The Beauty and Joy of Computing: Computer Science for Everyone

Brian Harvey, *bh@cs.berkeley.edu*
Computer Science Division, University of California, Berkeley

Abstract

"The Beauty and Joy of Computing" is a computer science course for undergraduate non-majors that combines a deep programming experience with lectures, readings, and discussions about nonprogramming topics such as the social context of computing and the future and limitations of computing. The course is designed to appeal to a wide range of students, including women and underrepresented minorities. The programming half of the course uses BYOB, an extension to Scratch adding first class procedures, lists, and objects. The course has been chosen as one of the pilots for a coming (2016) high school Advanced Placement exam. Our current work includes further curriculum development, an NSF-funded teacher preparation program, and the implementation of SNAP!, a new browser-based version of BYOB.

Keywords

Computer science curriculum, teacher education, programming language, BYOB, Scratch

The Curriculum

Berkeley has a 14-week semester, and our course "The Beauty and Joy of Computing" (BJC) meets seven hours per week: two lecture hours, four lab hours, and one discussion hour. Out-of-class assignments include reading, writing, watching videos, and pair programming projects. The non-programming aspects of the course help dispel the "nerd" image of traditional computer science courses, and our course has been very successful in attracting students from nontechnical majors. The course has been taught five times over the past three years, and half of the students have been women. Almost half of the students in the top fifth of the class have also been women. Table 1 shows the topic list for the course.

This paper is mostly about the programming part of the course, but it should be emphasized that our success in attracting and retaining students is due in large part to the social context included in the curriculum. Our textbook is *Blown to Bits* (Abelson *et al.*, 2008), which presents some of the social issues of the Internet era in a style that manages to be both accessible to lay readers and deeply informed by specific technical issues. The book, like our course, is generally positive about computer technology, while including a critical appraisal of unexpected consequences.

BYOB (Build Your Own Blocks) was presented at Constructionism 2010 (Harvey and Mönig, 2010). It is based on Scratch, a language designed for 8–12 year old users at MIT (Resnick *et al.*, 2009), using a novice-friendly drag-and-drop interface, eliminating many of the difficulties beginners experience in editing a program text. Our extended version adds capabilities intentionally left out of Scratch, most notably first class procedures, so that we can teach recursion and higher order functions.

Week	Lectures	Labs	Discussion
1	Abstraction	Broadcast, Animation, Sound	Welcome
2	3D Graphics, Video Games	Loops, Variables, Random, If	Computer Anatomy
3	Functions, Programming Paradigms	Procedures, Lists	Video Games (social implications)
4	Algorithms, Order of Growth	Lists, Algorithms	
5	Concurrency	Complexity, Concurrency	Complexity
6	Recursion		
7	Social Implications, Recursion	Recursion	
8	Social Implications, Human-Computer Interaction	Recursion	Social Implications
9	Game Theory, Industry Guest	Applications that Changed the World	Midterm review
10	Artificial Intelligence, Applications that Changed the World	Online Midterm	Artificial Intelligence
11	Lambda and Higher Order Functions		
12	Distributed Computing, Academic Research (guest lecture)	Distributed Computing	Lambda, HOF
13	Limits of Computing, Future of Computing	Project work	Open discussion
14	Cloud Computing, Summary	Project, Online Final	Final Thoughts

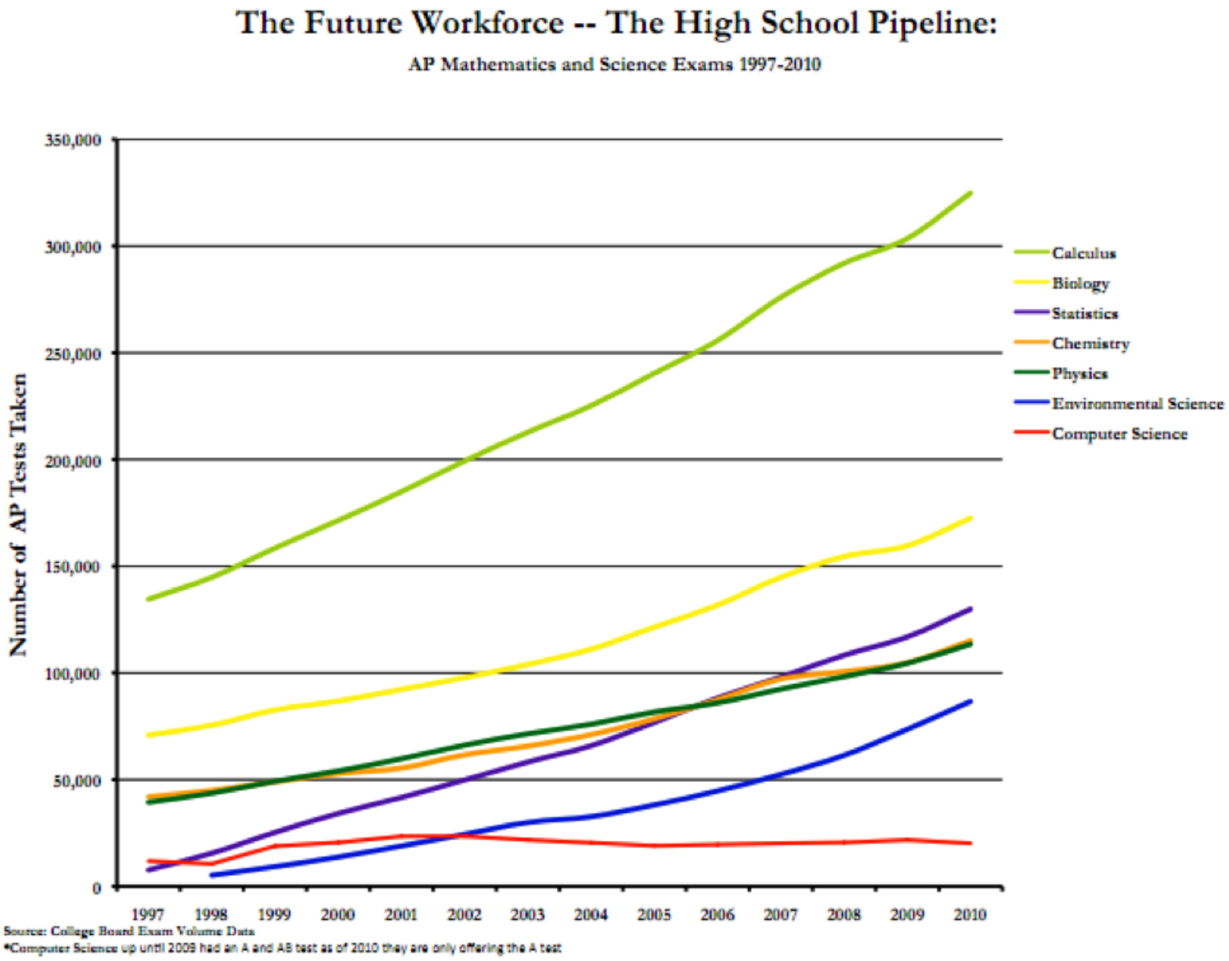
Table 1. Beauty and Joy of Computing curriculum

All of our course materials are available free of charge through the course web site (`http://bjc.berkeley.edu`), including lecture videos, Moodle-based lab units, the BYOB software, and even the textbook.

Advanced Placement "Computer Science: Principles"

In the United States, curriculum policy is set by each local school district (each city, roughly), with some input from the state governments. This makes a widespread curriculum reform much harder to implement than it would be in a country with a national education policy. The only *de facto* exception is that secondary schools can offer university-level courses through the Advanced Placement (AP) program run by the College Board, a private non-profit organization. To ensure uniform standards, students get AP credit by taking a national standardized AP exam. Changes to the exam are publicized in advance, and so the change is promptly reflected in every school, without a complicated reapproval process.

There is a Computer Science AP course, which consists entirely of Java programming at the level of a first semester university course for CS majors. Java, which is both syntactically and semantically complicated, is probably not the ideal first programming language for non-specialists. And, indeed, Computer Science is by far the least popular AP course, and the percentage of students taking AP CS has been flat while other math and science AP courses, also traditionally unpopular, have grown dramatically in recent years (Figure 1). Women and minorities, especially, have avoided AP CS.



Source: College Board Exam Volume Data
*Computer Science up until 2009 had an A and AB test as of 2010 they are only offering the A test

Figure 1. Advanced Placement test takers by subject.

Before the Internet and mobile computing platforms, computers were used by specialists, and the unpopularity of computer science was unsurprising. But most young people today are adept at computer gaming, social networking, and online media. Sites such as YouTube and Flickr have made young people creators, not just consumers, of online media.

Because the number of university Computer Science students has not kept up with the demand for computer programmers, the National Science Foundation (NSF) has initiated several efforts to make CS courses more popular. They are focusing on the secondary school level because students' choices are often already made before they attend university. As one part of this focus, the NSF has teamed with the College Board to develop a new course, "AP CS: Principles," that will be equivalent to a university breadth course for non-majors, rather than a first course for majors. (The old AP CS will still be offered.) (*CS Principles*, 2012.)

The new curriculum is still in development. There are design documents, including a list of seven "Big Ideas" (programming is one of them) and desired skills outcomes. The College Board chose five pilot sites in 2010–11, and 20 more sites in 2011–12. Each site is teaching its own course design, with a range of programming environments, and indeed a range in the extent to which programming is part of the curriculum at all. We were one of the initial sites. Another initial site, the University of North Carolina (UNC) at Charlotte, also chose to use a modified version of our curriculum.

Compared to the published College Board course documents, our version is technically ambitious; we think we can teach recursion and higher order functions to a general high school population (or at least to college-bound students). And we think that these powerful ideas *are* an important part of the beauty of computing. Seeing the complexity of a fractal tree, and then seeing the simplicity of the recursive procedure that draws it, is an "aha! moment" you don't get from doing a Google search or making a poster in Photoshop, or even from writing computer programs with no control structure more powerful than a loop.

Can Constructionism Be Standardized?

In the Berkeley BJC course, 30% of a student's grade is based on midterm and final programming projects, chosen by each team of two students. (Another 15% comes from an online component of the midterm and final exams, so programming practice counts for almost half of the overall grade.) The semester ends with a "show and tell" session in which student teams present their projects to the entire class. These projects are what give the course most of its constructionist flavour, although students' written work is also a public artefact in the form of a course blog posting.

In this early pilot phase of the project, evaluation standards are up to each participating school. But when there is an official AP curriculum, it will be measured entirely by an exam, which, the College Board says, will be "language agnostic"; that is, no particular programming language will be used. Instead, programming *ideas* will be tested in the form of pseudocode.

Our own implementation of the course will not change. But we are hoping to spread our curriculum, including its programming-heavy aspects, through the medium of the AP. A focus on student-chosen projects doesn't fit well with a standardized test. Can we influence the coming AP test so as to encourage a constructionist approach in high school computer science? Or is "taking over the world" through the AP a Faustian bargain in which only factual knowledge (including knowledge about programming) will be emphasized in the secondary schools?

Even at Berkeley, we struggle with testing student mastery of a visual, rather than textual, programming medium. It's hard for students to write BYOB programs in a test booklet. Our solution has been to test students' programming ability in the lab, so that they use computers to write and submit their answers. (In the written half of the test, we can include pictures of programs and ask questions such as "find the bug in this program" or "draw the picture that this program would draw.") But a nationwide test can't rely on hands-on computing, both because of a lack of available computers and for fear of cheating.

More broadly, the entire AP program is a stressful, jumping-through-hoops experience for high school students who want to attend a high-ranked university, hard to reconcile with any humane approach to education, let alone Constructionism. In the past, students with a strong interest in a particular subject might take one or two AP courses. But today, college admissions officers expect applicants to have taken every AP course offered at their school; many students' high school experience is entirely AP. A computer science course offered as an AP will have to be very joyful indeed to excite students' enthusiasm.

Teacher Preparation: CS10K

One reason that not many students take the existing AP CS, besides the curriculum itself, is that many schools do not offer it, because they can't find qualified teachers. Anyone who can program computers well enough to teach the course can get a better-paying job programming. And young people who discover in themselves an interest in programming don't often choose the kind of education that leads to a teaching credential. The NSF, in addition to the CS: Principles curriculum development effort, is sponsoring a drive to prepare 10,000 high school computer science teachers qualified to teach the new course. Many of these will be existing teachers of computer applications or, in some cases, teachers of computer assembly and repair. (In many parts of the country funding for computers is more readily available through vocational-track budgets than through academic course funding.)

UC Berkeley and UNC Charlotte have been funded by the NSF to prepare teachers through summer workshops using BJC. We ran one pilot workshop in 2011, and are funded for five workshops with 20 teachers each during the summers of 2012–14. We've already scheduled three workshops this summer (2012) and are seeking additional funding to expand the program.

School districts or other regional groups that organize 20 participating teachers can apply for a workshop. We bring experienced workshop leaders to these locations. Each six-week workshop includes an initial week of face-to-face meetings with leaders and participants, followed by four weeks during which the participants take our online course from home (watching the lecture videos and doing the online lab work) with one weekly discussion meeting in which participants gather face-to-face and work with a Berkeley teaching assistant using Internet videoconferencing. The sixth week is again face-to-face and focuses on how teachers can translate the curriculum to the specific conditions (contact hours, student body, and so on) at their schools.

Bringing the workshop to the participants' location is important. During the 2011 pilot workshop, we had some remote participants who used videoconferencing to join the group, and in post-workshop surveys, both those remote participants and the local ones found that the necessary technology was a distraction, and, more importantly, the remoteness of some participants interfered with the bonding and collaborative work even of local participants, who reported that they felt guilty if they got together outside of the scheduled session times without the remote participants. Ideally, we would fly our teaching assistants to the workshop locations, but doing that four times for weekly half-day meetings would be very expensive, so we are trying the compromise of having the actual participants physically gathered together but with a remote TA.

During the four-week online course, we provide online assistance with the lab work. The BJC course at Berkeley has attracted a small army of course veterans who are enthusiastic enough about the course to volunteer their time as lab assistants, so we can help summer participants at very small extra cost.

SNAP!: An Online Reimplementation of BYOB

BYOB was implemented as an extension to the actual Scratch source code, written in Smalltalk. Scratch was designed with the goal of maintaining a responsive graphical user interface, and smooth animation of sprites, rather than with the expectation of composition of functions as a primary control structure. The result was a series of incremental modifications to nearly every part of the code. Scratch's lists were designed for iterative sequences of commands, not for building up with recursive reporters. These and other factors made BYOB projects very slow, and debugging BYOB difficult.

BYOB's developer, Jens Mönig, is currently working on a complete reimplementation, written in Javascript so that it will run in a web browser. This solves several problems for us. Because it's a completely redone design, projects run much faster. Because it runs in a browser, the new version automatically supports every new platform, including tablets and mobile phones, although the user interface isn't currently very usable on the small screen of a phone. Also, we've learned that school IT departments are reluctant to install software they've never heard of, and a browser-based implementation requires no installation. Eventually, running in a browser will enable new capabilities, such as embedding a project in a web page. The new version is called "SNAP!"; it was renamed because a few teachers objected to the original acronym.

As of May 2012, there is an alpha-test version, missing many features, but already quite powerful, available at `http://snap.berkeley.edu/run`. (While in alpha testing, we don't promise that saved projects will remain readable as development continues. We are hoping for a stable beta version by the time of the conference in August.)

Javascript tries to maintain the security of users' computers by limiting the ability of downloaded code to interact with the computer's filesystem and hardware. This is problematic for us both for saving projects and for interacting with real-world sensors and robots. The standard Web solution to the former problem is to store everything "in the cloud," which means that we would have to provide user project storage centrally, or else ask schools to run their own SNAP! servers, defeating the no-software-download advantage. A possible solution would be an *optional* software download to interface between SNAP! and the user's computer.

Further Curriculum Development

The UNC version of the course is different from the Berkeley version, for two main reasons: UNC has fewer student contact hours per week, and our collaborator there, Prof. Tiffany Barnes, was previously teaching an introductory course based on video game design in Gamemaker and wanted to include some of that curriculum in the BJC course. We anticipate that other schools will have similar need for flexibility in the curriculum.

We therefore plan to build curriculum materials with the same core ideas, but divided into modules from which each school can select the ones they need. One big example is that, even though BYOB supports object-oriented programming through sprite inheritance, there is no OOP curriculum in BJC. A different kind of example is that we are working with a Microsoft-sponsored program that uses the Xbox Kinect motion sensor as a device to be programmed, and we plan to develop curriculum modules for that.

We are also working with the Ensemble project (`http://www.computingportal.org`) to allow teachers outside of our group to contribute modules.

This raises the question, so far unanswered, of how different a course can be from the Berkeley version and still be called "BJC." Probably the modules will be categorized, and there will be minimum standards both in the big ideas of programming and in the social context of computing.

Acknowledgements

The BJC design team at Berkeley includes its instructor, Dr. Daniel Garcia (Principal Investigator on our NSF grants), and several graduate and undergraduate students, especially Colleen Lewis, Luke Segars, and Glenn Sudgen. Dr. Nathaniel Titterton is our lead education researcher. High school teachers Josh Paley, Sean Morris, Ray Pedersen, and Eugene Lemon have participated in the development and testing of the curriculum. Prof. Tiffany Barnes is our collaborating Principal Investigator at UNC Charlotte. BYOB was developed, and SNAP! is being developed, by Jens Mönig.

This work has been funded by NSF grants 1138596 and 1143566, and by Lockheed Martin.

References

Abelson, H., Ledeen, K., and Lewis, H. (2008). *Blown to Bits: Your Life, Liberty, and Happiness After the Digital Explosion,* Addison-Wesley.

Harvey, B. and Mönig, J. (2010). "Bringing 'No Ceiling' to Scratch: Can One Language Serve Kids and Computer Scientists?", Constructionism 2010.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). *Scratch: Programming for All.* Communications of the ACM, vol. 52, no. 11, pp. 60-67 (Nov. 2009).