WHATEVER HAPPENED TO THE REVOLUTION, PART 2: IN WHICH I GET SEDUCED BY THE LURE OF A NATIONAL CURRICULUM

Brian Harvey [1]

Abstract
Logo was supposed to be an instrument of revolutionary change in learning, not just another activity for plain old school. In 1986 I stepped on some toes by complaining that Logo people had lost that vision. I like to think I've never lost it. And yet, I find myself making more and more compromises in order to influence the coming national computer science curriculum.

Keywords computer science education, Logo, Scratch, Snap!, revolution, reformism

1. Background: The Revolution

In 1986 I posed the question "Whatever happened to the revolution?" in the call for papers for the Logo86 conference at MIT.

The earliest writing on education, such as in Plato, considered it from the standpoint of the needs of society, either economic, as job training, or political, as the preparation of good citizens. The progressive education movement, often said to begin with Rousseau, offered a revolutionary shift of viewpoint, to making the needs of the learner central. Seymour Papert explicitly situated Logo within progressive education, suggesting that the progressive revolution had heretofore been unsuccessful because of the difficulty in finding materials that will excite the curiosity of every child, but that the computer is "protean" enough (see, for example, the Preface to *Mindstorms*) to serve, if not every child, at least most of them, as an object to learn with.

If you're making a revolution, you don't have to prove that you achieve the goals of the *ancien régime.* Papert was impatient, as am I, with the apparatus of Group A and Group B, pre-tests and post-tests, "$p < 0.05$." In one famous or notorious proposal for funding, he suggested that the funding agency agree to support his project for five years, without oversight or progress reports, and at the end of that time, "if any observer is tempted to apply statistical methods, declare the experiment a failure." (I'm quoting from memory so I'm not sure those are his exact words, but they're close.) That proposal wasn't funded, and Papert learned to use less inflammatory rhetoric, at least when talking to funding agencies. But the principle remains intact; Logo was not meant to do a slightly better job of teaching curriculum, whether traditional or innovative.

[1] Computer Science Division, University of California, Berkeley, CA 94720-1776, USA

But by 1986, Roy Pea had published his statistical study purporting to prove that learning Logo didn't result in transfer to kids' math performance, and some Logo people had started doing our own statistical studies hoping to refute him. More importantly, Logo had become very popular with teachers, and yet school was still school: mostly lessons in how to sit down and shut up.

2. Background: Scratch, BYOB, and Snap*!*

The computing environment of the 1980s was one of the limiting factors for Logo. When the Scratch project started in 2003, computers were powerful enough to support drag-and-drop programming, and the Internet enabled the creation of a learning community independent of the schools. For these reasons, along with the brilliance of its design, Scratch has had exponential growth. Among its achievements, one of the most interesting has been the development of a carefully nurtured community led by children as well as by adults.

When I first saw a demo of Scratch, I wasn't impressed, because the user couldn't create procedures. There were scripts, but they didn't have names and couldn't be recursive. What kind of programming language is that? It seemed a step backward from Logo, despite the great GUI. They nevertheless insisted that I come to the first Scratch conference, in 2008. I did, and started making trouble about recursion, organizing an ad hoc session at which I demonstrated projects that were easy in Logo and hard or kludgy in Scratch.

(But the opening session at that conference was a panel of three kids. That was how I learned about the Scratch online community, which I love, and which is what made making trouble worth the effort. I wanted Scratch to be as good as it could be! I understood that leaving out user-defined blocks was a deliberate decision, not an oversight, but I thought, and still think, it was a mistake.)

At my session was a German programmer and lawyer named Jens Mönig. I'm a complainer, but Jens is a doer; he went home and built an extended Scratch with user-defined blocks. He called it "Build Your Own Blocks," or BYOB. He made a blog post about it, but didn't post the software for download; he viewed it as a proof of concept for the Scratch Team, and wasn't expecting many actual users.

At about the same time, there was a movement at university computer science departments to develop a "computer science for poets" course that would make the flavor of CS accessible to all students, not just scientists and engineers. Several schools (Harvard, for example) wanted to use Scratch, because of its famously non-intimidating interface, but discovered that Scratch wasn't a powerful enough language to support the course they wanted to teach. So they started with Scratch, used it for the first week or two of the course, and then switched to some text-based language such as Python, Java, or even Excel. At Berkeley, we had the same conversation, but weren't happy with that conclusion, which seemed to be an unfair "bait and switch." The text-based language would be just as intimidating in week three as it would have been in week one. What we needed was a super-Scratch that would meet all the needs of our course. We found Jens's work and eagerly got in touch.

BYOB made our course, "The Beauty and Joy of Computing" (BJC), possible. What Berkeley gave to BYOB was its first large-scale test with real users. The problems we found were of three main kinds: First and least interesting, we found many bugs, large and small. Second, Jens's user interface for defining a block was (appropriately, for a proof of concept pilot version) not up to the standard of intuitive graphical manipulation set by Scratch. Third, the ability to create named blocks supported the big idea of recursion, but not the other big idea of programming technique we wanted in the course, namely higher order functions. For that we needed the ability to create *anonymous* procedures — the Lisp lambda.

So, being a complainer, I started complaining to Jens. Being a doer, he responded with almost daily new versions. This was the beginning of a close and ongoing collaboration, which in its first year produced BYOB3, with a graphical interface for procedure definition and with first class procedures (lambda) and first class lists.

At that point, we still hoped that the Scratch Team would be so impressed with our work that they'd incorporate it into Scratch itself. For that reason, we wanted to make the minimum possible change to Scratch, knowing that the Scratch Team didn't want a complicated and therefore intimidating collection of primitives. At Berkeley we had for many years taken our introductory curriculum from Abelson and Sussman's *Structure and Interpretation of Computer Programs,* the best CS book ever written. So we knew that lambda, plus a way to call the procedures created by lambda, are all you need for a Turing-complete language, so we were confident that, given lambda, we could relegate any other control or data structures we needed to libraries written in BYOB itself. (We did decide that the ability to create anonymous data structures — lists, in Scratch terms — was important enough, and hard enough to build just with lambda, to justify adding the LIST reporter that creates an anonymous list.)

(They *were* impressed, but at that point they weren't convinced that the intellectual difficulty of the idea of defining procedures wouldn't drive away their primary audience of younger kids. It wasn't until Scratch 2.0 that they took the first steps toward user-defined procedures.)

At this point in the story comes my first example of a compromise of the revolution. We started the process of trying to rule the world of high school computer science, about which more later, and about a year in, we heard from three teachers, including one of our graduate students who had contributed a lot to the curriculum, that they felt uncomfortable using a piece of software called BYOB with kids. They all explained that *they* had a fine sense of humor, but that some hypothetical kid's hypothetical *parent* might complain. (European readers may not be aware that Americans have the quaint idea that if you insulate a child from any mention of an adult pleasure, such as alcohol, until his or her 21st birthday, he or she will then abruptly develop adult attitudes about moderation and so on, but that this will be spoiled if the child gets any hint prematurely that such pleasures exist.)

At the BJC weekly staff meeting at which this issue came up, one of my Berkeley colleagues said, "We have to change the name." Naturally, I complained. "What do you mean, 'we have to'? It's not our software, it's Jens's. He gets to say what it's called." It took quite a while for me to argue the rest of the team down to *asking* Jens if he would mind changing the name. Somehow it became my job to convey this message to Jens. I said, roughly, "my idiot colleagues want you to change the name because exactly three teachers have no sense of humor; I personally think that the name is perfect, and funny, and you should tell them to shove it. But it's up to you." Jens, as a rule, would much rather write code than get involved in office-political debates, so he declared that he didn't care about the name and would do whatever we wanted. Thus was born Snap*!*. That was the best alternative name any of us could think of, but it's pretty generic; try Googling it. (I insisted on the exclamation point to make it stand out from the food stamp program and the support group for survivors of abuse by priests, but we're still the fourth link; Google seems to ignore the punctuation, even inside quotation marks. I also insisted that the name change wait for the release of version 4.0, which happened earlier this year, to minimize user confusion.)

3. Trying to Take Over the World

Our course is part of a coordinated national effort to reform the teaching of computer science. The effort has been led by the National Science Foundation. It is justified on two grounds:

- In almost every field, American universities are producing more Ph.D.s than there are jobs to absorb them. The only exception, at least according to some statistics, is computer science, where we have an acute shortage of people to fill the jobs. (This is a controversial point. Some argue that the jobs include many that do not require computer science education, from web design to accounting with spreadsheets.)
- In particular, almost all computer science students are white or Asian males, leaving out half of the population. (There's no doubt about this one.)

Computer science education is weak at every age level, but the NSF made the strategic decision to concentrate on the high school level at first, because that's when most students make career decisions. The goal is to offer a serious but broadly appealing CS course to every high school student in the country.

The United States education system is run by local school districts, following curriculum standards set by individual states. (There is an effort under way to change that, with the "Common Core" curriculum developed at the national level, but each state decides whether or not to follow it.) As a result, we can't do the sort of all-at-once educational reform effort common in other countries. The only exception, a *de facto* national curriculum, is the series of Advanced Placement (AP) exams offered by the College Board, an NGO established by a consortium of universities. AP courses do not have to be approved by each school district individually; almost all districts have already given blanket approval to them. Thus, a tactical decision was made that the new course would be offered as an AP. There's a cost to this choice: Kids who view themselves as not college-bound may be scared away from the course.

(There is already an AP CS course. To a first approximation, nobody takes it. All the other APs, even calculus, have seen exponential growth over the past decade. The AP CS A course has had slightly *declining* enrollment. It's a straightforward programming class using Java, a language not exactly designed to appeal to adolescent women. The new course, AP CS Principles, will be offered in addition to the existing CS A. The first CS Principles exam will be in spring of 2017.)

Heretofore, Berkeley CS has been content to design courses for our own students; indeed, we take pride in the ways in which our curriculum differs from everyone else's. But now, we have a shot at influencing how every high school student in the US is introduced to computer science. How could we not want that? For me, in particular, it's an opportunity to push functional programming, recursion, higher order functions — the same ideas I've been promoting since the Logo days.

4. In Bed with the College Board

I *hate* the Advanced Placement program. When I was a kid, it was a very different program. A kid would take one, maybe two, AP exams, in the area of his or her greatest interest. I took the math AP course, a calculus course, in the company of other kids who were fanatical in our interest in math. Today, though, AP courses are one of the ways kids fight to distinguish themselves in the eyes of university admissions officers. Like every arms race, this AP strategy is doomed to failure, because the other smart kids take as many AP courses as possible, too. There are high schools that offer *nothing but* AP courses — in effect, teaching a full freshman and sophomore university curriculum to high school students. AP courses involve twice as much homework as ordinary high school courses. This pressure ruins the lives of our brightest students. They're all addicted to caffeine; more than a few are addicted to Adderall and other stimulants borrowed from the officially ADHD kids. And the result is that many graduate from high school knowing everything, but with no real intellectual interest; all the subjects have become hoops to jump through.

In fact, a few years before this CS Principles effort began, I worked hard to be appointed to our faculty committee on admissions, where I tried valiantly to convince my colleagues to put less weight on the number of AP courses taken by applicants. I failed.

And yet, in 2010 I found myself a consultant to the College Board, working on piloting an AP course. How did this happen? My colleague Dan Garcia was part of the early discussions of the CS Principles idea, and we were the first to offer such a course to our own students, team-taught by Dan and me. In the planning for our course, I insisted that we start by paying no attention to the College Board curriculum principles, instead designing the course we thought best for Berkeley non-CS majors, and only later revisiting the College Board framework to make sure we satisfied it. Given our head start, it's not surprising that we were chosen as one of the initial five pilot sites for the curriculum.

The entire national process is a little strange. Usually, the College Board assesses what colleges and universities are actually teaching in a subject, and then designs an exam to match. That's how we got the CS A exam. (There used to be a CS AB exam, covering two college semesters, but it's no longer offered because it was even less popular than the CS A.) But now, for the first time, the College Board is trying to create a course that *isn't* taught at many schools, and about which there's no consensus curriculum. They are leading, not following. The result is a curriculum framework document with broad principles but few details. Most importantly, the course is supposed to include programming, but they don't want to choose a programming language for it. Thus, the initial five pilot sites, and the 20 second-round pilots, are all teaching quite different courses.

How can they write an exam under those conditions? My guess is that it'll cover the intersection of all the pilot curricula, i.e., not very much. For the programming questions, they've invented a "pseudocode" language, different from all the languages used in the pilots, but firmly in the C/C++/Java/Javascript/Python mainstream. But we'll find out in 2017. (One progressive aspect of the AP course is that, in addition to taking the exam, students must submit a "portfolio" of three assignments: a paper about social implications, a "big data" exploration, and a programming project. Students choose their own projects. This is much better than an exam entirely about facts and narrow skills, such as translating a number to binary.)

Our course is technically much more demanding than what the syllabus requires. Probably nobody else is including higher order functions! Indeed, the central research question for us is whether we're right that we can both attract underrepresented students to such a course and get them to succeed in it.

5. Compromising the Revolution

I hate grades! Grades are the enemy of learning; students can't take risks when they're afraid. When I was a high school CS teacher in the early 1980s, the first thing I did was get the school to agree that I could offer my courses pass/fail. That meant kids could work on what they wanted, that I could tell them if I thought they were doing a bad job without frightening them, and that they could argue back, colorfully, if they thought I was wrong. It was great!

Of course there was never a chance of my getting away with that at Berkeley. So I started compromising long before CS Principles came along. This compromise was its own punishment; over 25 years I've spent far too much time writing exams, grading exams, arguing with students about their grades, and dealing with students who cheat. I love teaching, but I hate all that. It's why I retired early.

At least I didn't keep my feelings secret from the students. Here's an excerpt from the first-day handout in my *Structure and Interpretation of Computer Programs* course:

> If it were up to me, we wouldn't give grades at all. Since I can't do that, the grading policy of the course has these goals: It should encourage you to do the course work and reward reasonable effort with reasonable grades; it should minimize competitiveness and grade pressure, so that you can focus instead on the intellectual content of the course; and it should minimize the time I spend arguing with students about their grades.

And, when it came time to write the first-day handout for our team-taught pilot course, here's Dan's attempt to do justice to my views and his own more conventional ones:

> For the most part, we would prefer to teach this course *without* grades. What a wonderful concept, learning for learning sake! However, even though we can't change the "system" overnight, we can create grading policies that support learning as much as possible.

What a concept! Learning for learning's sake! Indeed.

I hate curriculum! In my high school class, I required a very minimal experience with programming in Logo, and a very minimal experience with word processing. (Remember, this was before everyone had a computer.) But after that, students were free to do whatever they wanted, as much or as little as they wanted. (Besides being pass/fail, my intro class was variable credit, so a kid who turned out not to enjoy computers could stop early, taking one unit of pass instead of two units of fail. On the other hand, kids who loved it could work extra hard and get as many as four units.)

At Berkeley, I was immediately seduced by the best computer science book ever, *Structure and Interpretation of Computer Programs.* It's so great, it made me feel good about having a curriculum, especially because it was a course for CS majors, which made it reasonable to expect them to learn what we considered important.

But I still believe, really, that the way to get high school kids excited about computers would be to have no curriculum at all, so there would be absolutely no sense of jumping through hoops. I'm not

sure I could get away with that today even at one high school. Certainly it's not compatible with wanting to take over the world. And so here I am telling kids exactly what they should learn.
In fact, I recently submitted an NSF proposal that, among other things, would involve a collaboration with EDC (Education Development Center), a nonprofit curriculum developer, to turn our still rather sketchy curriculum into what high school teachers will recognize as a "real" curriculum, with scope and sequence, a teacher's manual, daily lesson plans, the works. (The EDC lead on this proposal is Paul Goldenberg, whom many of you know.) This is so unlike me! Why do I want to do it? Partly, to be honest, because I don't trust people to maintain the technical rigor of our curriculum unless it's cast in stone. I'm trying to force my opinions on my (intellectual) offspring, like the old people who end up getting murdered in Agatha Christie books.

During an argument a while ago, Dan was astonished (and then *I* was astonished that he didn't realize this about me) when I said that as far as I'm concerned, higher order functions are the entire point of our effort on BJC. "Don't you care about getting girls into computer science?" he asked. "Sure," I said, "but a lot of smart people all across the country are doing that. It doesn't need me." And so I am become a proponent of curricular rigidity.

I hate MOOCs! Well, let me not exaggerate. MOOCs are great if you're stuck in a rural intellectual wilderness and it's MOOCs or no education at all. But I hate MOOCs as a way to save money on education by allowing fewer teachers to teach more students. I believe passionately that the personal relationship between teacher and learner is the main point of education.

So of course we're making a BJC MOOC. Several, actually, because the MOOC experts believe that courses have to be short, three weeks or so, in order for people to finish them. So we have BJC part 1, BJC part 2, and so on. At least we're going to host it on edX, the noncommercial MOOC platform, rather than on a for-profit platform. (Worst of all is the one set up by the University of California itself, one of whose leaders addressed a CS faculty lunch meeting by saying, "You professors think you're doing education, but you're really not. You're just doing content delivery. It's not education until students pay money and are awarded official credit.")

I hate schools! Clearly I've been compromising on this one all my life, partly to earn a living and partly because schools have a monopoly on kids. But schools, even good schools with loving teachers, are so stultifying! They leech the curiosity out of kids, turning them into academic strategists: How can I get through this with the least injury to my soul and my self-esteem? At the university, at least our students are volunteers, even if quite a few engineering students are forced into it by their parents, when they'd really rather study drama or history.

In some ways BJC is less of a compromise than most curricula. The centerpiece of the course is a project done by students in pairs, chosen by the students themselves. This is so much better than a project chosen by adults as "culturally relevant" to the students! Let them decide for themselves what's relevant. Still, we teach binary (which has *nothing* to do with computer science, which is all about abstracting away the computer hardware so programmers can think about the problem they want to solve instead of about how computers work) and "big data" basically because the College Board says we have to.

6. Conclusion

I can’t help feeling that the BJC effort is worthwhile, for several reasons. At the same time, I also can’t help feeling that it’s not the work I was called to do, when I first got interested in education by reading Paul Goodman and John Holt and James Herndon and A.S. Neill. I’m hoping you can help me resolve this intense inner conflict!