Integrating Computing through Task-Specific Programming for Disciplinary Relevance: Considerations and Examples

By Mark Guzdial and Tamara Shreiner

Computing is the most powerful and flexible media that humans have ever invented. In the 1970s, the computer interface as we know it today (e.g., windows with multiple fonts and styles, menus, a pointers) was invented by a group of researchers who saw the computer as humans' first *meta-medium* – it can represent any other medium (from text to photographs to video), plus it is interactive (Kay & Goldberg, 1977). The first part of this book introduced and defined the computational thinking skills, concepts, and perspectives that we might want students to develop. Chapters 4 and 5 introduced ways to help students develop these skills, and whether we might have courses specifically focused on computational thinking or if we would want to integrate computational thinking into other subjects. This chapter suggests that we might *use* the power and flexibility of computing to enhance student learning *about* computing. "Plugged" activities are where students actively explore the computing at the computer. We argue that plugged activities *use* computing to enhance student learning about the discipline and about computing in ways that "unplugged" activities simply can't. No other medium can come close to what a computer can do.

At the same time, as researcher Amy Ko writes[1], programming is the most powerful way of interacting with the computer, but also the least usable. Programming can be hard for novices without previous computing experience or without the mathematics knowledge that many programming languages expect. It can be challenging to deal with programming constructs and concepts, such variables, loops, and Boolean conditions. Most importantly, the complexity of programming can be something *extra,* an extraneous load when using computing to learn disciplinary ideas*.* Does learning to get the computer to repeat something a million times help with learning to read or understanding why plants need light to grow?

Fortunately, we can use the *flexibility* of computing to improve the *usability* of programming. In this chapter, we describe some of the work that we are doing to make new programming languages explicitly for specific tasks in a disciplinary context. Rather than spending a bunch of time getting good at a general programming language, we ask teachers and students to spend a short amount of time to use a small programming language for a particular task. We call these *task-specific programming (TSP) languages.* You might think of them as *Teaspoon (TSP) languages* – they add a teaspoon of programming that brings some computing spice to the project. They can't be used across the curriculum, or maybe not even on many days in a single subject. Instead, TSP languages are about making it *easier* to *integrate computing to enhance learning in other subjects* ***for a subject-specific task***.

[1] https://medium.com/bits-and-behavior/programming-languages-are-the-least-usable-but-most-powerful-human-computer-interfaces-ever-invented-7509348dedc#

The Technology Acceptance Model (Lee, Kozar, & Larsen, 2003) predicts that teachers will only adopt technology if teachers perceive that the technology is *useful* (e.g., facilitates learning towards standards and objectives) and is *usable* (which includes computer interface usability but also context, like fitting into course schedules). We argue that computing integration will occur in non-CS classes when (a) the computing is *useful* for achieving disciplinary goals and (b) the computing is *usable*. If the cost in usability is too high (e.g., takes too much time to learn the programming language) or the benefit in usefulness is too low (e.g., students don't learn enough about the discipline), then adoption is unlikely.

I. Plugging with Scratch, Snap!, and other block-based languages

If an elementary or middle school teacher wants to use programming in their class, they will most likely reach for Scratch, Snap!, or one of the other block-based programming languages. Programming languages used by professional software developers are mostly just prose text using specialized keywords (e.g., **if, print,** and **for**) with punctuation unusual in natural language (e.g., heavy use of semi-colons, colons, and curly braces). We call those *general-purpose programming languages* because is very little that you cannot build with these general-purpose programming languages like Python, Java, and C++.Block-based languages turn programming into a process of selecting and assembling colorful, jigsaw-like blocks. Some of the block-based languages are as powerful as general-purpose languages, but many have been simplified for students beginning at learning to program. The meaning of the blocks is still a direct mapping from those specialized keywords (e.g., there are blocks for **if** and **for**). So rather than typing, students drag and drop these blocks to construct programs without dealing with any of the strange punctuation nor having to type those keywords exactly right.

The most popular of the block-based languages is Scratch[2] with well over 30 million users. Scratch was created after observing what students most liked to do in the Intel Computer Clubhouses (J. Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010; J. H. Maloney, Peppler, Kafai, Resnick, & Rusk, 2008; Resnick et al., 2009). The most popular computational activities in these Clubhouses students were *choosing* to do were photo manipulation (e.g., in Adobe Photoshop) and music creation (e.g., with keyboards and drum machines). Scratch was designed to give students the ability to use programming with their favorite kinds of activities, such as making animations, telling stories with animations and music, and building video games (Figure 1).

[2] https://scratch.mit.edu/

Figure 1: A Geography Review Game in Scratch[3]

Over 30 million users will likely attest to the ease of programming in Scratch. Many classes teach with Scratch. Harvard's introductory course, CS50, starts out using Scratch, because it's so easy to get started.

Scratch and Scratch Jr (Flannery et al., 2013) are terrific languages for exploring the ideas of computing. Even before students learn ideas like variables and iteration, there are more fundamental ideas of computing that students need to understand. These include the ideas that *programs are assembled out of basic elements*, and *different orderings of elements can sometimes have the same result,* and even that *the program determines the computer's behavior (there's no magic)*.

But what if you don't teach computing, and instead you want to use computing to teach mathematics, reading, science, and social studies in new ways? Can you use Scratch for *your* tasks? Can you use Scratch for whatever topics you need to teach, using the power and flexibility of the computational medium? Maybe, but Scratch may not be the simplest tool anymore. Scratch can be used for building scientific simulations or analyzing data, but it's harder than what Scratch was originally designed for. Scratch is not what you would choose if you wanted students to build a website or an application for a cellphone.

[3] Screenshot from https://scratch.mit.edu/projects/1837844/

There are block-based tools designed to do other kinds of things. App Inventor is explicitly designed to create mobile apps for Android phones. Snap! is a more general-purpose programming language than Scratch or App Inventor, with amazing support for building sophisticated graphics and manipulating digital media like sounds and video. Still, all of these feature multi-colored jigsaw-like blocks that are dragged-and-dropped rather than typed.

Pencil Code[4] (led by Google employee, David Bau) is another block-based programming languages. It allows students to use blocks and JavaScript (a textual general-purpose programming language) interchangeably. Any program can be viewed as blocks or as lines of textual JavaScript code. Figure 2 is part of the Pencil Code program to represent a chatbot that pretends to be Lady Macbeth[5]. This project is being used by middle school classes in literature and English, where students modify the code to change what Lady Macbeth says or to create a chatbot for a different character. The program searches for key words in input statements from the user, then outputs a response. If the user asks a question like "Who are you?", the rule **when "name","who"** would be triggered (because the word **"who"** appears in the input), then the chatbot would respond **Lady Macbeth.**

[4] http://www.pencilcode.net

[5] Screenshots are from Pencil Code with the example at https://docs.google.com/document/u/0/d/1RCP3G2f9QUeRxKGSXcO9QhjX7mK9YDE4Bqeqnh8q3UM/mobilebasic

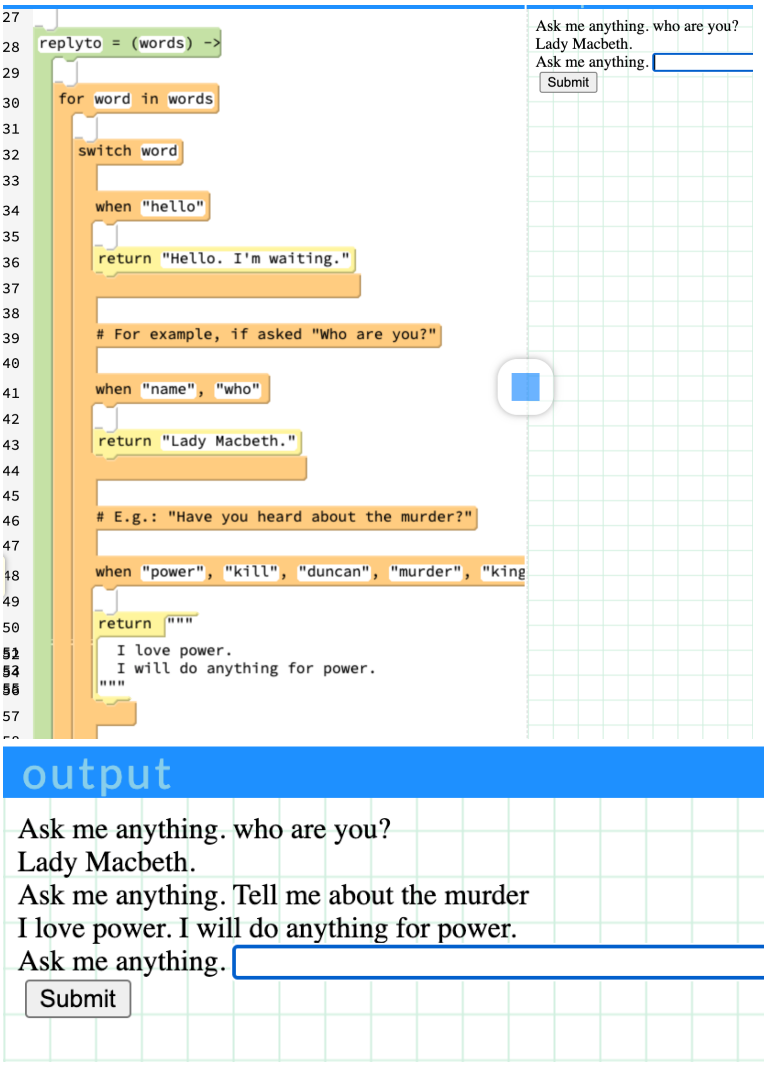

Figure 2: Building a Chatbot for Middle School English

Building a chatbot can be a terrific activity for getting students to think about what makes a character unique. How would a character respond in a way that is particular and representative of that character? Defining a chatbot invites the student to think about a character of interest, and then executing the chatbot program makes the character seemingly come alive. The power of the computer to be interactive makes the chatbot more than just an exercise in character analysis. Chatbots can be shared among students. Students can interact with other students' chatbots to try them out.

Creating a chatbot even in Pencil Code exposes underlying complexity that is not relevant to the class. Note the **replyto = (words) ->** at the top of the program. At the bottom of the program, the student finds a complicated **while** loop (Figure 3). It reads input phrases from a user (interlocutor) and breaks them up into words.

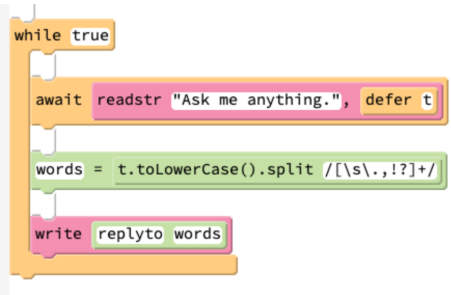

Figure 3: The bottom of the Chatbot example

Pencil Code offers a block-based form on top of a general-purpose programming language (JavaScript), which means that it has enormous expressive power. The cost of greater power is greater complexity. Our claim is that *the closer you move towards a general-purpose programming language, the more there is to learn.* You can have one programming language for many tasks, but learning all of that programming will be hard. *The programming becomes easier when the fit is closer between what you are trying to do and the tasks that the programming language was designed for.* We believe that this insight can lead to a new class of programming languages designed to be easier for disciplinary teachers to adopt.

II. Task-specific curriculum with general purpose languages

Researchers and developers are creating examples of task-specific curricula that use general purpose languages. Several of these can be found at https://www.bootstrapworld.org/. For example, *Bootstrap:Algebra* teaches algebra ideas around functions and variables by having students build video games. The students write functions that define the position of video game elements in the *next* frame in terms of the *last* frame. The students do not program the repeated generation of frames of the program – that's done for them with specialized code behind the scenes from the student.

The results of Bootstrap:Algebra are impressive in terms of improving student learning in (Emmanuel Schanzer, Fisler, & Krishnamurthi, 2018; E. Schanzer, Fisler, Krishnamurthi, & Felleisen, 2015). Students learn to solve word problems better through their experience with Bootstrap. A striking result is that students after learning Boostrap:Algebra are less likely to leave questions *blank* on an exam with story problems. Students learned how to get started on a problem, even if they didn't get to a solution. For mathematics teachers, Bootstrap:Algebra is clearly *useful* – it addresses their learning objectives.

Boostrap:Algebra takes about 25 hours of class time to implement. Some of that time is teaching programming, and some of it is teaching the curriculum components (e.g., how the video games are created), and it's probably not possible to tease out the distinctions between mathematics, video games, and programming contexts. Much of the time is spent talking about algebra ideas, but in a programming context. Variables and functions are the same in algebra and Bootstrap. There is some translation from what appears in the textbooks and what appears on the screen, which the Bootstrap team has made into a classroom practice called "circles of evaluation." By thinking through these mappings between mathematical notation and

programming, Bootstrap:Algebra is made *usable*. Students are not spending time learning programing content that gets in their way of doing video games in algebra class.

Boostrap:Algebra is a great example of taking a general-purpose programming language (either Racket or Pyret) and wrapping around it a task-specific curriculum (building video games) to make it work for a given school topic (variables and functions in algebra). The idea is to enhance students' learning about algebra by adding in the excitement and rigor of programming. A computer program requires the student to specify the variable and function *right*. It won't work otherwise. But once it is right, the student gets a video game to work. By working in the programming language, the student gets a check on understanding about the course content (a kind of formative feedback) and the motivation of making something real through the power of the computation.

Programming languages such as Logo (including its descendants Netlogo and Starlogo) and Pascal are designed to be simpler than professional programming languages, but are still complete general-purpose programming languages. All the features of general-purpose programming (from repetition to conditionals) are still there, which makes them powerful and full-featured. However, it's too easy for students to slip from an explainable subset of the language which can be focused on a particular task into the other features which are harder to explain – it's for this reason that Racket uses language levels to define sublanguages (Findler et al., 2002). We are proposing programming languages that are made even simpler by not even providing the more sophisticated features. Only the features that are useful for the task are implemented.

III. Example: Programming for Data Visualization in Social Studies classes

Can we get that computational power without investing as much class time in order to learn enough programming? In our research, we are fine-tuning the programming language for specific tasks to explore just how easy we can make. We want programming to be something that takes no more than 10 minutes to learn, for tasks that might be useful only for an hour lesson. For example, we are working on a data visualization tool that introduces programming for use in social studies classes that introduces programming.

Much of the work in integrating computing into other subjects has focused on STEM areas. While we may wish more students to take science and mathematics courses, not all students do. *All* students take social studies. The diversity of students in almost any history class is far greater than the average computer science class. Integrating programing into social studies classes makes a dramatic improvement in reach.

A. Why Data Literacy is Important for Social Studies Classes

Data literacy—the ability to read, analyze, interpret, evaluate, and argue with data and data visualizations—is an essential competency in social studies education. The National Council for

the Social Studies' College, Career, and Civic Life (C3) Framework for Social Studies State Standards (NCSS, 2013) recommends that by the end of second grade, students know how to use and construct maps, graphs, and other data visualizations and that they will continue working with data visualizations throughout elementary, middle, and high school. Such recommendations for data literacy are reflected in curriculum standards from all fifty U.S. states and the District of Columbia, which invariably require that students interpret, create, and use data visualizations from elementary school through high school (Shreiner, 2020). Standardized assessments of social studies, such as the National Assessment of Educational Progress in U.S. History and the SAT subject area tests in U.S. and World History also include items that require students to demonstrate proficiency with data visualizations (NAGB, 2010). Furthermore, social studies textbooks, trade books, and periodicals are filled with a wide variety of data visualizations. In the case of social studies textbooks, data visualizations become increasingly prevalent and complex as students move through school, and as many as 90% of them provide information not found in the surrounding verbal text (Fingeret, 2012; Shreiner, 2018). Perhaps most importantly, a core mission of social studies educators is to prepare students for informed and competent citizenship (NCSS, 2017). And in a society where data visualizations are regularly used to communicate information about problems, policies, and trends, or persuade people to vote for a particular candidate or agenda, an informed citizen must be a data-literate citizen (Bowen & Bartley, 2014; Franklin et al., 2015; Gould, 2017).

However, social studies teachers don't necessarily feel prepared to teach data literacy. In a recent survey of 242 practicing teachers, fewer than a quarter of respondents reported regularly teaching data literacy, or feeling that they could do so effectively (Shreiner & Dykes, 2020). This neglect of data literacy may be due a lack of teacher preparation and resources– 97% of teachers said they had no coursework or professional development to prepare them for teaching data literacy in social studies, and over half said they lacked classroom resources to help them teach.

For students, a lack of data literacy instruction may come at a high cost. Roberts, Norman, and Cocco (2015) found in a study of 156 third graders that comprehension of so-called graphical devices, which include data visualizations, was positively correlated with reading comprehension, suggesting that students who increase graphical comprehension might increase overall reading comprehension. In a think aloud study with 27 elementary, middle, and high school students, Shreiner (2019) found that students across grade levels tended to ignore data visualizations when using a history text to reason about historical questions, but that 85% of the students who initially ignored the data visualization in the passage they were reading later reported that it was helpful in answering the historical question with which they were tasked. Analysis of students' responses after considering the data visualization revealed that it helped students develop a richer context for the historical situation under study, an important but difficult aspect of historical reasoning.

Although it is often assumed that data visualizations are easy to understand, research indicates that students are likely to face numerous challenges as they attempt to make sense of data visualizations or integrate them with other information (Brugar & Roberts, 2017; Duke, Martin,

Norman, Knight, & Roberts, 2013; Maltese, Harsh, & Svetina, 2015; Roberts et al., 2013; Shah & Hoeffner, 2002; Shah, Mayer, & Hegarty, 1999). Brugar and Roberts (2017) found in a study of 326 elementary students that even when children attempted to use visual displays that included maps and graphs as sources of meaning in a text, they were challenged in doing so, answering questions related to the visual displays incorrectly more often than questions related to verbal written text. Such challenges seem to continue into adolescence and adulthood. In a think aloud study that included eight high school students, Shreiner (2009) found that while using bar and pie graphs to grapple with a political problem, students could extract basic information from the graphs, but did not employ evaluative strategies indicative of more expert analysis, such as sourcing, contextualizing, and considering methodological factors. Börner et al. (2016) concluded in a study of 127 participants aged eight to twelve and 146 participants aged 18 or older that a high proportion of both groups could not name different types of data visualizations or interpret them beyond basic reference systems.

Given the importance of data visualizations and associated challenges, teaching data literacy from elementary through secondary school is critical. Importantly, researchers suggest that it is not enough to learn data literacy skills in courses like mathematics alone. Different contexts can influence readers' comprehension of data visualizations, so teaching data literacy skills in multiple contexts is important, especially in light of all the different contexts in which data visualizations appear (Shah & Hoeffner, 2002). In history, for example, maps and graphs are used for unique, discipline-specific purposes. Maps help historians conceive of space, place, and time in concert, and at both small and large scales. Maps make the invisible processes visible—revealing ways that people moved over long stretches of time, or how diseases or languages spread. Likewise, by aggregating, compressing, and reducing complexity until obscure patterns and relationships become clear, graphs make it easier for historians to grasp incredibly large processes of change. Graphs are also critical in the historical inquiry process for testing hypotheses and providing evidence for historical interpretations about how and why past changes occurred.

B. Helping Students to Build Data Visualizations

We want students learning data literacy in their history classes to build data visualizations as part of an inquiry process. Building a data visualization is specifying a computational process – what data should the computer process and how in order to build a visualization? A data visualization over hundreds or thousands of years of history *requires* the power of the computer. A human is unlikely to go through two hundred years of population data to build a chart by hand, for example. There are tools designed for middle school students to use in making data visualizations (such as CODAP (Finzer & Damelin, 2016)), but they may not make visible to students that they are *specifying a* program. If we asked students to specify their data visualization as a program, students might lose confidence or be distracted by the details of programming. We have ample evidence that programming can reduce student self-efficacy (Kinnunen & Simon, 2012). In our work with teachers, we have found that our social studies teachers like Vega-Lite (Satyanarayan, Moritz, & Wongsuphasawat, 2017) for its power and

flexibility, but they find the programming language complicated and off-putting (Guzdial & Naimipour, 2019; Naimipour, Guzdial, & Shreiner, 2019).

For teachers to use the power of computing, we need programming tools that are both *useful* and *usable*. We are working on tools that *scaffold* the process of connecting data visualization and programming for students. Our goal is to use the power of computing to enhance learning of data literacy in history classes while also helping students to learn concepts and skills in computing. We want students to have the ease of block-based programming, or even better usability.

In our tool (Figure 4), students specify visualizations with pull-down menus. In this example, a student is comparing the populations of the United Kingdom and France from 1800 to 2018. We always show two visualizations because historical inquiry often begins with two pieces of data or accounts that do not agree (Bain, 2000).

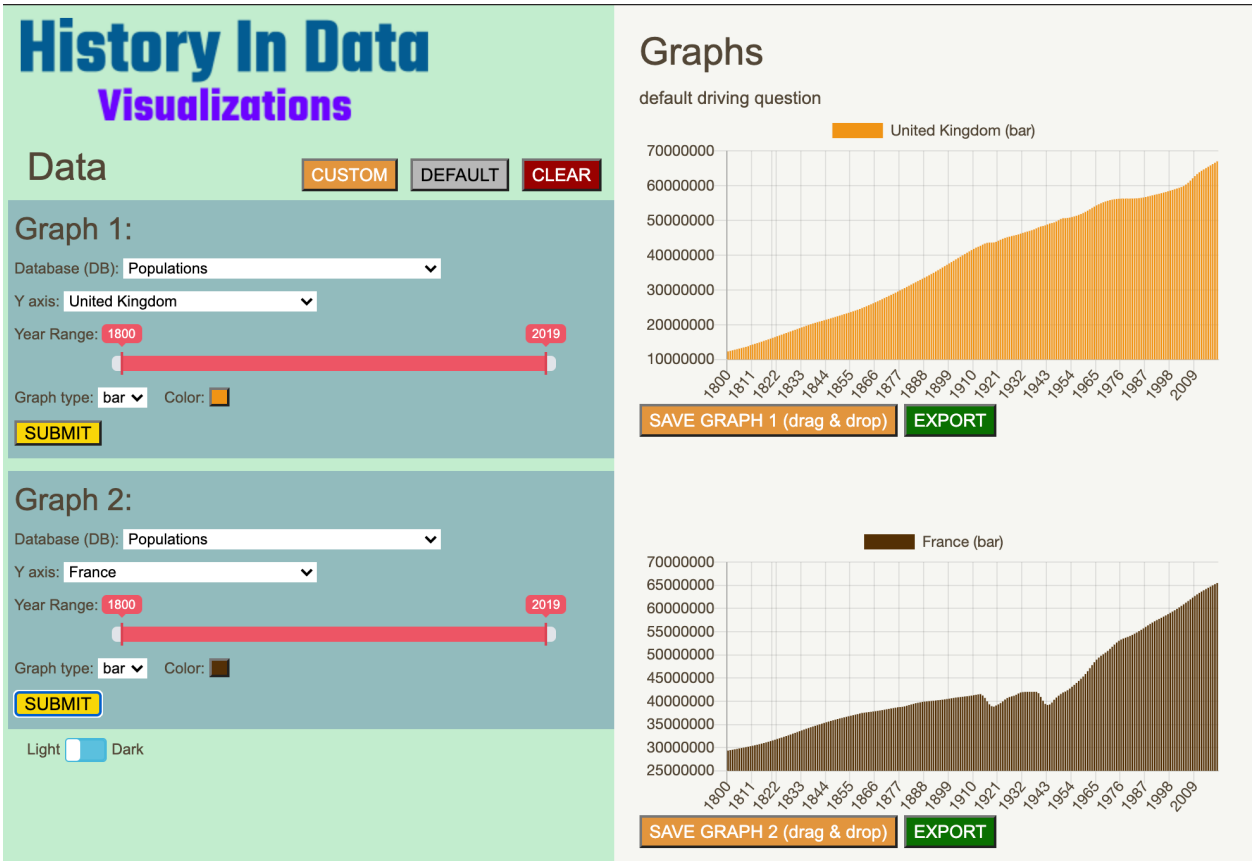

Figure 4: Data Visualization of Populations in the United Kingdom and France

The visualizations become the focus of inquiry. The student notices (perhaps with some scaffolding) that France has two clear dips in its population during World Wars I and II that the United Kingdom does not have. Both France and the UK were combatants in the World Wars. Didn't both countries suffer many casualties from their military? What might explain the larger relative dips in France's population?

With curriculum and a tool, we are supporting a historical inquiry process. The student can drag the France and UK graphs into saved spots on the right, and explore with other visualizations. What might explain the relative population drops? Maybe it has to do with the amount of time that the combatants were in the wars. The US entered late in both wars. In World War II, the Belgian military lasted only 18 days against the Nazis. In Figure 5, our student decides to compare the United States and Belgium.

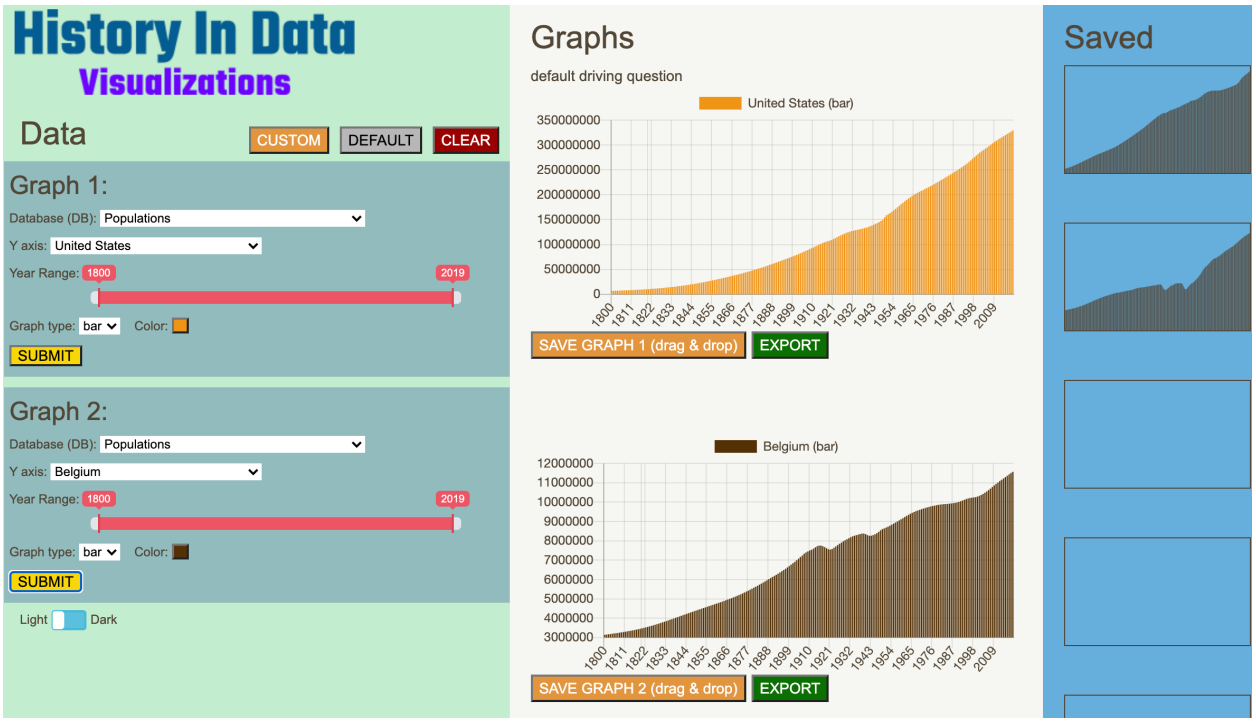

Figure 5: Comparing the Populations of the United States and Belgium

There are clear dips in Belgium's population, but not much in the US. At this point, the student would probably discard the earlier hypothesis that the dips were due to casualties of soldiers in the war, but might consider a new one. Maybe it has to do with the war being on the ground in the country. The student could continue building visualizations to test the hypothesis, e.g., by exploring the population drops in neighboring countries during the World Wars, or looking at the populations in other countries during the time periods of other wars.

As visualizations pile up in the Saved graphs area, the student might lose track. Which graph was which? By clicking on them, a textual description of how the graph was generated appears on top of the graph (Figure 6). We use a program representation like the one used for Vega-Lite. The program is *declarative*, which means that it specifies the visualization but does not specify the steps of the program (which is sometimes called *procedural* or *imperative* programs). The program lists keywords on the left and values for those keywords on the right, using a format called ***JSON***.

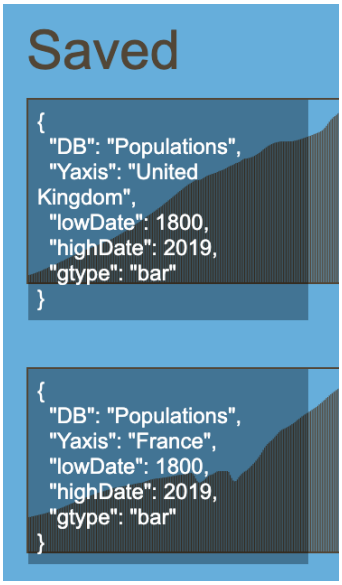

Figure 6: Saved visualizations with pop-up scripts visible

We show the program as a concise description of how the graph is presented. The student does not *write* the program. We present the program as a useful description to *read*.

We have a second version of our visualization tool (Figure 7) where scripts are literally at the center of the interface. Students can specify visualizations in the same way, by making choices in the pull-down menus. As they specify visualizations, the visible script updates. Students may also edit the script directly (e.g., change the *Yaxis* to *"Germany"*). As either the pull-down menus or textual script is changed, the other representation is updated to match, and the graph is re-drawn to match. The pull-down menus and textual scripts are multiply-linked representations (Vosniadou, De Corte, Glaser, & Mandl, 2012), which helps students to understand the mappings between the visualizations, the scripts, and the menu settings.

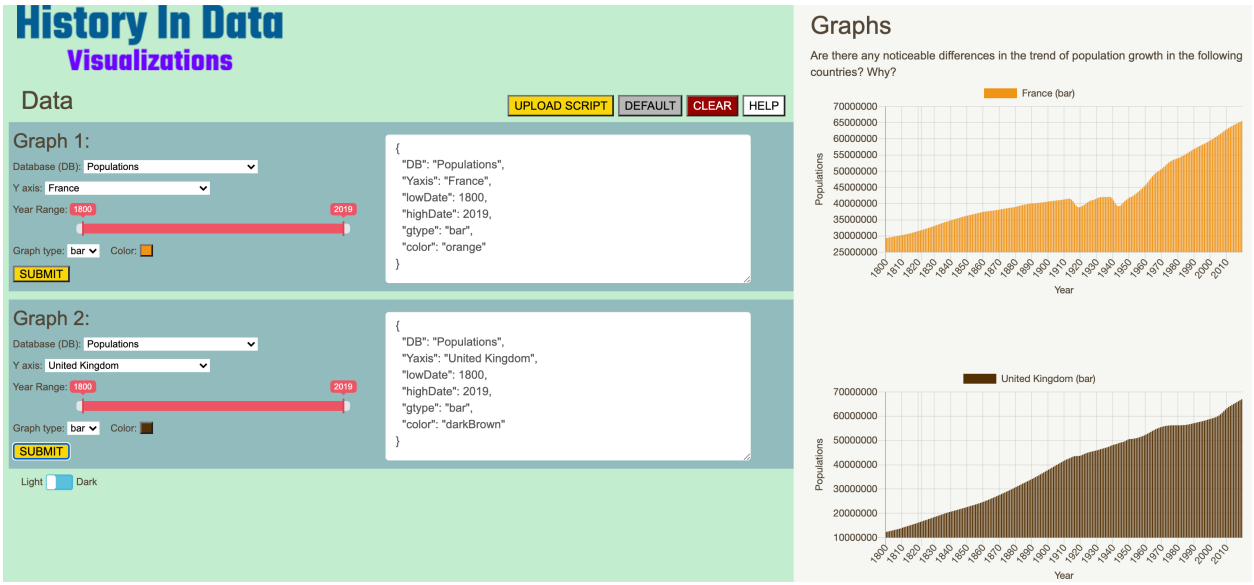

Figure 7: Visualization tool where scripts are available as a multiply-linked representation

Our hypothesis is that specifying a visualization with pull-down menus is even easier (i.e., less complexity, less cognitive load, and fewer errors) than assembling a set of blocks to construct a similar visualization. By providing both representations, our goal is to scaffold students in seeing textual programming as readable, accessible, and usable, much as Pencil Code does for JavaScript.

JSON is an unusual notation for general-purpose languages, but it's similar to the notation used in the visualization language Vega-Lite. Social studies teachers in our participatory design classes are interested in using Vega-Lite (Naimipour, Shreiner, & Guzdial, 2020). If students will one day use Vega-Lite, we might expect transfer from the experience. But even if students never use Vega-Lite, we hypothesize that students using our scaffolded visualization tools would learn new understanding of what programming and computing is about. Use of this visualization tool can be a place to learn that programs specify output deterministically, and that getting the syntax right is necessary for a program to run. We would need to design curriculum around the use of this visualization tool to make explicit that this is programming in order to prepare students for future learning (Grover, Pea, & Cooper, 2014).

IV. Example: Programming to build Chatbots in English classes

Earlier in the chapter, we suggest that creating a chatbot is a powerful activity in which to use computing to enhance learning about literature. Defining a chatbot requires students to think carefully about what makes a character unique. How would this character respond to an interlocutor? How would one character's responses differ from another's? Deeply analyzing characters is a useful activity in an English Language Arts (ELA) classroom.

The Pencil Code example in Figure 2 defines a chatbot meant to represent Shakespeare's Lady Macbeth. We argue (using Figure 3) that there are details in the Pencil Code example that are a distraction from the activity of defining chatbots in order to explore a character.

We have defined a new chatbot language which simplifies the process of defining chatbots. Figure 8 is shows part of the program for defining Lady Macbeth in our Teaspoon chatbot language[6]. In this language, there are no quote marks. There is no use of programming constructs like **while**. There are no variables. Instead, the programming language consists of **if** clauses that match **any** of these words to the interlocutor's input. The **if match** rules are tested from top to bottom. Whenever any rule matches, no other rules in the program are tested. So, more specific forms (e.g., matching to **all of green,apple**) can appear before more general mores (e.g., matching to **any of apple**).

```
if match any of hello, hi
brief response Hello. I'm waiting.
end match

if match any of name, who
brief response Lady Macbeth
end match

if match any of power, kill, duncan, murder, king
long response
I love power.
I will do anything for power.
end response
end match

if match any of prophecy, prophesy, prophesies
long response
I think the prophesies are true,
but I think that we have to make them
come true.We can't just rely on fate.
We have to take matters into our own hands.
end response
end match
```

Figure 8: A program to create a Lady Macbeth chatbot

Figure 9 shows the execution of the Lady Macbeth chatbot. It works exactly the same as the Pencil Code version.

Our Teaspoon chatbot language has fewer punctuation details than the Pencil Code version. It is a much more limited language. Using this language creates an opportunity for students to learn computational ideas like what an **if** conditional does. It could also be used to learn fundamental concepts in learning trajectories for programming (Rich, Strickland, Binkowski, Moran, & Franklin, 2017) like *programs are assembled out of basic elements*, and *different orderings of elements can sometimes have the same result,* and even that *the program determines the computer's behavior* (there's no magic*)*.

[6] http://teaspoon.livecodehosting.com/chatbot/index.html

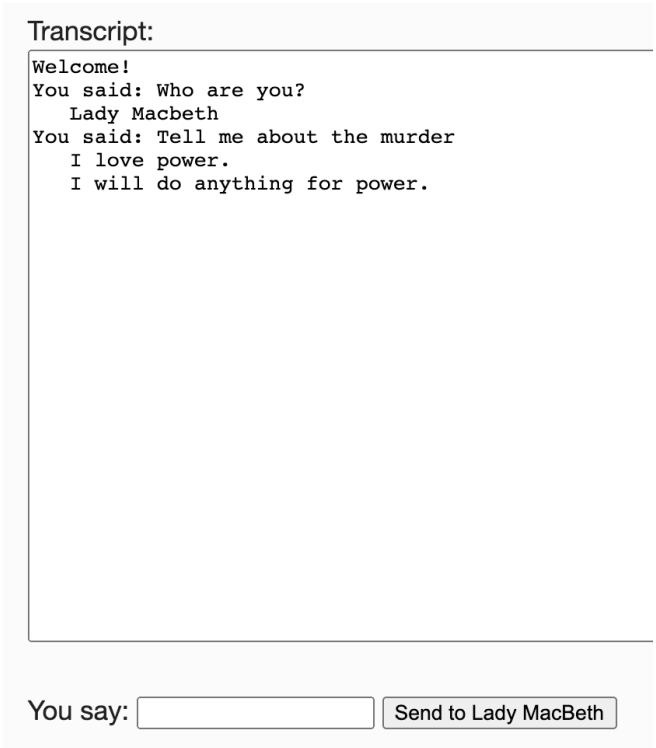

Figure 9: The Lady Macbeth Chatbot executing, exactly like the one in Pencil Code

V. Teaspoon Computing as a Preparation for Future Learning

Our work with task-specific programming ("teaspoon") languages is still at an early stage of research. We have been working with teachers in a participatory design process as we have iteratively defined and developed our languages. Our future work will involve K-12 students and how the use of our tools support their learning of disciplinary ideas.

We return here to the Technology Acceptance Model that we started the chapter with. In order to adopt technology for data literacy, the teacher must believe that the technology can help to achieve their learning goals (e.g., address a student learning challenge) while fitting into their existing structures and constraints (e.g., available class time), and that the teacher can successfully implement the activity (e.g., self-efficacy stemming from knowledge and experience) (Holden & Rada, 2011). Our first question has been whether we can define computing to be integrated into other subjects that will actually be *adopted* by teachers. Languages like Scratch and Pencil Code have existed for years, and are used in courses other than computer science. But our best evidence suggests that less than 10% of high school students in any US state see any computing education (Parker, 2019; Parker & Guzdial, 2019). If we define languages that support the *tasks* that teachers find valuable (perhaps even *impossible* without the technology), we hope that we can create programming that will actually be *adopted.*

We argue that our tools can be *useful* in meeting teachers' goals. For social studies teachers, data literacy is part of state standards. Some brave early adopter ELA teachers are trying chatbots. Our research is about showing that teaspoon languages might make programming *usable* enough to be adopted.

It's an open research question if we can design instruction using these teaspoon languages such that (a) student learning is enhanced in the domain and (b) students learn about computing, too. Our task-specific languages are not based on general-purpose languages. At a surface-level, there is little about traditional programming languages that students might be learning. These are trade-offs to explore in the research, which will involve both design of tools and design of the associate curriculum. What gets emphasized when?

We believe that teaspoon languages can be a place to prepare for future learning in computing (Grover et al., 2014). Students are actually *programming* in these languages. They are addressing the concepts appearing at the earlier stages in the Rich et al. trajectories of student learning in computer science (Rich, Strickland, Binkowski, & Franklin, 2019; Rich et al., 2017). For example, students will specify programs, make mistakes, and need to debug. The challenge will be to construct contexts so that students recognize programming and computing when they face it again. Another research question about teaspoon languages is whether students get bored with such a small and simple core of programming. Do we provide enough programming? Frankly, if we achieve our learning *and* adoptability goals, and leave students wanting *more programming*, we will have achieved everything we could hope for.

In summary, we believe that plugged activities offer so much to support student learning. Computer programming is powerful and flexible. Our challenge is to also make it usable enough that it is useful to meet teacher and student needs.

References

Bain, R. B. (2000). Into the breach: Using research and theory to shape history instruction. In P. N. Stearns, P. C. Seixas, & S. S. Wineburg (Eds.), *Knowing, teaching, and learning history : national and international perspectives* (pp. 331-352). New York: New York University Press.

Börner, K., Maltese, A. V., Balliet, R. N., & Heimlich, J. (2016). Investigating aspects of data visualization literacy using 20 information visualizations and 273 science museum visitors. *Information Visualization, 15*, 198-213. doi:10.1177/1473871615594652

Bowen, M., & Bartley, A. (2014). *The basics of data literacy: Helping your students (and you!) make sense of data*. Arlington, VA: National Science Teachers Association Press.

Brugar, K. A., & Roberts, K. L. (2017). Elementary students' challenges with informational texts: Reading the words and the world. *The Journal of Social Studies Research*. Retrieved from http://dx.doi.org/10.1016/j.jssr.2017.02.001

Duke, N. K., Martin, N. M., Norman, R. R., Knight, J. A., & Roberts, K. L. (2013). Beyond concepts of print: Development of concepts of graphics in text, preK to grade 3. *Research in the Teaching of English, 48*, 175-203.

Findler, R. B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., & Felleisen, M. (2002). DrScheme: A programming environment for Scheme. *Journal of functional programming, 12*(2), 159.

Fingeret, L. (2012). *Graphics in children's informational texts: A content analysis.* (Doctoral dissertation Doctoral dissertation). Michigan State University, East Lansing, MI. Proquest Dissertations Publishing database. (3524408)

Finzer, W., & Damelin, D. (2016). *Design perspective on the Common Online Data Analysis Platform (CODAP).* Paper presented at the Paper presented at American Educational Research Association (AERA) conference, Washington DC.

Flannery, L. P., Silverman, B., Kazakoff, E. R., Bers, M. U., Bontá, P., & Resnick, M. (2013). *Designing ScratchJr: Support for early childhood learning through computer programming.* Paper presented at the proceedings of the 12th international conference on interaction design and children.

Franklin, C. A., Kader, G. D., Bargagliotti, A. E., Scheaffer, R. L., Case, C. A., & Spangler, D. A. (2015). *Statistical Education of Teachers*. Retrieved from

Gould, R. (2017). Data literacy is statistical literacy. *Statistics Education Research Journal, 16*(1). Retrieved from https://iase-web.org/documents/SERJ/SERJ16(1)_Gould.pdf

Grover, S., Pea, R., & Cooper, S. (2014). *Expansive framing and preparation for future learning in middle-school computer science.* Paper presented at the International Conference of the Learning Sciences (ICLS) Conference.

Guzdial, M., & Naimipour, B. (2019). *Task-Specific Programming Languages for Promoting Computing Integration: A Precalculus Example.* Paper presented at the Proceedings of the 19th Koli Calling International Conference on Computing Education Research, New York, NY, USA.

Holden, H., & Rada, R. (2011). Understanding the Influence of Perceived Usability and Technology Self-Efficacy on Teachers' Technology Acceptance. *Journal of Research on Technology in Education, 43*(4), 343-367. Retrieved from https://doi.org/10.1080/15391523.2011.10782576

Kay, A., & Goldberg, A. (1977). Personal dynamic media. *IEEE Computer*, 31-41.

Kinnunen, P., & Simon, B. (2012). My program is ok–am I? Computing freshmen's experiences of doing programming assignments. *Computer Science Education, 22*(1), 1-28.

Lee, Y., Kozar, K. A., & Larsen, K. R. T. (2003). The technology acceptance model: Past, present, and future. *Communications of the Association for information systems, 12*(1), 50.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education, 10*(4), 16:11-16:15.

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). *Programming by choice: urban youth learning programming with Scratch.* Paper presented at the SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education, New York, NY, USA.

Maltese, A. V., Harsh, J. A., & Svetina, D. (2015). Data visualization literacy: Investigating data interpretation along the novice-expert continuum. *Journal of College Science Teaching, 45*, 84-90.

NAGB. (2010). *U.S. History Framework for the 2010 National Assessment of Educational Progress*. Retrieved from Washington, DC:

Naimipour, B., Guzdial, M., & Shreiner, T. (2019). *Helping Social Studies Teachers to Design Learning Experiences Around Data: Participatory Design for New Teacher-Centric Programming Languages.* Paper presented at the Proceedings of the 2019 ACM Conference on International Computing Education Research, New York, NY, USA.

Naimipour, B., Shreiner, T. L., & Guzdial, M. (2020). *Engaging Teachers in Front-End Design: Developing Technology for a Social Studies Classroom.* Paper presented at the Proceedings of the ASEE/IEEE 2020 Frontiers in Education Conference.

NCSS. (2013). Scholarly rationale for the C3 Framework. In N. C. f. t. S. Studies (Ed.), *Social Studies for the Next Generation: Purposes, practices, and implications of the College, Career, and Civic Life (C3) Framework for Social Studies Standards* (pp. 82-91). Silver Springs, MD: National Council for the Social Studies.

Parker, M. C. (2019). *An Analysis of Supports and Barriers to Offering Computer Science in Georgia Public High Schools.* (Human-Centered Computing). Georgia Institute of Technology,

Parker, M. C., & Guzdial, M. (2019). *A Statewide Quantitative Analysis of Computer Science: What Predicts CS in Georgia Public High School?* Paper presented at the Proceedings of the 2019 ACM Conference on International Computing Education Research, New York, NY, USA.

Resnick, M., Maloney, J., Monroy-Hern\'a, n., Andr\'e,s, Rusk, N., Eastmond, E., Brennan, K., . . . Kafai, Y. (2009). Scratch: programming for all. *Commun. ACM, 52*(11), 60-67.

Rich, K. M., Strickland, C., Binkowski, T. A., & Franklin, D. (2019). *A K-8 Debugging Learning Trajectory Derived from Research Literature.* Paper presented at the Proceedings of the 50th ACM Technical Symposium on Computer Science Education.

Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017). *K-8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals.* Paper presented at the Proceedings of the 2017 ACM Conference on International Computing Education Research, New York, NY, USA.

Roberts, K. L., Norman, R. R., & Cocco, J. (2015). Relationship between graphical device comprehension and overall text comprehension for third-grade children. *Reading Psychology, 36*, 389-420. doi:10.1080/02702711.2013.865693

Roberts, K. L., Norman, R. R., Duke, N. K., Morsink, P., Martin, N. M., & Knight, J. A. (2013). Diagrams, Timelines, and Tables – Oh, My!: Fostering Graphical Literacy. *The Reading Teacher, 67*, 12-23. doi:10.1002/TRTR.1174

Satyanarayan, A., Moritz, D., & Wongsuphasawat, K. (2017). Vega-lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics, 23*(1), 341-350.

Schanzer, E., Fisler, K., & Krishnamurthi, S. (2018). *Assessing Bootstrap: Algebra students on scaffolded and unscaffolded word problems.* Paper presented at the Proceedings of the 2018 ACM SIGCSE Technical Symposium.

Schanzer, E., Fisler, K., Krishnamurthi, S., & Felleisen, M. (2015). Transferring skills at solving word problems from computing algebra through bootstrap. *Proceedings of 46th ACM Technical Symposium on Computer Science Education*, 616-621.

Shah, P., & Hoeffner, J. (2002). Review of graphic comprehension research: Implications for instruction. *Educational Psychology Review, 14*, 47-69. doi:10.1023/a:1013180410169

Shah, P., Mayer, R. E., & Hegarty, M. (1999). Graphs as aids to knowledge comprehension: Signaling techniques for guiding the process of graph comprehension. *Journal of Educational Psychology, 91*, 690-702. doi:10.1037/0022-0663.91.4.690

Shreiner, T. L. (2009). *Framing a model of democratic thinking to inform teaching and learning in civic education.* (Doctoral dissertation). University of Michigan, Ann Arbor, MI. Proquest Dissertations Publishing database. (3354111)

Shreiner, T. L. (2018). Data literacy for social studies: Examining the role of data visualizations in K-12 textbooks. *Theory & Research in Social Education, 46*, 194-231. doi:https://doi.org/10.1080/00933104.2017.1400483

Shreiner, T. L. (2020). Lies, damned lies, and statistics in social studies: How social studies state standards address data literacy across grades and disciplines. *Journal of Curriculum Studies*.

Shreiner, T. L., & Dykes, B. (2020). *Teaching Data Literacy for Social Studies: Teacher Practices, Beliefs, and Knowledge* Paper presented at the American Educational Research Association Annual Meeting, San Francisco, CA.

Vosniadou, S., De Corte, E., Glaser, R., & Mandl, H. (2012) The use of multiple, linked representations to facilitate science understanding. In*. International perspectives on the design of technology-supported learning environments* (pp. 51--70): Routledge.