Recovering Constructionism in Computer Science: Design of a Ninth-grade Introductory Computer Science Course

Chris Proctor, *proctor5@buffalo.edu*
Graduate School of Education, University at Buffalo--SUNY, Buffalo, USA.

Jenny Han, *jlhan@stanford.edu*
Department of Computer Science, Stanford University, Stanford, USA.

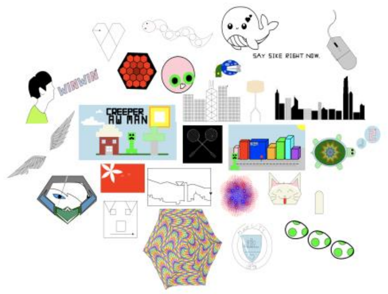
Jacob Wolf, *jwolf@isf.edu.hk*
International Schools Foundation Academy, Pokfulam, Hong Kong.

Krates Ng, *hnkng@isf.edu.hk*
International Schools Foundation Academy, Pokfulam, Hong Kong.

Paulo Blikstein, *paulob@tc.columbia.edu*
Teachers College, Columbia University, New York City, USA.

Abstract

Constructionism provided an early justification for children to study computers, but today's dominant approaches to K-12 computer science education are vulnerable to some of the same critiques Papert (1980) made of traditional schooling. In this paper, we identify three themes of Constructionism (computing cultures, material intelligence, and liberatory pedagogy) and use them to analyze existing approaches to K-12 computer science education. We then use these themes as design goals for a Constructionist ninth-grade introductory computer science course which is currently being implemented. This paper is part of a larger research project whose goal is to demonstrate the feasibility of a course focused on fully realizing the ambitious epistemological goals of Constructionism. As we contribute to a vision for K-12 computer science education, we hope to help recover the central role of Constructionism.



Keywords

Computer science education, Constructionism, computer cultures, material intelligence, liberatory pedagogy

Introduction

In most contemporary computer science classes the computer is used to put children through their paces, to provide exercises of an appropriate level of difficulty, and to dispense information. This sentence, which we believe accurately describes today's K-12 computer science landscape, also paraphrases the first sentence of *Mindstorms* (1980). Papert critiques the status quo of schooling and articulates a different vision of computer-supported education. Today, as computer science starts to be taken up broadly as a K-12 discipline, the leading implementations are vulnerable to the same critique which helped to justify teaching computer science in the first place.

This irony is the starting point for this study, in which we report on the design of a Constructionist computer science course which is currently being implemented in a high school in Hong Kong. This study is part of a larger research project whose goal is to demonstrate what might be achievable in one year of a Constructionist computer science course and to document how it comes about. As exploratory research, the goal is not to claim that the design of this course could be or should be implemented at other schools, particularly at schools with access to different resources. Nor is the goal to criticize projects like Hour of Code (Code.org), which prioritize expanded participation over any particular learning goals, and which are designed to scale up across existing conditions at schools around the world. However, we are concerned that in making compromises to scale up computer science education, current manifestations of K-12 computer science may have given up too quickly on the ambitious epistemological goals articulated by Constructionism.

In this paper, we surface three central themes of Constructionism, frame a design goal around each, and briefly consider several existing introductory computer science courses with respect to these goals. We then describe the context and design of an introductory ninth-grade computer science course, and close by grounding this design in an ongoing research project and considering how that project might help shape our vision of for K-12 computer science.

Background

In this section, we articulate three themes of Constructionism which frame our design goals.

Computer cultures

Constructionism is rooted in the belief that knowledge does not exist in a vacuum but rather lives and grows in situated context (Ackermann, 2001). Powerful thinking with computers requires a computer culture in which to participate. A computer culture provides ideas and media--tools to think with--as well as norms and practices to guide participation, define the community, and shore up the identities of participants. Just as Piaget's constructivism was rooted in the relationship between an organism and its environment (Fosnot & Perry, 1996), computer science is something people do within a computer culture.

Almost forty years ago, Papert imagined "the computer cultures that may develop everywhere in the next decades" (Papert, 1980, p. 20) Today, our environment is profoundly shaped by, made from, and mediated by, computers. We live in digital worlds which permeate, augment, and co-constitute what we perceive to be the real world. Youth growing up today almost universally participate in digital media, extensively and in important ways (Anderson & Jiang, 2018). These activities constitute rich and diverse computer cultures. They are to varying degrees emergent, viral, and engineered. Even though powerful ideas from computer science do not always flourish in these cultures, we propose that such cultures should be considered funds of knowledge (Moll, Amanti, Neff, & Gonzalez, 1992) which can be employed in the practice of constructing knowledge.

Material intelligence

Papert suggests that one reason computer cultures had not yet emerged in the late twentieth century was a relative "poverty in materials… from which intellectual structures can be built" (1980, p. 20). While our built environment and cultural practices offer innumerable opportunities to engage with and benefit from algebra and geometry, the claim was that fewer such learning opportunities existed for the powerful ideas of computing. This is certainly no longer the case. The computational infrastructure mediating our digital worlds is a leaky abstraction which constantly exposes the algorithms and computational properties with which it is designed, providing innumerable opportunities to encounter computational phenomena and to become powerful by making use of computational ideas. For example, the everyday practice of navigating social media involves informal social network analysis. Managing identities online requires constantly thinking in layers across interfaces. The essence of a meme relies on the practice of abstraction.

At the same time, many forms of work are now characterized by specialized computational media and practices in which computers are used intentionally and metacognitively as tools for thinking about thinking. diSessa refers to the ability to interface with a representational medium, for social practices or toward cognitive ends, as "material intelligence." Using computers as tools for thinking--and for thinking about thinking-- was of central importance to Papert. Wilensky (2010) uses the term "restructurations" to describe how knowledge can be reformulated via new representational forms which make different properties available. The protean nature of computers is powerful not just because they can take on many different forms to meet our existing needs (as in an app store), but because they support richer understandings of the structure of problems.

Liberatory pedagogy

Finally, we see Constructionism through a critical pedagogy lens, where education is a political act and computers could function as agents of emancipation (Blikstein, 2008). Instead of accepting an education "where children are segregated from society and segregated among themselves by age and put through a curriculum," Papert argued in agreement with Paulo Freire for a "problem-posing education" that encouraged students to approach important problems around themselves and in their worlds (Papert & Freire, n.d.). Each of the previous two themes contributes to a pedagogy of liberation, which we analyze in terms of Berlin's (1969) positive (freedom to) and negative liberty (freedom from).

An education grounded in computer cultures could be particularly well-suited to address students' agency of self-determination by connecting to their existing funds of knowledge (Rodriguez, 2013). Students (along with teachers, parents, and community members) must be treated as agents in the classroom with equal power to construct knowledge. To do this, teachers must understand students' backgrounds and cultural practices as wealths of experience which can be employed in the practice of constructing knowledge. Such a pedagogy promotes positive liberty by developing a classroom space where students can develop their identities while forming relationships with powerful ideas. These relationships contribute to learners' agency by expanding what they can do and understand, and by increasing their ability to contribute their own thoughts, ideas, or extensions of ideas to a conceptual domain (Boaler, 2003).

Developing material intelligence with computing could be particularly effective in supporting negative liberty, or freedom from oppression. Beyond defensively learning how to keep oneself safe online, learning how computing works could support youth in understanding how computation shapes our ideas about the world and our place in it, a computational analogy to Freire and Macedo's "reading the word and reading the world" (1998). Just as Freire connects a reading of words to a writing of words, we can connect understanding the impact of technology to creating technology that has an impact. As the negative social consequences of computing become more apparent in our world and joining the computing workforce perhaps loses some of its idealistic luster, a critical understanding of computer science may become an essential tool in utilizing technology to resist and deconstruct oppression.

Comparison with existing approaches

These three Constructionist themes can help structure the design space of possible ways of teaching introductory computer science courses (Nelson & Ko, 2018). In this section we briefly position several other projects with respect to the three themes.

First, Hour of Code's goal is "broad participation across gender and ethnic and socioeconomic groups" (Code.org) Toward this end Hour of Code is designed to be self-contained, requiring no teacher or community and usable on a wide variety of devices (or even without a computer at all). The scripted, puzzle-like activities are embedded in cultural worlds which might be familiar and appealing to a wide variety of students (e.g. programming sprites to dance to a soundtrack), and guide users toward understanding how to control those worlds using block-based programming. If Hour of Code succeeds at connecting with youth cultures (including informal computer cultures), the cost is very little development of material intelligence. The tools provided obviously cannot be used to make anything real. (Even in open-ended environments such as Scratch, students have trouble viewing their programming as real or building on their experience for future learning.) If Hour of Code has liberatory potential, it is in changing attitudes toward future computer science learning opportunities.

If Hour of Code can be dropped into any classroom, Exploring Computer Science (ECS) is a full introductory computer science curriculum supported by professional development (Goode & Margolis, 2011). Like Hour of Code, ECS is focused on broadening participation and stresses "ease of implementation for teachers and maximal engagement for students," but ECS is also focused on developing computer science knowledge (About ECS). ECS prioritizes computational thinking over programming; much of the curriculum is focused on working with computational problems and ideas in a social context rather than on implementing working programs. Therefore, we view ECS as being highly committed to building computing cultures while not emphasizing as much material intelligence. ECS could be seen as liberatory both in its focus on providing students access into the world of computer science, and in cultivating agency within the computing cultures it supports.

Finally, Beauty and Joy of Computing (BJC) is an introductory computer science curriculum which contrasts with ECS in its heavy emphasis on programming (Harvey, 2012). BJC is taught in Snap!, a block-based variant of Scratch which supports more explicit restructuration of mathematical ideas. While BJC also provides pedagogical support for diverse learners, in comparison with ECS its emphasis is more on powerful mathematical and computational ideas than it is on connecting to students' existing cultures or using computing in those worlds. As its name suggests, BJC stresses the liberatory potential of the ideas themselves. Like Hour of Code and ECS, BJC is taught in a block-based language which prioritizes accessibility over the ability to create personally-meaningful projects in domains already important to students.

Design of the Course

These examples of existing computer science courses suggest a general tradeoff between providing broad access to computer cultures and engaging deeply with material intelligence. Our broader research goal is to question whether this tradeoff is necessary. By reconfiguring the relationship between computer cultures and material intelligence, we hope to show that these goals can be mutually supportive and can result in new liberatory possibilities.

Context

The research and development of this course takes place at a bilingual private school in Hong Kong during the 2019-2020 school year. A teaching team of three instructors, two of which are associated with the university-based research group, works with twenty-eight Grade 9 students, divided among 2 classes. For the majority of the students, this course is their first exposure to computer science. For the school, this is the first iteration of a Constructionist computer science course. Notably, the material and financial resources available in this setting made low student-to-teacher ratios, highly-qualified teachers, and one-to-one computing possible.

At the same time, the students, the course, and the school face pressure to succeed in the context of both the Chinese and the overseas educational systems (notably, U.S. and U.K.). They are under pressure to demonstrate success through grades, AP, IB scores, college admission, and have historically relied on extrinsic motivation to achieve this performance. Thus, while the context of this research provides atypical access to resources, it also provides stringent demands that a Constructionist computer science class be able to demonstrate success on traditional terms as well as its own.

Overview

In designing the course, the central goal is to create a rich, diverse community of people making things with code, through which they can develop personal relationships with powerful ideas. The course is designed to help all students learn to interact with code as an expressive, evocative medium, which helps to structure thought. At the same time, the course is designed to support computational literacy, connecting with students' existing ways of reading and writing.

The course is composed of five curriculum units intended to introduce students to various computing domains and programming paradigms, while supporting progressive growth in particular skills. Each unit introduces new skills and concepts, and then devotes significant time to an open-ended student project in a real-world computing domain through which they will encounter programming challenges in authentic contexts and, with teacher support, learn what they need to solve them.

Table 1. Summary of curriculum units.

Topic	Project	Paradigm
Turtle drawing	A drawing (optionally using makerspace to etch/cut)	Imperative
Data science	Data-based argument about our digital worlds	Functional
Games	Create a game	Object-oriented
Networking	Create a networked microservice	Reactive
Web applications	Create a web application for real-world users	Human-computer interaction, collaboration on larger system

Each unit consists of labs (for student-driven exploration), mini-lessons (for just-in-time teaching), assignments (for review and practice of concepts), and projects (for open-ended creation). Below, we outline three central design goals, each attached to the aforementioned themes of Constructionism, and provide examples of specific decisions in our course design that align with our goals. In practice, we cannot separate the themes of Constructionism into three distinct design decisions, but we structure below accordingly for clarity.

Let students drive liberatory pedagogy

We dedicate most of our class time to student-directed engagement with course materials (called labs). Not unlike science labs where researchers plan and conduct experiments, computer science labs introduce groups of students to concepts, practices, and tools in computer science by giving them project-based problems to solve. Such a problem-posing education necessitates learning by doing. Consider the following examples:

Navigating the Terminal. In their first lab, students learned to navigate the terminal by exploring a directory with text files, subdirectories, and python files that played like an adventure game when students used the terminal commands to run the programs, change directories, read the text files, and other basic command-line tasks. Importantly, students were not told what to learn or memorize. Instead, students were provided with simplified documentation of various commands, and they chose to learn the terminal commands that they felt were most important or necessary in the moment.

Introduction to Loops and Lists. In this lab, students used print statements, lists, and loops for the first time. Rather than read about the syntax of a for-loop or why it may be useful, students experienced the concept for themselves. They explored a list of subway stops in their district and wrote different loops to traverse through the stops. Notably, we never provide answer keys to the labs or assignments. This design decision shows students that there is no one correct approach to a computer science problem; instead, we encourage students to consider and share multiple solutions in their groups.

The teaching team relies on a collection of norms for interactions in a student-driven class (see Figure 1). As instructors, we minimize our time in front of the class. Sometimes, we offer guidance "mini-lessons" about concepts to the entire class before they begin the lab. More frequently, this manifests in holding “mini-lessons” for individuals or small groups of students in the moment, when they get stuck on a lab or assignment. When one student grasps a concept or resolves a bug, we invite them to act as “student experts” and teach their peers. In this way, students drive the knowledge creation in the classroom and become agents of meaning-making.

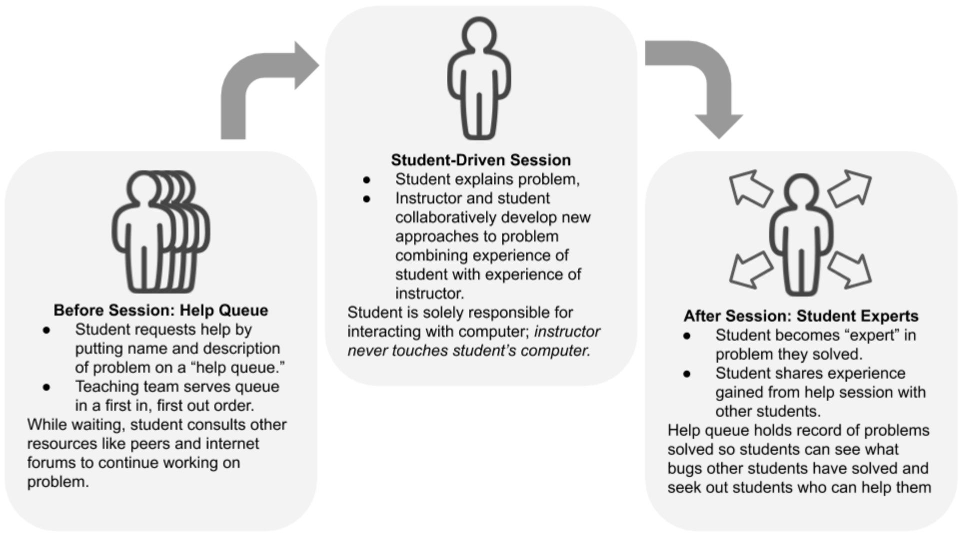

Before Session: Help Queue
• Student requests help by putting name and description of problem on a “help queue.”
• Teaching team serves queue in a first in, first out order.
While waiting, student consults other resources like peers and internet forums to continue working on problem.
Student-Driven Session
• Student explains problem,
• Instructor and student collaboratively develop new approaches to problem combining experience of student with experience of instructor.
Student is solely responsible for interacting with computer; *instructor never touches student’s computer.*
After Session: Student Experts
• Student becomes “expert” in problem they solved.
• Student shares experience gained from help session with other students.
Help queue holds record of problems solved so students can see what bugs other students have solved and seek out students who can help them

Figure 1. Process and norms of getting help including requesting a help session, doing a help session, and sharing knowledge after a help session.

Develop material intelligence using practical tools with high ceilings.

In designing the course, we paid particular attention to the tools we chose for our students to use to engage with computation. We were particularly interested in the Constructionist idea that powerful tools help students develop deep connections to a conceptual domain. We decided to use tools which we believed would increase students’ material intelligence, their agency to engage with computation. We decided to use a UNIX/Linux terminal user interface, Atom, and Python as the development tools for our class (Figure 2). Importantly, each of these tools provides an open-ended development experience with very high ceilings on what they can be used to create. Further, these tools are all regularly used in computer science practices from personal to academic to industry, allowing our students’ use of the tools to interface with broader practices of computing.

This approach is far from the sandboxed strategy adopted by many computer science curricula, and required significant up-front setup, downloading and configuring software and development environments. The tradeoff is that students are now able to powerfully use their computers as tools for general purpose computing in this class and beyond. In the first weeks of the course, students learn the basics of how to navigate the file system using a command line interface, how to create and edit Python files using the text editor, execute and debug them from the command line. Soon thereafter, students were using version control (git) to clone starter projects, track their own progress, reflect on their process (using a customized template for commit messages), and dialogue with teachers as they revise their work.

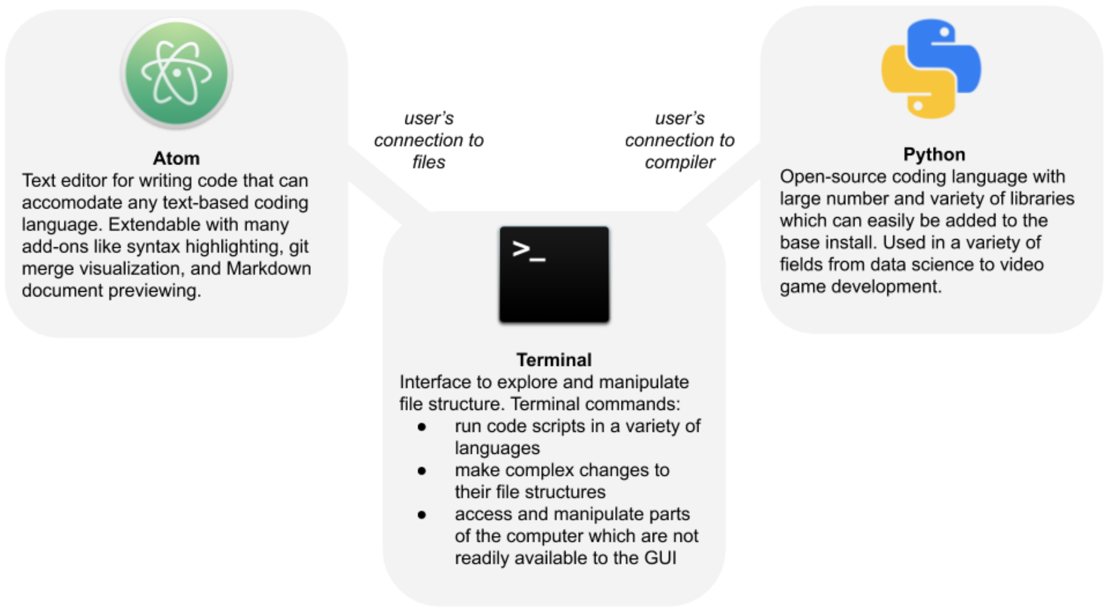

Atom
Text editor for writing code that can accomodate any text-based coding language. Extendable with many add-ons like syntax highlighting, git merge visualization, and Markdown document previewing.
user's connection to files
user's connection to compiler
Python
Open-source coding language with large number and variety of libraries which can easily be added to the base install. Used in a variety of fields from data science to video game development.
>_
Terminal
Interface to explore and manipulate file structure. Terminal commands:
● run code scripts in a variety of languages
● make complex changes to their file structures
● access and manipulate parts of the computer which are not readily available to the GUI

Figure 2. Visualization of tools and workflow for code development in our class: find/open/create files with Terminal, write Python scripts as files with Atom, use Terminal to run Python scripts.

Implement personally meaningful projects draw in students' computer cultures

Finally, we designed our class to rely on students' own experiences with computers while expanding their ability to understand and interact with their digital worlds. We feel that the most effective way to engage students with the "underlying representational form" (diSessa, 2001, pg. 24) of computation is to ask students to draw in elements of their own computer cultures. To do this, we spend a significant amount of time at the end of each unit supporting students as they develop a project which utilizes the tools and concepts explored in the unit. The goal of this project is always to create something personally meaningful: a piece of art, a data-informed answer to a question about their worlds, a game inspired by their own favorite game. Some of these projects are individual asking students to draw from their independent experiences while others are group projects asking students to mesh their computer cultures with the computer cultures of others. Further, these projects often expect students to build upon the work of others, hacking and remixing code for their own purposes.

This approach to projects has been particularly generative in creating space for students' computer cultures in our classroom. At the end of our Turtle drawing unit, many students' projects featured their own computer cultures. One student created her own version of a meme showcasing her efforts while another student invoked his computer culture by integrating Minecraft characters into his drawing (Figure 3).

Students' engagement with personally meaningful projects also allows students to connect other literacies to their computer cultures. Many students' projects connected outside identities, interests, and skills to the classroom computer culture, potentially enriching both (Figure 4).

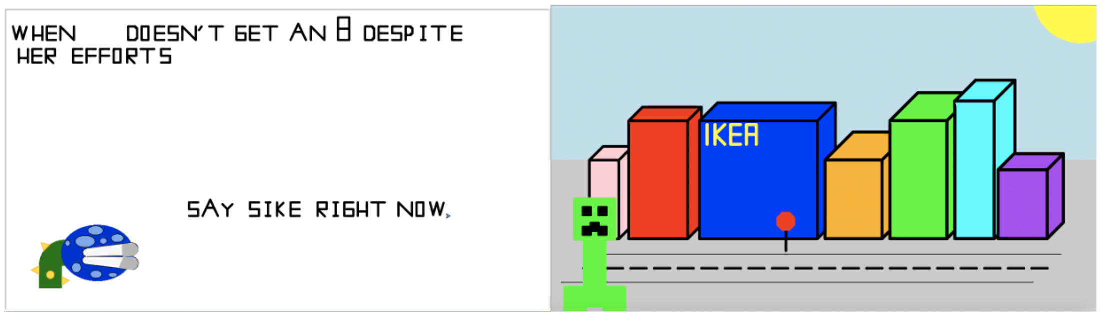


Figure 3. Students using elements of their computer cultures, such as memes (left) and Minecraft (right), as inspiration for their projects.

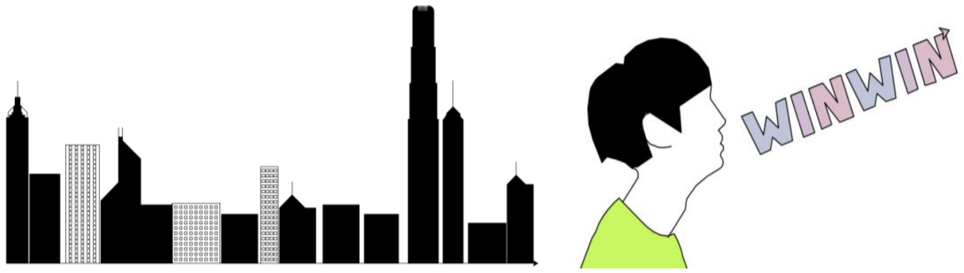


Figure 4. Students using elements of their cultural identities, such as their Hong Kong heritage (left) and Korean pop music interests (right), as inspiration for their projects.

Future Considerations

As we finish the first semester of the course, we reach a point where we can share preliminary results, limitations, and next steps. We are currently conducting research to substantiate the initial positive results we see from our perspective as reflective practitioners. Our intention is to revise and then share the curriculum broadly and to learn about what kinds of support will be needed when it is taught at other schools. Because our choice of tools requires access to lower levels of the computer, it would not be possible on a phone, tablet, or browser-based laptop. The hardware required may make this curriculum less accessible and scalable for other schools. It also requires significant human resources as teachers will need to familiarize themselves with these tools, which are more complex than other beginner-friendly platforms. The teacher must also feel human and disciplinary agency when using the tools. For many schools, this may present a difficulty if the computer science teacher is not a computer scientist by training.

Even though we have passed up interfaces designed for accessibility such as web-based and block-based environments, initial results suggest that all of our students have been able to access and participate in working with computational ideas using a powerful representational medium. The fact that we are using tools common to real-world formal and informal computing contexts also provides numerous ancillary benefits. Students can also begin to develop practices of using real-world resources such as man pages, library documentation, and support forums. Student curiosity about how things work is often rewarded by excursions into important phenomena ranging from their operating systems to random number generation to the layers of abstraction supporting user interfaces.

Conclusion: What kind of CS for all?

As computer science has gained recognition as a mainstream K-12 subject in school, the question of what kind of computer science we want has also begun to gain belated traction. Several recent frameworks have articulated different motivations for teaching computer science (Blikstein, 2018), visions for the field (Santo, Vogel, & Ching, 2019), and theoretical framings (Kafai, Proctor, & Lui, 2019). However, the dominant approaches in practice make compromises between the goals of broadly accessible computer cultures, deep engagement with material intelligence, and liberatory pedagogy. This paper develops the design rationale for a course and a research project seeking to recover the original spirit of a Constructionist approach to computer science, and in doing so, to question whether these goals could be mutually supportive rather than at each others' expense.

References

About ECS. Retrieved from http://www.exploringcs.org/about/about-ecs.

Ackermann, E. (2001). Piaget's constructivism, Papert's Constructionism: What's the difference. *Future of learning group*, 5(3), 438.

Anderson, M., & Jiang, J. (2018). *Teens' Social Media Habits and Experiences*. Pew Research Center.

Berlin, I. (1969). Two Concepts of Liberty. In *Four Essays on Liberty* (pp. 118–172).

Blikstein, P. (2008). Travels in Troy with Freire: Technology as an Agent of Emancipation. In C. A. Torres & P. Noguera (Eds.), *Social Justice Education for Teachers* (pp. 205–235).

Blikstein, P. (2018). *Pre-College computer science education: A survey of the field*. Mountain View, CA: Google.

Boaler, J. (2003). Studying and Capturing the Complexity of Practice—The Case of the "Dance of Agency." *International Group for the Psychology of Mathematics Education*, 1, 3–16.

Code.org. *How to teach one Hour of Code with your class*. Retrieved from https://hourofcode.com/ac/how-to.

diSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy*. Mit Press.

Fosnot, C. T., & Perry, R. S. (1996). Constructivism: A psychological theory of learning. *Constructivism: Theory, Perspectives, and Practice*, 2, 8-33.

Freire, P., & Macedo, D. (1988). *Reading the Word, Reading the World*.

Goode, J., & Margolis, J. (2011). Exploring Computer Science: A Case Study of School Reform. *ACM Transactions on Computing Education*, 11(2), 1–16.

Harvey, B. (2012). The beauty and joy of computing: Computer science for everyone. *Proceedings of Constructionism*, 33-39.

Kafai, Y., Proctor, C., & Lui, D. (2019). From Theory Bias to Theory Dialogue: Embracing Cognitive, Situated, and Critical Framings of Computational Thinking in K-12 CS Education. In *Proceedings of the 2019 ACM Conference on International Computing Education Research* (pp. 101-109). ACM.

Moll, L. C., Amanti, C., Neff, D., & Gonzalez, N. (1992). Funds of knowledge for teaching: Using a qualitative approach to connect homes and classrooms. *Theory into Practice*, 31(2), 132–141.

Nelson, G. L., & Ko, A. (2018). On Use of Theory in Computing Education Research. *Proceedings of the 2018 ACM Conference on International Computing Education Research - ICER '18*, 31–39.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. Basic Books.

Papert, S., & Freire, P. (n.d.). *The Future of School*. Retrieved from http://www.papert.org/articles/freire/freirePart1.html.

Rodriguez, G. M. (2013). Power and Agency in Education: Exploring the Pedagogical Dimensions of Funds of Knowledge. *Review of Research in Education*, 37(1), 87–120.

Santo, R., Vogel, S., & Ching, D. (2019). *CS for What? Diverse Visions of Computer Science Education in Practice*. New York, NY: CSforALL.

Wilensky, U., & Papert, S. (2010). Restructurations: Reformulations of knowledge disciplines through new representational forms. *Constructionism.*