# Grounding How We Teach Programming in Why We Teach Programming

**Chris Proctor,** *cproctor@stanford.edu*
Graduate School of Education, Stanford University

**Paulo Blikstein,** *paulob@stanford.edu*
Graduate School of Education, Stanford University

## Abstract

This article extends a 2005 taxonomy of languages and environments for teaching computer programming to the current field of pedagogical programming languages and environments. The original taxonomy organized tools according to the barriers to learning programming they addressed, and the strategies used to surmount or avoid the barriers. Updating the taxonomy was not entirely successful; some strategies have emerged as widely-used best practices. As computers and programming have become more prevalent in everyday life, it has become harder to distinguish programming from non-programming, and harder to distinguish tools for learning from ordinary computational tools. Programming, formerly a specialized activity, has developed into a computational literacy comprising many different forms of cultural interaction. We propose a new taxonomy based on DiSessa's analysis of the structure of literacy. The new taxonomy allows learners, teachers, and designers to ground decisions about how to learn or teach programming in literacy aims expressing why they want to learn to program .

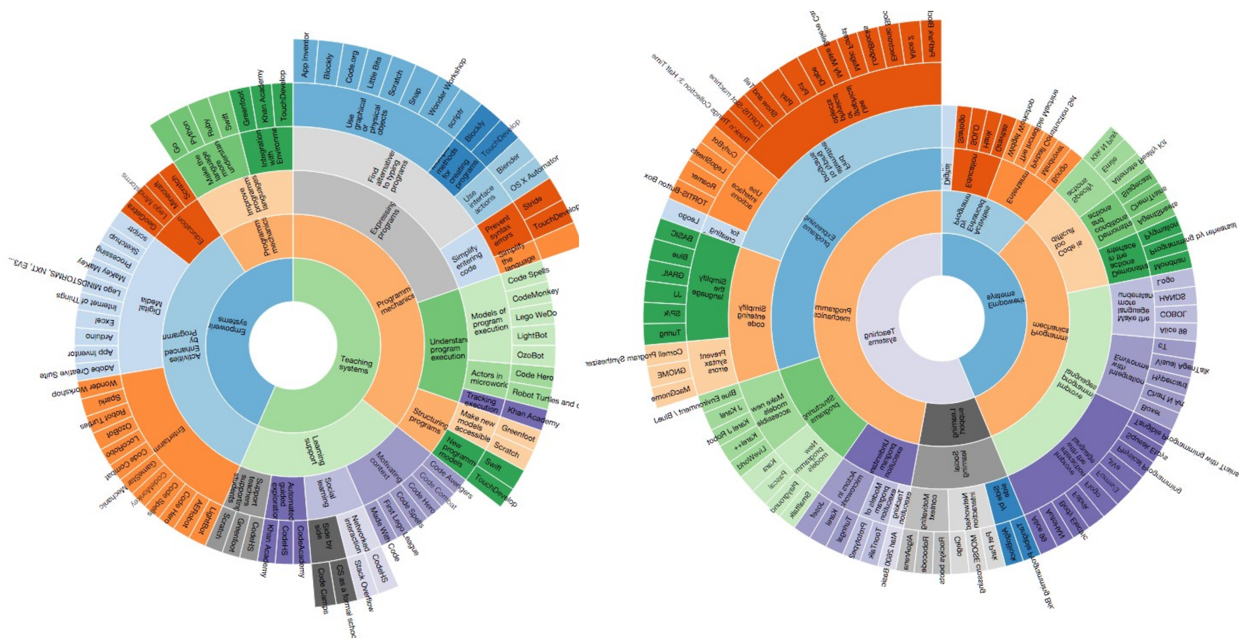

*Figure 1. Changes in the landscape of tools for teaching programming, 2005-2016*

## Keywords

computational literacy, design, programming languages, computational thinking, educational technology

# Introduction

The last decade has seen a surge of popular interest in learning to program, driven by the ever-deepening penetration of computers and mobile devices into our lives, an identification of programming as an important vocational skill, and celebrity endorsements going as far as the president of the United States. In response to this demand, there has been a proliferation of new programming languages and environments designed to teach programming to novice programmers. In 2005, Kelleher and Pausch published a taxonomy of languages and environments designed to teach programming, categorizing them in terms of how they sought to assist learners. This article extends Kelleher and Pausch's taxonomy to the current field of pedagogical programming languages and environments. We use our review as an indicator of which culture of programming is becoming more prevalent--shifts in the distribution of pedagogical approaches point to changes within the culture of computing and to the changing role of computing in the broader culture.

Despite these new tools and their many antecedents, it has always been hard to become a proficient programmer. Programming is a literacy demanding skills at many levels, spanning fluency with the symbols and syntax of programming languages, to practices such as structuring programs and debugging, to socialization into the community of programmers and development of a computational imagination. The earlier taxonomy defined programming narrowly, as "the act of assembling a set of symbols representing computational actions," allowing programmers to "express their intentions to the computer" and "predict the behavior of the computer." (p. 83) In the past decade, as computing has become a dominant medium in education, entertainment, journalism, and work, it has become more important to treat programming as a cultural practice, and to differentiate the levels of skills needed to be literate in the medium. The following three activities illustrate computational literacy at different levels:

Writing a program to control your household thermostat.
Hiring a programmer to change your newspaper's commenting policy.
Discussing the tradeoffs between freedom and security in a cyber security law.

The pedagogical tools to facilitate mastery of these activities will differ not just in methods, but also in aims. A second goal of this article, therefore, is to taxonomize the components of computational literacy, and to sort the tools for learning to program according to the (often implicit) computational literacy activities they aim to teach their users. Framing teaching tools in terms of the forms of literacy for which they provide affordances will allow us to critique the shallow vocationalism underlying some efforts to "teach kids to code," to identify projects that have the potential to be socially transformative, and to suggest new directions for designers of educational programming languages and environments.

# Comparing Pedagogical Approaches: 2005-2016

Kelleher and Pausch's taxonomy begins with a high-level division between "teaching systems," tools designed to serve as an intermediate step toward full-powered programming languages and "empowering systems," which aim to empower users directly, making full-powered languages unnecessary. The taxonomy then divides and sub-divides domains of barriers to programming. For example, most of the "teaching systems" address "mechanics of programming," which is divided into "expressing programs," "structuring programs," and "understanding program execution." The taxonomy then divides into groups of programs which take similar approaches to removing or bypassing the barrier. For example, in addressing the challenge of "expressing programs," some tools "simplify the language," while others "prevent syntax errors." After surveying the current field of languages and environments designed to teach programming, we organized them using Kelleher and Pausch's original taxonomy. The diagrams below show the 2005 taxonomy and its 2016 update. (See Appendix I for descriptions of the languages and environments taxonomized.)

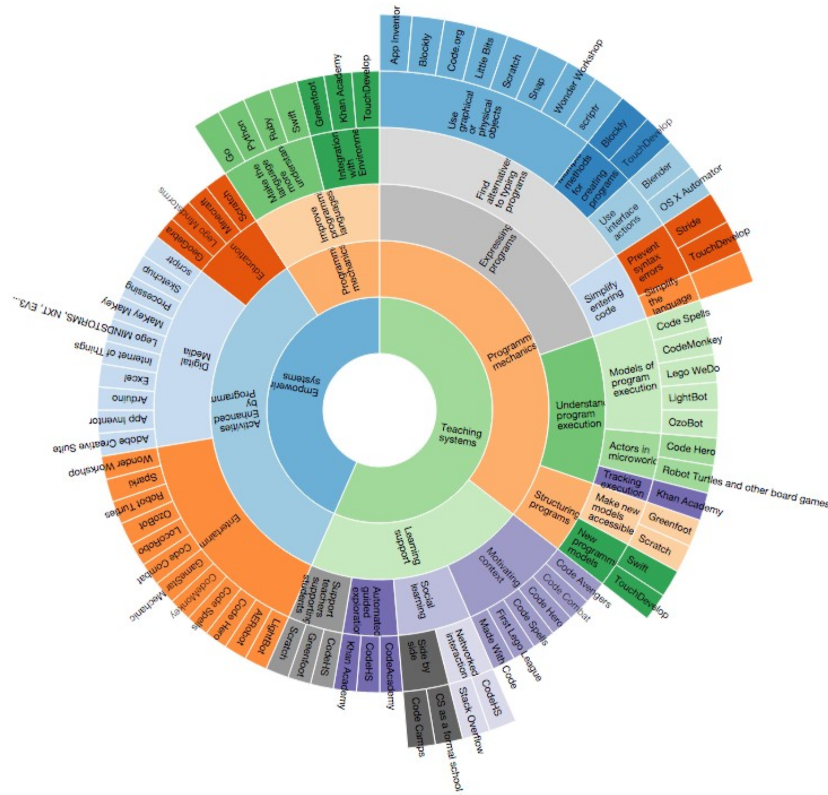


*Figure 2. Kelleher and Pausch's taxonomy (2005) (chrisproctor.net/research/teachinglanguages/2005)*

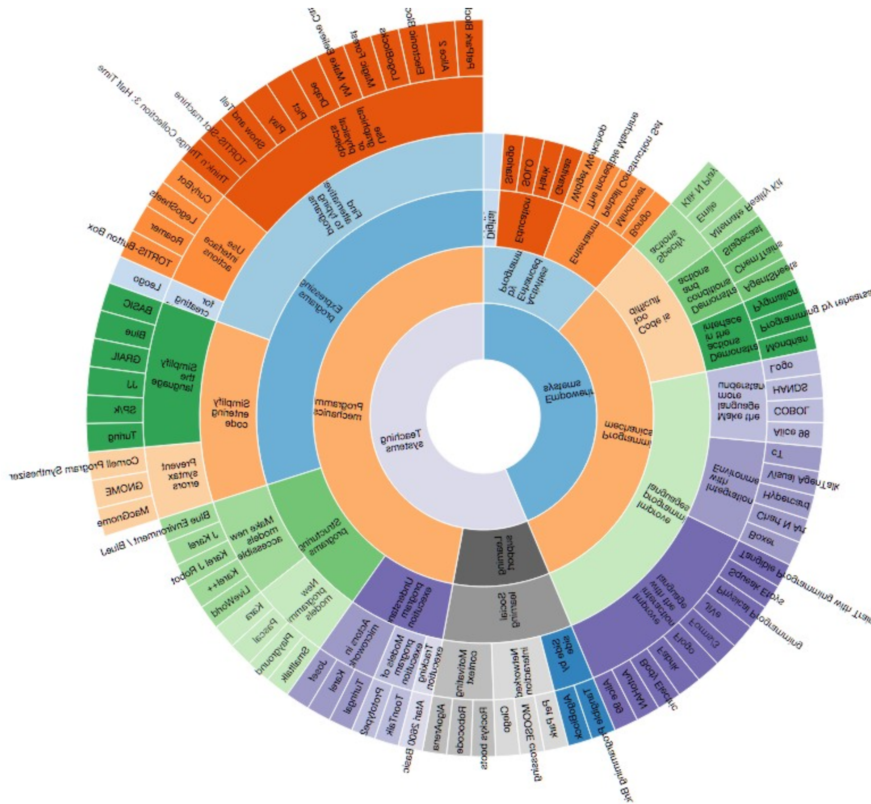

*Figure 3. Updated taxonomy (2016) (chrisproctor.net/research/teachinglanguages/2016)*

Comparing these two snapshots, with an interval of a decade, reveals similarities and some clear shifts. Emphasis continues to be put on helping learners overcome (or circumvent) the mechanics of programming; block-based programming remains a dominant paradigm for allowing users to express programs using affordances which prevent the possibility of some classes of syntax errors. In contrast, there are fewer efforts today to simplify the process of

entering code by simplifying the language or preventing syntax errors. This suggests that two dominant strategies have emerged: teaching languages either avoid textual languages entirely, or engage with them fully. Two other significant changes within the category of teaching systems are the growth of tools that model program execution and the growth of social learning support. Over the last decade, rich computer graphics have become much more pervasive; it is now technically easier to model the execution of a program by animating blocks as they are executed, or by demonstrating the effect of a program via an actor in a microworld. Similarly, with the increasing universality of networked computers, it is much easier to incorporate social support for learning into a programming environment. Components of identity such as race, gender, and language have been identified as important factors in teaching programming, and sources of marginalization in the dominant culture of programming; some efforts at teaching programming are explicitly focused on supporting marginalized identities. One final trend to note in the category of teaching systems is the use of gamification to provide a motivating context. Several projects repurpose familiar game environments, such as the dungeon crawler role-playing game, so that the player scripts the avatar's actions rather than controlling them directly.

The other broad approach identified by Kelleher and Pausch, "empowering systems," has undergone even more significant shifts. General-purpose programming languages have become more expressive, user-friendly, and designed to be easily learned--an important language characteristic as the rate of technological change accelerates. One core task of any software developer is keeping up to date with new paradigms, patterns, languages, and frameworks. Consequently, there are now many more general-purpose programming languages included in the taxonomy. The category of "activities enhanced by programming" has also grown dramatically. The cultural activities of education, entertainment, and media production have all shifted toward the digital medium; participation in these activities increasingly demands the ability not just to write but to code. It is therefore not surprising that languages and environments designed to teach people how to program would do so in the context of empowering them to participate in cultural activities.

While Kelleher and Pausch's taxonomy serves as a useful lens for broad generalizations across time, this analysis would not be a strong basis for more specific comparisons. Partly this is due to a lack of methodological precision: we are not confident that either taxonomy was an exhaustive survey of the field. More fundamentally, we found that the taxonomy's categories, as well as the boundaries of its domain, are no longer stable. Three categories of ambiguity impede the task of taxonomizing languages and environments designed to teach programming: First, as the field has matured, new projects have multiple goals and draw on multiple strategies for making programming more accessible. For example, Microsoft's TouchDevelop provides a block-based interface, allows the user to switch to a text-based interface, allows the user to specify functionality with a graphical interface, dynamically points out syntactical errors, is focused on digital media production (especially games), and has social sharing built into the development environment. The decision of where to place TouchDevelop becomes quite arbitrary.

A second ambiguity lies in determining whether a programming language or environment is primarily designed to make programming more accessible--either by acting as a "teaching system" or as an "empowering system." User interface, user experience, and explicit teaching have become primary design consideration of many new programming languages. Yukihiro Matsumoto, the designer of Ruby, described his central goals for the language as to "help every programmer in the world to be productive, and to enjoy programming, and to be happy." (2008) Similarly, Guido van Rossum, the creator of Python, wrote, "Our aim is provide tools to support users when they are learning programming and when they are employing those skills in their homes and offices." (1999) As these and other new general-purpose languages have become popular languages for teaching, it has become difficult to identify a distinct subset of programming tools which are designed for teachability or accessibility.

Finally, as our dominant cultural medium has become computational and interactive, it has become difficult to distinguish programming from non-programming. For example, Adobe Photoshop can be viewed as a graphical user interface for composing matrix operations on a raster of pixels; users can apply computational concepts such as saving a sequence of commands for later use, defining new functions, and controlling the flow of data from one function to another. It is possible to explicitly script Photoshop using one of several mainstream

programming languages. Should Photoshop be considered a programming environment? If so, similar cases can be made for Microsoft Excel, Facebook, Twitter, email, all our apps, and many of the new appliances we use at home and work ranging from photocopiers to Roombas to Internet-of-Things light bulbs. Computational culture has gone mainstream and code is becoming a literacy. While these shifts reduce the utility of Kelleher and Pausch's taxonomy, they underscore the importance of systems which make computation accessible, and therefore the need for making sense of how tools can help.

## A Taxonomy Based on Literacy Aims

The original taxonomy, which organized tools according to how they support learning programming, depends on a definition of programming which is no longer adequate to describe the range of cultural practices that now comprise our interactions with computational media. We propose using DiSessa's analysis of the structure of literacy to recast programming as computational literacy, and therefore as a means of bringing more specificity to what we mean by teaching programming. DiSessa (2000) defines literacy as "a socially widespread deployment of skills and capabilities in a context of material support (that is, an exercise of material intelligence) to achieve valued intellectual ends." (p. 19) DiSessa identifies three subsets of skills which comprise literacy: First, the material pillar involves "external, materially-based signs, symbols, depictions, or representations" which afford "particular modes of mediated thought." (p. 6) When we read, the written word mediates our thought, permitting us to grapple with substantial ideas, provoking us to make undiscovered connections between ideas, and framing points of view, among other functions. This array of mental capacities forms DiSessa's second pillar, the cognitive pillar of literacy. Finally, the social pillar names the social niches and genres (socially-constructed patterns of meaning) which share a basis on a representational form. Socially-constructed text-based meaning implies shared expectations for readership, authorship, context, and form.

We can identify computational activities that correspond to these pillars as well. For example, writing a website or debugging a script primarily involve engagement with code, the basic material form of computation. The cognitive pillar is dominant when we use computational thinking to restructure a problem or develop a model of a situation. Similarly, collaborating on GitHub or playing multiplayer online role-playing games are examples of computational literacy which engage primarily with the social pillar. These three pillars enable a categorization of tools for enabling computational literacy according to their literacy aims, or what they aim to enable users to do. These aims may be explicitly stated or they may be implicitly present in the affordances provided to users.

We can further refine our analysis by distinguishing between literacy aims which assume a consumer role and those which assume a participant role. These roles correspond somewhat to the practices of reading but not writing (consumer) and both reading and writing (participant). However, critical reading goes beyond the more passive consumer role, while heavily-scripted participation, lacking a mechanism for creating new meanings, can be most appropriately identified as the consumer role. As with the three pillars of literacy, it will not be possible to draw a clean line between consumer and participant roles; for example the practice of copying and pasting code written by others falls somewhere between consumer and participant. (Perkel, p.4) Nevertheless, it is important to distinguish roughly between literacy aims which assume the consumer role and the participant role. Promoting a more participatory "remix culture" or "read-write culture" has been a central commitment of media activists such as Lawrence Lessig (2008, p. 28) and progressive educators whose goal is student empowerment. Drawing on the three pillars of literacy, and the two broad literacy roles of consumer and participant, we can pose criteria for a new taxonomy based on a tool's literacy aims:

**Material (Consumer):** Does it aim to teach users how to interact with code? Does it provide affordances for learning to interact with code?
**Material (Participant):** Does it provide affordances for or serve as a tool for using code to create new things?
**Cognitive (Consumer):** Does it aim to develop recursive thinking, modularity and abstraction, heuristics, or other "mental tools" from the field of Computer Science? (Wing 2006) Does it provide affordances for learning to think in terms of computation?

**Cognitive (Participant):** Does it provide affordances for or serve as a tool for applying computational thinking in original and personally-meaningful ways?
**Social (Consumer):** Does it aim to teach users how to interact with computational artifacts created by others, or provide affordances for doing so?
**Social (Participant):** Does it provide affordances for or serve as a tool for personally-meaningful computational expression? Does it support users in active, critical participation in computation-based communities?

Using these questions, we re-taxonomized the tools for teaching programming (now broadened to include tools for enabling computational literacy); the results are presented in Figure 4. While most of the tools engage most strongly with one literacy, many also have literacy aims spanning more than one of the pillars.

## The Future of Teaching Programming

This new taxonomy of the tools used for enabling computational literacy should be of use to both users and designers. The new taxonomy allows users (both learners and teachers) to select tools which are designed to support their own aims. The learning goals of a middle-school who wants to dabble with programming will be very different from those of a business owner seeking new ways to model the flow of her inventory, or a policy-maker who wants to consider how to apply ethical frameworks to the digital domain.

Teachers can now ask of these tools the questions that are important for other pedagogical curriculum and tools. Do they envision an empowered, participatory role for students? Is the social vision embodied by the tool's affordances congruent with the goals of the teacher or the school? For example, a programming curriculum which attends to nothing but the material pillar of literacy will be just as incomplete as an English curriculum which focuses exclusively on mechanics and which never offers students the opportunity to write anything they care about. In the field of secondary English education, teaching grammar in a contextualized, constructivist manner is a widely-accepted best practice. (Weaver, 1996, p.174-175) If this principle also hold when teaching programming, it suggests that tools with literacy aims spanning multiple pillars will be most effective.

One clear implication for designers of tools for enabling computational is the necessity of considering the literacy aims to be supported. Many of the tools we analyzed in this study had no clear explicit or implicit literacy aims; they do not ground user interaction in assumptions about why the user is learning to program. This trend corresponds with the popular demand that children should learn to program without a clear rationale for why children should learn to program, or with vague justifications based on future employability.

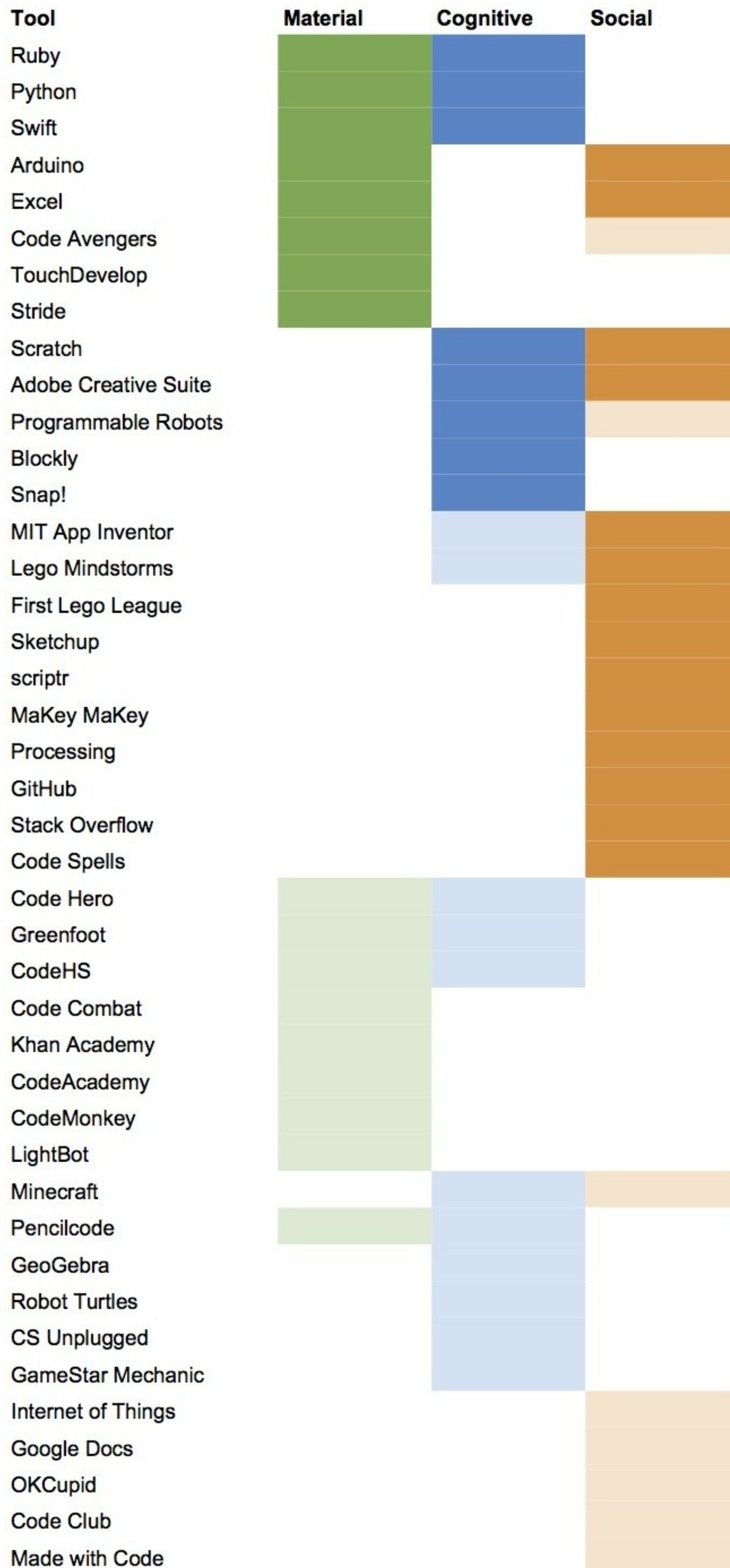


*Figure 4. Tools for teaching programming organized by literacy aims*
*(Light colors indicate a consumer role; Dark colors indicate a participant role)*

Shifting the focus away from pedagogical tactics--strategies for supporting computational literacy--is not meant to suggest these are unimportant. Rather, this shift in focus suggests that this field is maturing. A decade ago, many tools were developed to help users overcome a particular barrier using a particular strategy. Today this remains an active field of research, but a set of dominant strategies or best practices seems to be emerging. Most importantly, considering pedagogical tactics in the context of a tool's aims allows designers to consider not just whether a pedagogical strategy will be helpful, but whether it will be helpful toward a particular end. As our society shifts toward a reliance on computation as a cultural medium, we will have to renegotiate what kind of society we want to have and the roles we want to have in it; this is an opportunity to address longstanding issues of marginalization and disempowerment. Designers of tools for interfacing with this new medium have both the opportunity and the responsibility to be clear about the roles for which they are preparing their users.

## Appendix I: Notes on Tools for Teaching Programming

A table containing detailed notes on the programming languages and environments described in this paper are available at http://chrisproctor.net/research/teachinglanguages, where interactive versions of the visualizations included in this paper are also available.

## References


DiSessa, A. (2000). Changing Minds: Computers, Learning, and Literacy. Cambridge, MA: MIT Press.

Kelleher, C. & Pausch, R. (2005). Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. ACM Computing Surveys, Vol. 37, No. 2, June 2005.

Lessig, L. (2008). Remix: Making Art and Commerce Thrive in the Digital Economy. New York: Penguin Press.

Perkel, D. (2006). Copy and Paste Literacy: Literacy Practices in the Production of a MySpace Profile. Informal Learning and Digital Media: Constructions, Contexts, Consequences.

Matsumoto, Yukihiro. (2008). Ruby 1.9. [Recorded Google Tech Talk]. Retrieved from https://www.youtube.com/watch?v=oEkJvvGEtB4

van Rossum, G. (1999). Computer Programming for Everybody: A Scouting Expedition for the Programmers of Tomorrow. [DARPA Funding proposal] Retrieved from https://www.python.org/doc/essays/cp4e/

Weaver, C. (1996). Teaching Grammar in Context. Portsmouth, NH: Heinemann.

Wing, J. (2006). Computational Thinking. Communications of the ACM, Vol. 49, No. 3, March 2006.