The language of programming: a cognitive perspective

Evelina Fedorenko[1,2,3], Anna Ivanova[3], Riva Dhamala[4], & Marina Umaschi Bers[4]

[1] *Psychiatry Department, Massachusetts General Hospital, USA*
[2] *Psychiatry Department, Harvard Medical School, USA*
[3] *Brain & Cognitive Sciences Department, Massachusetts Institute of Technology, USA*
[4] *Eliot-Pearson Department of Child Study and Human Development, Tufts University, USA*

**Correspondence: evelina9@mit.edu (@ev_fedorenko) or Marina.Bers@tufts.edu (@marinabers)*

Abstract. Computer programming is becoming essential across fields. Traditionally grouped with STEM (science, technology, engineering, and math) disciplines, programming also bears parallels to natural languages. These parallels may translate into overlapping processing mechanisms. Investigating the cognitive basis of programming is important for understanding the human mind and could transform education practices.

The growing importance of computer programming

In the automated economy, computer programming is becoming an essential skill across diverse fields and disciplines. As a result, countries all over the world are exploring the inclusion of computer science (CS) as mandatory in the school curriculum. For example, the US government launched the "Computer Science for All" initiative in 2016 to introduce programming at all educational levels. Similar initiatives are taking place across Europe, Asia, and South America.

The growing importance of CS education underscores the urgency in characterizing the *cognitive mechanisms* and the corresponding *brain circuits* that support the acquisition and use of computer programming skills (**Box 1**). This basic knowledge is needed to guide the design of curricula, assessments, and educational policies regarding when and how to introduce computer science in schools, and inform the implementation of teaching strategies and disciplinary integration.

Furthermore, an understanding of the cognitive and neural basis of programming can contribute to the general enterprise of deciphering the architecture of the human mind. Computer programming is a cognitive invention, like arithmetic and writing. How are such emergent skills acquired? Presumably, they rely on phylogenetically older mechanisms, many of which we share with other animals. But which mechanisms? And how do these mechanisms and new domains of knowledge interact with the evolutionarily older and ontogenetically earlier-emerging ones?

Programming as problem solving

Traditionally, many researchers have construed programming in terms of problem solving, dividing it into distinct steps: problem comprehension, design, coding, and debugging/maintenance [1]. As a result, when describing the cognitive underpinnings of programming, researchers have often focused on the early stages of program planning and the ability to break down a problem into discrete units (later dubbed "computational thinking" [2]). Studies that have probed the process of coding itself have often resorted to evaluating overall cognitive load [3]. Thus, empirical research has lagged behind in exploring the relationship between mechanisms that underlie programming and other cognitive skills, in particular, language ability. Despite an abundance of metaphoric descriptions linking computer and natural languages,

such as the use of the terms "syntax" and "semantics" [4], the problem-solving approach has continued to dominate the discourse in the field of computer science education and sometimes eclipsed research exploring other cognitive mechanisms potentially involved (see **Supplementary Materials** for a more detailed overview).

Beyond STEM: an alternative construal of CS

In spite of the lack of rich and detailed characterization of the cognitive bases of computer programming, educators have long made assumptions about the relationship between programming and other cognitive skills. These assumptions have shaped the treatment of CS in schools across the world as mathematically/problem-solving oriented, and, when integrated in the curricula, CS has been grouped with STEM disciplines [5]. However, some have argued for an alternative construal of programming – an approach that has become known as "coding as literacy" [6]. The key idea is this: when you learn a programming language, you acquire a symbolic system that can be used to creatively express yourself and communicate with others. The process of teaching programming can therefore be informed by pedagogies for developing linguistic fluency.

The term "programming languages" already implies parallels to natural language. However, to rigorously evaluate the nature and extent of potential overlap in the cognitive and neural mechanisms that support computer vs. natural language processing, it is critical to delineate the core components of each process and formulate specific hypotheses about representations and computations that underlie them. Here, we propose a framework for generating such hypotheses.

With respect to ***knowledge representations***, both computer and natural languages rely on a set of "building blocks" (words and phrases in natural language, functions and variables in computer languages) and a set of constraints for how these building blocks can combine to create new complex meanings. Studies dating back to the 1970s have noted this parallel, as evidenced by the occasional reference to the "semantics" and "syntax" of programming languages [4], but few have investigated this distinction experimentally (see **Supplementary Materials**). Whereas the technical meanings of these terms in linguistics and CS differ, their usage highlights that both natural and programming languages rely on meaningful and structured representations.

With respect to ***computations***, a multi-step processing pipeline appears to underlie both comprehension and generation of linguistic/code units (**Fig. 1**). In *comprehension*, we start with perceptual input and are trying to infer the intended meaning of an utterance or to decipher what a piece of code is trying to achieve. In doing so, we initially engage in some basic perceptual processing (auditory/visual in natural languages and typically visual in computer languages), and then attempt to recognize the building blocks and the relationships among them. For longer narratives and extended pieces of code, we need to not only understand each utterance/line, but also infer an overall high-order structure of the text/program.

In ***generating*** linguistic utterances or code, we start with an idea. This idea can be a simple one and require a single sentence or line of code, or it can be highly complex and require a whole extended narrative or multi-part program. For simpler ideas or sub-components of complex ideas, we need to figure out the specific building blocks to use and to organize them in a particular way to express the target idea. For more complex ideas, we first need to determine the overall structure of the narrative or program. Once we have a specific plan for what we want to say, or what a piece of code would look like, we engage in actual motor implementation by saying/writing an utterance or typing up code statements. It is worth noting, however, that in both generating linguistic texts and computer code, top-down planning may be supplemented with more bottom-up strategies where certain fragments of the text/code are produced first or borrowed from previously generated text/code, and then the overall structure is built around those.

During these comprehension/generation processes, we also engage in other, plausibly similar, mental computations. For example, we can recognize errors – others' or our own – and figure out how to fix them [1]. When processing sequences of words and commands, we plausibly engage in predictive processing: as we get more input, we construct an increasingly richer representation of the unfolding idea, which, in turn, constrains what might come next. And during generation of utterances or code, creative thinking comes into play, affecting the very nature of the ideas one is trying to express, as well as how those ideas are converted into sentences/code. Finally, we may need to consider the intent of the producer or the state of mind of our target audience – abilities that draw on our mentalizing (Theory of Mind) capacities (although mentalizing about the computer itself might be maladaptive; see **Supplementary Materials**).

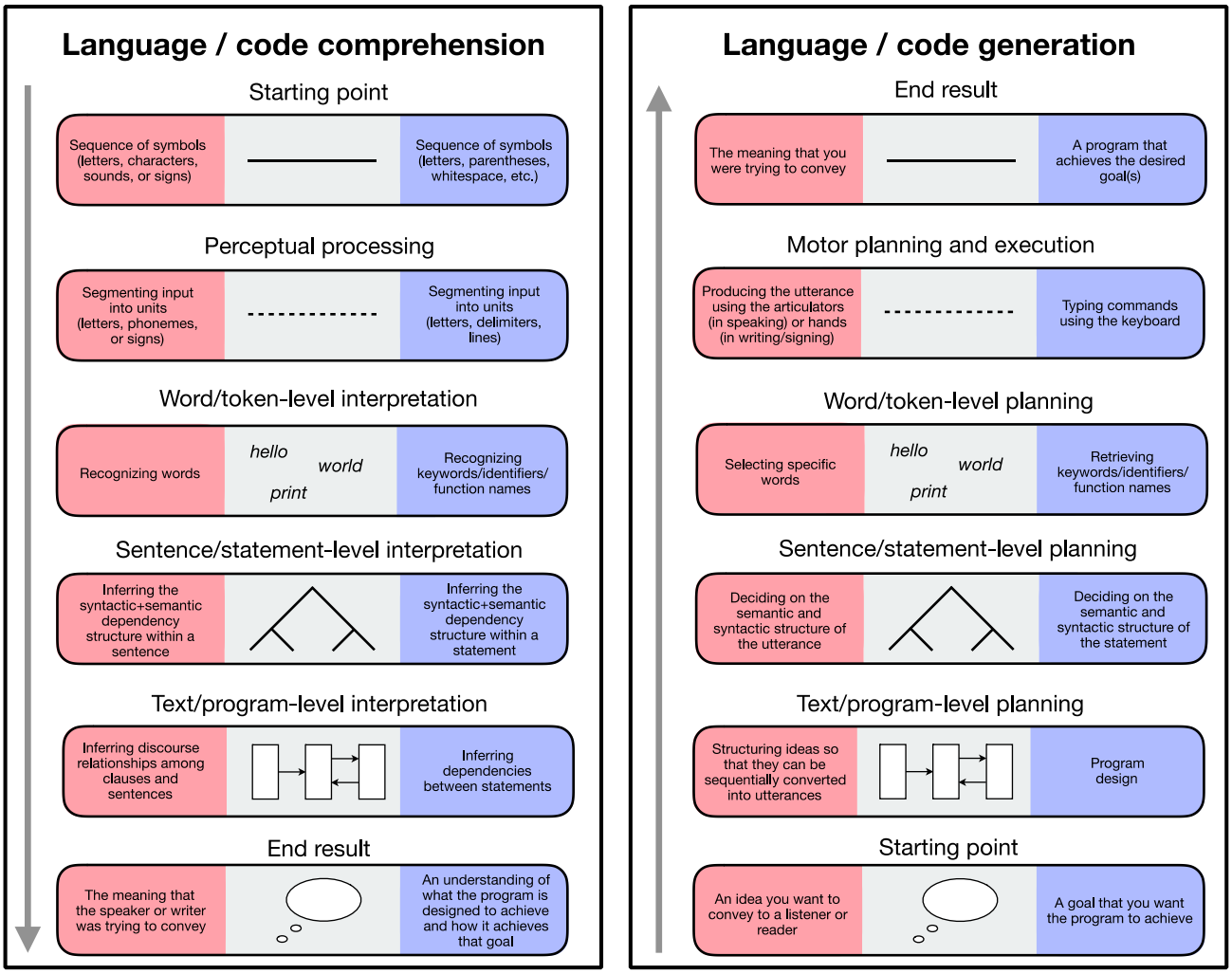

Figure 1. *Hypothesized parallels between natural language (red) and computer programming (blue) at different processing stages. Both sets of cognitive processes rely on the combinatorial nature of their inputs and outputs, which consist of phonemes/letters that make up words/identifiers, which combine into sentences/statements, giving rise to paragraphs and functions, and finally yield texts/utterances/programs (cf.* ***Supplementary Materials*** *for possible differences).*

It is also worth noting that the vast majority of programming languages directly rely on programmers' knowledge of natural languages (specifically, English). Keywords, variable names, function names, and application programming interfaces follow naming conventions that indicate their function; it has been

shown that "unintuitive" naming increases cognitive load [7] and hinders program comprehension [8]. The importance of natural language semantics is further highlighted by the fact than non-native English speakers often struggle to learn English-based programming languages [9]. Further, computer code is usually accompanied by comments and, for larger pieces of software, documentation, which serve to scaffold program comprehension. As such, the process of working with code necessarily involves tight integration of computer and natural language knowledge.

The machine learning community has already begun to exploit structural similarities between code and natural language by applying natural language processing (NLP) techniques to analyze code [10]. Developmental psychology researchers have also begun to explore those parallels. A case study [11] demonstrated that knowing a programming language can facilitate acquisition of reading ability. Pilot studies with preschoolers and kindergarteners have also shown that programming can facilitate language processing, as young children's sequencing abilities significantly improved after they received coding interventions [12]. Neuroimaging studies of programming, although in their infancy, hint at potential overlap between language- and code-processing brain regions [13]. Such findings, along with the theoretical framework presented above, call for direct investigations of cognitive and neural overlap between language and code processing.

Concluding remarks

The growing importance of computer programming underscores a need to conduct rigorous research probing the cognitive architecture that underlies programming abilities. We have highlighted potential parallels between programming and natural languages. Although the comparison is not perfect and some mental computations likely differ (see Supplementary Materials), future empirical studies should consider the hypothesis that programming draws on some of the same resources as natural language processing, in addition to the traditional proposal whereby programming shares computations with math, logic, and problem solving. If this hypothesis finds empirical support, we need to re-conceptualize the way CS is taught, especially in early childhood, when children are learning to read and write. CS learners might also benefit from techniques employed in foreign language classrooms, such as constant exposure to the language and learning by doing. Making progress in deciphering the cognitive and neural bases of computer programming may therefore yield fundamental insights about how to optimally design curricula, policy, and educational interventions, as well as new programming languages for children that might draw on pictorial and not only textual interfaces [15].

BOX 1: *Why now? The urgency of understanding the cognitive underpinnings of computer programming.*

Given the growing demand for programming skills, educators have been increasing efforts to make CS classes available to students. The question of whether CS should be grouped with STEM or with languages has sparked countless debates, with some citing Dijkstra who claimed that "mastery of one's native tongue" is key to competent programming [14] and Papert's early vision of "learning to program as a second language" [15], and others pointing to the decades-long tradition of viewing programming as principled problem-solving. Lawmakers have also weighed in on the issue. In 33 states, CS fulfills a math or science requirement, with Texas and Oklahoma providing an option to count it as a foreign language (https:/code.org/promote). At the federal level, legislators have proposed an initiative to award grants to schools that count CS toward either a math/science or a foreign language requirement.

There is a marked lack of scientific research that would support any of those initiatives. Although programming may, to some extent, recruit both STEM and language skills, no studies of programming have so far examined the exact division of labor between these cognitive domains. As new educational policies are introduced, understanding the cognitive underpinnings of programming can help guide those decisions.

Acknowledgements. This work was supported by an NSF EAGER award (FAIN 1744809, "The cognitive and neural mechanisms of computer programming in young children: storytelling or solving puzzles?") to Marina Bers and Evelina Fedorenko. We also thank three anonymous reviewers for their helpful comments and suggestions.

References

[1] Dalbey, J. and Linn, M. C. (1985) The demands and requirements of computer programming: A literature review. J Educ Comput Res 1, 253-274

[2] Wing, J. M. (2006) Computational thinking. Communications of the ACM, 49, 33-35

[3] Nakagawa, T. et al. (2014) Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Comp Proc of the 36th ICSE*, pp. 448-451, ACM

[4] Shneiderman, B. and Mayer, R. E. (1975) Towards a cognitive model of programmer behavior. Computer Science Department, Indiana University

[5] Guzdial , M. & Morrison, B. (2016) Growing computer science education into a STEM education discipline. Communications of the ACM, 59, 31-33

[6] Bers, M. U. (2019) Coding as Another Language: Why Computer Science in Early Childhood Should Not Be STEM. In *Exploring Key Issues in Early Childhood and Technology: Evolving Perspectives and Innovative Approaches* (Donohue, C., ed), Routledge

[7] Fakhoury, S. et al. (2018) The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load. In *Proc Int'l Conf Program Comprehension (ICPC)*, 286-296

[8] Lawrie, D. et al. (2006) What's in a Name? A Study of Identifiers. In *Proc Int'l Conf Program Comprehension (ICPC)*, 3-12

[9] Guo, P. J. (2018) Non-Native English Speakers Learning Computer Programming: Barriers, Desires, and Design Opportunities. In *Proc CHI*, 396. ACM

[10] Allamanis, M. et al. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys (CSUR), 51, 81

[11] Peppler, K. A. and Warschauer, M. (2011) Uncovering Literacies, Disrupting Stereotypes: Examining the (Dis)Abilities of a Child Learning to Computer Program and Read. International Journal of Learning and Media, 3, 15-41

[12] Kazakoff, E. R. and Bers, M. U. (2014) Put your robot in, put your robot out: Sequencing through programming robots in early childhood. J Educ Comput Res, 50, 553-573

[13] Siegmund, J. et al. (2014) Understanding source code with functional magnetic resonance imaging. In *Comp Proc of the 36th ICSE*, pp. 378-389, ACM

[14] Dijkstra, E. W. (1982) How do we tell truths that might hurt? In Selected Writings on Computing: A Personal Perspective, pp. 129-131, Springer

[15] Papert, S (1980): Mindstorms: Children, computers and powerful ideas. Basic Books, Inc.

Supplementary Information

I. Prior work on the relationship between programming and different cognitive functions

Attempts to develop cognitive models of computer programming date back to the 1970s (e.g., Brooks, 1977; Weinberg, 1971). Most researchers have construed programming in terms of problem solving (e.g., Dalbey & Linn, 1985; Ormerod, 1990; Pea & Kurland, 1984; Pennington & Grabowsky, 1990). Out of those, studies that have probed the process of coding itself tended to compare specific programming constructions (e.g., Sime, Green, Guest, 1977; Ganon & Hornig, 1975) or evaluate overall cognitive load (Bergersen & Gustafsson, 2011; Ikutani & Uwano, 2014; Nakagawa et al, 2014; Nakamura et al., 2003).

Empirical research has lagged behind in exploring the relationship between mechanisms that underlie programming and other cognitive skills, in particular, language ability. Early research by Sime, Green and Guest (1973) used concepts from Chomskyan linguistics (specifically, recursion depth) to test whether similar cognitive constraints apply to computer language processing. However, the problem-solving approach still dominated the field, despite an abundance of metaphoric descriptions linking computer and natural languages, such as the use of the terms "syntax" and "semantics" (Shneiderman & Mayer, 1975) or parallels drawn between learning to program and learning to read (Pea & Kurland, 1984). Murnane (1993) identified a strong need for programming to be examined in conjunction with natural language processing, but few studies have followed through.

A number of researchers have examined the link between learning to program and other cognitive skills, such as metacognition and creativity (e.g., Clements, 1986; 1995; Liao & Bright, 1991), as well as abstract reasoning (e.g., Lye & Koh, 2014; Rich et al., 2014). Others have examined effectiveness of CS concept learning by analogy (Hoc & Nguyen-Xuan, 1990) and the role of context (Chao, Feldon & Cohoon, 2018). However, rigorous empirical investigations of computer programming abilities are far and few between; most existing studies are outdated, and thus do not take into account the current understanding of human cognitive architecture.

II. A need for controlled rigorous experimentation to characterize the cognitive basis of programming

Multiple researchers have highlighted the fact that the field of programming language research has very few rigorous empirical studies (Hanenberg, 2010; Tishy, 1998; Stefik & Hanenberg, 2017). Kaijanaho (2015) found that out of 156 analyzed articles in programming language design, only 22 used randomized controlled design. Uesbeck et al. (2016) report that not a single paper from the International Conference on Functional Programming (ICFP), which started in 1996, met the standards of an empirical scientific study (as defined by the WWC Handbook by the Institute of Education Sciences). Many software researchers shy away from conducting human studies entirely, viewing them as "too difficult to design, too time consuming to conduct, too difficult to recruit for, and with too high a risk of inconclusive result" (Buse, Sadowsky & Weimer, 2011; as cited in Ko, LaTosa & Burnett, 2013). As a result, programming language choices tend to rely on mathematical reasoning rather than empirical evidence about cognitive processing (Stefik & Hanenberg, 2017), making it impossible to evaluate real-time code processing in the mind and brain.

III. Differences between computer and natural language understanding

The main aim of the article has been to highlight possible parallels in cognitive mechanisms underlying comprehension/generation of code and natural language. However, we recognize that not all parallels

will hold at all processing stages. Below, we outline several domains where computer and natural languages diverge.

Parsing. Visual structure plays a substantially more prominent role in code than in text, which means that its processing is less sequential (Busjahn et al., 2015). In addition, program structure is much more predictable (has lower entropy; Hindle et al., 2012), allowing programmers to get the gist of the program by skimming it instead of reading word by word (although natural language readers also make use of predictability by skipping low-entropy words; Rayner & Well, 1996). Finally, choosing the order in which the program should be read is often determined by the control flow, meaning that processing the meaning of a chunk of code will determine which part of the code the programmer will turn to next.

Semantics. When learning to program, a novice inevitably needs to learn a new set of concepts (Hoc & Nguyen-Xuan, 1990). Those concepts allow programmers to simulate execution flow within the machine and remove ambiguity present in natural language (e.g., inclusive vs. exclusive "or"). When the semantics of newly learned concepts overlaps with natural language semantics, learning is enhanced, and vice versa (Stefik & Siebert, 2013). Thus, computer and natural languages rely on a partially overlapping set of conceptual primitives; the fact that the nature of this overlap affects program learning makes it even more important to study it.

Pragmatics and Theory of Mind. Perhaps the most interesting set of differences between the two language types is the nature of the message recipient: natural language utterances are directed toward another human, while programs are mainly intended for machines. Both represent a form of communication; however, assumptions about natural language communication do not apply to coding, leading novice programmers to commit errors through improper knowledge transfer (Bonar & Soloway, 1983). Pea (1986) describes this misconception as "the idea that there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers". Although not explicit, this assumption may affect the process of program learning by leading programmers to transfer their "common ground" assumptions (Clark, Schreuder & Buttrick, 1983) to a computer that may not share them (Brennan, 1998). That said, the fact that such transfer mistakes happen at all, point to the fact that novice programmers often resort to their natural language knowledge when learning to code.

References for the Supplementary Information

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 10-13). ACM.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, *9*(6), 737-751.

Brennan, S. E. (1998). The grounding problem in conversations with and through computers. *Social and cognitive approaches to interpersonal communication*, 201-225.

Buse, R. P., Sadowski, C., & Weimer, W. (2011). Benefits and barriers of user evaluation in software engineering research. *ACM SIGPLAN Notices*, *46*(10), 643-656.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... & Tamm, S. (2015). Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension* (pp. 255-265). IEEE.

Chao, J., Feldon, D. F., & Cohoon, J. P. (2018). Dynamic Mental Model Construction: A Knowledge in Pieces-Based Explanation for Computing Students' Erratic Performance on Recursion. *Journal of the Learning Sciences*, *27*(3), 431-473.

Clark, H. H., Schreuder, R., & Buttrick, S. (1983). Common ground at the understanding of demonstrative reference. *Journal of verbal learning and verbal behavior*, *22*(2), 245-258.

Clements, D. H. (1986). Effects of Logo and CAI environments on cognition and creativity. *Journal of Educational Psychology, 78(4),* 309-318.

Clements, D. H. (1995). Teaching creativity with computers. *Educational Psychology Review*, *7*(2), 141-161.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. *Journal of Educational Computing Research*, *1*(3), 253-274.

Hanenberg, S. (2010). Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '10). ACM, New York, NY, USA, 933-946.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., & Devanbu, P. (2012). On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)* (pp. 837-847). IEEE.

Hoc, J. M., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In *Psychology of programming* (pp. 139-156). Academic Press.

Kaijanaho, A. J. (2015). Evidence-based programming language design: a philosophical and methodological exploration. *Jyväskylä Studies in Computing*, (222).

Ko, A. J., Latoza, T. D., & Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, *20*(1), 110-141.

Liao, Y. K. C., & Bright, G. W. (1991). Effects of computer programming on cognitive outcomes: A meta-analysis. *Journal of Educational Computing Research*, *7*(3), 251-268.

Lye, S. Y., & Koh, J. H. L. (2014). Review on teaching and learning of computational thinking through programming: What is next for K-12?. *Computers in Human Behavior*, *41*, 51-61.

Ormerod, T. (1990). Human cognition and programming. In *Psychology of programming* (pp. 63-82). Academic Press.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of educational computing research*, *2*(1), 25-36.

Pea, R. D., & Kurland, D. M. (1984). On the cognitive effects of learning computer programming. *New ideas in psychology*, *2*(2), 137-168.

Pennington, N., & Grabowski, B. (1990). The tasks of programming. In *Psychology of programming* (pp. 45-62). Academic Press.

Rayner, K., & Well, A. D. (1996). Effects of contextual constraint on eye movements in reading: A further examination. *Psychonomic Bulletin & Review*, *3*(4), 504-509.

Rich, P. J., Bly, N., & Leatham, K. R. (2014). Beyond cognitive increase: investigating the influence of computer programming on perception and application of mathematical skills. *Journal of Computers in Mathematics and Science Teaching*, *33*(1), 103-128.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, *5*(1), 105-113.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1977). Scope marking in computer conditionals—a psychological evaluation. *International Journal of Man-Machine Studies*, *9*(1), 107-118.

Stefik, A., & Hanenberg, S. (2017). Methodological irregularities in programming-language research. *Computer*, *50*(8), 60-63.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education (TOCE)*, *13*(4), 19.

Tichy, W. F. (1998). Should computer scientists experiment more?. *Computer*, *31*(5), 32-40.

Weinberg, G.M. (1971). *The Psychology of Computer Programming.* Van Nostrand Reinhold: New York.

References on CS initiatives mentioned in the main body

Balanskat, A., & Engelhardt, K. (2014). Computing our future: Computer programming and coding-Priorities, school curricula and initiatives across Europe. *European Schoolnet.*

Bocconi, S., Chioccariello, A., Dettori, G., Ferrari, A., Engelhardt, K., Kampylis, P., & Punie, Y. (2016). Developing computational thinking in compulsory education. *European Commission, JRC Science for Policy Report.*

Burning Glass Technologies (2016). Beyond Point and Click: The Expanding Demand for Coding Skills. *Retrieved at: https://www.burning-glass.com/research-project/coding-skills/ (Nov 27, 2018)*

code.org (2018). 2018 State of Computer Science Education. *Retrieved at: https://advocacy.code.org/. Accessed on 2018-11-27.*

Fayer, S., Lacey, A., & Watson, A. (2017). STEM occupations: Past, present, and future. *Spotlight on Statistics*.

Livingstone, S. (2012). Critical reflections on the benefits of ICT in education. Oxford review of education, 38(1), 9-24.