Low-Tech Computing

developing and teaching a programming-free introduction to computer science

Evan Silberman

Division III Committee
Lee Spector, chair
Chris Perry

❧

May 2011
Hampshire College
Amherst, Massachusetts

Contents

Abstract

The traditional introductory course in computer science makes extensive use of computer programming. Historical, philosophical, theoretical, and foundational aspects of computer science are deferred to upper-level courses for majors or discussed in passing as end-of-semester respites from "real work" in technical courses. I worked to develop a curriculum for a first course in compute science that inverts these biases, and guides students through the intellectual foundations of computer science from these lower-tech perspectives. In this essay, I describe the syllabus I created and the topics I covered, discuss outcomes, and suggest avenues for implementation and further development of innovative curricula in computer science.

Chapter 1

Reworking the curriculum

Computer science is a field that has never been entirely sure of its role within the academy. Especially at institutions without a parallel computer engineering track, CS departments are obliged to serve two distinct groups of students: those seeking pre-professional training in computer programming, and those interested in the academic practice of computer science. This plays out against a background of a steady decline in enrollments in computer science programs (frequently cast as a "crisis" by commentators) and has led to significant amounts of hand-wringing about the availability of computing professionals and the strategic position of the US.

As a student at a small liberal arts college, I have given relatively little thought to my future employability, and I don't particularly think that the strategic position of the US tech industry should be the primary concern of computer science educators. But I do think that working to increase computer science enrollments is important, and I think achieving this goal will require rethinking the curriculum of introductory courses in computer science.

The vast majority of introductory-level computer science classes follow a "programming-first" model, in which students are taught the basics of structured procedural and object-oriented programming using a single programming language or environment. [2] The most recent Join Task Force on Computing Curricula report on computer science acknowledges several persistent issues with the programming-first tradition, and with programming courses generally:

> 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 under-

> lying 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. [2]

Programming a computer is a complex and frequently frustrating task, requiring a high amount of mastery of the computing environment. Programming novices are hard-pressed to develop a capacity for algorithmic thinking when they are constantly grappling with a syntactic minefield. The programming-first model may not be fundamentally flawed, but if it is a local maximum in the space of introductory computer science curricula, a new approach to the first-year program is going to have to take what might seem like a radically different approach.

My core idea is a simple one: I propose to introduce students to computer science without using computers and without teaching programming. In this essay, I will describe a first course in computer science, suitable for students with no prior computer science background and no specific mathematics prerequisites, that provides a theoretical, mathematical, and philosophical introduction to the foundations of computer science, guided by the historical narrative of the development of computing.

1.1 Low-Tech Computing: A course overview

The course I outline and describe herein was initially conceived in Spring 2010, developed in Fall 2010, and taught in Spring 2011 under faculty supervision. This is not a strictly documentary record of the course as it was taught: it incorporates changes and ideas developed and recommended after the fact alongside course materials, activities, and assignments as they were originally used.

I had put together my first reasonable-sounding course description that summer, and for a while it was my mission statement as I was working on coming up with course topics:

> Computer science is a discipline founded on a singular and counterintuitive hypothesis—that any device or system designed to per-

> form calculations is no more powerful than a specific and very simple abstraction of computation. The details, implications, and contradictions of this hypothesis will concern us in this course. We will quickly build up a model of algorithmic computation, the Turing Machine, and explore its plausability as a universal model. We will then discuss the core results of computability theory, including Turing's solution to the halting problem. We will move on to further topics in foundational computer science, including time and space complexity, the P vs. NP problem, randomness and information, the philosophy of artificial intelligence, and other topics as time and interest allow. We will use the methodology of mathematical proof throughout the course to support our assertions, but there are no specific mathematical prerequisites.

The course was developed and redeveloped extensively throughout the fall of 2010. My first outlines of course content were heavy on mathematics and light on narrative. I generated a big list of topics and skills I wanted to teach: logic, set theory, computability, complexity. All of those things were in the final syllabus in one form or another, but early on, I was somehow planning on teaching a fairly standard lecture course on theoretical computer science, with a fairly demanding amount of math involved.

Apart from a meeting with my advisor where he pointed out that a big list of traditional topics didn't amount to an innovative syllabus, a couple of things made me rethink my direction and put some energy into figuring out how I was going to make my course accessible. I had come up with the idea of handing out a "math diagnostic" early on in the class, so that I'd be able to gauge how acquainted my students were with math and present material accordingly. The version I came up with looked something like this:

> For each of the mathematical and logical statements below, translate the statement into English words to the best of your ability.
>
> 1. $f(x) = x^2$
> 2. $A = \{1, 2, 3\}$
> 3. $A = \{x \mid x \text{ is even}\}$
> 4. $A \subseteq B$
> 5. $A \subseteq \mathbb{N}$

6. $A = \{x \mid x = 2y, y \in \mathbb{N}\}$
7. $B = \{A \mid A \notin A\}$
8. $x \in A \leftrightarrow \exists y(y \in \mathbb{N} \wedge x = 2y)$

A couple of people who I showed this to pointed out that it was terrifying. "If you give this to people on the first day of class," I was told, "they are going to get scared and run away". I had another conversation with a friend of mine around this time. She is an exceptionally bright science student, and is currently aiming for a career in neurobiology and neuropsychology. Her pre-college education was entirely at home, adhering to the venerable hippie pedagogy of "unschooling". Until she started college at 16, she never sat down and formally did coursework—except in mathematics. Math she ground through in the conventional way, and inevitably developed a distaste for it, as do most students forced to grind through high school mathematics. She also suggested to me that if the audience for my class was going to include erstwhile math-haters, that I should be careful not to front-load them with the idea that they will be doing math. Their observations saddened me, but I decided they were right. I wanted to attract students who didn't yet know that there is math to like in the world, and I didn't want to burden my course with the conventional trappings of math class, in the form of extensive homework assignments or chapter tests or what have you. So I resolved to be a bit more subtle about it: I would get students acquainted with the language and spirit of mathematics without necessarily obliging them to do any math right off the bat.

Fast-forward to January, and I had a syllabus. The final course description for Low-Tech Computing read 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.

The bulk of the syllabus was divided into three major units, covering about three weeks of class time each:

1. The Foundations Crisis: an introduction to mathematics using Bertrand Russell's work on *Principia Mathematica* as a frame tale
2. Algorithms and Computation: defining abstract computers and investigating their capabilities
3. Down and Dirty with Digital Logic: learning about logic by building electronic circuits[1]

Thanks to Hampshire College's flexible academic program, I was able to conduct the course as a student-led independent study activity for four students, for which they received credit equivalent to a normal faculty-taught course. Three of my students were first-years, and one was a transfer student in roughly his second year of college.

1.2 Adopting and adapting the curriculum

Curricula for computer science programs of study at colleges and universities most often use as a starting point the CC2001 computer science recomendations ([2]) and their interim revision, CS2008 ([4]), produced by a joint task force of the IEEE and the ACM. While I have never been privy to curriculum planning for a computer science department, I imagine it involves a lot of horse-trading, arguments about "service courses", and hand-wringing about how important the

[1] As I will discuss in section 4, this unit and the previous one were taught in reverse order, but the order presented here is clearly better, so that's how I'm presenting it.

standards are. As such, as an aid to possible adoption, I outline in this section the areas in the CS2008 Body of Knowledge for which the Low-Tech Computing syllabus covered at least one core hour.

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. Completely skips past finite state machines and CFGs, of course, but those almost always appear in advanced-undergraduate theory of computation courses. To my mind, the halting problem is the core of the core of the core.

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 here in the latter half of a course if time is left for topics. Certainly nobody should leave a compsci course that focuses on the fuzzy stuff without reading "Computing machinery and intelligence".

The course as I taught it, with lots of weird logistical issues and misfires attributable to my inexperience at teaching and random fluctuations of attendance, had at least 21 hours of core topics from the CS2008 Body of Knowledge. 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 ones that are alien to core undergraduate computer science courses. But bringing them together and teaching them as an introductory course is definitely unusual, if not unique, and hopefully this sketchy breakdown encourages other educators to consider drawing from my experience when designing future courses.

Chapter 2

The foundations of mathematics

If we were to ask an imaginary college freshman to sum up their high school math experience in one word, we should not be surprised to hear them say "traumatic". College students are high school math survivors. From algebra to calculus by way of conic sections, trigonometry, and polar coordinates, students are obliged to spend five years memorizing an endless assortment of arithmetical methods just long enough to pass the test. The value of such experiences is highly questionable, and certainly has little to nothing to do with mathematics as it is practiced by mathematicians. Even worse, alternative and experimental approaches to learning math tend to be found only in classes aimed explicitly at the mathematically disinclined; high-achieving students sensitive to college admissions committees feel compelled to stick with the AP Calculus class.

Secondary math education in the US is uselessly burdened by the legacy of the Cold War. Sputnik went up, and the call went out to educate a generation capable of calculating missile trajectories - a task already on its way to being performed by programmable computers. Single-variable calculus, the capstone of the conventional high school mathematics sequence, is taught as a decontextualized and tedious series of calculations. Techniques for integrating and such must be retained by students exactly as long as it takes to do a decent job on a test.[1]

Donald Knuth once wrote, "Science is what we have taught computers to do. Everything else is Art." A liberal arts approach to the foundations of computer science, which is what I sought to create, should not sacrifice mathematical rigor.

[1] Paul Lockhart thoroughly and convincingly indicts the K–12 math curriculum in *A Mathematician's Lament*, and I would refer the doubtful reader to that book rather than belabor the point from my less advantaged perspective.

I wanted my students to be exposed to the language of mathematical notation and the method of mathematical proof. Proofs are central to the mathematical endeavors of theoretical computer science, so I sought to teach my students the role that proofs play in the construction of the body of mathematics.

Mathematics is a discipline with a rich and dramatic intellectual history. A mathematics course for novices (and most high school graduates are relative novices in the areas that inform contemporary mathematics) ought to explicitly situate its subjects into that intellectual history. Low-Tech Computing's first class session introduced the conventional modern picture of mathematics as understood by working mathematicians. I asked students to read the first section of [1] and we discussed the components of this picture, especially the centrality of proof to the body of mathematical knowledge. One of the underlying questions of the first unit is "what makes mathematicians certain about mathematical facts?", and our starting point is the idea that proofs are convincing logical arguments that a mathematical statement is true.

2.1 Reasoning about infinity: Cantor's diagonal proof

I gave my students their first taste of "real math" by presenting Cantor's storied diagonal proof. A major theme 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. As a warm-up to the diagonal proof, we discussed Zeno's Paradox:

> Imagine Achilles chasing a tortoise, and suppose that Achilles is running at 1 m/s, that the tortoise is crawling at 0.1 m/s and that the tortoise starts out 0.9 m ahead of Achilles. On the face of it Achilles should catch the tortoise after 1s, at a distance of 1m from where he starts (and so 0.1m from where the Tortoise starts). We could break Achilles' motion up as we did Atalanta's, into halves, or we could do it as follows: before Achilles can catch the tortoise he must reach the point where the tortoise started. But in the time he takes to do this the tortoise crawls a little further forward. So next Achilles must reach this new point. But in the time it takes Achilles to achieve

> this the tortoise crawls forward a tiny bit further. And so on to infinity: every time that Achilles reaches the place where the tortoise was, the tortoise has had enough time to get a little bit further, and so Achilles has another run to make, and so Achilles has in infinite number of finite catch-ups to do before he can catch the tortoise, and so, Zeno concludes, he never catches the tortoise. [5]

The rapid mathematical reeducation continues with a first proof: Cantor's famous diagonal proof that the reals are uncountable. The diagonal proof of Cantor's theorem is a great first proof to sketch for computer science students:

- The proof is very visual. While not a picture-proof by any means, the diagonal argument can be easily and succinctly diagrammed.
- The elements involved are familiar. Everyone is acquainted with the natural numbers, certainly, and the real number line should hardly be alien. Sets may be a new concept, but the whole relationship of the proof to sets and set theory can be elided in service of communicating the core idea.
- The result is counterintuitive. The average individual's common-sense notion of "infinity" is likely either that infinities are incomparable or that infinities are all identical; Cantor's result gets us used to the idea that our common-sense notions about mathematical objects are often wrong.
- The proof techniques will come up again. Proofs by contradiction in general and diagonalization proofs in particular recur later in the course material: Gödel's incompleteness theorem and the undecidability of the halting problem both make use of analogous arguments.
- Infinity is sexy.

My resolve to acclimatize my students gently to math weakened a bit, and after the class session on diagonalization I gave out this short assignment, to be handed in at the next class:

> We might have the intuition that there are not as many even natural numbers as there are natural numbers. After all, for every even number, aren't there two natural numbers? Prove that this intuition is wrong, and that there are, in fact, as many even numbers as there are natural numbers.

Two of my students came up with the right answer (showing that there's a one-to-one correspondence between the natural numbers and the even numbers), and the other two were headed in the right direction.

This is a good spot for a more ambitious instructor to add a bit of mathematical material to the course. Given an extra class session or two, the course could introduce sets, functions, and relations in a bit of detail, rather than in an abbreviated and imprecise manner while presenting the diagonal proof like I did. You'd get a couple more core hours from DS/FunctionsAndRelations into the course, which might be necessary for a teacher trying to partially adapt Discrete Structures I (see [2]) to follow the narrative of LTC.[2]

2.2 Onward to Logicomix

The story of the foundations crisis is a compelling one, replete with interesting characters. For proof (as it were) we can look at the graphic novel *Logicomix*.[3] Written by novelist Apostolos Doxiadis and Berkeley computer scientist Christos Papadimitriou, the comic, subtitled "an epic search for truth", follows Bertrand Russell's on-again off-again affair with mathematical and philosophical logic. The authors take some historical liberties, mainly to allow their protagonist to meet and talk to his contemporaries in the community of logicians and to be present at a couple of famous talks, but the intellectual history presented in the book is accurate.

Using *Logicomix* is certainly not an obligatory component of the course. One could easily assemble readings from Russell's writing and writings about Russell and his contemporaries that would be more formal, more accurate, and contain more mathematical and logical content. But approachability is an important goal for the introductory or pre-introductory CS course, and for this course in particular, where I was not assuming any particular mathematical background.

My selection of Logicomix as a principal course reading ended up being critical to the shape of the course.[3] 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. Towards the end of the course, my students said

[2] I feel like what I'm saying here is "with just a bit of tweaking, you can take what I did and make it boring again!"

[3] *Logicomix* was really *the* primary course reading when I taught the class, because I struggled so much to find appropriate reading material, but I will comment more on that later.

that they liked that the intellectual angles that *Logicomix* shows off were relevant throughout the class.

2.3 Russell's paradox and the trickiness of self-reference

Self-reference, in forms both formal and poetic, is an eternal wellspring of delight to math and computer types. I rather wish I had put together a class session during the first part of the course focusing on playful and artistic aspects of self-reference, such as Quines (both natural, like "yields truth when appended to its quotation", and programmatic), *The Treachery of Images*, *If on a winter's night a traveller*, and others.

Russell's paradox is introduced in *Logicomix* as a big second-act twist. Its ultimate resolution (the rejection of naïve set theory) is beyond the scope of the novel, but at the time, it caused a tumult in the mathematical world. While students had not been introduced to a formal logical calculus, it was still possible to fill in the details of why the contradiction introduced into set theory by Russell's Paradox was of such great concern. In formal logic, any conclusion follows if you take a contradiction as a premise. If we assume that 1=2, or that the set of all sets that do not contain themselves somehow exists, we can technically prove whatever we like. This is, of course, an untenable state of affairs for a mathematical system that is supposed to be consistent: Russell and his logicist and formalist contemporaries identified *truth* with *proof*. True things were things you could prove, and if you could prove it, it was true. So if a contradiction existed in the theory of sets, then how could it be true? Russell tried to save it with the ill-fated theory of types.

One of my regrets about my first iteration of this course was that I was basically learning tiny bits of the philosophy of math in an unsystematic way as I went along, trying to bring in some "more serious" material to my students while we went through *Logicomix*. The take-away from the Foundations Crisis, as I see it, is that the platonic, formalist/logicist, and intuitionist pictures of math are along the lines of the parable of the blind mice and the elephant.[4]

[4] This is, unquestionably, the best parable.

2.4 1 + 1 = 300 pages

The syntax and argument of *Principia Mathematica* is so arcane that asking students to read it would be an invitation to a baffled and disheartened room full of people. But there is no better illustration of *Principia Mathematica* than *Principia Mathematica* itself, and I see no problem with simply baffling students for educational purposes. Every institution is likely to have a copy in the library.[5] I brought a couple copies of Volume One to a class session, and instructed students to simply peruse it.

The exercise wasn't staggeringly successful—it was one of several instances where I was less prepared to lead a discussion than I should have been—but I think it was still instructive.

The "big assignment" for the first unit of the course was a short essay responding to *Logicomix*:

> How important is logic to mathematics and to philosophical argument? What are the circumstances when a formal logical argument is required to convince us that something is true? To put it concretely, what is the value in a 300-page formal proof that one plus one equals two, if there's any value at all?

What is its value, after all? And what is its value for us as learners of mathematics? Certainly none of us needs 300-plus pages of theoretical apparatus to be convinced that 1 + 1 = 2. We've been quite certain of that since a very early age. And outside of the realm of pure logic, contemporary proofs are quite alien from the formalist/logicist mode of *Principia Mathematica*.

For Russell and Whitehead, laboring in service to Hilbert's program, the identification of proof with truth, of syntax with semantics, required that proof be not an argument, but a *demonstration*. The system of *Principia Mathematica* really was supposed to be a "machine to manufacture tautologies"—to prove that the whole of mathematics was true by necessity. If they wanted to create a system that could be used to prove everything, they had better make sure they can prove something as simple as integer addition.

[5] If Hampshire College's tiny library has a copy, I promise you can get your hands one one.

2.5 Things fall apart: Gödel's incompleteness theorem

Logicomix concludes, of course, with Kurt Gödel dashing Russell's dreams of a provably consistent and complete mathematics to pieces. Sketching Gödel's actual proof is beyond the scope of the course (I've certainly never read it), but it's certainly possible to present a sketch of the idea of the proof in one class session. The important feature of Gödel's incompletness theorem for our purposes is that, if we can twist our formal system into making statements about itself, we can construct a statement that, in effect, asserts its own non-provability. If the sentence is true, we can't prove it inside the system, and if it's provable in the system, then it's not true. Since we don't want our mathematics to be able to prove anything false, we are forced to admit that there are true sentences of our mathematics that we cannot prove.

In the first few pages of [9], Raymond Smullyan poses a toy version of the problem: given just a few symbols, can we write a "true" sentence that a certain toy machine can never print? The answer, lexically, is of the form "'has no proof when appended to its quotation' has no proof when appended to its quotation". The paradoxes of self-reference and the insidiousness of diagonalization have bitten us again.

The take-away: math is a bit more mysterious than "calculemus"; we can design systems as carefully as we like but they may have unanticipated behavior; formal proof is a method of mathematics, not necessarily a method for uncovering absolute truth. Unless it is! Hard to take a clear stance here. Also, sneaking in some math in the process.

Chapter 3

No-tech computing: 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]

It was time to move on to the world of algorithms. In order to present the Turing machine model without leading up to it from the world of finite automata, and without the benefit of an analogy to students' programming experience, I had to make the case that a formal abstract model for a computing procedure was plausible and necessary to begin with.

The need for such a model arose, of course, because of Hilbert's posing of the Decision Problem. Hilbert hoped to find a procedure that could automatically decide statements of number theory. Without a systematic way of characterizing algorithms, mathematicians were free to try whatever they felt like trying, but to prove that there was *no* algorithm would require formalization and a different proof technique. [8]

The features of the formal TM system are more easily understood once the informal notion of an algorithm is understood. In the first pages of *Theory of Recursive Functions and Effective Computability*, a book by Hartley Rogers, Jr. that is afforded the status of "classic" by people more in a position to make such determinations than I am, the features of this informal notion are elucidated:

1. An algorithm is given as a set of instructions of finite size.

2. There is a computing agent, *usually human,* which can react to the instructions and carry out the computations.
3. There are facilities for making, storing, and retrieving steps in a computation.
4. [T]he computation is carried out in a discrete stepwise fashion, without use of continuous methods or analogue devices.
5. [C]omputation is carried forward deterministically, without resort to random methods or devices, e.g., dice.

> Virtually all mathematicians would agree that features 1 to 5, although inexactly stated, are inherent in the idea of algorithm....A straightforward approach to giving a *formal* counterpart to the idea of algorithm is, first, to specify the symbolic expressions that are to be accepted as sets of instructions, as inputs, and as outputs, and, second, to specify, in a uniform way, how any instructions and input determine the subsequent computation and how the output of that computation is to be identified. [7]

By understanding the constraints imposed on the sorts of processes that we call algorithms, we are able to make sense of models for those algorithms—one model in particular, actually.

3.1 Turing machines

My presentation of the Turing machine model was based largely on Sipser's in [8], though his notational conventions and reliance on the concept of "languages", which I kept having to awkwardly gloss as "problems" or "sets of strings", may not have been completely appropriate for my audience. While I restricted my presentation to single-tape decision machines, I alluded to the fact that there are many changes one can make to a Turing machine that don't change its computing power. I gave the example of a two-tape Turing machine, and quickly sketched how it could be simulated on a one-tape machine, and thus could compute exactly the same things as the one tape machine.

3.2 The halting problem: Scooping the loop snooper

The proof of the undecidability of the halting problem is the punchline to two narratives that began in the first couple weeks of the course: diagonalization and self-reference. By first running through the proof, closely analogous to Cantor's theorem, that undecidable problems exist at all (though I was caught with my pants down by Lee on one question about this proof), we reintroduce diagonalization. Our demonstration that there are not just undecidable problems, but there are even undecidable problems that we care about, like halting, is also subtly diagonal, and more overtly exploits the paradoxes of self-reference: Russell's "set of sets which do not contain themselves" is analogous to the machine in the halting proof which accepts only those machines that don't accept themselves. Russell's paradox is linguistically intelligible; in the halting proof, we have a formal construction that illustrates a problem of self-reference.

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. While I passed out a short excerpt of the readable, yet notationally-dense, Sipser book when covering the mechanics of Turing machines, on the day I presented the halting proof, our text was Geoff Pullum's incomparably enjoyable "Scooping the Loop Snooper", which is the best proof of the halting problem in rhyming doggerel that I know of.[1]

The poem posits, for the sake of contradiction, the existence of a procedure (a TM, in our construction) P, which, given a TM description, determines whether it "defines a routine that eventually halts" on a given input. Our gloss of this in terms of the single-tape TM model is: if we give P a description of another TM M and an arbitrary string w as input, P will *accpet* if running M with w as input would eventually halt, and will *reject* if running M with w as input would loop infinitely.

Of course, since the punchline to the halting story is that the halting problem can't be decided, we have to show that P can't exist, and we do that by showing that, by assuming P's existence, we have created a contradiction.

> For a specified program, say A, one supplies,
> the first step of this program called Q I devise
> is to find out from P what's the right thing to say

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

of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,
and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q? [6]

The answer, of course, is that Q's behavior will always contradict P's prediction. If P predicts that $Q(Q)$ will loop forever, then $Q(Q)$ will halt, and if P predicts that $Q(Q)$ will halt, then $Q(Q)$ cheerfully runs forever. P, which was supposed to predict the looping behavior for every TM, has been shown to be an impossibility:

By reductio, there cannot possibly be
a procedure that acts like the mythical P. [6]

The first question that students are likely to ask after understanding the halting proof is whether this is the only significant undecidable problem. While we can of course answer "no", with reference to the result of Rice's theorem, we can also prove a somewhat less dramatic result that still gives us an idea of how many undecidable problems there are. Using an argument very much like Cantor's diagonal proof, we can show that there are countably many Turing machines and uncountably many decision problems. *Even if* this were the only class of problem we cared about computing, there are still *uncountably many* problems we can't compute with a TM, since we only have countably many Turing machines. [8]

3.3 Other undecidable problems: Proofs by reduction

The unit on the theory of computation concluded with a problem set. I spent a class demonstrating the method of proofs by reduction, and the idea that a problem being reducible to another means that one problem is "at least as hard" as

another. Since we proved independently that the halting problem is undecidable, we can prove that other problems are undecidable by showing that they are at least as hard as the halting problem. I put together a short problem set, with problems adapted from chapter 5 of [8], which my students did pretty well on—they grasped how to do the proofs and how to convincingly present them.

Chapter 4

Delving into logic

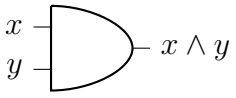
After all this time talking about logic, hearing about the exploits of the logically inclined, and employing logical arguments in proofs, it's time to get our students to actually engage with logic at a low level. In the traditional second-year discrete math course, the unit on logic covers things like truth tables, predicate calculus, derivation rules, and quantifiers. This is clearly good and important material for beginners, and an iteration of this course that skewed a bit harder than mine did could probably get away with doing some of this logic material more traditionally and pick up a couple more CC core hours in the process.

I opted to abandon the abstract and intellectually pure approach to boolean logic in favor of getting down and dirty with jumpers and breadboards. This may have introduced some complexity that wouldn't have cropped up if I had confined the material to truth tables and formulas, but it also added a physical and experimental dimension to the class that wouldn't have existed otherwise. One of the principle 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.

When I taught the course, I covered the material I discuss in this section in between that described in 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 I messed up, and that this unit should come after the undecidability problem set. 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.[1]

The circuit project was essentially a "guided lab": students were equipped 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 what each gate did, such as this excerpt for AND:

> **English** The AND operator outputs truth *if and only if* all of its inputs are true. Equivalently, the AND operator outputs falsity *if and only if* at least one of its inputs is false.
>
> **Symbols** $x \wedge y$
>
> **Truth table** Fill in this truth table:
>
> | x | y | $x \wedge y$ |
> |---|---|---|
> | 0 | 0 | |
> | 0 | 1 | |
> | 1 | 0 | |
> | 1 | 1 | |
>
> **Circuit symbol** AND gates should look familiar:
>
> 
>
>
> 1. Draw a boolean circuit for AND using NOT and NAND gates.
> 2. Redraw that circuit using only NAND gates. You can do this with just two gates.

[1] I would love for a historian to teach a version of this class, or for a computer scientists to co-teach with a historian. Or a philosopher. Or a philosopher and a historian. Or a theorist and a systems expert.

3. Wire up your circuit so that DIP switches 1 and 2 are the inputs to an AND gate and the output controls the LED. Check that your output is correct for all possible inputs. If you're having trouble, double-check that your jumpers between 4011 output and input pins are placed correctly.

The interdisciplinary scientist will relish the opportunity to teach a speedy recap (or perhaps belated introduction) to basic DC electronics. Knowing Ohm's law or Kirchoff's rules may not be strictly essential for completing the adder project, but on the other hand, a computer science major who doesn't know what Ohm's law is seems about as impoverished to me as a physics major who doesn't know what Turing machines are. (Meanwhile the mathematicians likely shake their heads in amusement at both the physicists and the computer scientists for going so thoroughly astray.)

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. 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 fashion.

4.1 Booleans to binary: encoding numbers into logic

The identification between boolean logic and binary arithmetic is a critical concept for understanding how digital computers operate and are built. In the conclusion of the circuit project, students join forces and use their electronic bits and pieces to build a very basic computer—one that can add two 2-bit numbers together. By building a simple adder from purely logical components, the plausibility of building math on top of logic may finally start to seem plausible: Russell devoted endless pages to proving that $1 + 1 = 2$, and we used a great deal of wire. We knew the answer already, but we may now be convinced that are models are worth something.

In the project's big finale, each student is equipped with two quad-NAND chips, so that they can wire up the AND and XOR gates that make up a half-adder. They are instructed to set everything up so that they have long jumpers for their two inputs and one output emerging in an obvious way from the breadboard. Then, with a little guidance and some extra OR gates, these half-adders can be assembled into a full-adder, then wired appropriately into some switches and LEDs so that we can prove not only that $1 + 1 = 2$, but also that $2 + 1 = 3$ and even $3 + 2 = 5$![2]

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 logic unit, with my lecturing rating rather worse in a short survey. Later on, I had one student tell me that the physicality of working with circuits helped 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.

I think the project would have benefitted from some more intense hardware prototyping. While obviously "easy" compared to soldering and PCBs, building circuits on breadboards required students to get acquainted fairly quickly with an environment that they had no particular experience in in order to solve problems not directly related to simply getting the damn thing to work—which is a lot like the criticism I have of how programming-first classes work. A slightly more toy-like environment, with some of the circuit components hidden inside a cheerful plastic box, inputs and outputs visualized more easily, and fiddly jumper wires replaced with something closer to A/V patch cables might have greased the wheels without sacrificing too much "real"-ness. The "magic boxes" used in an introductory-level course on computer organization at the University of British Columbia use this idea: students build logic on a breadboard, but things like input switches and output lights are built into a separate PCB with well-marked headers.

[2] For the record, things didn't go this way when I ran the class. I intended to have everyone build a half-adder and then combine them, Voltron-like, into the toy computer, but I was led astray by the giant breadboards I found in Lemelson and a moderate lack of planning.

Chapter 5

Further topics

I don't think there's ever been a computer science course that didn't have time built into the syllabus for "topics", which seems to be professorial secret code for "things that I personally find enjoyable to teach". In the case of Low-Tech Computing, the few classes of topics-time that we had when the semester drew to a close were devoted to a couple of things that I would have felt remiss not including in the course, that were interesting and important, and would serve as a bit of a preview of the sorts of things that students might expect to find in future computer science coursework.

5.1 Time complexity and P vs. NP

It hurt a little bit when I had to consign complexity theory and the P=NP problem to the "topics" portion of the course, rather than spending a whole two weeks or so on it. So, if you have 50 or so minutes to talk about time complexity, what do you talk about?

In keeping with my "don't be too afraid of the details" ethos, I wanted students to have a real introduction to the distinctions between problems in P and problems that were NP-complete, and not just say "some problems are slow, does it have to be that way?" in a hand-wavy fashion. I talked about the idea of the runtime of a Turing machine algorithm, and how we talk about runtime in terms of a function relating the size of the input to the number of steps that the algorithm takes. Defining polynomial runtime doesn't require math that scary. What's important next is to remind students of the dramatic difference between functions that grow polynomially and those that grow exponentially. Discussion of nonde-

terminism and computation trees can follow, but of course the most important step is for students to get an idea of what the "P vs. NP" question is asking.

5.2 Artificial intelligence

I felt that I would be completely irresponsible if I conducted a class about the foundations of computer science and did not ask students to read "Computing machinery and intelligence", even if they'll probably have to read it again someday. Of all the topics that the class covered, AI is the only one where it is viable to hand students the paper that founded the discipline, written by its instigating genius, and expect them to understand it and discuss it and engage with it. I was sick the day we discussed the article in class, and barely could string sentences together. Luckily for me, my students didn't need much of my help. There are countless angles to take on the Turing test and the idea of AI in general; the teacher's role with AI novices should not be to remonstrate vigorously with the dualists or spend half an hour ripping apart the Chinese Room argument.[1]

5.3 Outcomes and the future

When I asked my four students towards the end of the semester if they thought they would take another computer science course, answers ranged from "definitely" to "maybe" to "definitely not". Hardly scientific, but it wasn't a unanimous "no", so I probably didn't completely screw it up.[2]

So where could these students go from here? There may not be some perfect next-generation curriculum of my own mad devising waiting for them, but they do probably have some options. With a little bit of gumption, a student who did well in this course should be prepared to tackle an advanced undergraduate course in the theory of computation, such as one taught using [8], as long as they are prepared to have a much more challenging encounter with mathematical notation and writing proofs. With slightly less gumption, they could take a first course in algorithms. Students who liked the math and the philosophy better

[1] I did assign Searle's "Minds, brains, and programs" somewhat offhandedly as the "obvious" additional reading for this class session, but it wasn't really necessary, and the paper (while classic) is difficult (one might say poorly written), and students didn't get that far with it. So they can leave that alone until they take philosophy of mind or a proper AI course.

[2] There would presumably be more to say if I were writing this after my students completed their oral exams.

could proceed to a course in logic taught by either department. They should also be well-positioned to be seduced by machine functionalism in a philosophy of mind course.

As for those who leave this class wanting nothing more than to learn how to program, I don't know if I have a good suggestion. My judgement on the utility of college-level introductory programming courses must be reserved, since I never took one, but I suspect many of them are neither good at teaching programming nor good at introducing computer science. This may be thoroughly idealistic,[3] but I think the best way to learn to program is to sit down with a technical book (or just with a Web browser) and start figuring out how. It's little use learning to program without also learning to enjoy it, and it's hard to teach enjoyment. It is a subtle problem, and one that merits more attention than I can spare in this paragraph.

[3] For more idealism see section 7 below.

Chapter 6

Reflections and next steps

6.1 Reading

The perfect book for teaching this class with doesn't exist yet. *Logicomix* may be a perfect book to *use*, but it's not a textbook, and it doesn't cover any mathematical ideas in technical detail. My students wished that I had assigned more reading, and one suggested that providing lecture notes for some of the more technical material would have been helpful. Early drafts of my Division III contract implied that I was going to essentially undertake the task of writing a textbook during the fall semester, and while it's probably a good thing that I didn't try to do that, having a textbook at all would have been a big advantage for this class. The collection of topics I presented and the order and level of detail with which I presented them did not match up with any textbooks that I could easily get access to online. While a more "standard" discrete math textbook might have been helpful, the expense to utility ratio for my students would have likely been far too high.

Textbooks come into being in a variety of ways, but it certainly seems to me that the most interesting ones grow organically out of courses that people actually teach. If I have the opportunity in the future to write a textbook based on this work, it will likely be after I have taught a course like this one three or four more times, and have the inestimable benefits of experience. Since my hoped-for audience of people who are *already* computer science professors can't wait a decade or two for me to get around to writing a textbook, though, they'll have to do a bit of the legwork that I didn't do to track down a collection of textbook chapters and primary sources that cover things like Gödel's incompleteness theorem,

Turing machines, reducibility, and logic in an appropriate fashion.

6.2 Worrying

I definitely got better at teaching over the course of the semester. I suppose that is not too surprising, since I had to do it twice a week. Early on in the class I felt like a complete impostor much of the time, and that I was saying things that I didn't actually know were accurate—I barely knew anything about the philosophy of mathematics, so how could I presume to teach people about it? I asked a professor of mine how long it takes that feeling to go away, and she thought for a moment and said "I don't know if it ever goes away...it's more that you just stop caring."

That said, my confidence increased towards spring break. I was getting better at planning classes that filled the scheduled class time (or at least a solid hour) without lots of flailing about, and I was less reliant on preparing really detailed notes. At the same time, however, I was really struggling to make progress on the Thing I was supposed to be writing (i.e. this very essay). Teaching a handful of first-years was one thing, but to document that curriculum and pitch it at an audience of educators with years of experience teaching this same material and debating at length how it should be taught—that was another thing entirely. Several times I tried to start working, only to immediately worry that what I was trying to say was unjustified. I hadn't tracked down any longitudinal studies of computer science student outcomes or data on how many CS1 students became majors or really much of any empirical background on computer science education in general before designing my syllabus and starting the class. It was just something I was trying out because it seemed like a good idea, essentially. Why should anyone else care?

It took some prodding from my advisors, but I finally swallowed my anxiety and just got to writing. After all my worrying that I couldn't produce something that would manage to integrate documentation of my course, advocacy of my ideas, and reflection on my successes and failures, I seem to have done so.

Chapter 7

Radical technical education: A modest manifesto

Depending on who you ask, the Internet revolution is either the beginning of a hyper-transparent, completely-democratized, highly-networked, decentralized, techno-utopian future, or the vehicle of a worldwide, plutocratic panopticon and system of control. The truth, as tends to be the case with these sorts of things, likely lies somewhere in the middle. As someone who would like to see things play out more towards the techno-utopian end, I think that the only thing that will get us there is a considerable effort to educate young people about technology.

Since this is a *modest* manifesto, it consists of just a few bullet points:

1. Teach kids to touch-type in elementary school. Touch-typing can make a world of difference in how users think of their personal computers. If this isn't already happening everywhere, it should be.

2. Favor fundamental principles over specific tricks and technologies at every level. Requiring computer science students to use bloated Java IDEs and modeling tools is no less barbaric than spending weeks teaching middle schoolers to use Microsoft Office because that's what they'll be using in their future careers as corporate drones.

3. Don't evangelize any one platform to the exclusion of all other, but make sure that students understand the trade-offs between, say, iOS and Android, Linux and Windows, Apple computers and commodity hardware,

laser and inkjet. Nobody can make an informed choice without understanding what their choices are.

4. Many programmers lament the death of mandatory tinkering and the absence of BASIC prompts on new computers. This is, of course, a win for the vast majority of users, since they now have computers they can use. With that much chrome, it's even more important to teach the foundations of computing, since many students will have barely scratched the surface of what computers can do.

5. Somebody needs to fix the stupid math curriculum.

6. Little variations on the same old thing aren't going to attract underrepresented groups to computer science classes and computing careers. I would not be surprised to learn that the vast majority of college computer science majors first learned to program before college.

7. Academics will always move slowly. Let students help you move faster. If you are a student, spend some of your energy creating new ways for other students to learn before you find yourself slowing down.

Acknowledgements

Inestimable thanks are owed to all the following people, places, and things:

- Lee Spector and Chris Perry
- Laura Sizer
- Tatiana Soutar
- Ananda Valenzuela
- Lyndie Wood and Kat Mott
- Zaidee, Schmidt, Alex, and Arielle
- Mom, Dad, Magdalen, Oma, Shirley, and Eleanor
- The Omen and its Office
- Deathfest
- Throwback Pepsi and Mexican Coke
- Citalopram and Lorazepam

Bibliography

[1] James Robert Brown. *Philosophy of mathematics: A contemporary introduction to the world of proofs and pictures*. 2nd ed. New York and London: Routledge, 2008.

[2] The Joint Task Force on Computing Curricula. *Computing Curricula 2001: Computer Science*. Final task force report. IEEE Computer Society and Association for Computing Machinery, 2001.

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

[4] Interim Review Task Force. *Computer Science Curriculum 2008: An interim revision of CS 2001*. Final task force report. IEEE Computer Society and Association for Computing Machinery, 2008.

[5] Nick Huggett. "Zeno's Paradoxes". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2010 edition. 2010. URL: `http://plato.stanford.edu/archives/win2010/entries/paradox-zeno/`.

[6] Geoffrey K. Pullum. *Scooping the Loop Snooper*. 2008. URL: `http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html`.

[7] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability*. Cambridge, MA: MIT Press, 1987.

[8] Michael Sipser. *Introduction to the theory of computation*. 2nd ed. Course Technology, 2006.

[9] Raymond M. Smullyan. *Gödel's Incompleteness Theorems*. Oxford Logic Guides 19. New York and Oxford: Oxford University Press, 1992.

Low-Tech Computing: The History, Theory, and Philosophy of Computer Science

Evan Silberman
ejs07@hampshire.edu
Spring 2011

Course Description

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.

Trivial Details

Evan's email (he checks it constantly): ejs07@hampshire.edu

The course website: http://stuff.jklol.net/ltc/. There's not much here yet, (just a copy of the syllabus), but it will contain links to online readings and syllabus updates as plans change. It may also eventually look pretty.

Class meets 2:30-3:50 in the Hill-Urbina Meeting Room on the third floor of the library, but you probably know that already, since here you are!

Class Sessions and Assignments

January 31
Administrivia; introduction to the course.

Part I: The Foundations Crisis

February 2
Practical mathematics. What is mathematics? What do mathematicians do? What makes mathematicians certain about mathematical facts? What are mathematical facts anyway?

February 7
Infinity and beyond. Starting slightly before the beginning—A mathematician, an idea, a proof, a disruption of our common sense—Infinity and its reality—Diagonalization.

February 9
Mr. Bertrand Russel & his wondrous Paradox. What is logic? What is set theory? Why should they be the foundations of mathematics? Is anything wrong with those foundations? What is self-reference?

February 14
1+1=300 pages. Principia Mathematica: What it set out to do and what it did. Formalism, intutionism, and platonism. The epistemology of proof.

Assignment 1
Logicomix response. Due in class February 21.

February 21
Gödel. The Incompleteness Theorem(s). Its (their) impact on math, philosophy, and logic. Proof sketch.

February 23
Part I wrap-up. Talking about the *Logicomix* assignment. Stray threads.

Part II: Down and Dirty with Digital Logic

February 28
Drawing and building circuits. Drawing electrical/logical circuit diagrams. Using breadboards and electronic components.

Assignment 2
Begin circuit-building project. Completed write-up due to Evan's box (1394) by noon on March 11th.

March 2
A less-than-epic search for truth. Crash-course in formal logic. Syntax and semantics of logical connectives. Truth tables.

March 7
Bits are numbers. Counting in binary. Parallels between binary arithmetic and logical operations.

March 9
Building a respectable computer. We'll assemble the half-adders you've been building into a multi-bit adder. We might even be able to add two 4-bit numbers together!

March 21
Part II wrap-up. What have we learned about logic and numbers from our toy computers? What is computation? How might we go about defining it? Are our computational "proofs" of logical/numeric properties as good as logical/formal proofs?

Part III: Computation (At Last!)

March 23
Turing and his context. Flash-back to the Foundations Crisis. What is the Entscheidungsproblem? What is an algorithm? Wouldn't it be nice to be able to just set a proof-machine going until we knew everything?

March 28
The Church-Turing Thesis. The Turing Machine: a model for computation. Equivalence of models.

March 30
The Halting Problem. (Less German.) What is the Halting Problem for Turing Machines? Proof that the Halting Problem is undecidable. Diagonalization again!

April 4
Other undecidable problems. Proofs by reduction. Techniques for proving problems to be undecidable.

Assignment 3
Undecidable problems. Problem set drawn from Sipser chapter 5, etc., proving various things to be undecidable by TMs. This will likely be challenging. That's why the next class session is a...

April 6
Problem session. Questions and answers about the problem set.

April 11
Part III wrap-up. Present some complete solutions from the problem set. Other directions in computability theory.

Part IV: Topics in Computation

These classes are highly subject to change.

April 13
Intractability and the world-famous P vs. NP problem. What is intractability? What is non-determinism? What are "P" and "NP" exactly, anyway?

Assignment 4
Time to start preparing for the oral exam!

April 18
Intractability, heuristics, and cognition. If our brains are computers, then how do they solve problems quickly?

April 20
Artificial intelligence. What is AI? What is the Turing test? Reading "Computing machinery and intelligence".

April 25
Connectionism. What is a neural network? How do connectionist models change how we think about computation?

April 27
Evolutionary computing. What are genetic algorithms? What is genetic programming? Does biological natural selection do computations? (Perhaps not quite.)

May 2
Randomness and information. What is information? What is randomness? How do we tell if aliens are sending us signals?

May 4
Oral exams.

Challenge Problem:
Even Numbers

February 7, 2011

We might have the intuition that there are not as many even natural numbers as there are natural numbers. After all, for every even number, aren't there two natural numbers? Prove that this intuition is wrong, and that there are, in fact, as many even numbers as there are natural numbers.

Any reference material (including class readings) must be cited in your response. Please type your response (need not be more than one page) and bring it to class on Wednesday. I don't want late work on this assignment; please just hand in the *best attempt* you can make in the next couple days, even if it doesn't get very far. And feel free to contact me with questions; I will respond promptly.

(Also, notice how the wording of this problem explicitly ascribes an independent, abstract reality to numbers—the numbers "are", and we're trying to prove something about the numbers that "there are". Isn't this kind of creepy?)

Logicomix Assignment

February 14, 2011

Logicomix has introduced us to a great many ideas in logic, mathematics, and philosophy. For your first assignment, I want you to respond to some of those ideas as represented by the characters in the novel. In an essay of a few pages (at *least* a smidge over two, probably not more than six), address the following question:

How important is logic to mathematics and to philosophical argument? What are the circumstances when a formal logical argument is required to convince us that something is true? To put it concretely, what is the value in a 300-page formal proof that one plus one equals two, if there's any value at all?

Due date Monday, February 21, in class, typed, physical paper, with your name on it

Guidance In this paper, you are to present an argument for your *own* views, based on your reading of *Logicomix*, our class discussions, and your outside experience. It should also demonstrate your comprehension of the material from *Logicomix* and from class, by relating your views to those of the philosophers and mathematicians we have learned about.

Please cite page numbers when referring to specific parts of *Logicomix*. You do not need to do a bibliography entry for *Logicomix*; other sources (the use of which is entirely optional and not in any way expected) should have in-text citations and a bibliography entry in the style of your choosing.

Building Logic Circuits

Part 1

February 28, 2011

Overview

Your goal for this project is to build a device that can do some basic math out of primitive logical elements. In fact, you will start with only *one* logical operator available to you: NAND. Luckily, that's all you need! You will design circuits built out of only NAND gates that compute the NOT, AND, OR, and XOR operations. Then you will test your designs in the real world using electronic circuit components. Once we have built up or complement of logic gates, we'll attempt to combine forces to build a very simple computer that will add up some binary numbers. Then we will use nanoscale silicon chip fabrication processes to build a multicore central processing unit (not really).

Expectations

You can either answer all the questions for this part of the project in the space provided on this handout, or you can type/write your solutions up separately. You should *not* collaborate on this part of the project. Next week when we start trying to build adders you will be working collaboratively. You should try to generate your own solutions to all of these problems. If you resort to Google, please append an informal bibliography to whatever you turn in. If you get a solution from some other source, you must explain in some detail why that solution works, and discuss whether you were on the right track when you were making your own attempt.

1 NAND

Your 4011 IC is known as "quad two-input NAND". And what does that mean, exactly?

English Applying NAND ("not-and") to a set of inputs yields truth *if and only if* at least one of the inputs is false. Alternately, NAND yields falsity *if and only if* all of the inputs to NAND are true.

Symbols $\overline{(x \wedge y)}$

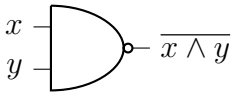
Symbolic logic usually doesn't include a special symbol for NAND; instead we use the overbar, indicating NOT, over the $\wedge$ for AND. We haven't defined those things yet but we'll get there.

Truth table A *truth table* defines the behavior of a logical operator by listing all the possible inputs to the operator alongside the operator's output in each case. Since we're dealing with two-input operators, there are four possible inputs to NAND that we have to define.

x	y	$\overline{x \wedge y}$
0	0	1
0	1	1
1	0	1
1	1	0

Math *Exercise*

Circuit diagram Every logic gate has an exciting circuit diagram equivalent.



Questions

a. Write down an arithmetical expression—that is, one in terms of addition, subtraction, multiplication, division, the inputs x and y, and some numbers—that is equivalent to NAND. Check that the outputs of your function match the truth table above.

b. Wire up your circuit so that DIP switches 1 and 2 are the inputs to a NAND gate on the 4011 and the output controls current through the LED using the transistor. Check that your output is correct for all possible inputs.

c. Show your circuit to a friend who isn't in the class and explain to them what it is doing as best you can without employing pencil and paper, additional visual aides, or any other class materials aside from the circuit. Have them initial and date here when you have done this. They may feel free to rate your explanation.

2 NOT

English Unlike the other gates in this project, NOT accepts only one input. Its output is the opposite of its input. NOT gates are thus also known in the world of circuit design as *inverters*.

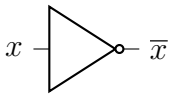
Symbols $\overline{x}$

Truth table The truth table for NOT is simpler than that for NAND since there is only one input, and thus only two cases to worry about.

x	$\overline{x}$
0	1
1	0

Math NOT can be described numerically by the expression $\overline{x} = 1 - x$. (Check that this works.)

Circuit symbol NOT gates are often indicated by a circle located after the input wire or before the output wire on other gates. An inverter all on its own looks like this:



a. Draw a boolean circuit using NAND gates that outputs $\overline{x}$ for an input x. (Hint: you can do it with just one gate.)

b. Wire up your circuit so that DIP switch 1 is the input to a NOT gate and the output controls the LED. Check that your output is correct for all possible inputs.

3 AND

English The AND operator outputs truth *if and only if* all of its inputs are true. Equivalently, the AND operator outputs falsity *if and only if* at least one of its inputs is false.

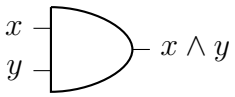
Symbols $x \wedge y$

Truth table Fill in this truth table:

x	y	$x \wedge y$
0	0	
0	1	
1	0	
1	1	

Math *Exercise*

Circuit symbol AND gates should look familiar:



a. Write down an arithmetical expression—that is, one in terms of addition, subtraction, multiplication, division, the inputs x and y, and some numbers—that is equivalent to AND. Check that the outputs of your function match the truth table above.

b. Draw a boolean circuit for AND using NOT and NAND gates.

c. Redraw that circuit using only NAND gates. You can do this with just two gates.

d. Wire up your circuit so that DIP switches 1 and 2 are the inputs to an AND gate and the output controls the LED. Check that your output is correct for all possible inputs. If you're having trouble, double-check that your jumpers between 4011 output and input pins are placed correctly.

4 OR

English The OR operator outputs truth *if and only if* at least one of its inputs is true. Equivalently, the OR operator outputs falsity *if and only if* all of its inputs are false.

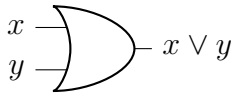
Symbols $x \vee y$

Truth table Fill in this truth table:

x	y	$x \vee y$
0	0	
0	1	
1	0	
1	1	

Math $x \vee y = x + y$ (Assuming that anything greater than 1 is also true, or, perhaps, that $1 + 1 = 1$.)

Circuit symbol



Additional fun fact De Morgan's theorem provides the following two equivalences:

$$\overline{(x \wedge y)} = \overline{x} \vee \overline{y}$$
$$\overline{(x \vee y)} = \overline{x} \wedge \overline{y}$$

This may come in handy when you're trying to figure out a circuit for OR. Or it may just distract you from a better solution.

a. Draw a boolean circuit for OR using NOT, NAND, and AND gates.

b. Redraw that circuit using only NAND gates. You can do this with as few as three gates. If you come up with a design that uses more than four gates, you can stick with that, but when you build it on your breadboard you will have to rearrange your circuit to make room for a second 4011 IC. So your motivation for finding the minimal solution is that you won't have to do that.

c. Wire up your circuit so that DIP switches 1 and 2 are the inputs to an OR gate and the output controls the LED. Check that your output is correct for all possible inputs. If you're having trouble, double-check that your jumpers between 4011 output and input pins are placed correctly.

5 XOR

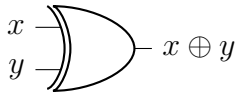
English The XOR ("exclusive-or") operator outputs truth *if and only if* exactly one of its inputs is true.

Symbols $x \oplus y$

Truth table

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	0

Circuit symbol



a. Draw a boolean circuit for XOR using NOT, NAND, OR, and AND gates.

b. Redraw that circuit using only NAND gates. You can do this with as few as four gates. (Really!) Again, if you come up with a design that uses more than four gates, you can use that, but when you build it on your breadboard you will have to rearrange your circuit to make room for a second 4011 IC. So your motivation for finding the minimal solution is that you won't have to do that.

c. Wire up your circuit so that DIP switches 1 and 2 are the inputs to an XOR gate and the output controls the LED. Check that your output is correct for all possible inputs. If you're having trouble, double-check that your jumpers between 4011 output and input pins are placed correctly.

d. Show your XOR circuit to a friend who isn't in the class (and isn't the same friend from question 1c) and explain to them what it is doing as best you can without employing pencil and paper, additional visual aides, or any other class materials aside from the circuit. Have them initial and date here when you have done this. They may feel free to rate your explanation.

Turing Machine Exercise

March 28, 2011

This exercise is taken from the Sipser book (*Introduction to the Theory of Computation*, second edition). As usual, if you collaborate with anyone or use outside sources to check or assist your work, please make a note on the assignment you hand in.

Refer to the TM named M_2 in Example 3.7 in the Sipser chapter and to the machine configuration sequence for the input `0000` illustrated on page 144. Give the sequence of machine configurations that M_2 enters for each of the following input strings:

a. `0`
b. `00`
c. `000`

Reducibility Problems

April 6, 2011

These problems are partially derived from Sipser, *Introduction to the Theory of Computation*, second edition. We will be having a problem session in class on Monday, April 11; your writeups are due on Wednesday, April 13. These problems are arranged in roughly increasing order of a property I'll call x, where x is based on my estimates of the objective difficulty of the problem, its subjective difficulty relative to what we've done in class, and how tedious the problem is to do. Your objective is to provide as many *good* solutions as possible; I'd rather you spend all week coming up with a good answer to a single problem than making a vague attempt at each problem.

Remember: working together is fine; however, each of you must write up your solutions individually, and please indicate who you collaborated with. And: using outside resources (such as a copy of Sipser) is OK (you might even find some solutions in the "Selected Answers" part of one of the chapters), but you must cite each source you used and describe how you used it, and if you find a complete solution in a source, you must demonstrate that you understand it.

1. A **multi-tape Turing machine** is just what it sounds like: a TM with more than one tape to work on during a computation. Prove that any 2-tape Turing machine M can be simulated on a single-tape Turing machine N by providing a general method for creating the simulation N based on M. Reminder: We briefly discussed a couple ways of simulating a 2-tape TM on a single-tape TM in class.

2. E is the set of Turing machines that don't accept any strings; we showed in class that this set is undecidable, since a TM that decided it could be used to decide the halting problem. Let Q (for "eQuivalent") be the set of pairs of Turing machines (M_1, M_2) such that M_1 and M_2 accept the same set of strings. Show that Q is undecidable by using a machine deciding Q to decide E.

3. Consider the problem of determining whether a two-tape TM ever writes a nonblank symbol on its second tape when it is run on input w. Show that this problem is undecidable by using a machine M that solves this problem to solve the halting problem. (Sipser problem 5.10)

4. Let R be the set of Turing machines M that have the following property: M is a TM that accepts the reverse of a binary string w whenever it accepts w. That is, if M accepts `01001`, it also accepts `10010`, and vice versa. Show that there is no Turing machine that decides the set R. (Sipser problem 5.9)

5. Let

$$f(x) = \begin{cases} 3x+1 & \text{for odd } x \\ x/2 & \text{for even } x \end{cases}$$

for any natural number x. Pick some starting number x and apply f repeatedly, obtaining a sequence of numbers. Stop if you ever hit 1. For example, if $x = 17$, you get the sequence 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1. Extensive computer tests have shown that every starting point between 1 and a large positive integer gives a sequence that ends in 1. But, the question of whether all positive starting points end up at 1 is unsolved; it is called the $3x + 1$ problem.

Suppose that the halting problem were decidable by a TM H. Use H to describe a TM that is guaranteed to state the answer to the $3x + 1$ problem. (Sipser problem 5.31)

Final Oral Exam

April 13, 2011

Your final assignment for this class will take the form of a twenty minute presentation and oral exam. There will be at least two examiners in the room with you, me and at least one special guest (plus your classmates). You must prepare to present your response to one of the questions below.

You must give a presentation of 10 minutes (±1 minute) in response to your question. You can present using any notes that you like, but the only visual aid you will have access to is the whiteboard and markers. After your presentation, the examiners will spend 10 minutes asking you questions about your response in order to probe the depth of your knowledge and understanding about the presentation topic.

You will likely need to do research beyond the confines of our class readings to present on some of the topics below. I am more than willing to point you towards useful resources if you ask; the research librarians are also fantastically helpful for this sort of thing—they may not know about computer science but they know how to find things. You must turn in an **annotated bibliography** (in whatever consistent and reasonable format you find pleasing) when you give your presentation.

The presentations will take place during the normal class session on **Wednesday, May 4**, our last day of class.

Questions/Topics

1. Is an artificial system capable of communicating using a human natural language plausible in principle, given what you know about computability and complexity? How should we measure the success of an AI communicator? Is the Turing test an adequate measure of machine intelligence? What sorts of advances are required to move from the current "chat-bot" state of the art to convincing, communicative intelligent systems? Are our brains computers?

2. Outline the historical and contemporary arguments over the validity of mathematical logic and its applicability to real-world problems. Some key questions for this topic are: What is the proper role of logic in mathematics and mathematical proof? What is the proper role of mathematical proof in the practice of mathematics? Why do arguments about the reality of infinity still linger? Did Gödel's incompleteness theorem really make much of a difference to how mathematicians work and think about their field?

3. Learn about cryptography. What are the technical foundations of current encryption systems? How would our current cryptographic protocols be impacted if it turned out that P=NP? How are they impacted by quantum computers? Is it possible to send a perfectly secure message? What is the societal role of secure communication?

4. A quote from John von Neumann goes "Anyone who considers [algorithmic] methods of producing random digits is, of course, in a state of sin." Research the topic of random and pseudo-random number generation. Where can we find randomness? What are some important applications of random numbers? Is it possible to inspect a string of bits and determine whether or not it is random? Is it possible to guess pretty well?

Challenge suggestion: Explain the notion of Kolmogorov complexity; give a sketch of the proof that the Kolmogorov complexity of a string is uncomputable.

5. Find out what the lambda calculus is and explain what it is and how it works. Sketch a proof that there is an undecidable problem in the lambda calculus. (I.e. the lambda calculus equivalent of the halting problem.) Use this proof to make an argument for the plausibility of the Church-Turing thesis.

6. Roll your own. Come up with a topic to research and present on. Topic proposals must be written up and handed in by Monday, April 18, be about a page long, and cite at least two sources. If your topic proposal is approved, that will be the topic I expect you to present on.