2

PROVIDING STUDENTS WITH COMPUTATIONAL LITERACY FOR LEARNING ABOUT EVERYTHING

Mark Guzdial

LEARNING ABOUT COMPUTING IS LEARNING ABOUT PROGRAMMING

Learning about computing almost always requires learning about programming. There have been some brilliant people, like Alan Turing and John von Neumann, who could think about computing without a language or notation, but those people are rare. It is analogous to learning mathematics, including addition, subtraction, and multiplication, without writing digits like "34.9" or symbols like "+."

Programming is defining a computation, something that a computer can do. A program describes a process. A program can be specified in any notation, so we should pick one that best suits the programmer and the domain. The most popular programming languages today are demanding, requiring students to use complex cognitive skills such as abstraction and decomposition of a problem into subcomponents. Programming does not have to be so complex and overwhelming. A simple programming language can still be effective for learning. Programming is a powerful tool for helping students learn in many different domains. I argue in this chapter that *providing students with the ability to program is providing them with a literacy that can be an advantage in learning about everything else*.

The term *computer science* first appeared in print in the *Journal of Engineering Education* in 1961 in an article by George Forsythe (Knuth 1972).

Forsythe described (in 1968) that he saw computation as a "general-purpose mental tool" that would "remain serviceable for a lifetime." Explicitly, *computer science* was defined as something that students could use to aid in their thinking and their learning, especially in STEM (science, technology, engineering, and mathematics) classes. In this chapter, I argue that the value of learning programming is even greater than what Forsythe described.

There are many possible benefits to students learning to program. The first section of the chapter lists many of these, ending with the most powerful—programming as a new kind of literacy. The next section explains why the computer can help with learning everything else, because it is the "master simulator." Finally, I argue that even a simple programming language can have enormous advantage in learning. We don't need all the power of C, Scheme, or Logo to learn with programming as a literacy.

WHY SHOULD STUDENTS LEARN TO PROGRAM?

Learning to program does not impart to the learner general problem-solving skills. There have been several studies looking for transfer from teaching programming to general problem-solving skills. Probably the first study investigating this claim was done by Roy Pea and Midian Kurland in 1984. David Palumbo completed a meta-review of the research relating learning programming and learning problem-solving (1990). Since then, the topic has been revisited, but I read Palumbo's results as painting a picture of programming as an opportunity to teach problem solving rather than an experience where problem-solving is learned automatically.

It is possible to teach problem-solving using programming, but problem-solving skills are not the automatic and direct result of learning to program (Grover and Pea 2013). Sharon Carver showed how to teach problem-solving with programming (Carver 1988). She wanted students to learn debugging skills, such as being able to take a map and a set of instructions and then figure out where the instructions are wrong. She taught those debugging skills by having students debug Logo programs. Students successfully transferred those debugging skills from Logo programming to the map task. That's significant from a cognitive and learning sciences perspective. But her students didn't learn much programming;

she didn't need much programming to teach that problem-solving skill. Other studies have found similar results (Grover, Pea, and Cooper 2015; Kalelioglu and Gülbahar 2014).

Fortunately, there are many other benefits of learning to program. These are described in the paragraphs that follow.

TO UNDERSTAND THE WORLD IN WHICH THEY LIVE

Simon Peyton Jones argued that computer science is a science like all the others (Peyton Jones 2013). We teach chemistry to students because they live in a world with chemical interactions. We teach biology because they live in a world full of living things. We teach physics because they live in a physical world. We should teach computer science because they live in a digital world.

Students live in a world where secret messages can be hidden inside of pictures and where machines can be infected with viruses. They live in a world where they own many computers, some of which do nothing more sophisticated than control their microwave oven. They do not need to know how all of this works at a level that they could build it (although they may want to). They do need to understand enough to troubleshoot the computing in their lives: for example, to know that it is unlikely for the internet to ever "break," but the router in their home can fail. They need to understand enough to protect themselves: for example, why running any arbitrary program downloaded from the internet may be dangerous for their security. They also need to understand that they can make their own apps and games and that anyone with any computer can invent something that is world changing. Students should know the basic principles of how their world works.

TO USE COMPUTERS MORE EFFECTIVELY

We all use computers ubiquitously, from the cellphones in our hands to the laptops on which we work. Does knowing how the computer works lead to more effective use of the computer? Are people who program less likely to make mistakes with software? Are they more resilient in bouncing back from errors? Can programmers solve computing problems (those that happen in applications or with hardware, even without programming) more easily?

I bet the answer is yes, but I am unaware of research results that support that argument. There are likely common elements to mental models that are used to understand the computational systems with which we interact. Some of those common elements may include the causal and repeatable nature of computers, which is unlike our everyday experience (e.g., your PowerPoint animations likely work exactly the same way every time). Programming may be a way to learn those common elements explicitly and efficiently.

TO INFLUENCE THEIR WORLD

The default behavior of users with computers is to consume. We consume books, videos, music, and commentary in an endless stream or scroll. The promise of programming is to turn digital consumers into digital producers who can use computing to have an effect on the world.

Yasmin Kafai calls this promise *computational participation* (Kafai 2016), and Tissenbaum, Sheldon, and Abelson (2019) call it *computational action*. The computer's connectivity, malleability, and representational power give students the ability to make digital products and share them widely. From YouTube videos to new apps, the computer provides a rich medium for creativity and a far-reaching distribution mechanism.

The question of the role of programming changes if we reframe programming. Imagine if programming was *not* a complex and hard-to-learn activity. What if learning to program was like learning to use a drawing app, a photo editing tool, or a video editor. If we think of programming as defining a process for someone else to use, then teaching students to program is giving them another way that they can create digital artifacts (i.e., stored and executable process) and share them with the world.

TO STUDY AND UNDERSTAND PROCESSES

Alan Perlis (first Association for Computing Machinery [ACM] Turing Award laureate) argued in 1962 that everyone on every campus should learn to program (Perlis 1962). He said that computer science is the study of *process*. He contrasted learning computer science with learning calculus.

Calculus is the study of *rates,* which is important for many disciplines. Perlis argued that *all* students need to learn about process, from managers who work on logistics to scientists who try to understand molecular or biological processes. Programming automates process, which creates opportunities to simulate, model, and test theories about processes at scale. Abelson, Sussman, and Sussman (1996) stated that mathematics is about formalizing declarative knowledge ("what is"), while programming is about formalizing imperative knowledge ("how to").

Perlis was prescient in predicting computational science and engineering. Today, people play "what-if" games with spreadsheets daily. We use computing to track our weather and our packages. Most professionals use a computer to explore models. The ability to construct models and test hypotheses by executing those models is one of the most powerful abilities that a computer can provide us. It is especially powerful because it extends a basic human capability—to imagine a possible future world. The computer can allow us to realize this world (at a level of fidelity that makes sense for our needs) and test it in simulation. Testing our imagined worlds is difficult to do at the level of precision that a computer affords.

TO HAVE A NEW WAY TO LEARN SCIENCE AND MATHEMATICS

Mathematics places a critical role in understanding our world. The power of mathematics in science is obvious, but the adoption of mathematics in society may be even more influenced by its importance for business. Without a doubt, the world runs on numbers.

Our notation for mathematics is mostly static equations representing models about the world. Increasingly, we are finding that representing code is different and gives us new insights. This is what Andy diSessa has been saying in his calls for computational literacy (2001). Bruce Sherin (2001), Idit Harel (1990), Yasmin Kafai (2014), Uri Wilensky (2016), and many others have shown us how code gives us a powerful new way to learn science and mathematics. Bootstrap:Algebra (Schanzer et al. 2015) teaches algebra with computing. Every student of mathematics should also be a student of programming, because it provides a different, dynamic notation for understanding mathematical ideas. When the programming context is tied to a real application (from image manipulation to video games), the computation can help to concretize the mathematical concepts (Wilensky 1991), which can make them more engaging and easier to learn.

TO BE ABLE TO ASK QUESTIONS ABOUT THE TECHNOLOGICAL INFLUENCES ON THEIR LIVES

C. P. Snow (1962) also argued for everyone to learn computing in 1962, but with more foreboding. He correctly predicted that computers and computing algorithms were going to control important aspects of our lives. He said, "I am asking whether we are now running into a position where only those who are concerned with the computer, who are formulating its decision rules, are going to be knowledgeable about the decision," and "It is not only that I am afraid of misjudgments by persons armed with computing instruments; it is also that I am afraid of the rest of society's contracting out, feeling that they no part in what is of vital concern to them because it is happening altogether incomprehensibly and over their heads." Snow would likely have agreed with Cathy O'Neil's premise in *Weapons of Math Destruction* (2016), that computer algorithms are not inherently objective and that programmers' biases may influence their judgments.

If we don't know about computing, we have "contracted out," in Snow's terms. We don't even know what to ask about the algorithms that are controlling our lives. It shouldn't be magic. Even if you're not building these algorithms, simply knowing about them gives you power. C. P. Snow argues that you need that power.

AS A JOB SKILL

The most common argument for teaching computer science in the United States is as a job skill. The original Code.org video (2013) argued that everyone should learn programming because we have a shortage of programmers. While the need for more programmers is important for supporting our technological society, that is not a good enough reason to put programming in front of every student. Moreover, that's not a reason to bear the enormous cost to change our school systems so that we have enough teachers to teach all those students. Not everyone is going to become a software developer, and it does not make any sense to train everyone for a job that only some will do.

But if you think about computing as a literacy, and not as a career, it becomes more clear that computing will be an important *component* job skill for many. Some fifteen years ago, we could already see that the ratio of professional software developers to people who program just as *part* of their job was somewhere between 1:4 and 1:9 (Scaffidi et al. 2005). A more recent analysis shows that, for the same job category, workers (who are not software developers) who program make higher wages than those comparable workers (in the same job category) who do not (Scaffidi 2017). Learning to program gives students new skills that have value in the economy.

Today, not everyone has access to computing education. It tends to be centralized in more urban/suburban and more affluent schools. Even when it's available, it is mostly White and Asian males taking the class (Margolis et al., 2017; Parker and Guzdial 2019). It is a social justice issue if we do not make this economic opportunity available to everyone.

TO DEVELOP A NEW LITERACY

Alan Kay and Adele Goldberg made the argument in the 1970s that computing is a whole new medium. In fact, it is humans' first meta-medium—it can be all other media, and it includes interactivity so that the medium can respond to the reader/user/viewer (Kay and Goldberg 1977). Computing gives us a new way to express ideas, to communicate to others, and to explore ideas. Everyone should have access to this new medium.

Kay (1977) described what the experience of using the computer as a literacy should be like: "Computer literacy is a contact with the activity deep enough to make the computational equivalent of reading and writing fluent and enjoyable.'" We can use Kay's perspective to contrast programming and textual literacy. We can and do study reading and writing for their own sake: for example, we read classics of literature and learn to compose our own essays. For most of us, the greatest power of reading and writing is that *every day* it enables us to express ideas, to communicate with others, and to understand our world. Literacy supports and affects how we learn. Programming can be studied for itself, and there are obviously full-time, professional programmers—just as there are full-time, professional writers. But programming can also be an everyday skill that can inform the way we think and communicate.

The computer's great power as a form of literacy is that it doesn't have to look like a computer. Kay (1995) pointed out that the computer as meta-medium could be anything else: "The computer is the greatest 'piano' ever invented, for is it the master carrier of representations of every kind. The heart of computing is building a dynamic model of an idea through simulation." The computer can be anything, which makes it a powerful tool for learning about everything. The most powerful aspect of the computer is the ability to encode models and execute them as simulations.

As Sherin (2001) demonstrated when he taught physics with Boxer, the computer provides a modeling capability different than equations. Algebraic equations are useful for describing *balance*. Given all but one of the variables in the equation, we can manipulate the equation to compute that last variable. Computer programs typically do not work the same way. Rather, computer programs can represent *causality*. Students learning a programming model of physics learn about how acceleration influences velocity and velocity influences position (Guzdial 1995)—a causal chain that is not obvious in kinematics equations.

THE COMPUTER AS A TOOL FOR LEARNING EVERYTHING

When computers were first being developed as tools for learning, the goal wasn't learning computer science. From Kemeny and Kurtz developing Basic, to Papert, Solomon, Feurzeig, and Bobrow developing Logo, the goal was using the computer to learn about *something else* (Guzdial and du Boulay 2019). Kemeny and Kurtz wanted everyone on campus to be able to use computing in their work. Papert and the Logo developers wanted students to learn about poetry, mathematics, and artificial intelligence.

In their seminal work "Personal Dynamic Media," Kay and Goldberg showed their new Smalltalk system being used in a wide variety of disciplines, with representations that matched the discipline. They used the new graphical user interfaces to represent circuit diagrams, music, art, animations, and a simulation of a hospital. Today, we recognize that each discipline has its own representations and ways of communicating, which is called *disciplinary literacy* (Moje 2015). The computer is powerful for teaching in all disciplines, in part, because it can support disciplinary literacy.

The interface and language of the computer doesn't have to look the way that computer scientists want it to look. We can adapt the language and interface to use the representations and abstractions of the domain. We want students to learn abstractions that are powerful and generalize, but these need not be abstractions that are native to the computer. There is nothing sacred about FOR loops, bits and bytes, or arrays and linked lists. Many domains have powerful abstractions. We can use the computer to teach any of those, to adapt to any of those abstractions, and to represent them in an authentic way.

HOW MUCH PROGRAMMING DOES A STUDENT NEED FOR LITERACY?

Programming languages are growing in size and complexity. The definition of equality (==) in JavaScript is a list of twenty-two dense rules (ECMA 2011), and that is one of the most basic operators. The number of primitives and the sizes of the libraries grow with every new release of a language. To "learn Python" is a significant challenge, one that can take years to achieve. Certainly, we cannot expect students to learn *all* of *any* language to be literate. So, how much programming does a student really need to be expressive and to learn?

Scratch is likely the most successful programming environment ever developed for children (Maloney et al. 2008, 2010), with tens of millions of users around the world. Empirical studies of students using the block-based programming language show that most students use very few of the capabilities of the language (Fields, Kafai, and Giang 2017). Most loops are simply forever loops. Few students use any Boolean expressions at all. Students don't need to know and use much programming to find Scratch compelling. Even a small bit of programming has expressive power that draws in tens of millions of students. What is likely more important than the Scratch programming language are the environment and community in which it is embedded.

Bootstrap:Algebra is a powerful way to teach algebra through programming (Schanzer et al. 2015). Students build video games by writing equations that describe the current frame of the video in terms of the previous frame, then translate those equations into code. The analysis process that students are taught in Bootstrap:Algebra helps them in solving word problems in algebra (Schanzer et al. 2018). But students don’t actually use much programming when building their video games. There is no explicit repetition (iteration or recursion). Students can improve their learning of algebra without learning everything that is in a modern programming language. Even a small bit of programming has power in enabling powerful learning outside of computing.

Of course, there is a purpose for all those other programming language features that aren’t used in Scratch or aren’t taught in Bootstrap:Algebra. The programming needs are dependent on the students’ goals. The important point is that students do not need to know *everything* in order to learn enough to gain benefits of computational literacy.

Consider a comparison with textual literacy. There are professionals who write for a living: for example, those who produce news stories or novels. Most people find value in writing even if they do not write for newspapers or publishers. Every day, people find value in writing letters and grocery lists with less sophisticated words or grammatical constructs. When people are learning a foreign language, they can often achieve basic communication with a limited vocabulary and few verb tenses. Similarly, there is value in even a small bit of programming.

WHY AREN’T WE THERE YET?

Over the last decade, the United States has made dramatic progress in increasing access to computing education. For example, in Georgia, 43 percent of high schools offer computer science classes (Parker and Guzdial 2019). However, only 1 percent of Georgia high school students take any of those computer science classes. In Indiana, 33 percent of schools offer computer science, but only about 2.5 percent of students ever take a computer science class (Guzdial 2019; Guzdial and Arquilla 2019; Parker and Guzdial 2019).

The reasons are complicated why students are still avoiding computer science, even when they have access to computing education. Certainly, one of the explanations is that not all computing education experiences are high quality. Some afterschool programs and internships dissuade students from continuing in computing (Weston et al. 2019). A more compelling explanation is that students do not see that computing is a pathway to achieving their goals (Lewis et al. 2019). Students who leave computing have a very different perception of the field than those who stay in computer science (Biggers et al. 2008).

One solution to give more students access to computing education is to find new ways to integrate computing across the curriculum. The idea is to follow the lead of Bootstrap:Algebra to find ways that programming can enhance learning in other subjects. If we can't convince students to come to programming and computational literacy, maybe we can bring programming to them and provide computational literacy to support the learning that students *are* interested in.

REDESIGNING PROGRAMMING FOR MICROWORLDS: TASK-SPECIFIC PROGRAMMING

Microworlds are one of the great inventions for using programming to teach a wide range of subjects. The idea of microworlds is to provide a limited subset of the programming environment with tailored operations that match the domain of the microworld. Seymour Papert (1980) first defined microworlds as a "subset of reality or a constructed reality whose structure matches that of a given cognitive mechanism so as to provide an environment where the latter can operate effectively. The concept leads to the project of inventing microworlds so structured as to allow a human learner to exercise particular powerful ideas or intellectual skills." Andrea diSessa (with Hal Abelson) built on this idea in Boxer (diSessa and Abelson 1986) and said in his book *Changing Minds* (diSessa, 2001): "A microworld is a type of computational document aimed at embedding important ideas in a form that students can readily explore. The best microworlds have an easy-to-understand set of operations that students can use to engage tasks of value to them, and in doing so, they come to understanding powerful underlying principles. You might come to understand ecology, for example, by building your own little creatures that compete with and are dependent on each other."

Typically, a microworld is built on top of a general-purpose language: for example, Papert used Logo and diSessa used Boxer. Thus, the designer of the microworld could assume familiarity with the syntax and semantics of the programming language and perhaps some general programming concepts like mutable variables and control structures. The problem here is that with Logo and Boxer, like any general-purpose programming language, it takes time to develop proficiency. They are large and complex things to learn, and learning those can get in the way of the powerful ideas or intellectual skills that Papert and diSessa are interested in.

Task-specific programming (TSP) aims to provide the same easy-to-understand operations for a microworld, but with a language and environment designed for a particular purpose. The task-specific programming language (TSPL) is purposefully limited in the abstractions and concepts needed for the tasks or explorations in the microworld so that programming becomes much easier to learn than a complete programming language. Some task-specific programming languages have been usable in only five to ten minutes (Chasins et al. 2018). The ease of use makes it possible to think about learning different concepts with different microworlds, that is, different task-specific programming languages. Perhaps an elementary or secondary school student might encounter several different TSPLs in a single year.

AN EXAMPLE TASK-SPECIFIC PROGRAMMING ENVIRONMENT

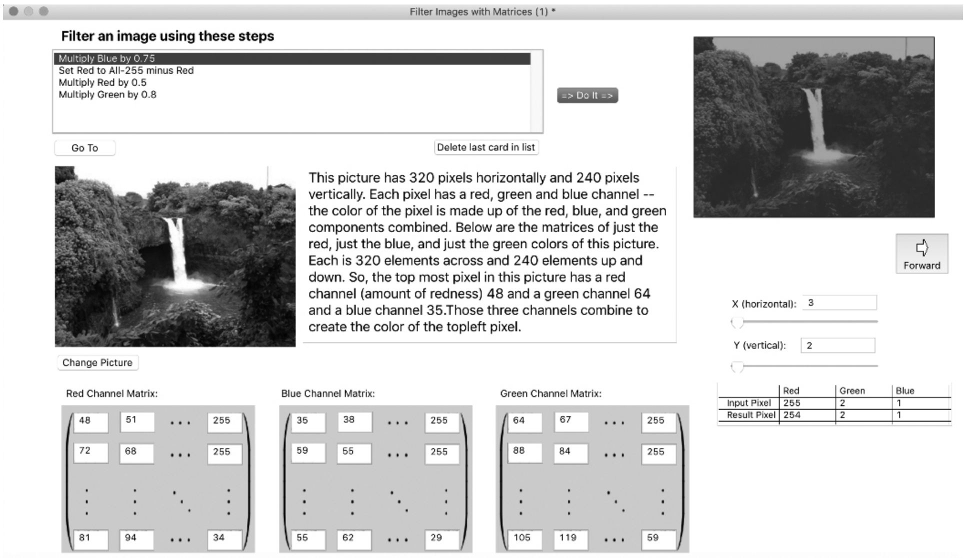
The domain for the following example task-specific programming environment is precalculus. The operations in this prototype environment are the simple matrix transformations taught in many precalculus curricula—matrix addition and subtraction and scalar multiplication. The concrete purpose in this microworld is the creation of image filters. The point of this prototype is to engage students in practicing the intellectual skills of matrix manipulation by engaging them in developing image filters. Image filters become the concrete purpose for learning the abstraction of matrix manipulation.

Figure 2.1 is the main screen for the prototype. Students see a picture (*left-hand side*) that is decomposed into matrices representing the red channel of the pixels in the picture (*bottom left*), and the green and blue channel matrices next to that. A set of matrix transformations is listed at the top left—this is the *program* that, applied to the input picture (*on left*), produces the output picture (*on right*). The Change Picture button changes the input picture so that the students can apply the operations to different pictures to see that the program processes an arbitrary picture to generate a similar image effect on all pictures.

The matrix transformations listed in figure 2.1 are a program, but the language is not typed (as in a textual programming language) nor assembled like a jigsaw puzzle (as in a block-based programming language). Instead, the statements are constructed with a purpose-built editor that is grounded in the disciplinary literacy of precalculus. Each matrix transformation is created and edited on a screen like in figure 2.2.

There are two possible transformations, which are selected by radio button:

- The red, green, or blue matrices can be redefined ("set") as the sum or difference between four matrices: red, green, blue, or a matrix where every value is 255. The matrices and operation (plus or minus) are selected with pull-down menus. In the example, figure 2.2, the red matrix (*top left*) is set to the difference of the green matrix and the blue matrix (*top middle*). The matrices are presented, using the notation



Multiply Blue by 0.75
Set Red to All-255 minus Red
Multiply Red by 0.5
Multiply Green by 0.8

This picture has 320 pixels horizontally and 240 pixels vertically. Each pixel has a red, green and blue channel -- the color of the pixel is made up of the red, blue, and green components combined. Below are the matrices of just the red, just the blue, and just the green colors of this picture. Each is 320 elements across and 240 elements up and down. So, the top most pixel in this picture has a red channel (amount of redness) 48 and a green channel 64 and a blue channel 35.Those three channels combine to create the color of the topleft pixel.

X (horizontal): 3

Y (vertical): 2

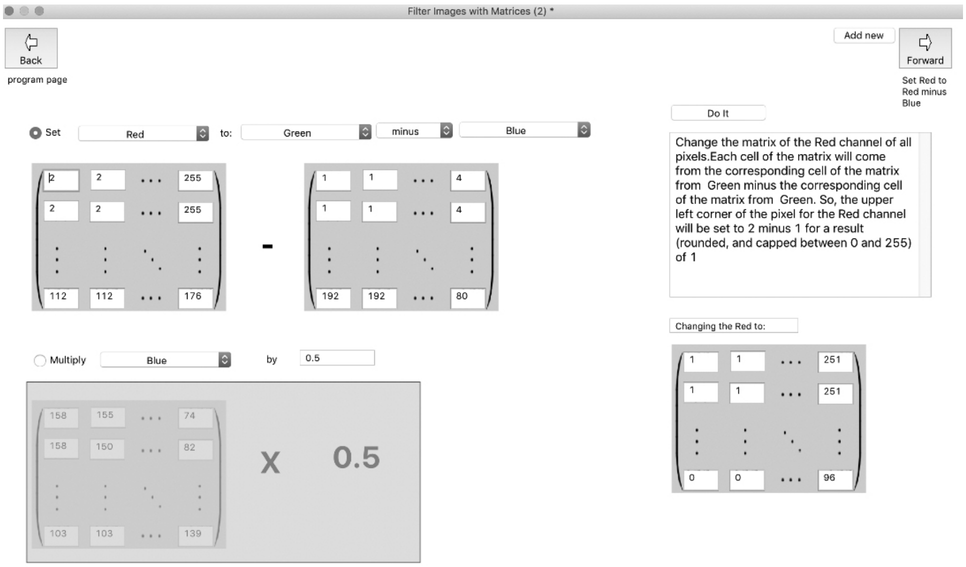
	Red	Green	Blue
Input Pixel	255	2	1
Result Pixel	254	2	1

Red Channel Matrix:
$$\begin{pmatrix} 48 & 51 & \cdots & 255 \\ 72 & 68 & \cdots & 255 \\ \vdots & \vdots & \ddots & \vdots \\ 81 & 94 & \cdots & 34 \end{pmatrix}$$

Blue Channel Matrix:
$$\begin{pmatrix} 35 & 38 & \cdots & 255 \\ 59 & 55 & \cdots & 255 \\ \vdots & \vdots & \ddots & \vdots \\ 55 & 62 & \cdots & 29 \end{pmatrix}$$

Green Channel Matrix:
$$\begin{pmatrix} 64 & 67 & \cdots & 255 \\ 88 & 84 & \cdots & 255 \\ \vdots & \vdots & \ddots & \vdots \\ 105 & 119 & \cdots & 59 \end{pmatrix}$$

2.1 Defining an image filter as a sequence of matrix transformations.



Change the matrix of the Red channel of all pixels.Each cell of the matrix will come from the corresponding cell of the matrix from Green minus the corresponding cell of the matrix from Green. So, the upper left corner of the pixel for the Red channel will be set to 2 minus 1 for a result (rounded, and capped between 0 and 255) of 1

2.2 Defining one matrix transformation.

commonly appearing in precalculus texts, with the output matrix (the new red matrix) appearing on the right. The *all-255* matrix can be used to compute the inverse of an image, by setting the red, green, and blue matrices to 255 minus the current value in the matrix.

- Alternatively, one of the matrices (red, green, or blue) can be multiplied by a scalar. The matrix can be selected by pull-down menu, and the scalar value is typed into a text area.

The image filter language is simple and grounded in the concepts and notation of precalculus. The image filter prototype is an example of task-specific programming to support learning matrix transformations for precalculus. Students may use this tool to meet a challenge (e.g., to generate a particular image manipulation effect) or to practice with tracing and using matrix arithmetic (e.g., in this given effect, what happens to pixels in the original picture whose RGB values are [128, 104, 12]). Our approach to adoption is informed by the work on SimCalc. In their scaling-up paper, Tatar et al. (2008) wrote: “Conversely, a wider path to adoption exists if one can engineer materials to support short-term use without extensive professional development and with a wide variety of pedagogical styles. In the short term, innovators may be able to make an earlier, more immediate impact on a wide audience and set a credible base of authentic improvement that can then serve longer term growth." This is exactly our approach. While task-specific programing tools may fit into project-based activities (Blumenfeld et al. 1991, 1994; Krajcik and Blumenfeld 2006), the goal is to be usable in a variety of activities.

We use our prototypes in participatory design sessions with teachers (DiSalvo 2016; Wilensky and Stroup, 1999). Our goal is to develop task-specific programming that teachers would find useful and would integrate into their classes, so we ask them to try it in the context of what students find challenging about precalculus. The prototype is an artifact to think with. Precalculus teachers learn and use it and then tell us what would *really* be useful to them. We then iterate on the design.

Sessions with precalculus teachers support our hypothesis that they can start using it in less than ten minutes. The general response from precalculus teachers has been guardedly positive. The teachers see that the microworlds aim to take an abstract concept in precalculus and ground it in a concrete application. They appreciate our attention to disciplinary literacy and to the learning outcomes for precalculus. Several of our informants saw the benefits of connecting precalculus to contexts that students found personally meaningful, like Instagram or Snapchat photo filters.

However, the teachers tell us that we are solving the wrong problems. While some students struggle with matrix notation and element-by-element operations, most do not. The hard parts of matrices in precalculus are matrix multiplication and determinants, and even convolution. Those parts are so difficult for students that matrices are often left out of a school's precalculus curriculum, which puts students at a disadvantage when they face linear algebra in undergraduate courses. We are currently iterating on this design.

A task-specific programming environment is unlikely to achieve all the goals described at the beginning of this chapter. Rather, task-specific programming may be an easier-to-use and easier-to-adopt programming experience than textual or block-base languages. Students will not use task-specific programming environments alone as a notational tool for computational literacy, but use of such tools may help students to gain understanding about the nature of programs, programming, and debugging. Task-specific programming may help students develop the first competencies on trajectories to learn programming (Rich et al. 2017, 2019).

CONCLUSION: FINDING PATHWAYS TO COMPUTATIONAL LITERACY

There are many reasons for students to learn programming, from understanding the digital world in which they live, to developing computational participation and action skills, to developing a new way to understand the world in which they live. Programming offers a powerful notation for learning and thinking that is unlike mathematical equations. The computer is the master simulator—it can look like any domain. Learning programming can be about learning domains that students are already interested in. Learning to program is not just about learning to become a software developer.

Achieving that vision may require us to rethink our programming environments. Languages developed for professional programmers, or developed for children in an earlier age with fewer computational end-interface skills, are unlikely to provide the affordances for learning that we can design in purpose-built environments. Task-specific programming is an approach for providing a pathway to computational literacy.

REFERENCES

Abelson, Harold, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs*. 2nd ed. Cambridge, MA: MIT Press.

Biggers, Maureen, Anne Brauer, and Tuba Yilmaz. 2008. “Student Perceptions of Computer Science: A Retention Study Comparing Graduating Seniors vs. CS Leavers.” In *SIGCSE ’08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. 402–406. https://doi.org/10.1145/1352135.1352274.

Blumenfeld, Phyllis C., Joseph S. Krajcik, Ronald W. Marx, and Elliot Soloway. 1994. “Lessons Learned: A Collaborative Model for Helping Teachers Learn Project-based Instruction.” *Elementary School Journal* 94 (5): 539–551.

Blumenfeld, Phyllis C., Elliot Soloway, Ronald W. Marx, Joseph S. Krajcik, Mark Guzdial, and Annemarie Palincsar. 1991. “Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning.” *Educational Psychologist* 26 (3 & 4): 369–398.

Carver, Sharon M. 1988. “Learning and Transfer of Debugging Skills: Applying Task Analysis to Curriculum Design and Assessment.” In *Teaching and Learning Computer Programming: Multiple Research Perspectives*, edited by Richard E. Mayer, 259–297. Hillsdale, NJ: Lawrence Erlbaum Associates.

Chasins, Sarah E., Maria Mueller, and Rastislav Bodik. 2018. "Rousillon: Scraping Distributed Hierarchical Web Data." In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology, UIST '18*. New York, 963–975.

Code.org. "What Most Schools Don't Teach." Streamed live on February 26, 2013, YouTube video, 5:43. https://youtu.be/nKIu9yen5nc.

DiSalvo, Betsy. 2016. "Participatory Design through a Learning Science Lens." In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. New York, 4459–4463.

diSessa, Andrea A. 2001. *Changing Minds*. Cambridge, MA: MIT Press.

diSessa, Andrea A., and Harold Abelson. 1986. "Boxer: A Recon-structible Computational Medium." *Communications of the ACM* 29 (9): 859–868.

ECMA International. 2011. *Ecma-262 Edition 5.1, The ECMAScript Language Specification.* https://262.ecma-international.org/5.1/#sec-11.9.3.

Fields, Deborah A., Yasmin Bettina Kafai, and Michael T. Giang. 2017. "Youth Computational Participation in the Wild: Understanding Experience and Equity in Participating and Programming in the Online Scratch Community." *ACM Transactions on Computing Education* 17 (3): 15:1–15:22.

Grover, Shuchi, and Roy Pea. 2013. "Computational Thinking in K–12: A Review of the State of the Field." *Educational Researcher*, 42 (1): 38–43.

Grover, Shuchi, Roy Pea, and Stephen Cooper. 2015. "Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students." *Computer Science Education*, 25 (2): 199–237.

Guzdial, Mark. 1995. "Software-Realized Scaffolding to Facilitate Programming for Science Learning." *Interactive Learning Environments* 4 (1): 1–44.

Guzdial, Mark. 2019. "Computing Education as a Foundation for 21st Century Literacy." In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19*. New York, 502–503.

Guzdial, Mark, and John Arquilla. 2019. "Is CS Really for All, and Defending Democracy in Cyberspace." *Communications of the ACM*, 62 (6): 8–9.

Guzdial, Mark, and Benedict du Boulay. 2019. "History of computing education research." In *The Cambridge Handbook of Computing Education Research*, edited by Sally A. Fincher and Anthony V. Robins, 11–39. Cambridge: Cambridge University Press.

Harel, Idit, and Seymour Papert. 1990. "Software Design as a Learning Environment." *Interactive Learning Environments* 1 (1): 1–32.

Kafai, Yasmin Bettina. 2016. "From Computational Thinking to Computational Participation in K–12 Education." *Communications of the ACM* 59 (8): 26–27.

Kafai, Yasmin Bettina, Quinn Burke, and Mitchel Resnick. 2014. *Connected Code: Why Children Need to Learn Programming*. Cambridge, MA: MIT Press.

Kalelioglu, Filiz, and Yasemin Gülbahar. 2014. “The Effects of Teaching Programming via Scratch on Problem Solving Skills: A Discussion from Learners’ Perspective.” *Informatics in Education* 13 (1): 33–50.

Kay, Alan C. 1977. “Microelectronics and the Personal Computer.” *Scientific American* 237 (3): 230–245.

Kay, Alan C. 1995. “Computers, Networks and Education.” *Scientific American* 272 (3): 148–155.

Kay, Alan C., and Adele Goldberg. 1977. “Personal Dynamic Media.” *IEEE Computer* 10 (3): 31–41.

Knuth, Donald E. 1972. “George Forsythe and the Development of Computer Science.” *Communications of the ACM* 15 (8): 721–726.

Krajcik, Joseph S., and Phyllis C. Blumenfeld. 2006. “Project-based Learning.” In *The Cambridge Handbook of the Learning Sciences*, edited by R. Keith Sawyer, 317–333. Cambridge: Cambridge University Press.

Lewis, Colleen, Paul Bruno, Jonathan Raygoza, and Julia Wang. 2019. “Alignment of Goals and Perceptions of Computing Predicts Students’ Sense of Belonging in Computing.” In *Proceedings of the 14th International Conference on Computing Education Research, ICER ’19*. New York, 11–9.

Maloney, John, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. “The Scratch Programming Language and Environment.” *ACM Transactions on Computing Education* 10 (4): 16:1–16:15.

Maloney, John H., Kylie A. Peppler, Yasmin Bettina Kafai, Mitchel Resnick, and Natalie Rusk. 2008. “Programming by Choice: Urban Youth Learning Programming with Scratch.” In *SIGCSE ’08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. New York, 367–371.

Margolis, J., R. Estrella, J. Goode, J. J. Holme, and K. Nao. 2017. *Stuck in the Shallow End: Education, Race, and Computing*. Cambridge, MA: MIT Press.

Moje, Elizabeth B. 2015. “Doing and Teaching Disciplinary Literacy with Adolescent Learners: A Social and Cultural Enterprise.” *Harvard Educational Review* 85 (2): 254–278.

O’Neil, Cathy. 2016. *Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy*. New York: Crown.

Palumbo, David B. 1990. “Programming Language/Problem-Solving Research: A Review of Relevant Issues.” *Review of Educational Research* 60 (1): 65–89.

Papert, Seymour. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Sussex, UK: Basic Books.

Parker, Miranda C., and Mark Guzdial. 2019. “A Statewide Quantitative Analysis of Computer Science: What Predicts CS in Georgia Public High School?” In *Proceedings of* *the 2019 ACM Conference on International Computing Education Research, ICER '19*. New York, 317.

Pea, Roy D., and D. Midian Kurland. 1984. "On The Cognitive Effects of Learning Computer Programming." *New Ideas in Psychology* 2 (2): 137–168.

Perlis, Alan J. 1962. "The Computer in the University." In *Computers and the World of the Future*, edited by Martin Greenberger, 180–217. Cambridge, MA: MIT Press.

Peyton Jones, Simon. 2013, September. "Computer Science as a School Subject." In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. Seattle, 159–160.

Rich, Kathryn M., Carla Strickland, T. Andrew Binkowski, and Diana Franklin. 2019. "A K-8 Debugging Learning Trajectory Derived from Research Literature." In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19*. New York, 745–751.

Rich, Kathryn M., Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2017. "K-8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals." In *Proceedings of the 2017 ACM Conference on International Computing Education Research, ICER '17*. New York, 182–190.

Scaffidi, Christopher. 2017. "Workers Who Use Spreadsheets and Who Program Earn More than Similar Workers Who Do Neither." In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Raleigh, NC, 233–237.

Scaffidi, Christopher, Mary Shaw, and Brad A. Myers. 2005. "An Approach for Categorizing End User Programmers to Guide Software Engineering Research." *ACM SIGSOFT Software Engineering Notes* 30 (4): 1–5.

Schanzer, Emmanuel, Kathi D. Fisler, and Shriram Krishnamurthi. 2018. "Assessing Bootstrap: Algebra Students on Scaffolded and Unscaffolded Word Problems." In *SIGCSE '18: Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. New York, 8–13.

Schanzer, Emmanuel, Kathi D. Fisler, Shriram Krishnamurthi, and Matthias Felleisen. 2015. "Transferring Skills at Solving Word Problems from Computing to Algebra through Bootstrap." In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. New York, 616–621.

Sherin, Bruce L. 2001. "A Comparison of Programming Languages and Algebraic Notation as Expressive Languages for Physics." *International Journal of Computers for Mathematical Learning* 6: 1–61.

Snow, Charles Percy. 1962. "Scientists and Decision Making." In *Computers and the World of the Future*, edited by Martin Greenberger. Cambridge, MA: MIT Press.

Tatar, Deborah, Jeremy Roschelle, Jennifer Knudsen, Nicole Shechtman, Jim Kaput, and Bill Hopkins. 2008. "Scaling Up Innovative Technology-Based Mathematics." *Journal of the Learning Sciences* 17 (2): 248–286.

Tissenbaum, Mike, Josh Sheldon, and Hal Abelson. 2019. “From Computational Thinking to Computational Action.” *Communications of the ACM* 62 (3): 34–36.

Weston, Timothy J., Wendy M. Dubow, and Alexis Kaminsky. 2019. “Predicting Women’s Persistence in Computer Science- and Technology-Related Majors from High School to College.” *ACM Transactions on Computing Education* 20 (1): 1:1–1:16.

Wilensky, Uri. 1991. “Abstract Meditations on the Concrete and Concrete Implications for Mathematics Education.” In *Constructionism*, edited by Idit Harel and Seymour Papert, 193–203. Norwood, NJ: Ablex.

Wilensky, U., K. Orton, D. Weintrop, E. Beheshti, M. Horn, and K. Jona. 2016. “Bringing Computational Thinking into High School Mathematics and Science Classrooms.” In *Transforming Learning, Empowering Learners: The International Conference of the Learning Sciences (ICLS) , Vol. 2*, edited by C. K. Looi, J. L. Polman, U. Cress, and P. Reimann. Singapore: International Society of the Learning Sciences.

Wilensky, Uri, and Walter Stroup. 1999. “Learning through Participatory Simulations: Network-Based Design for Systems Learning in Classrooms.” In *Proceedings of the 1999 Conference on Computer Support for Collaborative Learning*. Stanford, CA: International Society of the Learning Sciences, 80.