DOI:10.1145/1538788.1538803

Matthias Felleisen and Shriram Krishnamurthi

Viewpoint
Why Computer Science Doesn't Matter

Aligning computer science with high school mathematics can help turn it into an essential subject for all students.

In March 2008, the College Board (which administers the Advanced Placement (AP) exam) did the unthinkable by reducing a vibrant technology discipline, computer science, to the same level of unpopularity as a dead language, Latin. It achieved this by canceling an AP exam[2] in each area. Although ACM and other organizations provided data on the sustained levels of the other AP computer science exam, these statements mask the relative unpopularity of computer science compared to more traditional mathematical disciplines. Concretely, in 2007, a total of 19,392 students took one of the computer science AP exams, in contrast to 267,160 who took calculus and 96,282 who took statistics.[1]

Perhaps this isn't surprising. The three Rs—reading, 'riting, 'ritmetic—symbolize what matters in U.S. primary and secondary education. Teaching these three essential skills dominates the scholastic agenda in the minds of parents, educators, and legislators. Any new material competes with these core elements; if it isn't competitive, it is marginalized.

Computer science plays such a marginal role. A large part of the problem is due to how computing is portrayed to schools, parents, the people who allocate the education budgets, and the students. The high school curriculum is mired in teaching fashionable programming languages and currently popular programming paradigms. There is great churn in how to teach this complex content to people for whom its complexity is likely to be inappropriate. Never mind that the languages and perhaps even paradigms of today will have evaporated by the time the beginning students graduate.

This trend is not limited to high schools; it is repeated in the introductory college curriculum. Indeed, many high schools are merely reflecting the curricular confusion at the college level. Colleges, in turn, have a problem of their own: declining enrollments in computer science.

When enrollments decline, the leaders of the computer science education community routinely look for saviors: graphics, animation, multimedia, robotics, and games have all been cast in this role. Not that integrating such topics into a course on computing is necessarily bad; but such ideas are frosting, not essentials. This search for saviors pervades thinking about introductory college curricula, and much of it percolates to thinking at the secondary school level in the form of AP and pre-AP curricula. Others, wanting to offer alternatives, act embarrassed about programming, which is our field's single most valuable skill, and seek to marginalize it (for example, see the November 2005 *Communications* column titled "Recentering Computer Science"). Meanwhile, ACM's own press releases attempt to downplay the gravity of the situation.[3]

What our community should really aim for is the development of a curriculum that turns our subject into the fourth R—as in 'rogramming—of our education systems. This can not only address high school curricular concerns but can also become an integral part of general education and distribution requirements in college. One way of achieving this goal is to align computing through programming with one of the three Rs and to make it indispensable. An alignment with mathematics is obvious, promising, and may even help solve some problems in mathematics education.

Mathematics and Programming

All students must enroll in mathematics for most of their school years. Many of them already struggle with it. Does hitching programming to mathematics make any sense? Consider high school algebra. Bewildering exercises about flies flitting between trains do nothing to help students understand that algebra can actually be put to work. Algebra textbooks try hard to enliven their content with high-gloss color photographs, which we can immediately recognize as symptoms of failure, not a reprieve from it. In part, school algebra appears fundamentally dull to students because it appears to be all about numbers, which play at best a small role in the media-rich, interactive lives of students. We propose the paradigm of *imaginative programming*, which weds programming to al-

Table 1.

1	2	3	4	5	...	*x*
1	4	9	16	?	...	?

Figure 1.

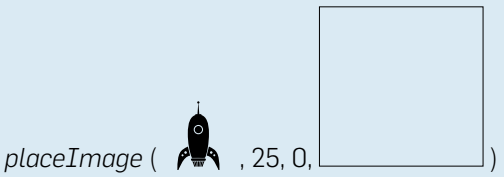


Figure 2.

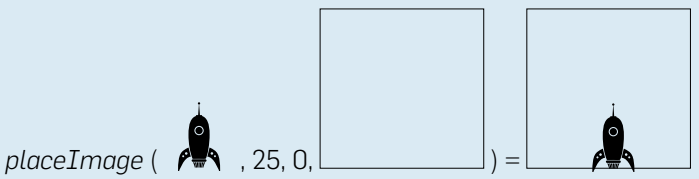


Table 2.

0	1	2	3	...	*t*
0	10	20	30	...	*height*(*t*) = ?

gebra through the use of rich media. By embracing these media, we can engage students while synergistically meeting the needs of math teachers. Indeed, we have already seen our curricular approach, described below, help students raise their algebra scores.

How Would This Work?

Let's make this vision concrete. Algebra textbooks contain exercises that ask students to determine the next entry in a table, such as Table 1, or to create a general "variable expression" that computes any arbitrary entry of the table. In Table 1, students are expected to say that 5 comes with 25 and x comes with $x \cdot x$. We might even hope to teach the student the notation $f(x) = x \cdot x$, but why would they care? This function means nothing to them outside their homework.

We can, however, show these students that modern arithmetic and algebra do not have to be about numbers alone. They can just as well involve images, strings, symbols, Cartesian points, and other forms of "objects." For example, Figure 1 is an arithmetic expression involving images in addition to numbers. The operator *placeImage* takes four arguments: an image (the rocket), two coordinates, and a background scene (the empty square). The value of such an expression is just another image, as shown in Figure 2. That is, algebraic expressions can both consume and compute pictorial results, enabling students to manipulate images using algebra.

Imagine asking students to determine a rising rocket's altitude after a given period of time. We could start with a table and the simplifying assumption that rockets lift off at constant speeds, as shown in Table 2. Because students understand that functions can produce images, not only numbers, we could even express this exercise as a problem involving a series of images and asking students to determine the next entry in Table 3.

By asking the student to define the function *rocket*, we are asking for a "variable expression" that computes any arbitrary entry of the table—just as we asked in the case of numbers. We would hope to get an answer like the one shown in Figure 3. A teacher may even point out here the possibility of reusing the results of one mathematical exercise in another, as shown in Figure 4. Students thus see the composition of functions and expressions, all while using mathematics as a programming language. In addition, students are motivated to learn more about mathematics and physics to improve these little programs.

With one more step, students can visualize this mathematical series of images and get the idea that constructing such mathematical series can be an aesthetically pleasing activity:

showImages(rocket, 28)

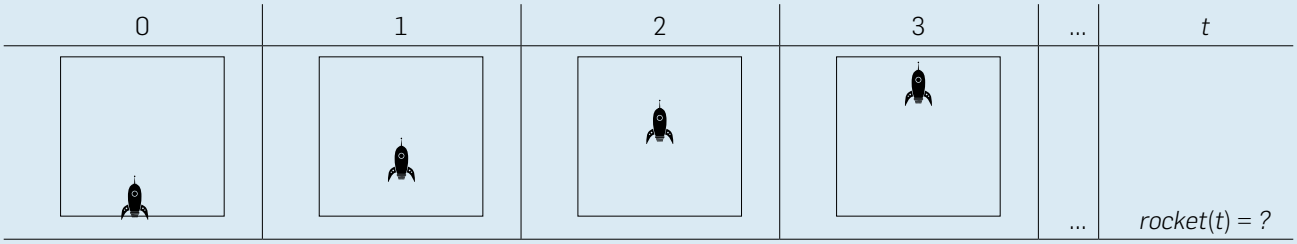
This expression demands that *rocket* be applied to 0, 1, 2, 3, 4, 5, etc., and that the result be displayed at a rate of 28 images per second. (Note how *showImages* furtively introduces the idea of functions consuming functions, because its first parameter—*rocket*—is itself a function.) Now we can tell students that making animated movies is all about using the "arithmetic of images" and its algebra.

Does It Really Work?

Readers shouldn't be surprised to find out that what we've described and illustrated here isn't just imagination or a simple software application for scripting scenes. A form of mathematics can be used as a full-fledged programming language, just like Turing Machines. In such a language, *even the design and implementation of interactive, event-driven video games doesn't take much more than algebra and geometry*. As students develop such programs they "discover" many concepts on their own simply because they want to add luster to their games—and, to formulate their improvements, they learn new mathematics and physics.

We have field-tested the beginnings of such a curriculum in the context of our TeachScheme! project for the past five years with a family of teaching languages that support images as first-class values. These languages

Table 3.



are all based on Scheme, but restrict the full language to protect students from its dark corners; the languages grow in sophistication with the students' understanding. The languages are implemented in DrScheme, a pedagogic integrated development environment (IDE). The rocket simulator example described in this Viewpoint is our "Hello World" program. Students at all levels—colleges, high schools, and middle school—react favorably to this curriculum. We also have numerous reports that students improve their performance in mathematics. In addition, formal evaluation shows the extremely positive impact this curriculum (see http://www.teach-scheme.org/ and http://www.bootstrapworld.org/) has on the way educators perceive computing and programming.

At the college level, this course follows a natural progression of programming on lists, trees, documents, graphs; abstraction; programming with first-class functions and accumulators; generative recursion; stateful objects; and many more computer science concepts. We have also worked out the transition to a second course, in Java, that builds on this knowledge. Preliminary field tests validate our conjecture that the transitions are reasonably smooth and never demand a fresh start. [a]

Figure 3.

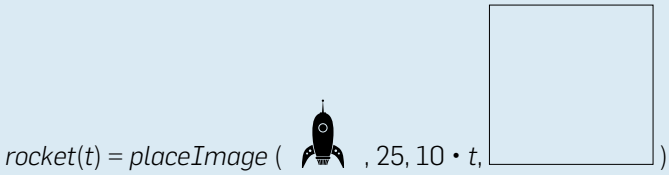
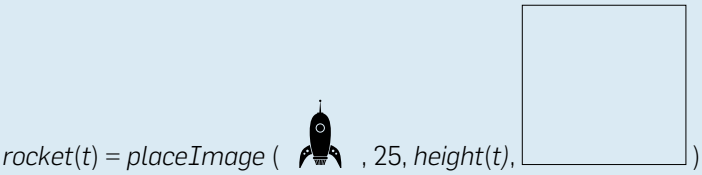


Figure 4.



What Makes It Work (and What Doesn't Work)

Any attempt to align programming with mathematics will fail unless the programming language is as close to school mathematics as possible. The goal of an alignment is to transfer skills from programming to mathematics and vice versa. While students quickly grasp small differences in syntax, they will mentally block if the *notion* of, say, "function" in programming significantly differs from the notion of "function" in algebra. Of course, some attributes of our approach are essential and others are accidental. We conjecture that, in addition to a language in harmony with mathematics, imaginative programming demands two more ingredients: the algebraic manipulation of images and symbolic data; and minimal overhead in the IDE for using these features.

As computer science educators, we must also demand a smooth, continuous path from imaginative programming to the engineering of large programs; otherwise beginning programming won't create skills that transfer to our discipline. Our decade-long curricular effort has been building one such path; others may produce different transitions.

Conversely, our community must realize that minor tweaks of currently dominant approaches to programming won't suffice. Even masking the `public static void main` of Java hides little when the body of the corresponding method has little to do with the mathematical formulation of a function. The complexity of object-oriented programming bears little fruit here: it makes no sense to teach students how to engineer the structure of large systems when they are yet to write any programs with a complexity worth structuring.

Functional programming languages, such as Haskell, ML, and Scheme, suffer from different, but equally bad problems. These languages are far too complex for novices; except for DrScheme, none support images as first-class forms of data or provide pedagogical IDEs. Their type systems are

a This material is being used in inner-city programs at several urban middle schools. It is in use at dozens of high schools. It has also been deployed at several universities of all sizes and styles in the U.S. and other countries. Representative institutions include Adelphi, Brown, Chicago, Rice, Northeastern, and Waterloo. Two German textbooks based on our material have appeared over the past two years. In India, a major corporation uses the material for its "bootcamp" for new employees. Our primary textbook has been translated into Spanish, Polish, Chinese, and (partially) German.

fascinating mazes suitable for exploration by researchers and hackers, but dispatch the average student in horror after just a few interactions.

The ideal language and the IDE for imaginative programming are still to be designed. If we develop them, educational stakeholders will see how programming provides students with an interactive, engaging medium for studying and exploring mathematics. Thus, it may just turn computing into an indispensable subject for all students, right up there with the other three Rs.

Crossroads

Our community is at a crossroads when it comes to tackling our educational needs. We can continue to search for more saviors and hope that somehow, somewhere computing will receive the respect it deserves. Or we can try to help ourselves and others by turning a piece of the core school curriculum into something that students find appealing and even exciting. Our proposal is just one way of moving in this direction. We don't know whether it will succeed at large scales; and we can't know yet what else our community will discover once we start the search. What we do know is that the savior-driven ways have had their chance for many years, and they have failed.

Acknowledgments

We thank Robby Findler and Matthew Flatt for their partnership over these dozen years. Kathi Fisler helped hone our thoughts in this essay. Emmanuel Schanzer turned our ideas into the Bootstrap middle school curriculum, and his efforts have greatly influenced our thinking.

References

1. College Board. AP: Exam Grades: Summary Reports: 2007; http://www.collegeboard.com/student/testing/ap/exgrd_sum/2007.html.
2. College Board. Important Announcement about AP Computer Science AB: Important Change for the 2009–2010 Academic Year; http://apcentral.collegeboard.com/apc/public/courses/teachers_corner/195948.html.
3. USACM. AP Computer Science is NOT Going Away; http://usacm.acm.org/usacm/weblog/index.php?p=593.

Matthias Felleisen (matthias@ccs.neu.edu) is Trustee Professor in the College of Computer Science at Northeastern University in Boston, MA.

Shriram Krishnamurthi (sk@cs.brown.edu) is an associate professor of computer science at Brown University in Providence, RI.