Low-tech computing: Developing a programming-free alternative to CS1

Evan Silberman
Hampshire College School of Cognitive Science
893 West Street
Amherst, MA USA
ejs07@hampshire.edu

ABSTRACT

The traditional introductory course in computer science makes extensive use of computer programming. Historical, philosophical, theoretical, and foundational aspects of computer science are often deferred to upper-level courses for majors. I worked to develop a curriculum for a first course in computer science that guides students through the intellectual foundations of the field from these lower-tech perspectives. I describe the syllabus I created and the topics I covered, discuss outcomes from the pilot class, propose avenues for further refinement of the concept, and suggest how this effort could be part of curricular reforms aimed at broadening participation in computer science.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and information science education—*computer science education, curriculum*

General Terms

Theory

Keywords

Computing education, CS1

1. INTRODUCTION

Learning to program is a cognitive challenge. Even learning to *use a computer* is a challenge—core paradigms of personal computer interfaces like the hierarchical filesystem are not as trivial to understand as the technically-enthusiastic community believes. It is quite possible that the conventional first course in computer science, generally a programming course focusing on object-oriented and procedural programming, severely limits the potential audience for advanced work in the field by skewing participation towards students who are already advanced or expert computer users.

SIGCSE 2012 Raleigh-Durham, North Carolina USA (declined)

The 2001 Computing Curricula report on computer science summarizes some of the oft-discussed drawbacks of the programming-first tradition:

> Programming courses often focus on syntax and the particular characteristics of a programming language, leading students to concentrate on these relatively unimportant details rather than the underlying algorithmic skills. This focus on details means that many students fail to comprehend the essential algorithmic model that transcends particular programming languages. Moreover, concentrating on the mechanistic details of programming constructs often leaves students to figure out the essential character of programming through an ad hoc process of trial and error. Such courses thus risk leaving students who are at the very beginning of their academic careers to flounder on their own with respect to the complex activity of programming. [5]

CS1 is where prospective computer science majors first encounter the discipline, and this course is taught in much the same way across a variety of institutions: an objects-first approach using Java dominates the landscape. [2] If computer science departments are to respond to declining enrollments and historical imbalances in participation by women and under-represented minorities, the programming-centric CS1 course needs to go under the microscope. Careful attention to the structure and curriculum of CS1 offerings have yielded promising results in the past. For instance, changes such as explicitly designating sections of CS1 as intended for students with prior experience, as a means of creating a less intimidating environment for true novices, were recently reported by Alvarado and Dodds to have increased participation by women in the computer science department at Harvey Mudd College. [1] More broadly, curriculum modernization efforts such as Stanford's, reported in [8], have yielded increased major declarations overall.

Within this context, I conceived, developed, and taught a programming-free introduction to computer science as part of my Division III thesis work at Hampshire. My other motivations for creating this course were multifarious: a survivor's dire assessment of high school math classes, an enthusiasm for interdisciplinary approaches to education, a belief that CS's contribution to a liberal education should go beyond teaching people to program, and pure curiosity about the viability of a programming-free approach.

1.1 Course description

I titled the course I offered "Low-tech computing" and advertised it as follows:

> Low-Tech Computing is an interdisciplinary first course in the history, theory, and philosophy of computer science. Our unconventional approach to the field will begin with exploring computing's origins in work done on logic in the early 20th century by Bertrand Russell, Wittgenstein, Gödel, and others. Next, we will use wires, LEDs, and very simple logic circuits so that we can see how logic and arithmetic are on some level interchangeable. We will return to our historical timeline and prove a foundational theorem of computer science, the undecidability of the Halting Problem, and learn about the life and work of Alan Turing and some of his contemporaries and successors. In the last part of the course, we will touch on a variety of computing topics, including computational complexity, the P vs. NP problem, artificial intelligence, evolutionary computing, and more. Readings will be drawn from primary, secondary, and popular literature, including the recent graphic novel *Logicomix*. Coursework will include a short paper, a multi-part project working with electronic circuits, a problem set, and preparation for and completion of a final oral exam. We will not be doing *any* programming; this course will not require any prior experience with computer programming or mathematics—or even computers in general, as long as you can check your email.

1.2 Students

Thanks to Hampshire's independent study policies, I was able to teach the course as a for-credit activity to four students in Spring 2011, supervised by my advisor. Three of the students were first-years in their second semester of college, one was a transfer student with roughly sophomore standing in his second semester at Hampshire. Two of the students were women; one of them had walked into a programming class the previous semester and exclaimed "Seriously? I'm the only woman in here?" Students' areas of interest included philosophy, women's and gender studies, and interactive media. At least one student had stopped taking math in high school after Algebra II. At least two of the first-year students needed to pass the course in order to fulfill a distribution requirement.

2. UNITS OF THE COURSE

2.1 Logical foundations

Many of the components of this course were selected and developed specifically to provide a counterpoint to the prevailing pedagogical modes in math and computer science. I wanted to provide a correction to the ahistoricity of the programming-first curriculum, and so the first few weeks of the course focused on the foundations of mathematics.

My selection of *Logicomix* [3] as a principal course reading ended up being critical to the shape of the first part of the course. Written by novelist Apostolos Doxiadis and Berkeley computer scientist Christos Papadimitriou, the comic, subtitled "an epic search for truth", is a narrative account of Bertrand Russell's work in mathematical and philosophical logic. The authors take some historical liberties, mainly to allow their protagonist to talk to some contemporaries he never met and to be present at a couple of famous talks, but the intellectual history presented in the book is accurate. The graphic novel introduced students in an informal way to concepts that would be presented formally in lectures or would crop up again later in the class, and we followed its historical narrative as we discussed the mathematical concepts below. Towards the end of the course, my students said that they liked that the intellectual angles that *Logicomix* shows off were relevant throughout the class.

A major theme of *Logicomix*, and of the first part of the course, was that seemingly intuitive mathematical notions can produce counterintuitive results that you might have to accept to move forward. The strange-but-consistent results that manifest themselves when we incorporate infinite objects or processes into our mathematics are an excellent example of this. The course's first taste of "real math"[1] was a presentation of Cantor's storied diagonal proof of the uncountability of the real numbers. The proof serves as an excellent example for a number of reasons: highly visual, deals with the most familiar mathematical objects (natural and real numbers; even the concept of sets can be left out of a sketch of the proof), yields a decidedly counterintuitive result (the existence of differently-sized infinities, and illustrates techniques (proof by contradiction, diagonalization) that will come up again in other contexts.

We next discussed Russell and Whitehead's *Principia Mathematica* and its goals. *Logicomix* portrays the creation of this work as an intense and frustrating labor of years, as the authors continually reassess their assumptions and strive to base their creation on irreducible axioms and irrefutable chains of logic. While the notation of *Principia Mathematica* is dense, rarefied, and idiosyncratic, something of its approach can be gleaned merely by flipping through the book. I encouraged my students to do so for part of a class session,[2] merely to gain a feel for the nature and scale of this attempt to construct an unassailable fortress of logic.

I gave my students a writing assignment at this point in the course, asking them to respond to the ideas in *Logicomix* and address an obvious practical question: "What is the value in a 300-page formal proof that one plus one equals two, if there's any value at all?" My students produced a range of responses, all of which demonstrated comprehension of the mathematical material in the novel. One student discussed the relationship between mathematical language and reality, another formulated a thesis about mathematical being a codification of our common-sense ideas about the world.

Logicomix ends, of course, with Gödel's incompleteness theorems dealing a major blow to Russell's program. We took a very abstract look at the incompleteness result, using the puzzle introduced by Smullyan in the first chapter of [10]. I set my students the task of finding a true but unprovable sentence of the small language Smullyan defined, and while it took a while for anybody to find the answer,

[1] As opposed to the sort of math first-year college students tend to have dealt with in high school, a critique of which is far beyond the scope of this article.

[2] Our quite small library somehow had space for two copies of its volumes.

we did then discuss how the structure of the Gödel sentence recalled Russell's paradox, and what the implications really were for projects like *Principia Mathematica*.

2.2 Algorithms and abstract machines

In *Logicomix*, the author-characters disagree about whether Russell's work in logic was a failure. Certainly Gödel's incompletness theorem proved that *Principia Mathematica*'s dream was a hopeless one. But as Papadimitriou's cartoon-proxy says, "You just can't go calling Russell's work in logic a 'failure'... the *Principia* is the basis of everything that followed!" [3] And what followed, of course, is the advent of the theory of computing, with Alan Turing its convenient hero.

Computer science students are often presented with the Turing machine model as an abstraction of the digital computers with which, by the time they take a course on the theory of computation, they have extensive experience. In order to present the Turing machine model without the benefit of an analogy to programming, we had to discuss what computation was, and why we might want to formalize it. I led a class discussion in which students did a surprisingly good job at surfacing the generally-accepted features of an algorithm, and they completed an activity writing instructions for paper-plane-folding in order to expose the perils of informality. I presented the Turing machine model in detail, based largely on Sipser's description in [9]. This may have been more technical than required for the ensuing discussion of computability, but the details are certainly helpful in understanding *why* computer scientists accept TMs as embodying the common-sense notion of an algorithm.

Before introducing the halting problem and proving its undecidability, we proved first that there *were* problems that could not be solved by any Turing machine, following [9]. The structure of this proof is practically identical to Cantor's diagonal proof, the first formal proof that students were exposed to in the course. Diagonalizing on TMs gives us some idea that there are probably problems we care about that we can't compute; surely at least one of the uncountable infinity of undecidable problems would be useful if we could decide it. The halting problem, of course, served as our example.

One of the benefits of approaching earlier material in concrete, intuitive, and accessible ways, while gradually working in bits and pieces of the relevant formalisms, is that by the time we arrived at the halting proof, I was able to explain the proof formally in a fairly standard way—my students nodded all the way through and asked smart questions. Of course, I also had to include something fun, and so our text for the halting problem was Geoff Pullum's incomparably enjoyable "Scooping the Loop Snooper" [7], which is the best proof of the halting problem in rhyming doggerel that I know of.[3]

Finally, I introduced the method of proving undecidability by reduction, and the idea that a problem being reducible to another means that one problem is "at least as hard" as another. Students completed a short problem set with several reducibility problems adapted from [9]. Performance on the assignment was fairly good—while nobody did a perfect job, every student grasped how to do the proofs and how to convincingly present them.

[3]It is the only such proof I know of, of course, but I don't see how it could be superseded.

2.3 Circuits and physical machines

One of the principal virtues of teaching people how to program is that when it works, they relish the sudden attainment of power over their unruly computers, and are able to interact with it in a way that provides immediate (and sometimes even useful) feedback. I wanted to include some of this interactivity and feedback cycle in the course despite my exclusion of programming. Other educators have been successful integrating guided, circuit-based lab activities into theoretically-based discrete math courses for majors. [6] So in the last major unit of the course, students were challenged to build (extremely) rudimentary computing devices out of digital logic circuits.[4]

I equipped each student with a breadboard, a 4011 quad-NAND CMOS integrated circuit, and the necessary jumper wires, resistors, transistors, switches, and LEDs to wire up logic gate demonstrations. Since NAND gates are, on their own, a complete basis for boolean logic, students were instructed to use their IC to wire up other basic logic gates: NOT, AND, OR, and XOR. They were guided through this by a packet asking them to use truth tables, math, and circuit diagrams to get a handle on how each gate behaved, and so that they could check that their physical circuits did what they expected.

My first computer science teacher memorably explained the paradigm of object-oriented programming by talking about "the A-E-I-O of OOP", for abstraction, encapsulation, inheritance, and overriding (which stood in for polymorphism for obvious poetic reasons). While the latter two cornerstones are fairly specific to OOP, abstraction and encapsulation are core concepts for programmers working with any high-level paradigm, and have resonances in the theoretical world as well. The project was intended to culminate with each student constructing a half-adder, so that between the four of them they could assemble a two-bit adder from their modules. Practical issues interfered and while we put the adder together I had to intervene in the process more than I had planned to. Even so, by building an adder out of half-adders, which are built out of AND and XOR gates, my students were exposed to the fruits of abstraction and encapsulation in a way that they might not have if they had learned boolean logic in the traditional paper-based fashion.

My students seemed to enjoy the circuit project, though they struggled with different parts of it. They all cited it as the most useful component of the unit, with my lecturing rating rather worse in a short survey. Later on, one student told me that the physicality of working with circuits helped

[4]When I taught the course, this section of the course actually went in between the previous two—i.e., after wrapping up *Logicomix* with Gödel's incompleteness theorem and before introducing the theory of computation with the idea of the algorithm. At some point, it struck me that the circuit material properly should come last. My students agreed with me when I polled them for some suggestions towards the end of the semester, so for strictly rhetorical purposes I am counterfactually presenting the logic circuits unit here. Presenting digital logic circuits—one might say "real computers"—after their abstract forebears better suits the historical narrative that shaped the class. This reorganization also suggests the possibility of an iteration of the course that condenses some of the earlier mathematical material in favor of spending more time on topics suggested by the work with logic circuits: von Neumann machines, computer architecture, the history of computer hardware, and the like.

her understand the abstract logical ideas, and that she had a moment where she really "got it". Another student, conversely, said that she found building the circuits to be more challenging than working with symbols and diagrams.

3. FINAL PRESENTATIONS

My four students were challenged to give ten-minute final presentations, to be followed by a question-and-answer period, at the conclusion of the course. I presented them with five fairly broad suggested topics to choose from, as well as the chance to present on a topic of their choice. Ultimately they each chose one of the topics I came up with, and gave their final presentations on logic, cryptography, randomness, and artificial intelligence.

While one student was noticeably underprepared, the presentations were all quite successful. Cautioned away from Wikipedia, students made use of both academic publications and credible online sources for background. Highlights of the presentations included a well-organized historical overview of cryptography, a confident explanation of Kolmogorov complexity, and an insightful observation that computers and brains aren't exactly equivalent because brains can't be programmed to solve arbitrary problems. All the students leveraged knowledge from the course to improve their understanding of presentation topics.

4. RELATIONSHIP TO CURRICULUM STANDARDS

The course as I taught it covered at least 18 hours of core topics from the CS2008 Body of Knowledge. [4] A department looking to stick to the standards could conceivably expand various course topics so that every hour of the course was a core hour from the curriculum, or at least one of the more universal elective topics (such as P vs. NP, which I introduced in one class session towards the end of the semester). The topics covered in Low-Tech Computing are not alien to core undergraduate computer science courses.

Body of Knowledge areas as described in [4] covered by my syllabus include:

DS/FunctionsAndRelations: 3+ core hours, covering sets, cardinality, countability, Russell's paradox, self-inclusion

DS/BasicLogic: 4+ core hours, covering logical connectives, truth tables, propositional logic (slightly), applications of logic in the manifestation of digital circuits

DS/ProofTechniques: 2+ core hours, covering the structure of mathematical proofs, proofs by contradiction, mathematical induction. Proofs by contradiction crop up repeatedly in the course and are the main proof technique discussed in relation to things like Cantor's theorem, Gödel's incompleteness theorems, and the halting problem.

AL/BasicComputability: 4+ core hours, covering the idea of the algorithm, the Turing Machine model, and the Halting Problem—the core of the core, in my view, and the basis of computing as a formal discipline.

AR/DigitalLogicandDataRepresentation: 4+ core hours. Logic gates, circuits, logic expressions, boolean functions, binary data, binary arithmetic.

IS/FundamentalIssues: 1 core hour. The one solitary core hour required by CS2008 in this knowledge area certainly will be repeated in an upper-level AI course, but it has a good home in the latter half of a course based on mine if time is left for topics. Certainly nobody should leave a historically-minded computer science course without reading "Computing machinery and intelligence", which we discussed in one of the final classes of the semester.

5. FUTURE DIRECTIONS

Students indicated in their feedback that they would have liked to be assigned more reading. Technical details that were presented in lecture were often unaccompanied by an appropriate reading assignment. Apart from [3], it was difficult to find book chapters or reference sources that covered a given day's topic with the right vocabulary and the appropriate level of detail for my students. The excerpts of [9] that I did distribute as reading material were probably too notationally dense. Many of the books and lecture notes I was able to get access to discussed topics from the perspective of a reader who was already acquainted and comfortable with programming. One book from my advisor's bookshelf even defined a computable function as "a function that can be implemented by a Pascal program". Other sources assumed more mathematical sophistication than my students would have. Future iterations of the course will require a greater breadth of readings, especially texts that discuss algorithms, Turing machines, the Church-Turing thesis, and the halting problem at a suitable level of detail. I also did not evaluate any of the various biographies or fictional histories of Alan Turing for their suitability as course readings; this genre could contain a gem worth including in the syllabus.

While the syllabus I used was substantially diverse, the range of topics covered could probably be expanded in order to present a more broadly representative cross-section of computer science. In particular, contemporary AI topics such as connectionism and evolutionary computation would be well-suited to the latter part of the course. The intersections between modern AI practice and brain science, evolutionary biology, and philosophy of mind are fun to explore and are important parts of computer science's interdisciplinary context.

The trap that computer science departments fall into is the creation of "fun" or "alternative" introductory courses that don't actually serve as gateways to the major. Many CS0 classes do not count towards major requirements, and even a program with an innovative and successful CS0 class may undo its own work by sticking to a CS1 syllabus that eschews the experimental or motivational content that fostered student success in the first place. In order to maintain any increase in interest that a curriculum starting with Low-tech Computing might generate, follow-up courses must lead into advanced work in CS without giving up on the experimental project. Specifically, CS majors must learn to program *eventually*, and thus programming courses that leverage and build on the theoretical background acquired in a course like Low-tech Computing should be developed.[5]

6. CONCLUSIONS

[5]Also the course needs a better name.

It is not possible to draw any empirically well-supported conclusions from an experimental course taught a single time to four first-year college students. In situ, I believe the effort was largely successful. My students had a range of backgrounds, academic interests, and mathematical comfort-levels, and they all indicated that they enjoyed the course in self-evaluations. Each student demonstrated a robust understanding of some sophisticated material through their final presentations. At the very least, my experience teaching Low-Tech Computing left me with no reason to doubt that a programming-later curriculum could succeed at drawing students into the field while also providing "one-and-done" students with a relevant, contextually-grounded introduction to some of the fundamentals of the field.

Persistent imbalances in the demographics of CS programs can only indicate that there are large populations of students who aren't well-served by, or at the very least aren't attracted to, the current status quo of CS education and the introductory sequence. Empirically-founded self-study and widespread institutional will to make changes will ultimately be required to correct these imbalances; locally, curricular experimentation can generate ideas for broader implementation.

7. ACKNOWLEDGEMENTS

Thanks to Lee Spector and Chris Perry for their conscientious and critical support, and to my students for their seriousness and enthusiasm.

This material is based upon work supported in part by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] C. Alvarado and Z. Dodds. Women in cs: an evaluation of three promising practices. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 57–61, New York, NY, USA, 2010. ACM.

[2] S. Davies, J. A. Polack-Wahl, and K. Anewalt. A snapshot of current practices in teaching the introductory programming sequence. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 625–630, New York, NY, USA, 2011. ACM.

[3] A. Doxiadis and C. H. Papadimitriou. *Logicomix: An epic search for truth*. Bloomsbury USA, New York, 2009.

[4] Interim Review Task Force. Computer science curriculum 2008: An interim revision of cs 2001, 2008.

[5] Joint Task Force on Computing Curricula. Computing curricula 2001: Computer science, 2001.

[6] E. Patitsas, K. Voll, M. Crowley, and S. Wolfman. Circuits and logic in the lab: toward a coherent picture of computation. In *Proceedings of the 15th Western Canadian Conference on Computing Education*, WCCCE '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM.

[7] G. K. Pullum. Scooping the loop snooper. `http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html`, 2008.

[8] M. Sahami, A. Aiken, and J. Zelenski. Expanding the frontiers of computer science: designing a curriculum to reflect a diverse field. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 47–51, New York, NY, USA, 2010. ACM.

[9] M. Sipser. *Introduction to the theory of computation*. Course Technology, 2nd edition, 2006.

[10] R. M. Smullyan. *Gödel's Incompleteness Theorems*. Number 19 in Oxford Logic Guides. Oxford University Press, New York and Oxford, 1992.