Java, to be a more general introductory course based on the theme of computational thinking. We believe that modern programming environments and languages simplify the task of writing code enough that we can cover basic computer science concepts and provide practical programming skills in a single course that is suitable for a broad cross section of students. In their programming assignments, students will be able to apply sophisticated analyses to data collected from a variety of different sources, including laboratory equipment, image databases, and the world-wide web.

Our two other introductory courses: 15-122 and 15-150 will also include many aspects of computational thinking. Their principal aim, however, is to start building the foundations for a more extended study of computer science. Whereas 15-110 should be a course that provides a general picture of the essence of computer science, the other courses can go into greater depth but with a narrower focus.

The need for reliable and secure programs

The consequences of writing buggy programs are becoming increasingly serious. As computers are increasingly used to control critical resources, such as heart pacemakers, antilock braking systems, and the national power grid, a simple bug can literally cause people to die. In addition, most security breaches in computer systems occur due to poorly written programs. For example, a buffer overflow exploit involves having an attacker supply a larger stream of data than was anticipated by the programmer, causing corruption of the program data structures. With the increasing number and sophistication of malicious adversaries in the world, systems are exposed to a very hostile testing environment that is likely to uncover even the most obscure bugs.

As an illustration, the recent Operation Aurora attacks on machines at Google, Adobe, and other companies gained initial entry by exploiting a weakness in Internet Explorer. A user's machine could become infected when a downloaded page caused malicious code to be executed by referencing an invalid pointer [7]. The attackers then inserted a back door into the companies' systems to access their source code repositories. The attack involved many steps of both gaining access and hiding its trail [10]. As this example illustrates, the existence of many, seemingly minor flaws in commercial software allows adversaries to tunnel their ways through many layers of security. Programmers must increase code quality to ensure that it is secure against attack. For example, Microsoft went from having a reputation for being somewhat cavalier about software quality to viewing code security as a primary concern [6].

Writing secure and reliable code requires programmers to be more disciplined in how they reason about and test their programs. It might have been common practice at one time for students to continually modify their programs, adding more and more complexity, until they could pass a set of tests representing "typical" cases. Now we want students to be able to argue both formally and informally that their programs will run under all possible conditions. We want them to understand the logical methods by which they can reason about programs, and to be familiar with the growing collection of tools that can aid systematic program development. Both of the introductory programming courses: 15-122 and 15-150 will go beyond the mechanics of writing code to include more on the fundamental principles of programming. These principles will be reinforced in 15-214, which will also introduce methodologies and tools for reasoning about large-scale software systems.

Preparing for parallel computation

For decades, hardware designers have used the ever-increasing number of transistors that can be integrated onto a single chip to produce faster microprocessors. We could reliably expect a program to run faster by simply upgrading to a newer processor. While the number of transistors that can be integrated onto a chip continues to double every 18–24 months, semiconductor manufacturers are no longer able to design circuits that will execute individual code sequences much faster [8]. Instead, they have shifted to a strategy of increasing the number of independent processors, or *cores*, integrated onto a single chip. Making a program run faster now requires that it be written in such a way that multiple parts can be executed in parallel. Opportunities for parallelism also arise in lower-level forms, where entire vectors of data can be processed in parallel. Our changing technology has led to a mismatch, where the widely used model of computation based on sequential execution does not reflect the parallel execution modes supported by the hardware. We must therefore shift to models that expose the many opportunities parallel execution within computations.

Current multicore processors only have 4 or 8 processor cores. Achieving such small-scale parallelism requires only minor modifications to existing programs. One simple strategy is to assign dedicated cores to background tasks, including monitoring network ports, scanning files for viruses, and backing up files. We can anticipate much higher degrees of parallelism in the future, but our current programming methods will not be able to go much beyond 16- or 32-way parallelism.

Our plan for revising (and renumbering) our data structures and algorithms course 15-210 is to teach students to *think parallel.* By this we mean that every aspect of how programs are expressed, how they are analyzed, and how they are optimized will be reworked to consider 1) how they would perform if there were unbounded computing resources, and 2) how they would perform if limited to n processors, with sequential execution being the special case of $n = 1$.

In terms of expressing programs, a first step is to eliminate constructs that artificially introduce sequential behavior. Rather than writing loops that sequentially step through a set of values, we will use constructs that express operations at a higher level, for example mapping a function to all elements of a set. The functional programming models they learned in 15-150 will help them reason about programs at this higher level of abstraction.

The course will present traditional key algorithmic techniques: such as divide-and-conquer, dynamic programming, and greedy methods, as well as ones specifically oriented to parallel computation, such as contraction and map/reduce.

Rather than writing and measuring parallel programs for specific machines, our plan is to express and reason about parallelism at a much higher level of abstraction. In particular, we can characterize a program in terms of its *work* W, expressing the total number of operations that must be performed, and its *span* S, expressing the longest sequential dependency. Given a system with n processors, the potentially achievable execution time would therefore be $\max(S, W/n)$. This measure covers the extreme cases of maximal parallelism (S) and purely sequential execution (W), as well as everything in between. This form of analysis is a natural extension of the asymp-

totic analysis of sequential programs that has proved so effective for reasoning about algorithms in machine-independent ways. For example, the traditional average case analysis of Quicksort generalizes to showing it has span $O(\log n)$ and work $O(n \log n)$.

Our coverage of parallel computing in 15-210 will focus on *deterministic parallelism*, where the outcome of a computation is fully determined by the data and not by some chance ordering of events. By contrast, some systems attempt to extract parallelism through *concurrency*, where a program is divided into a number of tasks that are executed on separate processor cores, with their interactions coordinated by synchronization operations. This is a more difficult and more error-prone way to improve program performance. As we will discuss, our coverage of concurrency will be more as a way to support distributed computing and the handling of asynchronous events.

4 Other Important Trends

To close out our discussion of trends for our field, we include several other recent ones. These are trends that will be dealt with directly in more advanced courses. We believe, however, that our revised 100- and 200-level courses will help lay the groundwork for them.

Concurrent and distributed computing

Distributed computing involves organizing a system as a number of concurrently active computing agents that coordinate their actions via a set of protocols. Whereas parallel computing is purely a technique to improve performance, a distributed system may exist to facilitate communication among independent entities (e.g., the set of Internet hosts) or to process asynchronous events (e.g., the requests to a server from multiple clients.) Distributed system design typically involves more concern for fault tolerance, reliability and security than for raw performance. The boundaries between the two classes of system are not precise, however. For example, the large-scale "cluster" computing systems used at companies such as Google [2] are structured using distributed system principles, but many are used to perform parallel computing over very large data sets. (One way Google has made it tractable to write programs for their distributed system infrastructure is to layer their Map/Reduce programming model [5], a form of deterministic parallelism, on top of it. This shields programmers from the pitfalls of managing low-level task coordination themselves.)

In our curriculum, we begin introducing students to the concepts of concurrency in 15-213, our introductory computer systems course. Our planned 15-214 course on software system construction will have much more extensive coverage of concurrency, as well as object-oriented abstractions. Both of these are key components of most large-scale, distributed software systems. We also have several advanced courses on the principles of distributed systems and web-based programming.

Data-intensive scalable computing

Whereas traditional high-performance computing focuses on maximizing the ability of systems to perform complex numerical computations, an emerging class of applications derives benefits from collecting and analyzing data sets consisting of multiple terabytes of data. (A terabyte is 10^{12} bytes. By reference, the set of all books in the U.S. Library of Congress, when converted into digital form, would require around 20 terabytes.)

At Carnegie Mellon, we've taken on *Data Intensive Scalable Computing* (or "DISC") as a major focus for our research efforts. We believe that the potential applications for data-intensive computing are nearly limitless, that many challenging and exciting research problems arise when trying to scale up our systems and computations to handle terabyte-scale datasets, and that we need to expose our students to the technologies that will help them cope with the increasingly data-intensive world. Realizing the promise of DISC requires combining the talents of people from across many disciplines, and Carnegie Mellon, with its strengths in engineering, computer science, and many application disciplines, has played a major role in getting universities involved in this style of computing [4].

From an educational perspective, we believe that our revised curriculum will provide students with a strong foundation for data-intensive computing. For example, the Map/Reduce programming model pioneered at Google [5] for mapping data-intensive applications onto large-scale, cluster computing systems has its roots in functional programming. Having students learn functional programming from the start, and evaluating the efficiency of algorithms from both sequential and parallel perspectives will serve our students well as they write programs for these machines. Our more advanced courses in distributed systems and parallel algorithms now include coverage of Map/Reduce programming and its applications.

5 Practical Matters

Our discussion so far has focused largely on the conceptual basis for our curriculum and courses. Some people have expressed concerns about how these changes will affect the ability of students to gain skills that will help them with their further studies (in both computer science and other areas) and for their careers. We address these concerns here. Carnegie Mellon has long been an institution that strives to teach students deep principles while preparing them for the workforce. We believe that our courses can serve both roles.

Preparing students for their careers

Of the 140 or so students who receive undergraduate degrees in computer science each year, around 35 go on to graduate school, including masters and PhD programs in computer science, as well as to business, law, and medical schools. The majority of our students enter directly into the workforce, at large computer companies (such as Microsoft and Google), at technology startups, and

also at companies that make extensive use of computer technology, especially financial services. Regardless of their employers, most of our students pursue technical careers. While we seek students who will become leaders, we count more VPs for engineering among our alumni than CEOs. Thus, while some undergraduate programs view their primary purpose as preparing students for advanced study or for managerial positions, we find that most of our graduates make direct use of the technical knowledge they gain from our program.

A significant tension exists in any educational program and in every course as to whether the purpose is to teach fundamental principles or to teach usable job skills. On one hand, useful skills help students get jobs, but on the other hand, they tend to become obsolete within 5–10 years. We want our students to get jobs but also to be able to have fruitful careers spanning perhaps 50 years.

We have designed our computer science program to satisfy both goals. Our courses combine strong foundations with deep experience involving real-life systems. In talking with employers, we are often praised for producing students who can be productive on their first day of work, while continuing to adapt to the many changes that occur over the years. We teach our students classical and current material as well as how they can learn new material themselves. We avoid teaching skills that are largely artifacts of current technology. We do not teach courses in .NET programming or in Cisco router configuration. We generally avoid proprietary systems and languages.

How important is the selection of programming language?

Discussions about course design in computer science often quickly focus on the choice of programming language. While we acknowledge that choosing the language(s) taught in a course requires careful consideration, we admonish people not to fixate so strongly on this issue. Languages fall into a small number of general classes. Once one has gained experience in one language in a category, it is fairly simple to pick up another, often by simply reading a language manual.

In conversations with recent alumni, we have been advised that it is more important that we promote a sense of flexibility—that students should be exposed to a variety of programming models and become comfortable with learning new languages through self study.

Note that, unlike others, we do not advocate eliminating or minimizing programming as a key aspect of introductory computer science. We continue to believe that writing programs is one of the core activities of a computer science education. Writing code is the means by which we convert general concepts of computing into concrete realizations and in the process gain a more complete understanding of the concepts. We want students—majors and nonmajors—to have the writing of code as one of the sharpest tools in their toolboxes when confronting new tasks.

What about Java?

The Java programming language has emerged as one of the most widespread platforms for large-scale software development. It has the advantages of (1) careful design and standardization to work in many different hardware and operating system configurations, (2) an extensive set of

libraries enabling application programmers to focus on high-level objectives rather than low-level implementations, and (3) increasingly sophisticated implementations that allow programs to have performance comparable to that of carefully crafted, highly tuned code. We also take pride in the fact that James Gosling, the developer of Java, is an alumnus of our computer science PhD program.

That said, we do not believe Java is the right language for introductory programming courses. As a first language, it has two major shortcomings:

- *It takes a lot of work to do simple things.* The overhead involved in writing even the most elementary Java program, with class definitions, invoking libraries to do simple printing, etc., get in the way of introducing the basic concepts of programming. Some introductory courses try to minimize the "Java-ness" of the coverage by having students write programs that make no use of class abstractions. Basically, they do C programming with a slightly different syntax. But, that is clearly a compromise, fully satisfying neither goal of minimizing the mechanics of writing programs or of teaching Java programming.

- *It hides too much about what's really going on.* One of Java's strengths is the rich set of libraries supporting different data abstractions. In using a dictionary data type, the user need not know whether it is implemented as a hash table or a balanced tree. That's great when the objective of a program is to implement some application, but it's not helpful when the objective is to teach students about algorithms and their performance. It is difficult even for experienced Java programmers to understand and optimize the time and space costs of their programs. For novice programmers trying to learn about efficient algorithms, the language and its runtime system mask important performance issues.

We would also argue that our current approach of teaching Java as an introductory language and then never revisiting the language later does an injustice to Java itself. Java provides many rich capabilities that are worthy of study in more advanced courses. Features such as introspection, type hierarchy, concurrency, and performance optimization deserve a more thorough treatment after students have gained greater sophistication in their understanding of programming languages and concurrent programming.

Our new 15-214 course on software system construction will include object-oriented programming, since this is an important approach for structuring complex software systems. They are currently planning on using Java for this course, for its support of objects, concurrency, and component-based software reuse. Using Java for this course will better situate Java within our curriculum.

What language do we plan to use in 15-110?

As mentioned, we believe that our new 15-110 course can convey the conceptual basis of computer science plus provide concrete experience in writing programs that solve practical problems. We plan to do this by using a so-called *scripting language*. Compared to languages such as C or Java, these languages have simplified syntax and type systems, and they have primitives that make it

easier to invoke other programs and to process strings of text or data. Our current plan is to use Python, but other choices are being considered as well.

Although the term "scripting language" seems to imply they are just useful for writing small scripts that automate some routine task, these languages are perfectly usable for writing sophisticated programs. They sacrifice some amount of performance, and they do not contain some of the features desired for assembling large-scale software systems, but they are well suited for the needs of an introductory computer science course. A number of other universities have also developed introductory courses based on languages such as Python.

Scripting languages work within interactive programming environments that foster an incremental, "learn by doing" approach to software development. Programmers write in small chunks, testing each chunk before integrating it into the rest of the code. This approach works well for writing small, application-specific programs, and is less time consuming than the more deliberative process required by the edit/compile/test/debug cycle of more conventional languages. Python has available a large set of libraries supporting scientific computing and graphics.

In discussions with faculty in other science and engineering departments, we have found them very receptive to the idea of switching away from Java, but then they often ask that we teach the language included as part of the MATLAB mathematical modeling system. We believe, however, that MATLAB is not the right choice for introducing students to the concepts of computational thinking. It really is a language for manipulating matrices, not a general-purpose programming language. We also believe that students can readily learn the language once they have become familiar with other languages and the underlying principles of computation. Python, for example, has libraries that allow the construction and manipulation of matrices in a style similar to that supported by MATLAB, and so the transition to MATLAB by students who have taken 15-110 should require little effort.

Our shift to Python will let us satisfy the twin goals of presenting the important concepts of computer science while also providing students with skills that they can apply to other courses and in their jobs. Experience at other schools lends credence to this optimistic outlook. But, we also do not view the shift to Python as the most important aspect of this course. Our larger goal is to get students to think about computer science in terms of a set of core principles and strategies, rather than to create programs in a particular language.

6 Conclusions

Although there are many details to work out, we are moving ahead with a revised set of introductory computer science courses that we believe will better serve majors and nonmajors alike, and will put Carnegie Mellon at the intellectual forefront of computer science education. We will shift away from a focus on the mechanics of programming and instead cover the general principles of computer science, based on the theme of computational thinking. We will have students learn to reason about programs in more systematic ways, recognizing the increasing need for secure and reliable programs. We will prepare students for a time when parallel computing becomes the main

method by which we can continue to push the limits of computer performance.

This document has only provided a high-level view of our planned structure for introductory computer science. Details on course offerings, requirements, and how we will transition to this new structure will be discussed elsewhere.

Acknowledgments

We are grateful to the faculty committee members who spent many hours studying the needs of all of the students taking introductory computer science courses and devising a plan for the future, under the guidance of Bob Harper. Other committee members include: David Andersen, Guy Blelloch, John Lafferty, Frank Pfenning, Andre Platzer, and Danny Sleator. Both they, and other faculty members have given valuable feedback on this document, especially: Jonathan Aldrich, Tom Cortina, Mike Erdmann, and Ananda Gunawardena.

Jeannette Wing has been very helpful in reviewing this document and helping refine our understanding of computational thinking.

References

[1] D. Anderson, G. Blelloch, R. Harper, J. Lafferty, F. Pfenning, A. Platzer, D. Sleator, and M. Stehlik. Recommendations for revising introductory computer science. Carnegie Mellon University, February 2010.

[2] L. A. Barroso, J. Dean, and U. Hölze. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[3] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] R. E. Bryant. Data-intensive supercomputing: The case for DISC. Technical Report CMU-CS-07-128, Carnegie Mellon University, 2007.

[5] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[6] S. Lipner. The trustworthy computer security development lifecycle. In *20th Annual Computer Security Applications Conference*, pages 2–13. IEEE, 2004.

[7] R. Naraine. Microsoft says Google was hacked with IE zero-day. *ZDNet*, Jan. 16 2010.

[8] H. Sutter. The free lunch is over. *Dr. Dobb's Journal*, 30(3), March 2005.

[9] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, March 2006.

[10] K. Zetter. Google hack attack was ultra sophisticated, new details show. *Wired Magazine*, Jan. 14 2010.

A Introductory Curriculum Review Committee Report

Attached is the report generated by the committee formed to review and recommend revisions to the introductory computer science curriculum.

Recommendations for Revising Introductory Computer Science

David Andersen Guy Blelloch Robert Harper (chair)
John Lafferty Frank Pfenning Andre Platzer Danny Sleator
Mark Stehlik.

February, 2010

1 Background and Motivation

In the Spring of 2009 SCS Dean Randy Bryant, CSD Head Peter Lee, and Associate Dean Klaus Sutner asked Robert Harper to form a committee to review the introductory curriculum in Computer Science with these considerations in mind:

1. The introductory curriculum had been largely stable for more than ten years, so it was time to review and reconsider our offerings in light of academic and industrial trends, especially the increasing importance of parallel computing.

2. The financial stringencies being faced by the university limited the available resources for any change.

3. Dean Bryant suggested that we reconsider the role of Java in the introductory and core curriculum.

2 Activities

The committee met almost weekly to discuss the current state of the curriculum, to isolate particular issues, and to propose alternatives that might better serve our goals overall. The committee collectively and individually sought input from a number of faculty within SCS, including Dave Feinberg, Tom Cortina, Klaus Sutner, Bill Scherlis, and Chris Langmead. The chair met with about a dozen faculty from across campus (in three groups representing CIT, MCS, and HSS) to inform them of proposed changes and to solicit recommendations. The chair consulted regularly with the URC, which conducted an informal survey of undergraduates

about the proposed changes. The chair met regularly with Dean Bryant to discuss progress. The chair made three presentations during the Fall semester of 2009: to the Dean's Advisory Board, to the Alumni Advisory Board, and to the Computer Science Department faculty. Committee member Mark Stehlik met with the Student Advisory Council to discuss the proposed changes.

3 Objectives

The committee formulated these objectives for the restructuring of the introductory curriculum:

- The introductory curriculum should stimulate excitement for Computer Science by emphasizing its enduring scientific and engineering principles and its wide applicability to many problem areas. It should avoid over-identification with the ups and downs of the computer industry.
- To support flexibility in admissions, including increasing the representation of women and minorities, the introductory curriculum should accommodate students with widely varying backgrounds and levels of preparation. For example, it should not rely too heavily on prior computing experience (for example, through AP CS), but should still challenge more-experienced students.
- The introductory curriculum serves two broad classes of students, those for whom the introductory courses are a terminal education in CS, and those for whom the introductory courses are preparation for the core curriculum and upper division courses. The needs of these two broad classes of students are often incompatible; the introductory curriculum should be restructured to better serve their needs.
- It is important to prepare students for summer internships as early as the end of the first year, but certainly (and in most cases more realistically) by the end of the second year.
- The introductory and core curricula should be brought up to date to reflect changes in the field over the last decade, and to anticipate, as best as possible, expected trends over the next decade. For example, as parallelism becomes prevalent, it becomes important for students to be taught to think of parallel, rather than sequential, computing as the ordinary case. Similarly, as formal methods become more important in ensuring software quality, it becomes more important for students to be proficient in the principles of logic and semantics that underly them.

- Although object-oriented programming (in its myriad forms) remains a dominant theme in industrial software development, the use of object-oriented languages, such as Java, at the introductory level introduces considerable complexity and distracts from the core goals at the introductory level. It seems preferable to give fuller coverage of OO design and implementation methodology to later in the curriculum to allow more focused concentration on basics at the introductory level.

- The revisions to the curriculum are to be undertaken within tight resource constraints.

Obviously these objectives are not entirely compatible! For example, current industrially relevant languages stress sequential programming at the expense of parallel programming. Moreover, the complexity of languages such as Java impede efforts to teach basic principles of program design and analysis. Put the other way around, if currently industrially relevant languages and techniques are to be emphasized, the goals of introducing parallelism and verification concepts are necessarily compromised. Put yet another way, the goal to prepare students for summer internships conflicts, to some degree, with the goal to teach new approaches to programming.

4 Current Structure

The current introductory curriculum streams both terminal and non-terminal students through the same set of courses. A prime objective is to prepare continuing students for 15–211 (Data Structures and Algorithms) and 15–213 (Computer Systems). The courses are structured to support both students with computing experience and those without. The main introductory courses are these:

- 15–110: Introduction to Programming. For students with no computing background as preparation for 15–121, and for non-continuing students. Taught using Java.

- 15–121: Introduction to Data Structures. Main preparation for 15-211. Students with computing background start here. Also serves as second course for IS students. Taught using Java.

- 15–123: Effective Programming in C and UNIX. Preparation for 15–213. Introduction to C programming and UNIX (command-line) tools. Taught using C and various scripting languages.

The current pre-requisite structure among these courses is as follows:

- 15–110 is required for 15–121 and 15–123.
- Either 15–110 or 15–121, and 21–127, are required for 15–251.
- 15–121 and 21-127 are required for 15–211.
- 15–123 is required for 15–213.
- 15–251 is required for 15–212.

Since 15-211 is taught using Java and 15-213 is taught using C, it is considered essential that the introductory curriculum be taught using these languages.

Discussions of the current curriculum raised a number of points, including these:

1. For most non-continuing students (other than IS students) there is little to be gained by learning the Java language. Moreover, most students perceive that the point of 15–110 and 15–121 is to "learn Java," and hence they see little value in taking the course. One symptom is that some students wait until their last semester to take 15–110 or 15–121 as required for their major.

2. Across campus many faculty complained that learning Java was not appropriate for their students, but still many faculty felt that the purpose of 15–110 and 15–121 is to teach a particular language. (The most commonly advocated alternative was MatLab, with some support for Python or similar scripting language, SQL for manipulating databases, and the C^{++} STL.) There is some support across campus for teaching computing fundamentals, but many faculty continue to perceive these courses as skills training.

3. For IS students there is strong support for learning Java because of its perceived relevance to the software industry. This recommendation is consistent with the pervasive attitude that introductory CS is a form of skills training, and does not address the question of what should be a fundamental education in CS for IS students.

4. The different sections of 15–110 use different textbooks and different teaching styles, and differ widely in their degree of rigor. It is often a matter of accident which students end up in which sections.

5. There is a strong overlap between 15–110 and 15–121, to the extent that in some cases half or more of the semester in 15–121 is spent on elementary programming techniques that are also covered in 15–110. One reason for the overlap is that students enter with widely differing backgrounds.

6. The structure and content of 15–110 and 15–121 is overly determined by the needs of 15–211, given that relatively few students taking the introductory courses in fact go on to more advanced computing courses.

7. There may be more effective means of delivering skills training for both continuing and terminal students. For example, one can imagine introducing mini-courses in Unix skills, or in using MatLab, that could address these needs. It is strongly felt that the introductory curriculum should focus on enduring principles, rather than on currently applicable skills and technologies.

Historically, 15–211 has suffered from the sometimes-conflicting objectives of introducing rigorous computer science as well as teaching programming. The Fall and Spring semesters have developed a different balance of emphasis on programming technique versus the algorithms component. It appears that Java exacerbates the problem because the language is relatively complex and does not admit a simple subset that could be used in this course.

There is an issue of overlap between 15–211, 15–251, 15–212, and 15–451, known as "the RSA problem" but which extends to more than just this one topic. There is some danger of perpetuating, or even aggravating, this problem in the revised curriculum, as discussed below. This problem should be addressed separately from the present proposal by more careful coordination among the courses.

5 Proposed Structure

The recommendations of this committee may be summarized as follows:

1. Reposition 15-110 to provide a fundamental grounding in the principles of computer science, primarily aimed at non-continuing students. It will also provide a foundation for students who may wish to transition to 15–122 or 15–150.

2. Redefine 15-211 (and renumber) to teach data structures and algorithms based on a parallel, rather than sequential, computational model.

3. Replace 15–121 by two new entry-level courses, 15–150 (Principles of Functional Computation) and 15–122 (Principles of Imperative Computation).

4. Refactor 15–212 into 15–150 and 15–211.

5. Introduce a new course on programming methodology to compensate for material lost from the current 15–211 and 15–212.

6. Relegate 15–123 to a skills elective to be taken before students take upper-level systems courses. It might also be useful to consider offering other skills electives, such as programming with MatLab, to satisfy demand.

These recommendations represent the consensus of the committee for the eventual outcome of restructuring our introductory CS curriculum. The process of making these changes requires careful planning not addressed in this document.

None of these courses is defined by the language in which it may be taught. This is not to say, however, that the choice of language is arbitrary or unimportant. On the contrary, language matters, and for this reason we have indicated likely choices in each case.

Reposition 15–110 The goal of 15–110 should be to provide a fundamental education in the principles of Computer Science for terminal students. This should include: (1) how to design programs that solve a computational problem; (2) basics of time and space complexity of algorithms and their role in designing programs; (3) the capabilities and limitations of computer arithmetic; (4) computing with aggregate data structures such as vectors, matrices, lists, or streams; (5) basics of data acquisition and analysis. Such a course would address the needs of a large majority of students across campus, and provide a foundation for learning popular languages and technologies used in various disciplines. For example, the proposed course addresses core ideas of database manipulation, of matrix and other numeric computations, and of scripting programs to build applications.

After analyzing a number of possible alternatives, it is recommended that the revised 15–110 be based on the Python programming language, and that it follow the *How to Design Programs* curriculum developed by Matthias Felleisen and his co-workers. There was some support for using the updated dialect of Scheme developed by Felleisen, but on balance of considerations it was felt that Python would be an acceptable compromise. It is important that a language for this course support convenient programming with high-level aggregate data structures, rather than their low-level representations, and that it be convenient for building applications that collect and analyze data available on the web.

Redefine 15–211 The current 15-211 focuses on using conventional imperative programming techniques to implement *sequential* algorithms on a variety of data structures. These methods are inherently sequential, stressing, for example, the iteration of primitive operations in loops, rather than aggregate operations on whole structures. Moreover, these methods stress *ephemeral*, rather than *persistent*, data structures—those for which operations irrevocably alter the representation of the

data structure in memory, rather than compute a new data structure based on one or more given data structures.

Modern approaches to algorithms treat parallel computation as the fundamental notion, with the sequential algorithms emerging as those with a low degree of parallelizability (ratio of overall work to critical path length). Moreover, for both reasons of parallelism and reasons of expressiveness, persistent data structures are increasingly important, but are neglected in current textbooks and curricula. The committee recommends that 15–211 be redesigned emphasize parallelism and persistence as fundamental algorithmic concepts, while continuing to cover basics such as asymptotic analysis, divide-and-conquer, dynamic programming, sorting, balanced trees, priority queues, dictionaries, and graphs. The course will continue to include programming projects that demonstrate the use of these algorithms in practical applications.

It is important to distinguish *parallel computation* from *concurrent computation*. Concurrency is concerned with the problem of coordinating multiple stateful computations by controlling how they interfere with each other. Parallelism, instead, is concerned with exposing the opportunities for truly simultaneous computation in a way that isolates their stateful effects so that no difficulties arise from their interaction. Parallelism raises no concerns about correctness, beyond those of the sequential case, but provides opportunities for much greater efficiency of execution.

To make this possible, it is essential to use a *functional programming model*, which stresses computation with data structures as aggregate values, rather than as objects represented in memory, and which supports interference-free simultaneous computation. For this reason it is essential that the revised 15–211 be taught using a language with functional programming capabilities, which would include Standard ML, Objective Caml, Haskell, or Scala. It must also support a clear cost model that allows asymptotic complexity to be assigned to programs written in that language; this appears to rule out Haskell as a choice for this course. Conventional imperative and, especially, object-oriented programming languages are not suitable for this purpose, because of their over-emphasis on mutation and sequentiality.

Replace 15–121, Refactor 15–212 It is recommended that the first-year introductory for continuing students consist of two courses, 15–150 Functional Computation and 15–122 Imperative Computation. These are intended to be entry-level courses[1] that provide a grounding in the two fundamental models of computation, the functional and the imperative, and prepare students for the 200-level core

[1]Though we expect that, as now, students with no prior programming experience would likely take 15–110 as preparation.

courses. These two courses are expected to be coherent in that they both are to place a strong emphasis on writing clean, correct code, they are complementary in that 15–150 will stress high-level abstraction and composition principles, whereas 15–122 will stress the realization of these abstractions in terms of lower-level mechanisms.

Course 15–150 is intended to teach functional programming, a style of programming that avoids (but does not rule out) mutation of data structures, and that emphasizes programming with data structures as aggregate values. There is a strong emphasis on recursive programming, and a corresponding emphasis on inductive methods for reasoning about such programs. Data structures are conceived abstractly in terms of the operations on them, rather than in terms of their realization in memory. Types play a central role in specifying programs and in delineating their modular structure. One may think of 15–150 as the first half- or two-thirds of the current 15–212. Appropriate languages for this course would be Standard ML, Objective Caml, Haskell, or Scala, according to what language is chosen for 15–211.

Course 15–122 is intended to teach imperative programming and methods for ensuring the correctness of programs. Higher-level abstractions are to be built up in terms of these representations. Students will learn the process and concepts needed to go from high-level descriptions of algorithms to correct imperative implementations, with specific application to basic data structures and algorithms. The emphasis is on iteration of basic operations such as assignment, and the most important reasoning methods are pre- and post-conditions and loop invariants. The course is explicitly intended to incorporate up-to-date methods and tools for mechanized analysis programs to prepare students for modern software development techniques. Standard languages such as C or C++ are not suitable for this course because their complexity and deficiencies impede both informal and mechanized reasoning techniques. An experimental safe subset of C is being developed as a vehicle for teaching this course, with a transition to full C at the end. The motivation for using a C subset is that it eases the transition to C itself and allows linking to existing C code, and, unlike C, has a semantics that is amenable to verification and analysis.

The proposed changes to 15–211 and the introduction of 15–150 imply significant changes to 15–212. Approximately two-thirds of the material in 15–212 would be moved into 15–150 and 15–211.

Programming Methodology The committee recommends that a new course on programming methodology be introduced that provides students with a grounding in techniques for building large-scale programs. This should include, but not be

limited to, object-oriented programming methods. Some material currently in 15–121 and 15–211 would be moved to this course. A proposal for this course, and its position in the curriculum, is under development.

Skills Courses The committee recommends that 15–123 be reduced to an elective that can be taken by students as preparation for advanced courses that require familiarity with Unix-based command-line tools.

More generally, the committee recommends that the need for skills electives be reviewed, and that new courses be introduced that fulfill such needs as may be identified. One candidate, in addition to 15–123, would be a course on MatLab, which is often requested by faculty across campus.

Any skills offerings should be clearly distinguished from the introductory and core curriculum as envisioned here.

Prerequisite Structure The following pre-requisite structure is proposed:

- 15–122 is required for 15–213.
- 15–150 and 15–122 are required for 15–211.
- either 15–122 or 15–150, and 21–127, are required for 15–251.

It is not yet clear how to position the proposed programming methodology course.

6 Concerns

Current resource constraints raise serious impediments for revision in both the near- and long term. Any changes would have to be phased in gradually, requiring double offerings for the next few years. Several tenure-track faculty have offered to help design and implement new course offerings, but this requires that their current teaching obligations be filled by some other means. Teaching track faculty would have to devote time to the transition to a new curriculum.

Currently IS students are required to take 15–110 and 15–121. Under the proposed revision it is no longer clear what is the appropriate second-level course for such students.

A roll-out plan for the proposed curriculum remains to be devised. Aside from the personnel issues alluded to above, there is significant turbulence to be expected as students shift from one regime to another. To some extent the changes can be made gradually, but dependencies among the courses would seem to require coordinated changes at multiple levels simultaneously.

The use of a non-standard language for 15–122 raises concerns, since it requires development of all new course materials and software, including a compiler for the language. The risk is mitigated by the fact that the proposed language is a safe C subset that compiles to C, and by the fact that Frank Pfenning is volunteering to develop this course and its infrastructure.

About one third of 15–212, including metaprogramming and concurrent programming, is not carried over into 15–150 and 15–211 under the proposed revisions. This represents a significant loss of content for which no obvious replacement is available under the proposed restructuring.