Computer programming goes back to school

Learning programming introduces students to solving problems, designing applications, and making connections online.

By Yasmin B. Kafai & Quinn Burke

We are witnessing a remarkable comeback of computer programming in schools. In the 1980s, many schools featured Basic, Logo, or Pascal programming computer labs that students typically visited once a week as an introduction to the discipline. But, by the mid-1990s, schools had largely turned away from programming. In large part, such decline stemmed from a lack of subject-matter integration and a dearth of qualified instructors. Yet there was also the question of purpose. With the rise of preassembled multimedia packages via glossy CD-ROMs over the 1990s, who wanted to toil over syntax typos and debugging problems by creating these applications oneself? This question alone seemingly negated the need to learn programming in school, compounded by the excitement generated by the Internet. Schools started teaching students how to best surf the web rather than how to delve into it and understand how it actually

YASMIN B. KAFAI (kafai@upenn.edu) is a professor of learning sciences at the Graduate School of Education, University of Pennsylvania, Philadelphia, Penn., with a secondary appointment in the Department of Computer and Information Science. **QUINN BURKE** (burkeqq@cofc.edu) is an assistant professor of education technology at the College of Charleston and a former high school teacher. Kafai and Burke are authors of the forthcoming book *Connected Code* (MIT Press, 2014).

R&D appears in each issue of *Kappan* with the assistance of the **Deans' Alliance**, which is composed of the deans of the education schools/colleges at the following universities: Harvard University, Michigan State University, Northwestern University, Stanford University, Teachers College Columbia University, University of California, Berkeley, University of California, Los Angeles, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Thinkstock/iStockphoto

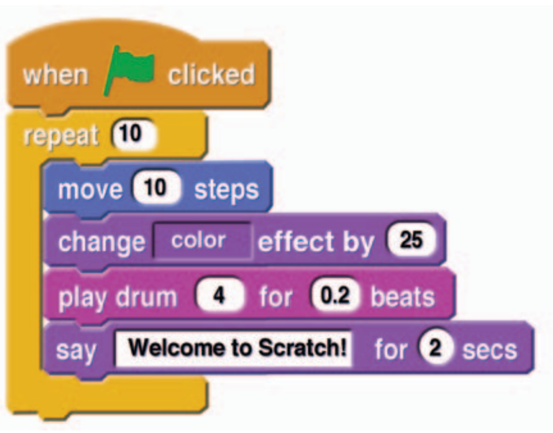
works. Schools largely forgot about programming, some deeming it entirely unnecessary and others labeling it too difficult to teach and learn.

But this is changing. In the past five years, we've seen a newfound interest in bringing back learning and teaching programming on all K-12 levels. But it's digitally based youth cultures, not schools, leading this revival (Kafai & Peppler, 2011). Computers seem to be accessible everywhere, particularly outside school, where children and youth are innovating with technology — often with hand-held devices — to create their own video games, interactive art projects, and even their own programmable clothes through electronic textiles. What's more, the same computers on which they create these items connect them to wider networks of other young users who share common interests and a similar commitment to connecting through making.

Schools may very well take a page from these informal communities of creative production and networked participation. After all, despite this surge of interconnected youth communities, very few youth are using their smart devices — laptop, iPad, iPhone, or Droid — for something other than the mass consumption of commercial media. These digital natives may be able to technically manipulate the latest devices, but their capacity to wield such devices critically, creatively, and selectively is decidedly less potent.

What then is the role of programming in facilitating more productive use of technology? And what is the role of schools in introducing programming to a wider array of youth, particularly given schools' own aborted attempts to teach coding in the past? How will schools address challenges of diversity and equity so prevalent in computing culture? Given these questions facing education as well as the economic viability of this country, we must first understand what computational thinking is, how we can teach it, and why the computational participation of online communities and traditional schools together offers new opportunities to engage students.

An example of Scratch coding



```
when [flag] clicked
repeat (10)
    move (10) steps
    change [color] effect by (25)
    play drum (4) for (0.2) beats
    say [Welcome to Scratch!] for (2) secs
```

Scratch is developed by the Lifelong Kindergarten Group at the MIT Media Lab. See http://scratch.mit.edu

What is computational thinking?

In 2006, Carnegie Mellon professor Jeannette Wing defined computational thinking as all "aspects of designing systems, solving problems, and understanding human behaviors" (2006, p. 6). Wing argued that understanding the world computationally gives a particular lens to understanding problems and contributing to their solutions. Computational thinking — while often strictly associated with computer science — actually is better understood as extending computer science principles to other disciplines in order to help break down the elements of any problem, determine their relationship to each other and the greater whole, and then devise algorithms to arrive at an automated solution. Computational thinking isn't limited to mathematics and the sciences but also applies to the humanities in fields such as journalism and literature.

Thinking like a computer scientist has the potential to better articulate and advance other academic disciplines. But how is computational thinking present and relevant in everyday life? Wing provides several examples. Consider cleaning up and sorting Lego brick pieces. If a child sorts the pieces as "all rectangular thick blocks in one bin," "all thin ones in another," and so on, computer scientists would call this hashing. Of course, most children (and adults) clean up heaps of Legos simply by just dumping them in one big bucket. But imagine if the child wanted to build a bigger project with Legos and needed to construct the project by selecting particular pieces in a set sequence. Looking through a large pile of Lego bricks each and every time would take far longer than looking through bricks organized by size, shape, and even color. Establishing these categories would reduce search time and let the builder concentrate on what he or she wanted to do in the first place: build, not search. That would be especially helpful when building more ambitious and precise structures.

Wing's definition of computational thinking provoked a wide-ranging response among computer scientists and educators concerning what qualifies as digital literacy. What does computational thinking contribute to reasoning and communicating in an ever-increasingly digital world? To what extent do schools encourage systematic problem solving across disciplines, breaking down problems and processes to determine relationships before reassembling? These aren't necessarily new questions for schools (Grover & Pea, 2013). Although computers have been in schools for 30 years, computational thinking hasn't become part of the curriculum. Teaching word processing and how to create PowerPoint presentations don't engage students in the deeper analysis needed to think more creatively and critically (Collins & Halverson, 2009). Most youth have no or very little conception of computer science as a discipline or how it could apply to their daily lives. In short, students need to know not only more about computer science but what it ultimately means to think more systematically in order to more efficiently solve all types of problems.

Teaching computational thinking

So what could computational thinking look like in schools? How could we teach it? The definition of computational thinking as designing systems, solving problems, and understanding human behaviors admittedly provides quite a broad berth here. Several professional groups like the Computer Science Teachers Association and nonprofits like Shodor have developed academic standards and instructional activities to make computational thinking more accessible for K-12 education. Programming has invariably played a role in all proposed curricula. Yet while programming figures prominently, no single programming language is deemed best by all proponents. Whether the language is Java/Java Script, Python, C/ C++, HTML or introductory languages like Scratch and Alice, teaching the underlying concepts conveyed by the language — not the language itself — is what's relevant.

So who is to say that teaching programming in these languages will have any greater success than what we witnessed in the 1980s with Logo's and Pascal's relatively brief foray into schools?

The answer, we argue, is that children have already been using code to create and share. Over the past decade, a plethora of youth-generated web sites have emerged committed to making and sharing programmable media online, be it video games, interactive art projects, or digital stories. Inherently do-it-yourself (DIY) in nature, web sites such as Newgrounds, Planet Kodu, Scratch Online, and Looking Glass (to name a few) encourage youth programming not so much as a learned discipline but as opportunities to create and share online. Within this DIY ethos of individual endeavor mixed with group feedback and collaboration, we see three key shifts in how youth are now learning computer programming:

#1. A shift from code to applications.

Rather than coding exercises for learning about algorithms and data structures, children now learn programming to create specific applications, be they video games or interactive stories. They are engaged by the potential to create something real and tangible that can be shared with others, converting the learning of programming — at least initially — from the study of an abstract discipline to a way of making and being in the world digitally.

#2. A shift from tools to communities.

Happily, the past decade has seen the development of many admirable introductory programming languages that have made coding a more intuitive, personal process. Scratch (http://scratch.mit.edu) and Alice (http://alice.org) are two primary examples. But developers are realizing that tools alone are not enough. Every tool needs an audience and the opportunity to bring like-minded creators together via the Internet. Accordingly, tools like Scratch and Alice now have extensive online communities of millions of young users. The latest version of Scratch — version 2.0 released this past spring — actually now exists entirely online so children can program and share from a single web site, tacitly highlighting the fact that the community of practice effectively has become the key tool for learning to code.

#3. A shift from creating "from scratch" to creating via "remix."

Programming is no longer an individual activity in which source code is hidden and closely guarded. In the spirit of the open-source movement, there is an increasing push to share one's underlying code and encourage participants to sample others' creations for the sake of adjusting and adding to them. With the idea that such openness heightens the potential for innovation, young users embrace sampling and sharing more freely, challenging the traditional top-down paradigm characteristic of computer science and of schools in general.

Broadly speaking, we view the three aforementioned shifts as a social turn, moving from a predominantly individualistic view of technology to one that includes a greater focus on the underlying sociological and cultural dimensions in learning programming and reconceptualizing computational thinking as computational participation.

Getting to computational participation

Who is actually participating computationally is a whole other story. The three shifts above are happening largely outside K-12 schools. Within schools, computer science education remains resolutely top down, focusing on instilling abstract principles before any direct application occurs. Within upper-level high school courses, such as Computer Science Principles, leading with abstraction is understandable given the breadth of topics to be covered. But the near total lack of computational participation in any earlier, introductory technology-based coursework means few students are even considering computer science principles, much less encountering computer science in their K-12 education.

GENDER AND RACIAL DISPARITY IN COMPUTER SCIENCE REPRESENTS A SIGNIFICANT HURDLE. LESS THAN 1% OF AP COMPUTER SCIENCE EXAM TEST TAKERS IN 2011 WERE BLACK AND ONLY 21% WERE FEMALE.

The numbers support this. Only 2,100 of some 42,000 high schools in the U.S. offer an AP computer science course (College Board, 2012). The number of introductory computer science courses has decreased by 17% since 2005. Such a drop is quite literally inexcusable given the Bureau of Labor Statistics' (2012) consistent listing of computer science-related jobs among the fastest growing professions in the country with over 4 million new positions expected by 2020. Gender and racial disparity in computer science represents another significant hurdle. Women make up 56% of all AP test takers, yet only 21% of those who took the AP computer science exam in 2011 were female, and only 29 of the test takers nationwide that year were black — less than 1% of the total.

In the 1990s and even in the early 2000s, these longstanding inequities were widely attributed to the digital divide. Much energy in the 1990s focused on providing access to computers, not curricula or pedagogy, with initiatives such as Net Day dedicated entirely to bringing computers into schools and connecting them to the Internet. Although such initiatives did some good addressing the access issue, there remained what media scholar Henry Jenkins (2006) called the "participation gap" when it came to children's usage of digital media in creative and critical ways.

Debunking the notion that access alone would address the equity issue, technology educator Mark Warschauer and Tina Matuchniak (2010) compared two schools with the same number of computers but in starkly different neighborhoods in terms of socioeconomics. While students had equal access to

Learning with technology has moved beyond the question of access to a question of what one is making and what one is sharing with computers.

computers in both schools, what was taught and what students learned in school differed greatly. Students in the upper socioeconomic neighborhood learned to work creatively and collaboratively with computers, at times even programming, while students in the low-income community were groomed for word processing and simply learning how to technically operate the machinery.

Participation in computing is not only about having access but also having quality curricula and pedagogy. Such quality can occur when computer programming allows children to produce, collaborate, and repurpose content that is personally meaningful. Focusing on computational thinking would remedy the lack of engaging curricula in K-12 technology courses and teach children the concepts and skills to solve problems algorithmically. Computational participation meanwhile focuses on the pedagogical practices and perspectives needed to meaningfully contribute in wider social networks, including but not limited to schooling. It is here, within the wider

All students need to know not only about programming but what it means to think more systematically in order to more efficiently solve all types of problems.

network of creative and critical thinkers, that educators can set new academic and social norms for what it means to meaningfully use technology.

Conclusion

Learning with technology has moved beyond the question of access to a question of what one is making and what one is sharing with computers. Moving from the digital divide to the participation gap has become the driving force toward what we call computational participation. Of course, incorporating computational participation will be no small step for schools, where individual achievement far outweighs group dynamic and collaborative effort in terms of academic achievement. Web 2.0 has taught us the importance of collaboration in facilitating more creative and cost-effective solutions to problems. Access to participation and collaboration in communities of programming is key to learning fundamental concepts and practices. Learning to program is also about learning to participate in the many digital publics and vice versa.

While only a few of us will become computer scientists who will write the code and design the systems that undergird much of our daily life, learning, and leisure, many will encounter the need for some form of programming at some point in our lives. All of us are and will remain users of digital technologies and thus will need at times to be able to critically and constructively examine designs and decisions that went into making them. In terms of the magnitude of what any literacy affords the individual, Paulo Freire estimated that "reading the word is reading the world." We see reading code very much about reading today's world in terms of understanding and having the opportunity to remake it. Schools, their leaders, teachers, and students play a critical role in realizing this opportunity. K

References

Bureau of Labor Statistics. (2012). Employment projections 2010-20. www.bls.gov/emp/

College Board. (2012). AP course audit. https://apcourseaudit.epiconline.org/ledger/search.php

Collins, A. & Halverson, R. (2009). *Rethinking education in the age of technology*. New York, NY: Teachers College Press.

Grover, S. & Pea, R. (2013). Computational thinking in K-12: A review of the state of the field. *Educational Researcher, 42* (2), 59-69.

Jenkins, H., Clinton, K., Purushotma, R., Robison, A.J., & Weigel, M. (2006). *Confronting the challenges of participatory culture: Media education for the 21st century.* Chicago, IL: MacArthur Foundation.

Kafai, Y.B. & Peppler, K.A. (2011). Youth, technology, and DIY: Developing participatory competencies in creative media production. *Review of Research in Education, 35*, 89-119.

Warschauer, M. & Matuchniak, T. (2010). New technology and digital worlds: Analyzing evidence of the equity in access, use and outcomes. *Review of Research in Education, 34* (1), 179-225.

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, *49* (3), 33-35.

Thinkstock/iStockphoto