Creating New Programming Experiences Inspired by Boxer to Develop Computationally Literate Society

Submission to Boxer 2022

Mark Guzdial, University of Michigan

Reaching Computing for All, Including Those Who Don't Making Spreadsheets

The spreadsheet is a great success in making programmable computational media available to a broad audience. Spreadsheets meet the user where they are – they can be used at various levels of computational sophistication. Bonnie Nardi explored this range, from those users who simply plug-in values, to those who define formulas, and finally to those who create new macros for those who define formulas (Nardi, 1993).

We probably would not say that those users who are simply plugging in values are developing "*computational literacy"* as diSessa defined it (diSessa, 2001). They are not developing the ability to read and express themselves in computational terms. They are not developing meta-representational competence (Disessa & Sherin, 2000). In my work with my colleague Dr. Tammy Shreiner, I get to interview pre-service social science teachers who tell me that they are competent at making data visualization "as long as it's with only pencil, paper, and ruler." They find spreadsheets too difficult or too confusing. Another told me that he can use Excel, but not Google Sheets. There is a class of users who aren't comfortable with spreadsheets.

In this paper, I describe a research agenda addressing the goal of reaching "*CS for All*," including those users/students for whom spreadsheets are overly complicated. How could we lower the barriers to programming so that it is approachable by all students? What kinds of meta-tools might be used take to create those kinds of highly-usable programming activities?

Teaspoon Languages: Task-Specific Programming Languages for Teachers' Tasks

In my research group, we are developing *Teaspoon languages.* Teaspoon languages are a form of task-specific languages (TSP, which is also an abbreviation for "teaspoon"). Teaspoon languages:

- Support learning tasks that teachers (typically non-CS teachers in our participatory design sessions) want students to achieve;

- Are programming languages, in that they specify computational processes for a computational agent to execute; and
- Are learnable in less than 10 minutes, so that they can be learned and used in a one hour lesson. If the language is never used again, the learning cost was low enough that it was worth the value of adding computation to the one lesson.

We say that *we're adding a teaspoon of computing to other subjects.* The goal is to address the goal of "CS for All" by integrating computing into other subjects, by placing the non-CS subjects first. We believe that programming can be useful in learning other subjects. Our primary goal is to meet learning objectives outside of CS using programming. Teachers (and students eventually) will be learning foundational CS content — but not necessarily the content we typically teach in CS classes. All students should learn that a program is non-WYSIWYG, that it's a specification of a computational process that gets interpreted by a computational agent, that programming languages can be in many forms, and that all students can be successful at programming. We focus on items at the earliest stage of computational learning trajectories (Rich et al., 2017).

Teaspoon languages offer an alternative strategy to achieve universal computational literacy. Let's set up a contrast between the *Hour of Code* from Code.org and Teaspoon languages — not that you'd have to choose. You could easily do both. The Hour of Code makes sure that students see CS as computer scientists see it, with a Turing-complete language, once a year for one hour. Teaspoon languages could appear multiple times in multiple classes every year, even though the CS is minimal. We might consider the relative strengths of each model. Which one leads to more CS learning for the long run? Which creates more of a school culture where programming is used across disciplines? Which one leads to students seeing how CS can be used everywhere?

Our Teaspoon languages owe a design debt to Boxer (diSessa & Abelson, 1986). They draw on the ideas of the box containing not just data (values, in the spreadsheet sense) but also program representations. In this paper, I describe two of our Teaspoon languages and how they connect to Boxer. Finally, I offer a third example of how the ideas of Boxer might lead to insights about tools for building Teaspoon languages.

Example #1: Pixel Equations

Pixel Equations was designed for an Engineering course in Detroit Public Schools where students were learning to visualize data. Pixel Equations is a Teaspoon language that has students program image filters. It was created for three learning objectives.

Figure 1: Pixel Equations

- First, that equations that describe sections of a graph can also describe sections of a picture. The equation is independent of the graph or the picture — we could select any input picture and apply the same filter to it.
- Second, that colors can be specified by equations, like the value *255* or the expression *2 * green*.
- Third, that we can select pixels with a conditional that tests the value inside the pixel. For example, we can select just those pixels whose blue channel is greater than 200.

Pixel Equations is simple enough that it's being adopted by middle school classrooms. Used at the simplest level, it's much like entering values into a spreadsheet. But it can be also used in a slightly more sophisticated manner – what's inside the "boxes" are equations. The boxes in Figure 1 really do contain program code. Those are Boolean expressions and mathematical expressions that might appear in JavaScript, Python, or Boxer. But all the other trappings of a programming language have been stripped away. There are no flow of control keywords, no loops, no creation of variables, and no functions. This is a low threshold step into programming.

Example #2: Counting Sheets

Counting Sheets is a Teaspoon language inspired by Elise Lockwood's research on teaching students about counting problems through the use of Python (Lockwood & DeChenne, 2019). Lockwood teaches students about counting problems (e.g., "How many two letter words can be formed from the letters ROCKET?" or "If you have three letters and four digits, how many license plates can you generate?"), by having students work with Python code that actually

generates all the possible outcomes. The Python code becomes a representation of process that can guide students' thinking about the counting process.

Counting Sheets supports expressing counting processes to generate outcomes, but with a spreadsheet-inspired notation. Figure 2 shows a program in which the first column will be drawn from ROCKET, the second column will use the items from the first column (***data1***) but not repeating whatever was actually selected for column 1 (***minus item1***) and only taking items *after* the current position (***index1***) when we don't care about order (e.g., we don't want both OR and RO). The third column is drawn from the second column (***data2)***, but subtracts ***item2*** (no replication), but using the Spanish keyword ***menos*** for *less*. Computing for All should include students who do not speak English or who want to use words from multiple languages (Salas et al., 2021; Sara et al., 2019).

The boxes in Counting Sheets are like spreadsheet cells, but the interface is simpler – no ribbons of icons, no menus, no graphing tools. Cells can contain literals or formulas. The execution model is different than in spreadsheets. Instead of evaluating each cell to determine its value, Counting Sheets exhaustively generates every possible combination of the items in the cells. It's data and code in the boxes, but with a different model of execution. It's still using a spatial metaphor, but there is no naïve realism. And it's a totally different model than the first example.

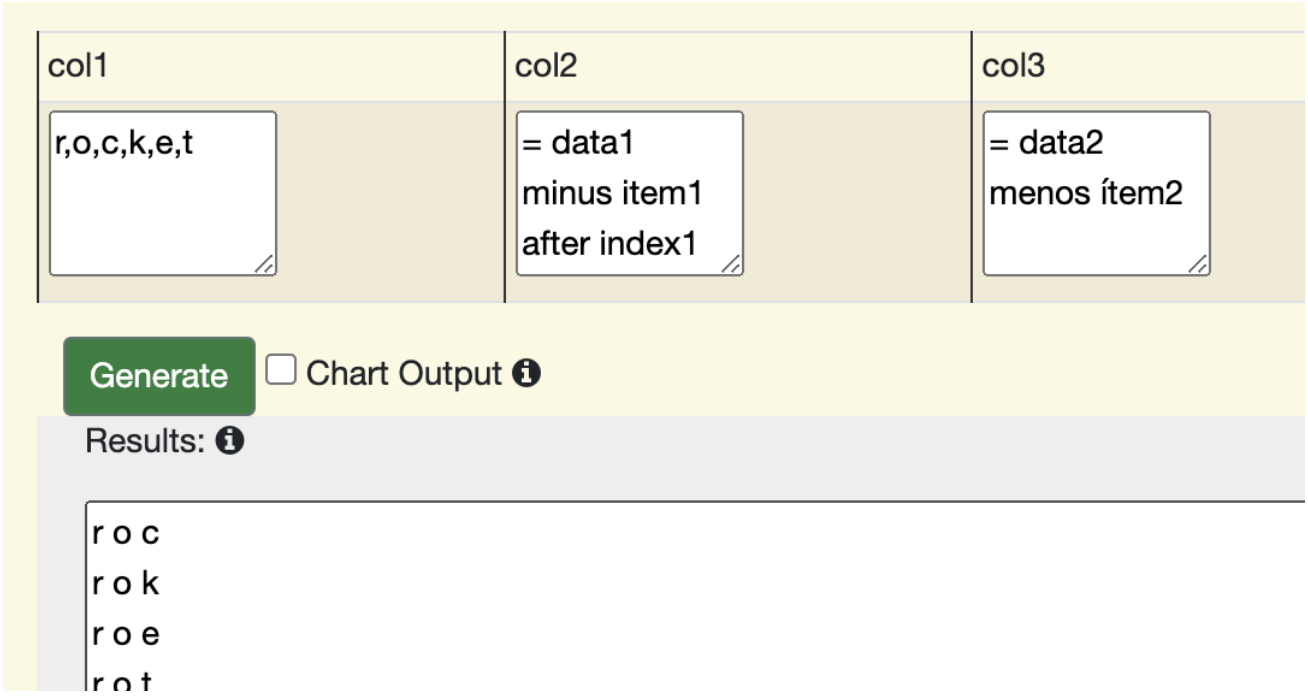

Figure 2: Counting Sheets Program for Three letter Words from ROCKET

Meta-Example 2: Counting Sheets underneath

For Teaspoon languages to succeed as a strategy towards computing for all, we will need many different Teaspoon languages – so that two or three will fit into non-CS subjects at every grade level. How will we generate so many languages? I suggest that Boxer offers some strategies.

I implemented the interpreter engine for both Pixel Equations and Counting Sheets. I'm a "senior" (read: old) faculty member in a CS department. Few full professors in CS program anymore. I am not particularly gifted in programming language development such that I can implement languages using tools standard tools like lex and yacc in short amounts of time. Rather, I cheat.

I am a longtime HyperCard programmer. Back in the late 1980's and early 1990's, I built many HyperCard stacks. In the HyperCard style, many (or even *most*) of your variables were visible as "fields." By simply extending the concept of those fields to Boxer's boxes, HyperCard becomes a tool for rapidly developing new Boxer-like tools. HyperCard is long-defunct, but *LiveCode* is a modern implementation Figure 3 shows the actual engine underlying the Counting Sheets interface shown in Figure 2. Counting Sheets is a Web page that calls the server implementation of LiveCode and uses Figure 3 as an interpreter.

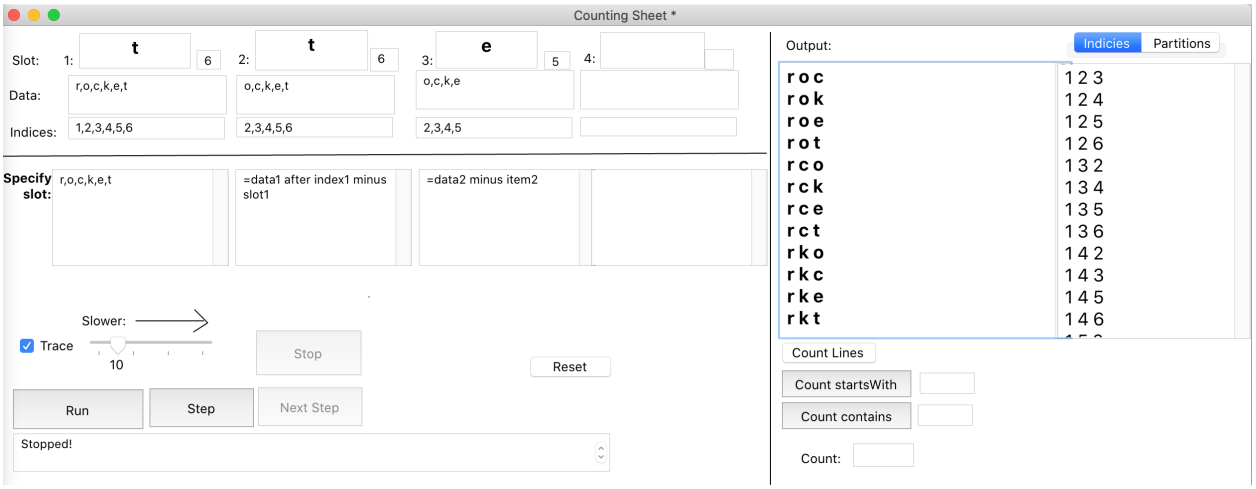

Figure 3: LiveCode stack implementing Counting Sheets, stopped mid-execution

Boxer may offer us more than a way of thinking about supporting computational literacy for all users. Boxer may also offer us a way of supporting rapid development of the underlying tools. Figure 3 shows the underlying representations of the Counting Sheets interpreter, all of which look like boxes in that they contain values, data representations, and programmatic code.

Boxer was built on ideas of a spatial metaphor and *naïve realism* (diSessa, 2001). The computational model of the system was visible in front of the user, and changing the text in the box was changing the value in the system. That's exactly how I implement Teaspoon languages in LiveCode. I can't keep all the variables in my head anymore, so I put them in boxes (fields) on the screen. I forget how the system is working, so I display internal variables, too. The features

which make Boxer approachable are also features that make implementing Teaspoon languages easier. There probably is a limit to this advantage. I may not build an entire operating system or a distributed transaction processing system using a spatial metaphor and naïve realism. However, for implementing new computational models and languages, it works well and allows us to build quickly.

Conclusion: Designing for a Computationally Literate Society

We build Teaspoon languages with an assumption that is antithetical to the original Boxer vision. Boxer was an attempt to develop a universal programming medium which might be used from kindergarteners to career professionals (diSessa, 2001). Rather than defining a universal programming medium that might be used at any age, Teaspoon languages expose just a small slice of programming with a tiny execution model in a task-specific interface and syntax. Teaspoon languages are rarely Turing-complete. You would be lucky to use them for more than one task, let alone across an entire domain and never between domains. However, we would hope to have many Teaspoon languages that, in total, would be about universal programming for any age.

Teaspoon languages are not offering a universal computational language. Rather, Teaspoon languages are about creating a *value* for universal computational literacy, with many *different* programmable representations. Teaspoon languages aim to be as close to the teacher's task (including their discipline and context) as possible, so that teachers can use computation as a tool to facilitate disciplinary learning. What we hope to transfer between tasks, domains, contexts, and ages is a value for using computation and multiple representations. That value is the basis for a universally computationally literate society. Boxer inspires this vision.

References

diSessa, A. (2001). *Changing Minds*. MIT Press.

diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM, 29*(9), 859-868.

Disessa, A. A., & Sherin, B. L. (2000). Meta-representation: An introduction. *The Journal of Mathematical Behavior*.

Lockwood, E., & DeChenne, A. (2019). Enriching Students' Combinatorial Reasoning through the Use of Loops and Conditional Statements in Python. *International Journal of Research in Undergraduate Mathematics Education*.

Nardi, B. A. (1993). *A small matter of programming: perspectives on end user computing*. MIT press.

Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2017). K-8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals.*ICER '17* Proceedings of the 2017 ACM Conference on International Computing Education Research, New York, NY, USA.

Salas, C. C., Gonzales, L., Evia, C., & Pérez-Ouiñones, M. A. (2021, 23-27 May 2021). Ideology of monolingualism: How ignoring bilingualism makes society less inclusive. 2021 Conference on Research in Equitable and Sustained Participation in Engineering, Computing, and Technology (RESPECT),

Sara, V., Christopher, H., Laura, A.-M., & Kate, M. (2019). *The Role of Translanguaging in Computational Literacies: Documenting Middle School Bilinguals' Practices in Computer Science Integrated Units* Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA.