Learning Basic Python Concepts Via Self-Explanation: A Preliminary Python ACT-R Model

Veronica Chiarelli (veronicachiarelli@cmail.carleton.ca)
Department of Cognitive Science
Carleton University
Ottawa, Canada K1S 5B6

Abstract

This paper presents a cognitive modelling approach to investigating student learning of computer programming concepts via self-explanation. Self-explanation involves explaining instructional material to oneself by generating inferences about the material. Here, we present a preliminary Python ACT-R model of novice and experienced students studying basic Python concepts and self-explaining. Our contributions include formalizing the self-explanation process and providing a framework that can be expanded to explore and simulate more aspects of this type of student study and learning in the domain of programming.

Keywords: self-explanation; programming education; Python ACT-R; cognitive modelling

Introduction

Learning to program is difficult (Du Boulay, 1986; Duran, 2020; Robins, 2019) with high drop out and failure rates in computer science classes relative to other courses. Research has shown that novices lack in-depth knowledge of computer science concepts and instead tend to approach programming activities in a superficial way (Robins, 2019). For instance, when aiming to comprehend programs, students tend to paraphrase the code in front of them rather than provide higher-level explanations of the program's function (Biggs & Collis, 1982).

Given these consistent difficulties, some research in computer science education has explored ways to help students learn more effectively. Some techniques are specific to topics of computer science while others draw inspiration from educational tools or approaches used in other domains. One general technique is self-explanation.

Self-explanation is the process of generating explanations of instructional material to oneself (Chi et al., 1989). To illustrate, self-explanation can involve making inferences on the domain concepts needed to generate worked-out example solutions and/or by making connections between various concepts. Self-explanation is highly beneficial for learning, (Chi et al., 1989; Chi et al., 1994; Renkl, 1999). While not all students spontaneously self-explain, Chi et al. (1994) found that explanations can be elicited by simply prompting students to self-explain instructional text. This result has since been replicated in other studies (Conati & VanLehn, 2000).

To date, the utility of self-explanation has been mainly investigated in domains other than programming and so work is needed to identify the mechanisms of self-explanation for this domain. This paper presents a preliminary model of how students learn through self-explanation of a short instructional text about the programming language Python. Before we present the model and results, we provide some background research to contextualize our findings.

Background

Chi et al. (1989) identified the self-explanation effect through their seminal study examining how students learn from instructional materials. The goal was to identify why some students learned more than others from studying activities. The student participants read a physics textbook and then studied examples and worked on problems. While they studied examples, they verbalized their thoughts, providing access to their reasoning. Student utterances were analyzed using qualitative methods and this analysis revelated that some utterances contained self-explanations while others corresponded to mere paraphrases of the instructional materials. The results showed that students who learned more produced more self-explanations than paraphrases and that their self-explanations expanded on the material and/or linked concepts or examples. The self-explanation effect has been replicated through studies in diverse domains like biology (Chi et al., 1989), math (Renkl, 1997) and programming (Recker & Pirolli, 1995). The focus on experimental work, and particularly studies aimed at characterizing the self-explanation phenomena and/or interventions to encourage this activity has meant that there is less work computationally modeling self-explanation. There are, however, notable exceptions that we now describe.

Cascade is a Prolog-based computational model of how students solve physics problems in the presence of examples (VanLehn, Jones, & Chi, 1992). Cascade can both study examples and solve problems. When studying, it "reads" an example and attempts to self-explain each solution step by deriving it using existing facts in its memory. If no appropriate rule can be found, Cascade self-explains the example using common sense and reasoning to derive a new rule. Good students modelled by Cascade use different strategies while studying examples than poor students do. Namely, good students self-explain examples that they are studying while poor students simply accept that the examples are correct without actively processing why that is the case. Running the model demonstrates the self-explanation effect.

That is, the results show that the number of simulated self-explanations is positively correlated with the number of correct problems solved by the model. This makes sense since self-explanation increases the likelihood that a student will encounter new rules or uncover impasses or gaps in knowledge that act as learning opportunities.

Jones and Fleischman (2002) investigated student learning about probability via faded examples (incomplete examples that require the students to fill in the missing material). The hypothesis was that more learning will take place if a student is challenged to complete faded examples as opposed to studying already fully worked out examples that can more easily be passively accepted as accurate. To test this hypothesis, they added a new knowledge base to Cascade to support computing probabilities. To evaluate the extended model, they conducted simulations of the model, and also compared the model actions related to learning probabilities from faded examples with a student learning from the same examples. The simulations revealed that faded examples resulted in more student self-explanations than completely worked-out examples and that faded examples exposed more impasses (knowledge gaps), thereby uncovering more learning opportunities. However, not all student learning was accurately modeled. For example, some students would learn the correct application of a rule over the course of several examples. Cascade was unable to capture that gradual progression.

Other work has focused on modelling problem solving without including example studying or self-explanation behaviours. Braithwaite and Pyke (2017) created a computational model of learning fraction arithmetic and compared it to student learning data. The model simulated problem-solving via reinforcement learning of rules, initially including various correct and incorrect rules and then increasing activation of selected rules as it progressed through example study and problem solving. The model was implemented to reinforce strategies that lead to correct problem-solving actions when applied correctly by increasing the strategies' activation. Because they were reinforced, those strategies became simultaneously more likely to be correctly selected to solve a problem and more likely to be incorrectly selected to solve a problem for which they were not appropriate. This reinforcement of strategies was implemented because the model assumed that the majority of student errors come from overgeneralizations of fraction arithmetic rules. Test runs revealed that the textbook-trained model accurately reproduces student performance data and that model simulations trained on unbiased distributions of examples and problems performed better on problems that are underrepresented in popular textbooks. The model accurately simulated student difficulties while learning and provided some evidence in support of the assumption that unrelated statistical properties also have an impact on student learning.

So far, we have described computational models that aim to simulate human problem solving and/or example studying. Other computational frameworks produce interventions to enhance learning. For example, Conati and VanLehn (2000) extended Andes, a tutoring system for the domain of Newtonian physics, to support example studying by encouraging self-explanation through prompts and feedback. The framework doesn't simulate self-explanation but rather assesses its presence or absence using a model of the student. For example, if a student using this system spends enough time viewing an example, they will not be prompted to self-explain because the model will deem that sufficient effort was spent on the example to result in learning. Results from an evaluation of the system showed that the tutor was a beneficial tool, specifically for increasing student learning in the early stages of example study.

Recker and Pirolli (1995) also created a type of computer tutor, in this case for programming education. They investigated how students learned programming skills through self-explanation using an embedded hyper-text non-linear environment to present the instructional materials and elaborations on the text versus a typical linear instructional text. They found that high ability students (labelled as high ability based on post-test scores) benefitted from the hyper-text environment, suggesting that the self-directed learning skills of those students enabled them to use the embedded elaborations to their advantage. Conversely, low ability students did not benefit from the experimental environment, suggesting that it may have increased cognitive load for these students. Analysis of self-explanations demonstrated that good students most frequently made comments about the domain (showing that their focus was on the content) while poor students most frequently made comments related to navigation (showing that they were more focused on interface features than on the lesson content).

The challenge of understanding how students learn and how to enhance student learning is one that has been approached in various ways. We contribute to this effort and introduce a preliminary model of learning via self-explanation in the domain of programming education.

Python Self-Explanation Model

The goal of the Python self-explanation model is to simulate student learning through self-explanation in the domain of learning to program. It simulates the process of novice or experienced students self-explaining a short instructional text about Python. In its current state, the model randomly assigns a pre-test score, then self-explains each line of text by retrieving existing knowledge from memory and producing that knowledge as a self-explanation and then finally it calculates the post-test score based on the self-explanations produced. For this preliminary model, learning is simulated by an increase from pre-to post-test scores.

Theoretical Foundation

Muldner, Burleson and Chi (2014) investigated how self-explanation helps students learn about emergent phenomena in a study where students were prompted to self-explain texts about diffusion. To investigate the impact of different kinds of self-explanations on learning, each utterance was labelled as either a macro-level explanation, a micro-level explanation, an inter-level explanation, a paraphrase, or other. The results showed that some explanations were more strongly related to student learning than others. We extend this framework to the domain of programming, modifying the characterization of the explanations to make them suitable for the programming domain.

In the present work, micro-, macro- and inter-level self-explanations are defined for the domain of programming as follows. Micro-level self-explanations correspond to utterances about the directly visible elements in a program (such as programming syntax) or in the instructional text. Macro-level self-explanations are inferences about high-level programming concepts, such as the idea behind a given code segment. Inter-level self-explanations act as a bridge between micro-level and macro-level concepts and, as such, they explain the connection between the directly visible elements in the instructional text and their use or purpose.

For example, the line of instructional text "*It is necessary to update the condition in order to eventually break out of the loop*." could be explained in the following ways. A micro-level self-explanation could be "*This works using conditions that can change*." This is micro-level because it focuses uniquely on the words *condition* and *update* which are found in the line of text. A macro-level self-explanation of the same line could be "*Just like if there is no stop sign, people will keep driving.*" This self-explanation shows an understanding of the purpose and function of a while loop. An inter-level self-explanation connects micro- and macro-level ideas. "*Just like you will wash one dish at a time until there are no more dirty dishes, this will repeat until some condition happens*" is an example of an inter-level self-explanation for that same line of text. In this model, paraphrases are just restatements of the instructional text.

As in Muldner, Burleson and Chi (2014), the present model defines learning as the increase from pre- to post-test scores. Based on the learning outcomes of Muldner, Burleson and Chi (2014), in the present work, the most learning occurs with inter-level self-explanations, followed by micro-level, then macro-level, and finally the least learning occurs with paraphrases. Since the model here has not yet been developed to acquire new rules and knowledge, it uses the experimental findings about the relationship between levels of self-explanations and learning (from other domains) to calculate a post-test score, as we will describe in more detail shortly.

Model Framework, Environment and Components

Modeling Framework. ACT-R is a well-known theory of cognition which includes theories of declarative memory, procedural memory, and a chunk and buffer system (Anderson, 2007). The original computer architecture of ACT-R was implemented using Lisp. That implementation restricts modelling to directly reflect the theory, so implementing some features can implicitly have side effects on other parts of the model. To allow for flexibility, Stewart and West (2007) created Python ACT-R. This framework has the three main components from ACT-R (a chunk-based communication system, a chunk storage system, and a pattern matching production system), but is implemented in Python. The Python code is based on the theory itself rather than being a direct translation from the original Lisp. Stewart and West (2007) thus demonstrated that the theory is separable from the code. Also, the simple module creation in of ACT-R and the ability to manually adjust more components makes it more flexible than Lisp ACT-R and promotes more extensive exploration of ACT-R theory, claims, and components. For these reasons, we used Python ACT-R as the basis for the present model. The model includes an environment and modules, described below.

Model Environment. The main component of the model environment is the instructional text. For the present work, the text describes the syntax and the concepts of "if statements" and "while loops" in Python. There are 13 lines of text in total. Each line is stored as an element in the instructional text environment and has an associated state. The lines are initially in a state of "read" to indicate that the line has yet to be read (and subsequently self-explained by the model). Each line of the text is also labelled as belonging to one of the three levels of knowledge (macro-, micro- or inter-level).

Other information stored in the environment includes a count of the number of macro-level, micro-level, and inter-level self-explanations as well as a count of paraphrases. All of these are initially set to 0 since no self-explanations have been produced before the model runs. During model execution, the counts are updated as each line of the text has been self-explained to keep track of the type of self-explanation produced. The environment also includes the pre-test score and the post-test score used to quantify learning. The calculation of these scores happens in the self-explain module (described below). Finally, the environment includes the experience level of the student being simulated by the model, either "novice" or "experienced". This experience level influences the model execution, reflecting that novices self-explain differently and have more to learn than experienced students.

Model Modules. A key benefit of self-explanation is the integration of new and existing knowledge, meaning that students make connections between the text and what they already know. In order for the model to simulate existing knowledge, the Python ACT-R declarative memory module is initialized to model a student's prior knowledge. Specifically, the model's declarative memory is initialized with chunks of domain information. Each chunk specifies the knowledge itself (a piece of existing knowledge), the level of that knowledge (macro-level, micro-level, inter-level), and the topic of the knowledge (a label indicating the topic of the existing knowledge). When the model runs, the declarative memory buffer is used to retrieve existing knowledge chunks. Some noise is added to the declarative memory to account for the fact that which chunk of knowledge is retrieved is not always predictable.

When the model self-explains a line of text, it attempts to retrieve a relevant chunk of existing knowledge from declarative memory. The relevance of a chunk is influenced by its topic and its knowledge level. A chunk is most relevant to a line of text if they have the same topic and if their knowledge level (inter-, micro-, macro-level) is similar. Knowledge levels were defined as partially similar to one another as follows. Inter-level was set to be partially similar to both macro-level and micro-level while micro-level and macro-level are set to be dissimilar. This choice was made based on the assumption that an inter-level explanation for a topic is always suitable since it serves as a bridge between the other concept levels. Meanwhile, micro-level and macro-level concepts are quite different and, therefore, it is less likely that a student would choose to produce a micro-level self-explanation for a macro-level line of text, for example.

The model also includes a self-explanation module that contains a production to change the state of a line of text from "read" to "self-explain" (to indicate that the line has been read and self-explained), a production to update the count of the different levels of self-explanations produced, and productions for updating the pre-test and post-test scores. Since this is a preliminary model, it does not yet simulate the process of taking the pre-test and the post-test. Instead, the pre-test score is randomly determined from a range of values depending on the student experience level. Novice student pre-test scores arbitrarily range from 30% to 40% based on the assumption that novices will not have the knowledge required to pass a programming pre-test. Experienced students are assumed to only have minimally more experience than novice students and, as such, their randomly chosen pre-test score will fall within the range of 45%-60%. Given that previous research has demonstrated that inter-level explanations are associated with the most learning, micro-level with slightly less learning and macro-level and paraphrases with the least learning (Muldner, Burleson & Chi, 2014), the following equation was used to determine a post-test score:

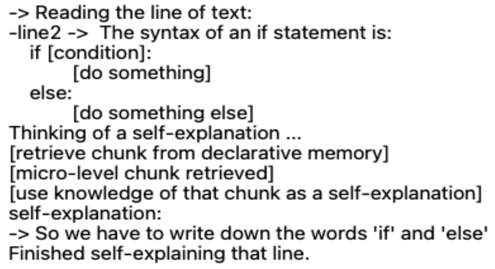
$$\text{Post-test score} = \text{S} + \left(\frac{1}{13}\right) \bullet \left(\text{I} + \left(\frac{3}{4}\right) \bullet \text{M} + \left(\frac{1}{2}\right) \bullet \text{A} + \left(\frac{1}{3}\right) \bullet \text{P}\right) \bullet (100\text{-S})$$

where S is the pre-test score, I is the number of inter-level self-explanations produced, M is the number of micro-level self-explanations produced, A is the number of macro-level self-explanations produced, and P is the number of paraphrases produced. With this formula, all simulated students will have post-test scores higher than their pre-test scores, which makes sense since it is assumed that students will not lose any programming knowledge by self-explaining the instructional text. Further, in order to match previous findings in other domains that show the most learning is associated with inter-level explanations, a perfect score in this model is possible if all self-explanations are inter-level. (Note this is based on an assumption that the findings in other domains hold in this domain which still needs to be verified experimentally.) All other possible scores are a function of the number of each level of self-explanation produced weighted by the relative amount of learning assumed to be associated with the given level of self-explanation.

Model Execution

A run of the preliminary model begins by manually setting the model parameter for experience level as either novice or experienced. While the same Python ACT-R parameters are used to produce self-explanations when simulating either type of student, the student type influences the levels of self-explanations produced. In the programming domain, novices have been shown to be more likely to provide micro-level self-explanations than any other level of self-explanation (Robins, 2019). So, if simulating a novice student, the model rehearses micro-level knowledge in memory, thereby making it more salient in memory and strengthening its activation. In other words, novice student simulations are more likely to retrieve micro-level knowledge when self-explaining. If the experience parameter indicates previous programming experience, then the model will rehearse inter-level knowledge since, unlike novices, experienced learners are known to have more complete schemas and can therefore connect different levels of ideas (Robins 2019). This is why inter-level knowledge is more likely to be retrieved during a simulation of an experienced student's self-explanations. Like all other levels of knowledge, macro-level knowledge still is added to declarative memory for every type of student, it is just not rehearsed and therefore is less salient and less likely to be retrieved. This is because Muldner, Burleson and Chi (2014) reported than macro-level self-explanations are least frequently produced by all students. Next, the model fires the pre-test production to randomly assign a pre-test score to the student, influenced only by the experience level.

The model then simulates self-explaining of the text. Specifically, it reads a line of instructional text and connects that text to existing knowledge. This is achieved by retrieving chunks of knowledge from declarative memory related to the topic and knowledge level of the text line (recall that all chunks in declarative memory are labelled with the topic and level). Figure 1 demonstrates a summary of the process of self-explaining a line of the instructional text. The model reads the line "*The syntax of an if statement is: if [condition]: [do something] else: [do something else]*", retrieves a micro-level chunk from memory, and produces the chunk's knowledge as a self-explanation "*So we have to write down the words 'if' and 'else'*".



```
-> Reading the line of text:
-line2 ->  The syntax of an if statement is:
   if [condition]:
         [do something]
   else:
         [do something else]
Thinking of a self-explanation ...
[retrieve chunk from declarative memory]
[micro-level chunk retrieved]
[use knowledge of that chunk as a self-explanation]
self-explanation:
-> So we have to write down the words 'if' and 'else'
Finished self-explaining that line.
```

Figure 1: Self-explaining a line of text.

As mentioned, knowledge levels have been assigned to lines of instructional text and to chunks in declarative memory. However, given the a priori specified similarity between knowledge levels and the effect of noise in declarative memory, the retrieved knowledge may not correspond to the knowledge level or topic of the line of instructional text. This simulates the fact that students may not always be able to retrieve related knowledge when they want to self-explain. For this model, any topic-relevant knowledge that is retrieved will lead to successful self-explanation since, for example, a macro-level line can be explained by inter-level knowledge. However, if the retrieved knowledge does not relate to the topic of the line of text, then that knowledge cannot be used to produce a self-explanation of the line resulting in the model disregarding the retrieved knowledge and, instead, just paraphrasing the line. Similarly, a failure to retrieve knowledge of any kind for a given line leads to a paraphrase of that line. After each line is self-explained, the count of each level of self-explanation is increased accordingly.

Finally, when the model has self-explained each line of the text, it calculates a post-test score using the post-test score formula previously described. The simulation displays the pre-test score, the number of self-explanations of each level produced, and the post-test score.

Results

Sample runs of the preliminary model were used to evaluate how it performed when modelling learning via self-explanation in the domain of programming. Table 1 displays results of running the Python self-explanation model 5 times as a novice programming student, and 5 times as a more experienced programming student. The table displays the pre- and post-test percentage scores and the percentage of learning gains along with the count of each level of self-explanation produced.

The results indicate that the Python self-explanation model does accurately simulate some findings of learning via self-explanation. Micro-level self-explanations were the most common type of explanation produced by novice student simulations and inter-level self-explanations were the most common type produced by experienced student simulations. The novice student simulation reflects experimental data showing that novice programmers focus on the line-by-line details rather than the overarching concepts or connections between the syntax and the concept (Robins, 2019). This model also accurately reflects prior findings that, with experience, more complete schemas exist connecting code to concepts thereby permitting inter-level self-explanations for experienced learners. The very low number of macro-level self-explanations as compared to inter-level or micro-level self-explanations matches the observations of Muldner, Burleson and Chi (2014) and is understandable in our simulations since the model rehearses micro-level or inter-level knowledge (depending on student experience level) but not macro-level knowledge. So, while chunks of all three levels of knowledge exist in declarative memory, macro-level chunks have not been rehearsed for the reasons stated above and are therefore less salient and less likely to be retrieved for self-explanation.

The relationship between learning gains and levels of self-explanation matches the relationship described in the Muldner, Burleson and Chi (2014) data. This is built into the model as the post-test score is a weighted function of the levels of self-explanations produced. For example, the highest learning gains for a novice come from S1, and for an experienced simulation, S10, both of whom produced more inter-level self-explanations than any other simulations with their experience level. So, inter-level self-explanations resulted in the most learning.

The results accurately indicate that there were learning gains for all simulated students. Further, novice runs of the model result in more learning than experienced runs. This seems reasonable since novice students simply have more to learn. However, since the pre-test scores are randomly selected and the post-test scores are simply calculated as a function of weighted level of self-explanations produced and the pre-test score, this result is hard coded into the model rather than being determined by simulating the pre-test and post-test in full, so these results are expected.

Table 1: Results of 10 sample runs.

Student	Pre-test	Micro	Inter	Macro	Paraphrase	Post-test	Gains
N S1	37	5	6	0	2	87	50
N S2	36	6	1	0	6	72	36
N S3	38	7	2	1	3	79	41
N S4	36	5	4	0	4	80	44
N S5	32	5	3	0	5	76	44
Average (Novice)	35.8	5.6	3.2	0.2	4	78.8	43
E S6	51	3	4	1	5	82	32
E S7	55	2	3	2	6	80	25
E S8	51	3	5	1	4	85	34
E S9	52	4	3	2	4	82	30
E S10	49	1	7	0	5	85	36
Average (Experienced)	51.6	2.6	4.4	1.2	4.8	82.8	31.4

Discussion and Future Work

As described in the background section, cognitive models can provide valuable insight into how students learn and can inform effective teaching strategies and interventions. Yet, due to the complexity and intricacies involved in learning, such as individual learner differences and differences in domains, a complete model of the learning process has yet to be created. The Python self-explanation model is a preliminary step for informing on learning via self-explanation in the domain of programming.

While the current model captures some aspects of self-explanation such as drawing on existing knowledge, it does not yet simulate the acquisition of new knowledge. If the learning process were expanded to include the ability to learn new rules through commonsense and general reasoning, as is the case with models like Cascade (VanLehn, Jones, & Chi 1992), then it would be possible to also simulate the pre- and post-test activities, as opposed to randomly producing a pre-test score and then a post-test percentage calculated using weighted counts of levels of self-explanations. Supporting new rule acquisition and subsequently modelling the pre- and post-tests could provide insight into mechanisms used by students when they apply the knowledge gained from the self-explanation exercise to a problem-solving test.

There are various other avenues for future work. One extension would be to model more types of students. Robins (2019) describes that in the field of programming education, three distinct clusters of students emerge. There are "stoppers" who withdraw from or abandon the activity quickly when they encounter difficulties, "movers" who trace code and try to navigate to a correct solution when they notice an issue, and "tinkerers" who react to problems by trying different tweaks of the code somewhat haphazardly and without code tracing. These clusters all pertain to program generation and so work is needed to determine if these clusters also emerge in activities that involve reading and explaining programs and instructional materials. If similar clusters of student types exist when self-explaining, these student types' self-explanation patterns could be modelled in addition to modelling differences in two experience levels. Alternatively, modelling the learning patterns of good and poor students as examined in Chi (1989) or of high and low ability students as in Recker and Pirolli (1995) could be an informative next step. A more sophisticated extension could include modelling individual differences on a continuum from novice to expert rather than modelling students as falling within specific experience or type categories.

Another avenue for future work involves adding capability to model student emotion during the self-explanation process. We began work for this step. Specifically, although it was not described here, our model includes a preliminary emotion module. Currently, all simulations produce states corresponding to motivated and happy at times, but also individual runs of the model will produce either frustration or boredom while self-explaining each line of text, influenced only by the student experience level. That is, runs of the model representing experienced students produce reports of feeling bored more often than frustrated while novice runs more frequently produce frustration. This was a first step in modelling some basic emotions guided by the assumption that novice difficulties lead more often to frustration while experienced pre-existing programming knowledge makes reading a basic instructional text a more boring exercise. The model does not yet take into account text complexity (e.g., which may commonly elicit frustration across all experience levels). Also, the emotion is not yet related to the level of self-explanation produced so, the fact that a student paraphrases because they cannot retrieve relevant knowledge, for example, does not make them any more likely to feel any frustration than if they, say, successfully self-explain a line with inter-level knowledge. This leaves a lot of room for improvement in the emotion module of future versions of the model including modelling a wider range of emotions, the connection between lines of text and emotions, the relationship between successful self-explanations and emotions, or even the intricacies of the various emotions associated with more types of students (such as the stoppers, movers, tinkerers, or good and poor students suggested previously).

Additionally, most existing data comes from self-explanation studies in other domains. Confirming that the same patterns emerge within the topic of learning to program would better inform this and future models.

Acknowledgements

Thank you to Robert West for providing guidance on Python ACT-R and to Kasia Muldner for providing feedback on this work and paper.

References

Anderson, J. (2007). 1 Cognitive Architecture. In *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press.

Biggs, J. B., & Collis, K. F. (1982). Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome). *New York: Academic Press.*

Braithwaite, D. W., Pyke, A. A., & Siegler, R. S. (2017). A computational model of fraction arithmetic. *Psychological Review, 124(5).*

Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13(2).*

Chi, M. et al. (1994). Eliciting self-explanations improves understanding. *Cogn. Sci. 18 (1994).*

Conati, C., & VanLehn, K. (2000). Toward computer-based support of meta-cognitive skills: A computational framework to coach self-explanation. *The International Journal of Artificial Intelligence in Education, 11.*

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research,* 2(1).

Duran, R. (2020). *Cognitive Complexity of Comprehending Computer Programs.* Aalto University.

Fleischman, E., & Jones, R. (2002). Why example fading works: A qualitative analysis using cascade.

Muldner, K., Burleson, W., & Chi, M. T. H. (2014). Learning from self-explaining emergent phenomena. *Proceedings of International Conference of the Learning Sciences, ICLS, 2*.

Recker, M. M., & Pirolli, P. (1995). Modeling individual differences in students' learning strategies. *Journal of the Learning Sciences, 4(1).*

Renkl, A. (1999). Learning mathematics from worked-out examples: Analyzing and fostering self-explanations. *European Journal of Psychology of Education, 14(4).*

Robins, A. (2019). Novice programmers and introductory programming. *The Cambridge Handbook of Computing Education Research.*

Stewart, T., & West, R. (2007). Deconstructing and reconstructing ACT-R: Exploring the architectural space. *Cognitive Systems Research*, 8(3).

VanLehn, K., Jones, R. M., & Chi, M. T. H. (1992). A model of the self-explanation effect. Journal of the Learning Sciences, 2(1).Hill, J. A. C. (1983). A computational model of language acquisition in the two-year old. *Cognition and Brain Theory, 6.*