Task-Specific Programming Languages for Promoting Computing Integration: A Precalculus Example

Mark Guzdial
mjguz@umich.edu
University of Michigan, Computer Science & Engineering
Ann Arbor, Michigan

Bahare Naimipour
baharen@umich.edu
University of Michigan, Engineering Education Research
Ann Arbor, Michigan

ABSTRACT

A task-specific programming language (TSPL) is a domain-specific programming language (in programming languages terms) designed for a particular user task (in human-computer interaction terms). Users of task-specific programming are able to use the tool to complete useful tasks, without prior training, in a short enough period that one can imagine fitting it into a normal class (e.g., around 10 minutes). We are designing a set of task-specific programming languages for use in social studies and precalculus courses. Our goal is offer an alternative to more general purpose programming languages (such as Scratch or Python) for integrating computing into other disciplines. An example task-specific programming language for precalculus offers a concrete context: An image filter builder for learning basic matrix arithmetic (addition and subtraction) and matrix multiplication by a scalar. TSPLs allow us to imagine a research question which we couldn't ask previously: How much computing might students learn if they used a multiple TSPLs in each subject in each primary and secondary school grade?

CCS CONCEPTS

• **Social and professional topics → K-12 education**; *Adult education*;

KEYWORDS

human-computer interfaces, programming languages, precalculus, participatory design

ACM Reference Format:
Mark Guzdial and Bahare Naimipour. 2019. Task-Specific Programming Languages for Promoting Computing Integration: A Precalculus Example. In *19th Koli Calling International Conference on Computing Education Research (Koli Calling '19), November 21–24, 2019, Koli, Finland.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3364510.3364532

1 INTRODUCTION

Computing is a transformational technology that influences students' daily lives. Yet, few students learn about this technology. In the few US states for which we have data, less than 1% of high school students take any computer science class, despite 30–50% of high schools in those states offering computer science [25]. The US national framework for K-12 CS Education [47] recommends that students should be able to use computing as a true literacy, for both reading (consuming or using computation) and writing (creating). How do we engage more students in gaining computational literacy? We have shown that learning computing in a *context* (such as manipulating digital media in Media Computation) can dramatically improve student engagement and retention within the discipline [14], in part because students have a greater sense of relevance and utility [15, 32].

Computing education research has supported the hypothesis that context can make computer science learning more successful. We see a historical argument for using other STEM subjects as that context [19, 26]. The goal for integration is to make a positive feedback loop. Consider physics, as an example. Learning computer science in the context of physics can lead to better learning of physics (e.g., as seen in [13, 38]) and can also make the computer science more relevant and useful [15].

We are exploring the hypothesis that we can using computing to provide a context that enhances learning in *other* subjects. For example, precalculus is a course that has the potential to be taught more effectively through the use of computing. The abstractions of precalculus (e.g., vectors and matrices, trigonometric functions, sequences, and series) are powerful and applicable to many domains, and programming can give students the opportunity to use these concepts in concrete and motivating contexts, such as media manipulation, 3-D animations, and music. Precalculus skills are critical for success in STEM education [33]. While precalculus enrollment is small (about 20% of US 11th graders [1]) compared to subjects like algebra or biology, precalculus courses in the US enroll more than double the *total* enrollment for computer science classes in the US [16, 25]. Can computing provide a concrete context for precalculus, the way that Media Computation provided a context for computer science?

The trick is to make "programming" *fit* precalculus (or other subjects), so that there is a synergy and not a competition for student attention. Programming experiences differ, and those differences matter. The notation used for programming (the *programming language*) plays a critical role in student success in programming and what they learn from the experience. For example, there are mistakes that students make in textual languages that they rarely make in blocks-based languages [43]. Most importantly, there is a difference in meta-representational power between different programming languages. The programming language can be a scaffold to promote the development of conceptual understanding [4, 5, 44].

Koli Calling '19, November 21–24, 2019, Koli, Finland

ACM ISBN 978-1-4503-7715-7/19/11…$15.00
https://doi.org/10.1145/3364510.3364532

In the end, it's the classroom teacher who makes computer science integration in the classroom work. The "teacher effect" explains more variance in student learning than any other factor [9]. Their "will and skill" and comfort with the "tool" determine the success of any technology integration effort [27]. *It's surprising, then, that the teacher is the stakeholder who typically has no voice in choosing the programming language used in integration efforts.* Tools are typically provided to teachers pre-defined. The reason in the past was the complexity of the design space. Designing and implementing programming languages has been a challenging problem. Teaching teachers and curriculum designers a new programming language takes time and effort.

Making new languages has been hard, but modern research in programming languages has reduced the complexity of creating new languages considerably. We have significant new tools for generating new *domain-specific programming programming languages*, such as the Racket language-oriented programming supports [7, 41], and Pharo meta-programming tools [6, 28]. Empirical studies show that domain-specific programming languages are faster to learn and lead to fewer errors than programming in general purpose programming languages [20].

The term *task-specific programming languages* [17] describes a new set of domain-specific programming languages in which users can be successful and complete tasks in literally minutes. Task-specific programming languages are domain-specific languages (in the programming language design sense) that have been designed around supporting specific user tasks (in the human-computer interface design sense). Probably the best example currently in the research literature is *Rousillon* is a task-specific programming language for web scraping, i.e. gathering data from Web pages, which is a common data science task [2]. Rousillon combines a programming-by-demonstration interface and a blocks-based programming language (Helena). Empirical studies comparing Rousillon with a popular web scraping language (Selenium) find that users of Selenium finish tasks in 25 minutes that complete novices can solve in Rousillon in 10 minutes.

1.1 Task-specific programming languages as microworlds

A powerful way to think about task-specific programming languages is as a *microworld*. Seymour Papert first defined microworlds [24] 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 built on this idea in Boxer, and said in his book *Changing Minds* [3]: "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, e.g., Logo for Papert and Boxer for diSessa. 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 Logo and Boxer, like any general-purpose programming language, take time to develop proficiency.

A task-specific programming language (TSPL) aims to provide the same easy-to-understand operations for a microworld, but with a language designed for a particular purpose. While that limits the abstractions and concepts that can be used, it makes it possible to think about different microworlds, i.e., different task-specific programming languages, in the same course. Perhaps an elementary or secondary school student might encounter several different TSPLs in a single year.

2 EXPLORATION OF TASK-SPECIFIC PROGRAMMING

We are currently exploring task-specific programming in two domains: social studies (specifically, history) and precalculus.

- For history, we are supporting the work of Tamara Shreiner who is developing a data literacy curriculum for social studies. Social studies curricula are increasingly relying on data visualizations [39], but teachers are uncomfortable with existing visualization tools [23]. We are testing new task-specific programming languages with history teachers.
- For precalculus, we are creating prototype task-specific programming languages to use as starting places for teacher commentary.

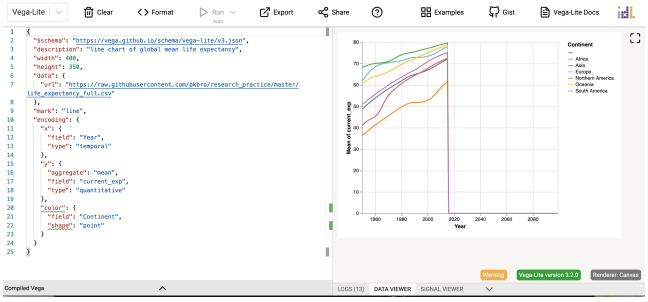
In both examples, we use a participatory design framing. We ask teachers in history and precalculus to be informants in a design process. In neither domain is there an existing practice of using computation. We provide examples that we *expect* will be insufficient, in order to prompt our informants to explore what might really help them. By starting with teachers, we believe that we dramatically increase the odds that the products we produce will be usable and adopted by teachers. We are explicitly designing task-specific programming languages as a strategy to integrate computing into other clases.

2.1 Pilot Study with Social Studies Educators using a TSPL

In a pilot study we ran in March 2019, we asked social studies educators to use a task-specific programming language, Vega-Lite [34], that they had never seen before to build visualizations in less than 20 minutes. Vega-Lite is only a language for information visualization – not for calculation or any other task. But that specificity makes for ease of use.

Our participants were pre-service social studies teachers taking a course on data literacy. We offered the teachers the opportunity to build their own visualizations using two languages with high domain authenticity [37]. Both languages accessed the same data source (UN data on life expectancies in different countries and on different continents) and generated the same initial visualization.

Figure 1: Vega-Lite Code and Editor as Used in the Pilot Study



- Activity 1 used JavaScript with the Google Charts [12], an industry-standard language and visualization package. The JavaScript code was 45 lines long and included an SQL query to access the data, and HTML for formatting the visualization. We used JSFiddle to provide a browser-based programming environment where the code and visualization would both be visible.
- Activity 2 used Vega-Lite [34], a research product that is being used by data scientists and computational journalists. The comparable Vega-Lite program was 25 lines code in a JSON format. Vega-Lite provides its own editor with specialized features, e.g., hovering over a keyword provides documentation and suggests for alternatives. (See Figure 1.)

We asked the teachers to pair up, and then randomly gave them one of two activity templates, following the process of Wilkerson [45]. Each introduced the data set, and then linked to the program in an in-browser editor. We guided them to make some small changes to the program (e.g., visualizing a different variable) then gave them options to explore. We gave the groups 20 minutes to explore, then led a discussion about their experience with prompts asking them about their design preferences of a programming language for social studies education.

All groups were able to generate a visualization, which was itself an interesting finding. Our teachers had very little prior experience with programming, with several participant commenting that the most similar activity they had ever had was changing the background of their MySpace page in middle school. Most groups who started with Vega-Lite were also able to try JavaScript, but not all the groups who started with JavaScript were to also complete the Vega-Lite visualization. The JavaScript groups complained that the program was "overwhelming" (repeated several times in both the discussion and post-survey), that the complexity "distracted from the data," and that they could not find good documentation. They told us that when they were making changes, "the intuitive way *didn't* work. We hunted all over to figure it out. But we couldn't." Working in pairs was a critical part of their social support, and several teachers told us how they made more bold changes to their visualization because they had a partner to help them back out if they failed.

When we asked teachers what tool they preferred if they were going to make their own visualizations, Vega-Lite was the favorite, but mostly because of the environment. Teachers preferred the documentation, and how the editor helped with syntax (with the hover-over tips). They also liked how the new visualization came up immediately when they made an edit, without hitting the Run button. The environment reduced the complexity of the task. They also suggested applications of the visualization languages in both History and Economics.

Several teachers compared the use of the programming language to Excel. They preferred the use of code for generating visualizations because of the ease of exploration. One teacher said that she'd never figured out "all the mouse clicks…I haven't watched the YouTube videos" to build sophisticated visualizations in Excel. "Here, it's a single word. It can be learned."

This pilot study gives us confidence that participatory design sessions can give us important feedback on task-specific programming. Teachers were able to complete the activities, evaluate the options in terms of the variables of interest, and give us new insights that we had not had previously. We are planning a second participatory design session for Fall 2019 where we will compare Vega-Lite to CODAP, a data analysis and visualization platform explicitly designed for high school students [8]. We are also designing a new visualization tool explicitly for Shreiner's curriculum, informed by our teacher-informants.

2.2 Prototype: Building Image Filters as a Precalculus Activity

In history, we have collaborators and teachers to work with. In precalculus, we have not yet found collaborators, so we build prototype task-specific programming environments as prompts, foils, and provocations – an artifact to respond to. We are using our initial prototypes in participatory design sessions with precalculus teachers, as in our session with social studies educators using Vega-Lite. Evidence on the use of prototypes in participatory design suggests that too "finished" a product squashes criticism and discussion [42]. Few design informants will question a beautiful design. We are aiming for tools that provoke teachers to tell us what they really need and want.

Our goal is to develop a tool that teachers would actually adopt. By inviting precalculus teachers to help design the language, we dramatically increase the odds that the tool might be adopted by real teachers, according to implementation science ([18]) and from research on moving research-based interventions into practice [10, 40].

Our first prototype allows students to define image filters (Figure 2) through matrix manipulations in the precalculus curriculum. The front page of the application shows an initial picture (lower left hand corner), a list of matrix operations (upper left corner), and a transformed picture (upper right hand corner). In some versions, we have developed a pixel-by-pixel inspector to compare the two pictures (lower right corner). In text and using standard mathematical matrix notation, we describe how the picture is composed of a red, green, and blue channel matrix.

On subsequent cards or pages, students describe matrix manipulations in English, via radio buttons and pull-down menus (Figure 3 and Figure 4). Students might choose to change the matrix of colors in a picture (top of Figure 3) by adding or subtracting matrices

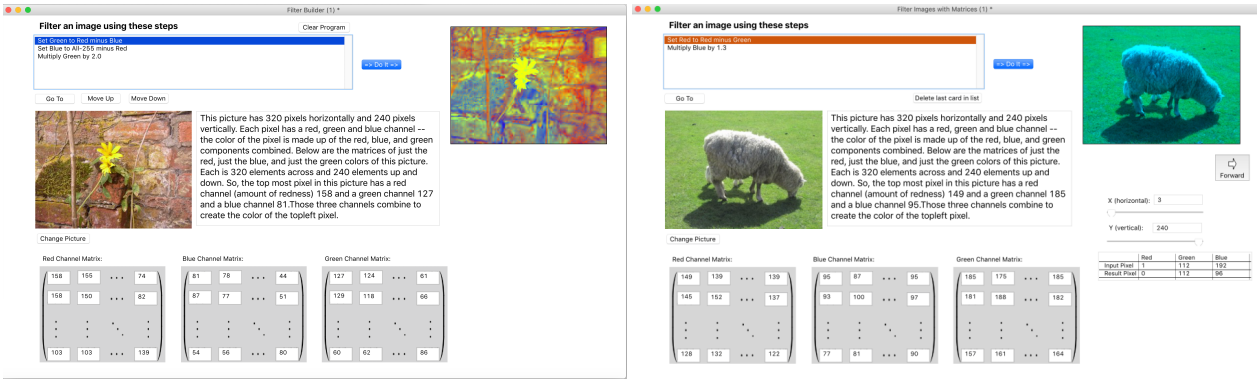


Figure 2: Two examples of a complete filter which is a program composed of multiple matrix manipulations.

which are selected via pull-down menus. The available matrices are red, green, blue, or *all-255*, a matrix of the same size where all cells contain 255 (the maximum value of a channel). Students might alternatively choose (via a radio button) to multiply a matrix by a scalar (e.g., to reduce or increase red in a picture) as in Figure 4. Once a student defines a matrix manipulation, the manipulation is explained to the student using the mathematics language and notation as it might appear in a precalculus textbook. In this way, the student's understanding of the matrix manipulations is rehearsed and connected to the classroom context.

The collection of operations, in sequence, is then a program which implements an image filter which can be applied to an arbitrary picture (Figure 2). Our goal is to make more concrete and relevant the precalculus matrix manipulations.

2.3 Response from precalculus teachers

We have had a few one-on-one participatory design sessions with precalculus teachers. We begin the session by asking the teacher what students find difficult about precalculus. What is hard to teach, and what is hard for students to learn? Then we ask about the purpose of matrix manipulations and wave functions (the context for our second prototype) in precalculus, to prime them to think about the concepts on which we are focusing. Then we show them the prototypes. The key questions are whether the tools are usable and meet a need – and if not (which is what we expect), what might be useful.

In general, most mathematics teachers are not excited about this prototype. While the image filters are "cool," the matrix manipulations we are practicing are the "easy" ones. Few students struggle with matrix addition and subtraction or with scalar multiplication. Teachers all agreed that this was simple and usable enough to fit into a single class session. Teachers appreciate our attention to disciplinary literacy [11, 22], in that we use the communications standards of mathematics, e.g., the matrix notations, operations, and language as they appear in precalculus textbooks, not as they appear in most programming languages. They have made several suggestions about what students find challenging in precalculus. With the teachers, we made sketches of new task-specific programming languages for precalculus, in order to get their in-the-moment feedback on our next generation of prototypes. Low-fidelity prototypes, like sketches, are common in HCI design [21, 46]).

3 CONCLUSION: IMAGINE A WORLD WITH MANY TASK-SPECIFIC PROGRAMMING LANGUAGES

TSPLs raise a tantalizing possibility for integrating computing into other disciplines, one that we could not think about with only our existing general purpose programming languages. If a TSPL can be learned and used within a single class session, then we might imagine several of them being used in a single course. We might have several in each course We might imagine a student studying algebra with Boostrap [35, 36], using Vega-Lite for data visualization in history class, and later using matrices to define image filters, as with our prototype.

How much computing might a student learn if they used multiple TSPLs before entering their first course specifically on computer science? Might they generalize some ideas about computing across different languages, and enter their CS course with a strong intuitive sense of "program," "programming," and "computation"? We are not arguing for *transfer*. Rather, we suggest that students might learn the first concepts in learning trajectories [29–31], where the focus is on the causal and repeatable nature of programs. Students might learn what a program *is*.

We are just starting to explore the research questions about task-specific programming. TSPLs may be even easier to use than block-based programming languages, allowing us to explore questions about how usable we can make programming. Our current TSPLs completely avoid concepts such as abstraction, decomposition, and programmer-defined data. That will unlikely to be true for all tasks, but it raises a great set of research questions. How do we characterize the learning tasks with which one can engage at a given level of computational complexity? How far can we go with using programming across the curriculum before we deal with the hard stuff? Rather than assume that we need to teach a Turing-complete language for integrating across the curriculum, we can instead explore just how much computing we *really* need to help students to learn with and about computing.

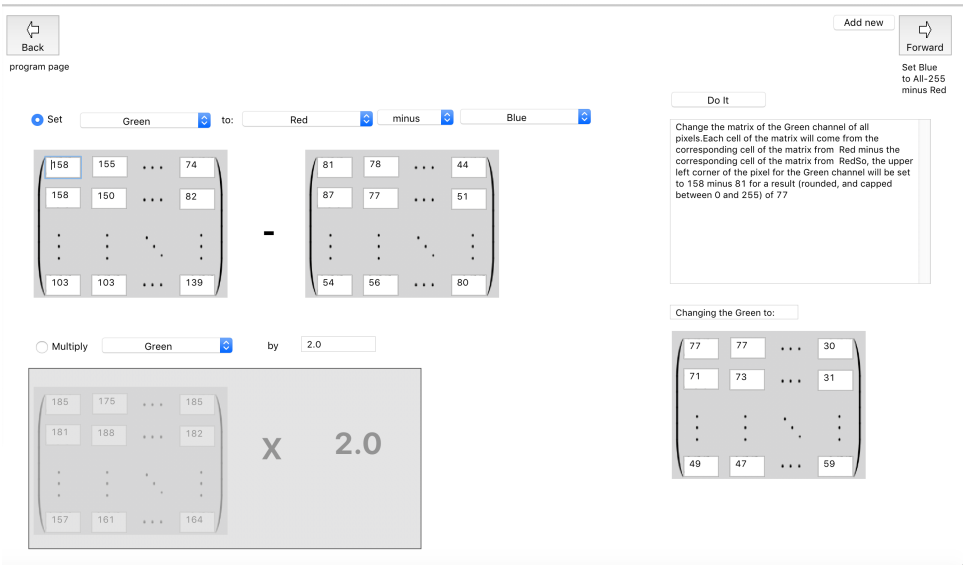


Figure 3: Describing manipulation a channel as an arithmetic over other channel matrices.

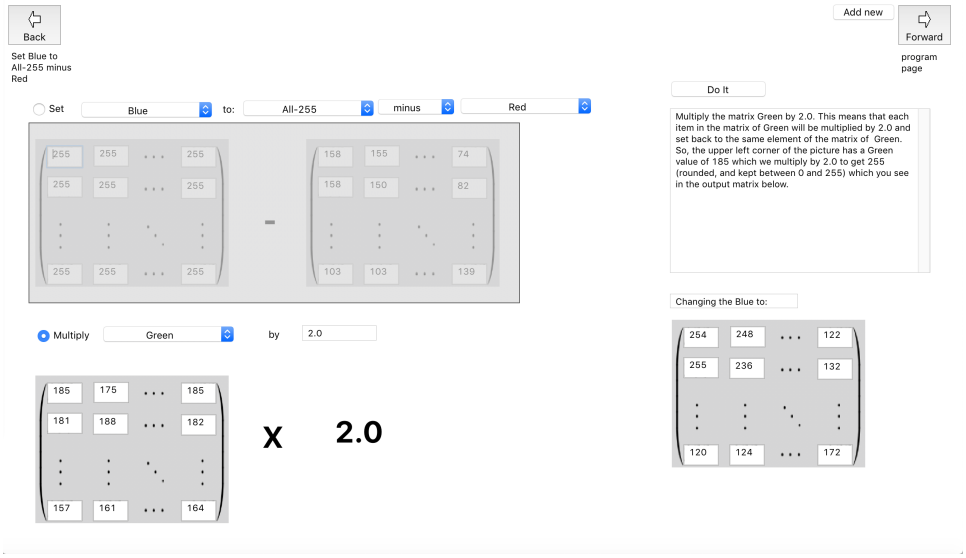


Figure 4: Describing manipulating a channel of a picture as matrix manipulation of a scalar.

REFERENCES

[1] J. Brown, B. Dalton, J. Laird, and N. Ifill. 2018. *Paths Through Mathematics and Science: Patterns and Relationships in High School Coursetaking.* National Center for Educational Statistics.

[2] Sarah E. Chasins, 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).* ACM, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

[3] Andrea diSessa. 2001. *Changing Minds.* MIT Press.

[4] Andrea A diSessa. 2004. Metarepresentation: Native competence and targets for instruction. *Cognition and instruction* 22, 3 (2004), 293–331.

[5] Andrea A Disessa and Bruce L Sherin. 2000. Meta-representation: An introduction. *The Journal of Mathematical Behavior* (2000).

[6] Stéphane Ducasse and Tudor Gîrba. 2006. Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems.* Springer, 604–618.

[7] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

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

[9] Barry J Fishman and Elizabeth A Davis. 2006. Teacher learning research and the learning sciences. In *The Cambridge Handbook of the Learning Sciences*, R. Keith Sawyer (Ed.). Cambridge University Press.

[10] J.E. Froyd, Charles Henderson, R.S. Cole, D. Friedrichsen, R. Khatri, and C. Stanford. 2017. From Dissemination to Propagation: A New Paradigm for Education Developers. *Change: The Magazine of Higher Learning* 49, 4 (2017), 35–42.

[11] Victoria Gillis. 2014. Disciplinary literacy: Adapt not adopt. *Journal of Adolescent & Adult Literacy* 57, 8 (2014), 614–623.

[12] Google. 2018. Google Charts: Interactive charts for browsers and mobile devices. https://developers.google.com/chart/.

[13] Mark Guzdial. 1995. Software-realized scaffolding to facilitate programming for science learning. *Interactive Learning Environments* 4, 1 (1995), 1–44.

[14] Mark Guzdial. 2010. Does contextualized computing education help? *ACM Inroads* 1, 4 (2010), 4–6.

[15] Mark Guzdial. 2013. Exploring Hypotheses About Media Computation. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 19–26. https://doi.org/10.1145/2493394.2493397

[16] Mark Guzdial. 2019. Computing Education As a Foundation for 21st Century Literacy. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. ACM, New York, NY, USA, 502–503. https://doi.org/10.1145/3287324.3290953

[17] M. Guzdial, W.M. McCracken, and A. Elliott. 1997. Task specific programming languages as a first programming language. In *Proc. 27th Annual Conference Frontiers in Education Conference 'Teaching and Learning in an Era of Change'*, Vol. 3. 1359–1360 vol.3. https://doi.org/10.1109/FIE.1997.632675

[18] Barbara Kelly and Daniel F Perkins. 2012. *Handbook of implementation science for psychology in education*. Cambridge University Press.

[19] Donald E. Knuth. 1972. George Forsythe and the Development of Computer Science. *Commun. ACM* 15, 8 (Aug. 1972), 721–726. https://doi.org/10.1145/361532.361538

[20] Tomaz Kosar, Nuno Oliveira, Marjan Mernik, Maria João Pereira, Matej Crepinsek, Daniela Cruz, and Pedro Henriques. 2010. Comparing general-purpose and domain-specific languages: An empirical study. *ComSIS–Computer Science an Information Systems Journal* (2010), 247–264.

[21] Youn-Kyung Lim, Erik Stolterman, and Josh Tenenberg. 2008. The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Transactions on Computer-Human Interaction (TOCHI)* 15, 2 (2008), 7.

[22] Elizabeth Birr Moje. 2015. Doing and teaching disciplinary literacy with adolescent learners: A social and cultural enterprise. *Harvard Educational Review* 85, 2 (2015), 254–278.

[23] Bahare Naimipour, Mark Guzdial, and Tamara Shreiner. 2019. Helping Social Studies Teachers to Design Learning Experiences Around Data: Participatory Design for New Teacher-Centric Programming Languages. In *Proceedings of the 2019 ACM Conference on International Computing Education Research (ICER '19)*. ACM, New York, NY, USA, 313–313. https://doi.org/10.1145/3291279.3341211

[24] Seymour Papert. 1980. *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

[25] Miranda Parker and Mark Guzdial. 2019. A statewide quantitative analysis of computer science: What predicts CS in High School?. In *Proceedings of 2019 International Computing Education Research Conference (SUBMITTED)*.

[26] Alan J. Perlis. 1962. The Computer in the University. In *Comptuers and the World of the Future*, Martin Greenberger (Ed.). MIT Press.

[27] Dominik Petko. 2012. Teachers' pedagogical beliefs and their use of digital media in classrooms: Sharpening the focus of the 'will, skill, tool' model and integrating teachers' constructivist orientations. *Computers & Education* 58, 4 (2012), 1351 – 1359. https://doi.org/10.1016/j.compedu.2011.12.013

[28] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. 2010. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*.

[29] Kathryn M. Rich, 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)*. ACM, New York, NY, USA, 745–751. https://doi.org/10.1145/3287324.3287396

[30] Kathryn M. Rich, 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)*. ACM, New York, NY, USA, 182–190. https://doi.org/10.1145/3105726.3106166

[31] Kathryn M. Rich, Carla Strickland, T. Andrew Binkowski, Cheryl Moran, and Diana Franklin. 2018. K–8 Learning Trajectories Derived from Research Literature: Sequence, Repetition, Conditionals. *ACM Inroads* 9, 1 (Jan. 2018), 46–55. https://doi.org/10.1145/3183508

[32] Lauren Rich, Heather Perry, and Mark Guzdial. 2004. A CS1 Course Designed to Address Interests of Women. In *Proceedings of the ACM SIGCSE Conference*. 190–194.

[33] Philip Sadler and Gerhard Sonnert. 2018. The path to college calculus: The impact of high school mathematics coursework. *Journal for Research in Mathematics Education* 49, 3 (2018), 292–329.

[34] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. 2017. Vega-Lite: A Grammar of Interactive Graphics. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (Jan. 2017), 341–350. https://doi.org/10.1109/TVCG.2016.2599030

[35] Emmanuel Schanzer, Kathi Fisler, and Shriram Krishnamurthi. 2018. Assessing Bootstrap: Algebra students on scaffolded and unscaffolded word problems. In *Proceedings of the 2018 ACM SIGCSE Technical Symposium*.

[36] Emmanuel Schanzer, Kathi 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*. ACM, 616–621.

[37] David Williamson Shaffer and Mitchel Resnick. 1999. "Thick" Authenticity: New Media and Authentic Learning. *Journal of interactive learning research* 10, 2 (1999), 195.

[38] Bruce L. Sherin. 2001. A comparison of programming languages and algebraic notation as expressive langauges for physics. *International Journal of Computers for Mathematical Learning* 6 (2001), 1–61.

[39] Tamara L Shreiner. 2018. Data literacy for social studies: Examining the role of data visualizations in K–12 textbooks. *Theory & Research in Social Education* 46, 2 (2018), 194–231.

[40] C. Stanford, R. Cole, J.E. Froyd, Charles Henderson, D. Friedrichsen, and R. Khatri. 2017. Analysis of Propagation Plans in NSF-Funded Education Development Projects. *Journal of Science Education and Technology* 26, 4 (2017), 418–437.

[41] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages as libraries. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 132–141.

[42] Vimal Viswanathan and Julie Linsey. 2011. Design fixation in physical modeling: an investigation on the role of sunk cost. In *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. American Society of Mechanical Engineers, 119–130.

[43] David Weintrop and Uri Wilensky. 2015. Using Commutative Assessments to Compare Conceptual Understanding in Blocks-based and Text-based Programs.. In *ICER*, Vol. 15. 101–110.

[44] Uri Wilensky and Seymour Papert. 2010. Restructurations: Reformulations of Knowledge Disciplines through new representational forms. In *Proceedings of the Constructionism 2010 Conference*, J. Clayson and I. Kalas (Eds.). Paris, France, 97.

[45] Michelle Hoda Wilkerson. 2017. Teachers, students, and after-school professionals as designers of digital tools for learning. In *Participatory Design for Learning: Perspectives from Research and Practice*, Betsy DiSalvo, Jason Yip, Elizabeth Bonsignore, and Carl DiSalvo (Eds.). Routledge.

[46] Tracee Vetting Wolf, Jennifer A Rode, Jeremy Sussman, and Wendy A Kellogg. 2006. Dispelling design as the black art of CHI. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*. ACM, 521–530.

[47] Pat Yongpradit. 2016. K-12 CS Framework. https://k12cs.org/. https://k12cs.org/