Decoding the Value of Computer Science

By Kevin Carey

In *The Social Network*, a computer-programming prodigy goes to Harvard and creates a technology company in his sophomore dorm. Six year later, the company is worth billions and touches one out of every 14 people on earth.

Facebook is a familiar American success story, with its founder, Mark Zuckerberg, following a path blazed by Bill Gates and others like him. But it may also become increasingly rare. Far fewer students are studying computer science in college than once did. This is a problem in more ways than one.

The signs are everywhere. This year, for the first time in decades, the College Board failed to offer high-school students the Advanced Placement AB Computer Science exam. The number of high schools teaching computer science is shrinking, and last year only about 5,000 students sat for the AB test. Two decades ago, I was one of them.

I have never held an information-technology job. Yet the more time passes, the more I understand how important that education was. Something is lost when students no longer study the working of things.

My childhood interest in programming was a product of nature and nurture. My father worked as a computer scientist, first in a university and then as a researcher for General Electric. As a kid, I tagged along to his lab on weekends, watching him connect single-board DEC computers into ring networks and two-dimensional arrays, feeling the ozone hum of closet-sized machines. By my adolescence, in the mid-1980s, we had moved to a well-off suburb whose high school could afford its own mainframe. That plus social awkwardness meant many a night plugged into a 300-baud modem, battling other 15-year-olds in rudimentary deep-space combat and accumulating treasure in Ascii-rendered dungeons without end.

Before long I wanted to understand where those games came from and how, exactly, they worked. So I took to programming, first in Basic and then Pascal. Coding taught me the shape of logic, the value of brevity, and the attention to detail that debugging requires. I learned that a beautiful program with a single misplaced semicolon is like a sports car with one piston out of line. Both are dead machines, functionally indistinguishable from junk. I learned that you are good enough to build things that do what they are supposed to do, or you are not.

I left for college intending to major in computer science. That lasted until about the fifth 8:30 a.m. Monday class, at which point my enthusiasm for parties and beer got the upper hand, and I
switched to the humanities, which offered more-forgiving academic standards and course schedules. I never touched Pascal again.

Fortunately, other students had more fortitude. They helped build and sustain the IT revolution that continues to make America the center of global innovation. But the number of people going this way is in decline. According to the Computing Research Association, the annual number of bachelor's degrees awarded in computer science and computer engineering dropped 12 percent in 2009. When Zuckerberg started Facebook, in 2004, universities awarded over 20,000 computer degrees. The total fell below 10,000 last year.

This "geek shortage," as Wired magazine puts it, endangers everything from innovation and economic growth to national defense. And as I learned in my first job after graduate school, perhaps even more.

There, at the Indiana state-budget office, my role was to calculate how proposals for setting local school property-tax rates and distributing funds would play out in the state's 293 school districts. I did this by teaching myself the statistical program SAS. The syntax itself was easy, since the underlying logic wasn't far from Pascal. But the only way to simulate the state's byzantine school-financing law was to understand every inch of it, every historical curiosity and long-embedded political compromise, to the last dollar and cent. To write code about a thing, you have to know the thing itself, absolutely.

Over time I became mildly obsessed with the care and feeding of my SAS code. I wrote and rewrote it with the aim of creating the simplest and most elegant procedure possible, one that would do its job with a minimum of space and time. It wasn't that someone was bothering me about computer resources. It just seemed like the right thing to do.

Eventually I reached the limit of how clean my code could be. Yet I was unsatisfied. Parts still seemed vestigial and wrong. I realized the problem wasn't my modest programming skills but the law itself, which had grown incrementally over the decades, each budget bringing new tweaks and procedures that were layered atop the last.

So I sat down, mostly as an intellectual exercise, to rewrite the formula from first principles. The result yielded a satisfyingly direct SAS procedure. Almost as an afterthought, I showed it to a friend who worked for the state legislature. To my surprise, she reacted with enthusiasm, and a few months later the new financing formula became law. Good public policy and good code, it turned out, go hand in hand. The law has reaccumulated some extraneous procedures in the decade since. But my basic ideas are still there, directing billions of dollars to schoolchildren using language recorded in, as state laws are officially named, the Indiana Code.

A few years later, I switched careers and began writing for magazines and publications like this one. It's difficult work. You have to say a great deal in a small amount of space. Struggling to build a whole world in 3,000 words, I thought back to stories my father would tell of consulting on the Galileo space-probe project. To make it work with 1970s technology and withstand the rigors of deep space, he and his fellow programmers had to fit every procedure—"take pictures," "fire rockets," "radio NASA back on Earth"—into a machine with 64K bytes of what we now
call RAM. That's about 1/60,000th of what you can buy in a cheap laptop at Best Buy today. So every program on Galileo was polished to a gleam.

Good editors know there is no Moore's law of human cognition that would double our ability to retain and process information every 18 months. Instead, the technology-driven surfeit of modern information has made the need for clarity and concision more acute.

My editors rarely said a word about words, in the sense of how to phrase an idea. The real work was in structure, in constructing an unbroken chain of logic, evidence, and emotion that would bring readers to precisely where I wanted them to be. I learned to minimize the number of operating variables (people). I also learned that while some talented writers can get away with recursive scene-setting, it is usually best to start at the beginning and finish at the end. I discovered that skilled writers, like programmers, embed subtle pieces of documentation in their prose to remind readers where they are and where they are going to be.

Writing, in other words, is just coding by a different name. It's like constructing a program that runs in the universal human operating system of narrative, because, as Joan Didion once said, we tell ourselves stories in order to live. Authors adhere to a set of principles as structured in their own way as any computer language. Publications worth writing for insist on Galilean standards of quality and concision.

Computer science exposed two generations of young people to the rigors of logic and rhetoric that have disappeared from far too many curricula in the humanities. Those students learned to speak to the machines with which the future of humanity will be increasingly intertwined. They discovered the virtue of understanding the instructions that lie at the heart of things, of realizing the danger of misplaced semicolons, of learning to labor until what you have built is good enough to do what it is supposed to do.

I left computer science when I was 17 years old. Thankfully, it never left me.

Kevin Carey is policy director of Education Sector, an independent think tank in Washington.