Putting a Teaspoon of Programming into Other Subjects (v3)

By
Mark Guzdial, Emma Dodoo, Bahare Naimipour, Tamara Nelson-Fromm, Aadarsh Padiyath

Programming is such a powerful tool that as early as the 1960's scholars like C.P. Snow and Peter Naur were worried about its potential negative impacts on society[1]. They called for everyone to learn to program in order to democratize access and to inform citizens about how computational processes worked. At the same time, Alan Perlis argued for teaching programming to all university students because of the disciplinary benefits. He saw that computing gave us a new way to understand in many disciplines. His words have proved to be prescient.

Historians, scientists, humanities scholars, mathematicians, and artists today use programming to advance the goals of their own disciplines. They are using programming for their own agendas, for problems other than professional software development. These professionals are likely using a broad range of tools, like Excel, MATLAB, Python, R, and JavaScript. Domain-specific languages (DSLs) are a class of languages designed explicitly for domain experts to use in solving problems for which programming is a useful tool.

We aim to bring domain-specific programming into classrooms. Only about 5% of American high school students take a course in computer science[2]. If we were to integrate programming into a broad range of classrooms, especially in subjects other than computer science, we might help to democratize access to programming and give all students the opportunity to learn programming for all the reasons that Snow, Naur, and Perlis raised, not for the purpose of creating more software developers.

DSLs are designed for solving problems in the domain. Students do not know the domain nor programming. Teachers outside of computing have domain expertise, but typically no programming experience. They are also gatekeepers. Teachers make the decisions for what to bring into their classroom and what will help their students the most.

Our approach is to make programming more accessible by making new languages that are even easier to use than DSLs. We work with social studies and mathematics teachers who do not currently use any form of programming in their classes. They have no interest in learning programming for its own sake. However, they want to incorporate activities that require students to program, as a way to learn more within their discipline. We design programming languages with and for these teachers. We create ***Task-Specific Programming (TSP) languages*** since they are made just for the teacher's tasks. We call these ***teaspoon languages*** – we are adding a teaspoon of programming into other-than-CS subjects.

The teaspoon languages we have built so-far have three defining characteristics:

[1] https://computinged.wordpress.com/2021/11/26/computer-science-was-always-supposed-to-be-taught-to-everyone-but-not-about-getting-a-job-a-historical-perspective/
[2] https://advocacy.code.org/stateofcs

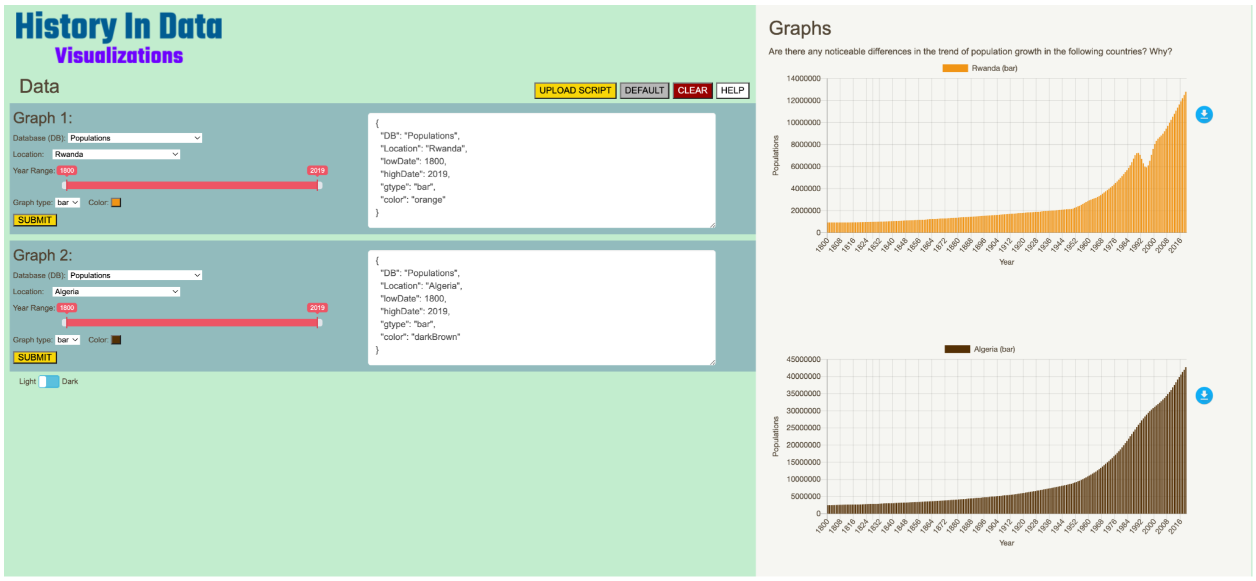
1. They can be used by students for a task that is useful to a teacher. Integrating into formal, mandatory schooling through teachers is the best way to get broad access and participation.
2. They are programming languages, i.e., a notation for defining a computational process.
3. They can be learned within 10 minutes, so students can *learn* and *use* them within a single class session. A larger language requires more time to learn. To be worthwhile, that cost has to be amortized across many sessions to be worthwhile. That is a big investment for a teacher who hasn't used programming previously..

Participatory Design of Teaspoon Languages

We started building teaspoon languages in collaboration with Dr. Tammy Shreiner, a history professor at Grand Valley State University. All US states require Social Studies classes (including history) to teach data literacy, but not all teachers include it in their classes. We have brought a variety of data visualization tools to her course for teachers, *Data Literacy for Social Studies*. We have her students try them out and tell us what would fit best into their courses. Some of these tools required programming and others did not.

Some teachers valued programming for specifying data visualizations. With a drag-and-drop tool, it's not always easy to understand how you got to a particular visualization or how to change it. Social Studies teachers often want to generate two visualizations for comparison, so efficiently specifying multiple visualizations is important to them. Teachers appreciated the declarative programming language in one of the tools we showed them, Vega-Lite, for its simplicity, clarity, and parsimony.

Figure 1 is a screenshot of a teaspoon language for data visualizations that we have developed. In *Data Visualization for Learning (DV4L)*, the two visualizations appear on the right. The code for defining the visualizations is in the center, inspired by Vega-Lite. Menus for controlling the visualization are on the left. These are linked. Changing the code changes both the graph and the settings in the menus. Changing the menus changes the graph and the code. There is a "Driving Question" above the visualizations on the right. Social Studies teachers told us that students need to be reminded *why* they're playing with visualizations, so we built the driving question into the interface.



Graph 1:
Database (DB): Populations
Location: Rwanda
Year Range: 1800 – 2019
Graph type: bar Color: 🟧

```
{
 "DB": "Populations",
 "Location": "Rwanda",
 "lowDate": 1800,
 "highDate": 2019,
 "gtype": "bar",
 "color": "orange"
}
```

Graph 2:
Database (DB): Populations
Location: Algeria
Year Range: 1800 – 2019
Graph type: bar Color: 🟫

```
{
 "DB": "Populations",
 "Location": "Algeria",
 "lowDate": 1800,
 "highDate": 2019,
 "gtype": "bar",
 "color": "darkBrown"
}
```

Are there any noticeable differences in the trend of population growth in the following countries? Why?

Figure 1: DV4L Scripting

We use participatory design methods to create teaspoon languages. We put existing programming languages in front of teachers as *design probes*, guide their use of the tools, then ask them to reflect on what worked and what didn't. We typically work closely with a *co-designer*, like Dr. Shreiner, who is both a teacher and a domain expert. The teachers who participate in our sessions are *design informants*. They inform us about their values, like the importance of having two visualizations at once, having the driving question be visible at all times, and of what they value in a code notation.

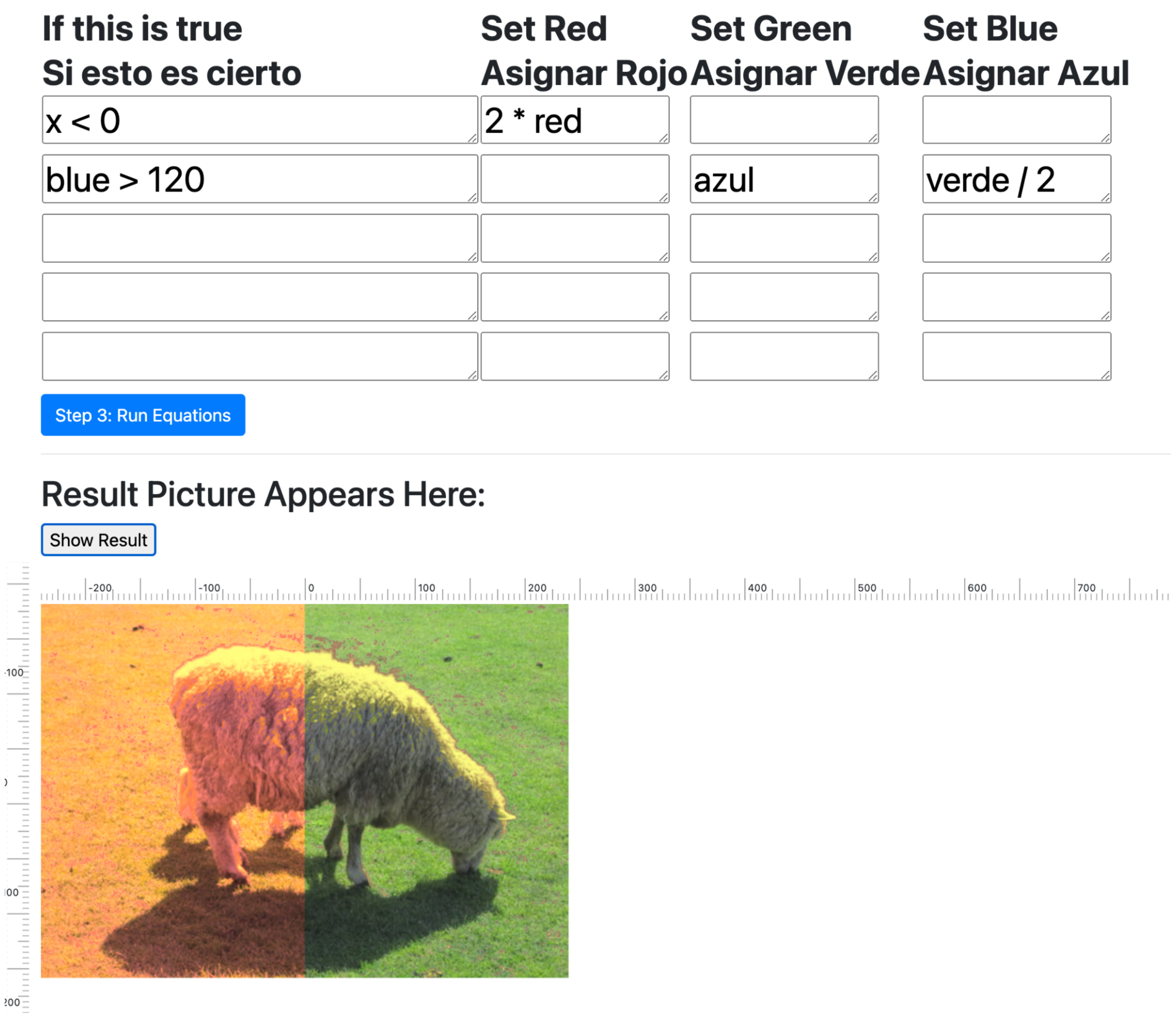
Pixel Equations (screenshot in Figure 2) is a teaspoon language for mathematics classes. We developed Pixel Equations for a collaboration between a Detroit Public School and robotics faculty at the University of Michigan. The Detroit school wanted to develop an Engineering course for their 11th grade students (about 16 years old) where they would be asked to visualize data from a sensor. We interviewed undergraduates who had solved the same problem in their robotics course to identify what was difficult about the task. We also worked with mathematics teachers and mathematics education researchers to identify learning objectives in the task.

Pixel Equations is used to define image filters. We identified three learning objectives that were particularly difficult for the undergraduates and were important to the teachers.

1. Equations that define parts of a plane (e.g., `x < 0`) can be used to define regions of a picture to which a filter might be applied.
2. We can use mathematics to define colors (e.g., to define the amount of red, green, and blue in a color).
3. Pixel colors can be queried (e.g., increase the red wherever `blue > 120`) to specify pixels to change.

In Pixel Equations, regions are specified in the first column. Wherever the logical expression is true, the right side color specification is applied. The three columns to the right specify the red, green, and blue components for the pixel color. Expressions can use the old value of the color to define a new value (e.g., `2 * red` to double the red in a color).

A Pixel Equations program is a program without **for** loops or explicit **if** statements. It's clearly limited in scope and only good for one task (defining image filters). Nonetheless, it's a motivating task that addresses learning objectives relevant to engineering, mathematics, and art classes.



If this is true Si esto es cierto	Set Red Asignar Rojo	Set Green Asignar Verde	Set Blue Asignar Azul
x < 0	2 * red		
blue > 120		azul	verde / 2

Result Picture Appears Here:

Figure 2: Pixel Equations

Building in support for Multilingual Students

Figure 2 demonstrates one of the features that we have been exploring in teaspoon languages. Pixel Equations supports the use of *red*, *green*, and *blue* for specifying color components, and also *rojo*, *verde*, and *azul*. Since we are defining entirely new programming languages, we do not have to be constrained in our choice of words. We do not have to maintain the hegemony of the English language in programming.

Wherever we can, we support non-English choices for keywords. A different teaspoon language, for defining chatbots, supports the mapping of keywords to many languages. We currently include English-based, Spanish-based, and Spanglish-based chatbot languages. We built our *Charla-bot* teaspoon language based on an explicit challenge from Dr. Sara Vogel, whose work on translanguaging in bilingual CS classes inspired us. We asked her to review an earlier version of the chatbot tool and asked, “Would you like this with Spanish keywords?” She responded, “Could you make it so that the students could pick their own keywords?”

Research Challenge #1: Programming is Interpretation

While our teaspoon languages are understandable and usable in under 10 minutes, the users are still programming. Whenever you are programming, you are specifying a process for a computational agent who doesn’t understand the human world. Mismatches occur between desired intent and actual execution. Teaspoon language programmers still have to learn how to debug.

Our work with teaspoon languages gives us new insights about what students need to learn to develop “computational thinking”. Beginning programmers have to understand that programming is not like a word-processor or presentation tool. The input (program) does not look like the output (execution). There is a process of interpretation or translation by the computer on the input to get to the output. Many users of smartphone or web apps have never experienced an explicit process of interpretation of a notation. We are working to help teaspoon language users to navigate programming bugs for the first time.

Research Challenge #2: Adoption

While our teaspoon languages score highly with teachers on measures of usability and usefulness, they have not been widely adopted. There are many barriers to adopting programming for the first time. Not many history teachers have computers for all their students. New Social Studies teachers might have learned about building data visualizations, but they are unlikely to push their more senior colleagues towards new methods in their first years. When pressed for time, teachers often jettison the technology first.

We are conducting studies to understand the factors that influence adoption. Today, we work closely with a handful of adopting teachers, to understand what works for them and how we can better facilitate their use. These teachers use teaspoon languages because they believe that they help students learn in their subject.

An Interdisciplinary Research Direction to Broaden Participation and Access

Teaspoon languages are a new target for programming language design, developed with HCI design processes with non-programmers to achieve education goals. Using Teaspoon languages really is programming, in the way that Snow, Naur, and Perlis meant it. Teaspoon languages are about democratizing access to the activity, about increasing understanding of computational processes, and about applying programming to advancing disciplinary goals. Maybe students will get interested and want to do more programming, but our goal is to meet the teacher’s disciplinary needs, not convert students to computing.

We see teaspoon languages as a strategy for broadening participation in *programming*. Few American high school students take a computer science class, but 100% of these students take history, mathematics, and science. Helping to integrate programming across the curriculum gives *all* students the opportunity to discover programming and how it can be useful to them in their lives and careers, even if they never become part of the computing field.