Algorithms and Programming

Literature Review

SUGGESTED CITATION: Computer Science Teachers Association. (2026). *Algorithms and programming: Literature review.* https://csteachers.org/k12standards/research

© 2026 Computer Science Teachers Association (CSTA).

We conducted a literature review to synthesize research on learning progressions for algorithms and programming. These topics have been studied extensively by researchers, so we focused on 11 articles that were primarily syntheses of this larger research base and that attended to stages of student understanding for the related subconcepts. The References section lists each article and the subconcepts and grade bands they covered.

Design Recommendations

The literature reviewed is in accord with many of the 2017 Computer Science Teachers Association (CSTA) K–12 Standards, so the existing standards provide a solid starting ground for the 2026 revision. Table 1 on the following pages details which articles provide evidence for the specified standards, with recommendations for the content and placement of standards. It is important to note that the majority of the reviewed literature concentrated on grades K–8. Consequently, only these grade bands are presented in the table.

Two areas within Program Development standards were not discussed in the reviewed articles, but are included as practices in the new pillars:

- There was no discussion of attribution and intellectual property (1A-AP-13, 1B-AP-14, 2-AP-16) in the reviewed articles. These topics appear in the Impacts and Ethics Pillar ("Practice 2: Respect users and other creators when creating computational technologies").
- There was no discussion of project roles or project timelines (1B-AP-16, 2-AP-18) in the reviewed articles. These topics appear in the Inclusive Collaboration Pillar ("Practice 2: Actively manage projects and project teams").

We recommend integrating these as practices within other standards instead of making these standalone standards.

Several articles discuss the importance of program comprehension (i.e., reading, interpreting, and communicating about code), which seems absent in the 2017 CSTA K–12 Standards (see articles [5], [6], and [11]). The gray boxes in Rich et al.'s (2018b) ([10]) learning trajectories for conditionals, repetition, and sequence (Figures 2–4) provide ideas that are relevant to program comprehension and that might be included as standards under the Algorithms subconcept. These trajectories are presented in the Control and Modularity section.

One article ([4]) examines how the choice of programming language—such as visual, block-based languages like Scratch, compared to text-based languages like Python—affects students' understanding of variable initialization (described further in the Variables section). Although the standards typically avoid citing particular programming languages, it might be beneficial to evaluate whether (a) any new content standards presume a level of understanding tied to the programming language type, and (b) it would be advantageous to incorporate standards that explicitly address the ability to transition conceptual knowledge across different programming language types.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Table 1. *Alignment between 2017 CSTA K–12 Algorithms and Programming Standards and the Reviewed Literature*

Subconcept	Grades K–2 (Level 1A)	Grades 3–5 (Level 1B)	Grades 6–8 (Level 2)
Algorithms	**1A-AP-08** Model daily processes by creating and following algorithms (sets of step-by-step instructions) to complete tasks. ***Supporting articles:*** [2], [11]	**1B-AP-08** Compare and refine multiple algorithms for the same task and determine which is the most appropriate. ***Supporting article:*** [2] *[2] also includes a goal of distinguishing between speeding up implementation and speeding up the algorithm.*	**2-AP-10** Use flowcharts and/or pseudocode to address complex problems as algorithms.
Variables	**1A-AP-09** Model the way programs store and manipulate data by using numbers or other symbols to represent information. ***Supporting article:*** [11]	**1B-AP-09** Create programs that use variables to store and modify data. ***Supporting articles:*** [4], [11] *[11] also notes operators at this level.*	**2-AP-11** Create clearly named variables that represent different data types and perform operations on their values. ***Supporting article:*** [11]
Control	**1A-AP-10** Develop programs with sequences and simple loops, to express ideas or address a problem. ***Supporting articles:*** [1], [11] *[1] mentions students being able to "create simple programs with simple cause and effect commands." [11] also notes events at this level.*	**1B-AP-10** Create programs that include sequences, events, loops, and conditionals. ***Supporting articles:*** [10], [11] *[10] identifies goals related to understanding conditionals/repetition that do not require creating programs. These ideas could be under Algorithms.* *[11] mentions parallelism at this level.*	**2-AP-12** Design and iteratively develop programs that combine control structures, including nested loops and compound conditionals. ***Supporting articles:*** [10], [11] *[11] also mentions parallelism at this level.* *[1] also mentions this topic for ages 4–7.*

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Subconcept	Grades K–2 (Level 1A)	Grades 3–5 (Level 1B)	Grades 6–8 (Level 2)
Modularity	**1A-AP-11** Decompose (break down) the steps needed to solve a problem into a precise sequence of instructions. ***Supporting article:*** [8]	**1B-AP-11** Decompose (break down) problems into smaller, manageable subproblems to facilitate the program development process. ***Supporting articles:*** [8], [11] *[7] provides some evidence that students at this age have a beginner-level understanding of decomposition.*	**2-AP-13** Decompose problems and subproblems into parts to facilitate the design, implementation, and review of programs. ***Supporting article:*** [11]
		1B-AP-12 Modify, remix, or incorporate portions of an existing program into one's own work, to develop something new or add more advanced features. ***Supporting articles:*** [5], [10], [11]	**2-AP-14** Create procedures with parameters to organize code and make it easier to reuse. ***Supporting article:*** [10]
Program Development	**1A-AP-12** Develop plans that describe a program's sequence of events, goals, and expected outcomes. ***Supporting articles:*** [1]	**1B-AP-13** Use an iterative process to plan the development of a program by including others' perspectives and considering user preferences. ***Supporting articles:*** [11] *[1] suggests this standard may be achievable at the prior grade band, stating students are able to "understand a situation from the point of view of others" and "create programs that reflect this understanding."*	**2-AP-15** Seek and incorporate feedback from team members and users to refine a solution that meets user needs.
	1A-AP-13 Give attribution when using the ideas and creations of others while developing programs.	**1B-AP-14** Observe intellectual property rights and give appropriate attribution when creating or remixing programs.	**2-AP-16** Incorporate existing code, media, and libraries into original programs, and give attribution.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Subconcept	Grades K–2 (Level 1A)	Grades 3–5 (Level 1B)	Grades 6–8 (Level 2)
Program Development	**1A-AP-14** Debug (identify and fix) errors in an algorithm or program that includes sequences and simple loops. ***Supporting articles:*** [1], [3], [5], [11] *[1] also calls out "distinguish and fix logical errors," which may be made more explicit in the current standard.*	**1B-AP-15** Test and debug (identify and fix errors) a program or algorithm to ensure it runs as intended. ***Supporting articles:*** [5], [9], [11]	**2-AP-17** Systematically test and refine programs using a range of test cases. ***Supporting articles:*** [9], [11]
		1B-AP-16 Take on varying roles, with teacher guidance, when collaborating with peers during the design, implementation, and review stages of program development.	**2-AP-18** Distribute tasks and maintain a project timeline when collaboratively developing computational artifacts.
	1A-AP-15 Using correct terminology, describe steps taken and choices made during the iterative process of program development. ***Supporting article:*** [11]	**1B-AP-17** Describe choices made during program development using code comments, presentations, and demonstrations.	**2-AP-19** Document programs in order to make them easier to follow, test, and debug.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Literature Review

When researchers started studying computer science education in the 1960s and 1970s, they paid a lot of attention to how students understood programming, how best to teach programming, and designing educational programming languages.[1] As a result, there is a much larger literature base related to Algorithms and Programming than to the other four concepts in the 2017 CSTA K–12 Standards. Several scholars have summarized this research and created two types of guides that show how students typically learn about this concept area: learning progressions and learning trajectories. *Learning progressions* describe the concepts and skills students are expected to gain as they learn about a domain, and often inform curriculum design and assessment development. *Learning trajectories* describe students' understanding of a domain as they progress in their levels of thinking, and often inform analysis of student thinking and help identify areas of additional support.

Standards define what students should know and be able to do regarding a topic, so learning trajectories and learning progressions can provide guidance on the content and ordering of standards. The following sections summarize the learning progressions and learning trajectories for (a) the broad topic of algorithms and programming, (b) variables, (c) control and modularity, and (d) program development and program comprehension. In reviewing these summaries, the reader should know that:

- Much of the research cited in this review focuses on the elementary and middle school levels.
- The type of programming language used can influence what students can and do learn about algorithms and programming.
- Many studies at the K–8 levels focus on block-based programming languages.
- The research cited in this review places more emphasis on creating code than on understanding existing code, though both are important for learning.

Algorithms and Programming

Three articles provide learning progressions covering the broad topic of algorithms and programming.

Zhang and Nouri's (2019) ([11]) literature review identifies the algorithms and programming skills K–9 students can develop using the Scratch programming language. They use the term *computational thinking* to describe these skills. Their learning progression (Table 2) identifies the concepts, practices, and perspectives students develop by three grade levels that overlap somewhat with the CSTA grade bands. They also note that some more advanced topics (e.g., "nested loops, conditional loops, loops/conditionals with variables, loops/conditionals with Boolean logic, parallel multiple events, systematic testing and debugging") are difficult to learn in Scratch and are often a focus at the seventh to ninth grade levels. Lastly, they indicate that the reviewed studies had a greater focus on concepts than on practices and perspectives because concepts are easier to assess.

---

[1] Guzdial, M., & du Boulay, B. (2019). The history of computing education research. In S. A. Fincher & A. V. Robins (Eds.), *The Cambridge handbook of computing education research* (pp. 11–39). Cambridge University Press.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Table 2. *Zhang & Nouri's (2019) Learning Progression for Computational Thinking (CT) Skills (Grades K–9)*

	5–9 years old Grades K–3	9–12 years old Grades 4–6	12–15 years old Grades 7–9
CT concepts	• Algorithm/Sequences • Event • Variable	• Sequences • Loops • Events • Parallelism • Conditionals • Operators • Data • Input/output	• Sequences • Loops • Events • Parallelism • Conditionals • Operators • Data • Input/output
CT practices	• Being iterative and incremental • Debugging • Predictive thinking • Reading, interpreting, and communicating the code	• Abstracting and modularizing • Being incremental and iterative • Multimodal design • Reusing and remixing • Reading, interpreting, and communicating the code • Testing and debugging	• Abstracting and modularizing • Being incremental and iterative • Multimodal design • Predictive thinking • Reusing and remixing • Reading, interpreting, and communicating the code • Testing and debugging
CT perspectives	• Connecting • Expressing • Questioning	• Connecting • Expressing • Questioning • User interaction	• Expressing • User interaction

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Focusing specifically on young children ages 4 to 7, Bers (2019) ([1]) argues that programming should be viewed as a literacy, not simply as a tool for problem solving. Drawing on two decades of research, she presents a learning progression (Table 3) that describes six stages young learners go through when using developmentally appropriate educational programming languages (e.g., tangible and graphical block-based languages like Scratch or KIBO). This learning progression was inspired by six stages of literacy development: emergent, coding and decoding, fluency, new knowledge, multiple perspectives, and purposefulness.

Table 3. *Bers's (2019) Coding Stages for Young Learners (Ages 4 to 7)*

Stages	Coding
1. Emergent	• Learn the variety of purposes of technology (hardware/software) • Recognize that technologies are human engineered • Explore what a programming language is and when it is used • Learn concepts of interface (e.g., turning on and off, different elements of interfaces, screens, inputs and outputs) • Understand the concept of symbolization and representation (i.e., a command is not the behavior, but represents the behavior) • Explore basic control structures, such as cause and effect • Identify when technologies do not work and the need to problem solve
2. Coding and decoding	• Learn a limited set of symbols (syntax) and grammar rules within a programming language • Understand that sequencing matters and that the order in which commands (symbols) are put together generates different behaviors • Create simple programs with simple cause and effect commands • Learn to engage in simple debugging by trial and error • Identify and fix grammatical errors in the code (i.e., make it work) • Instruction is specific to the developmentally appropriate programming language of choice (e.g., KIBO or ScratchJr)
3. Fluency	• Consolidation of all that was previously learned • Master the full syntax of the programming language and be able to create complex programs using control structures • Apply knowledge gained to create complex programs that are personally meaningful • Ability to apply all steps of the design process • Distinguish and fix logical errors in the code (i.e., a program runs, but it doesn't do what is expected) • Transition from 'learning to code' to 'coding to learn'

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Stages	Coding
4. New knowledge	• Ability to combine multiple control structures and create nested programs for making a complex project • Consolidation of debugging tools, allowing for more strategic debugging • Learning how to learn new commands
5. Multiple perspectives	• Understand a situation from the point of view of others and be able to create programs that reflect this understanding • Creating programs that involve user's input and that can span different expressive genres (i.e., games, stories)
6. Purposefulness	• Coding is skillfully used for one's own needs and purpose • Coding at high levels of abstraction requiring high skill and flexibility • Coders analyze, synthesize, and make judgments based on what they read • Coding is rapid and efficient

Dwyer et al. (2014) ([2]) present findings from focus group interviews conducted with fourth grade students, which included a CS Unplugged activity created by the authors. In relation to students' understanding of step-by-step instructions, they note that students were able to do the following without any instruction: recognize the need for specific instructions and identify what was lacking in instructions; however, students had difficulty creating their own instructions, remembering every attribute of the problem, or describing attributes with precision. In relation to students' understanding of algorithmic development, they note that students were able to create an algorithm related to sorting canisters and provide multiple ways to make their algorithm implementation faster; however, some students had difficulty creating algorithms that were scalable. Dwyer et al.'s resulting learning progression (Figure 1) shows what knowledge they expect fourth grade students to have about algorithms and programming before receiving any instruction (i.e., the lower anchor points) and the various levels of knowledge they might progress through by the end of their instruction.

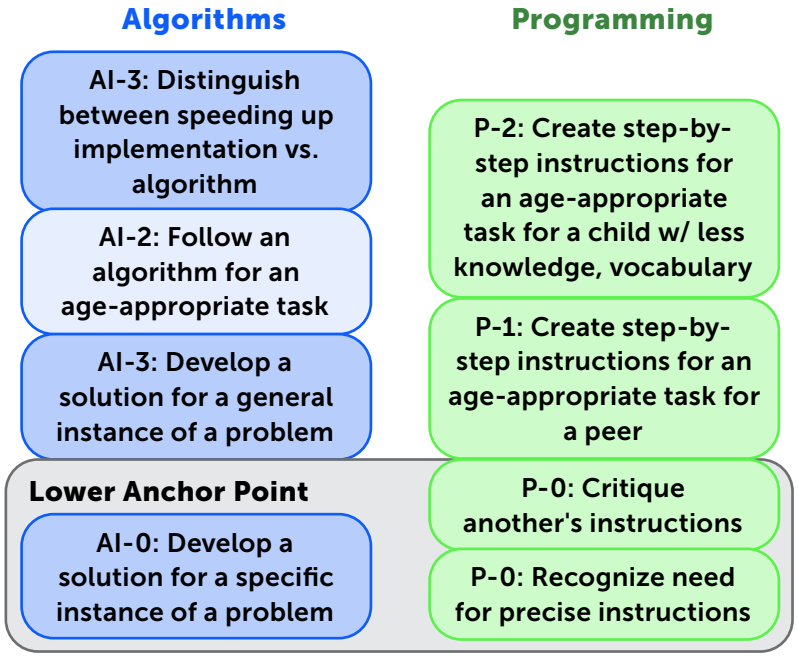

Figure 1. *Dwyer et al.'s (2014) Learning Progression for Algorithms and Programming at the Fourth Grade Level*

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Variables

While there was no learning progression or learning trajectory specific to the subconcept of Variables in the reviewed literature, one article discusses student understanding of these topics. Acknowledging that many students transition from visual block-based languages to traditional programming languages, Franklin et al. (2016) ([4]) examines how initialization of state and variables in Scratch might not transfer well to traditional text-based languages (e.g., C and Java). Drawing on observations of 9-to-12-year-old students, the authors identify the following as potential challenges for transferring knowledge of variables from one type of programming language to another:

- Text-based languages provide less syntax scaffolding.
- Visual block-based languages operate mainly on graphical data, and text-based languages operate mainly on strings and numbers.
- Traditional languages require variables to be initialized before use, but it is not obvious that this is important in Scratch.
- Traditional languages do not maintain a variable's end state, whereas Scratch does.

To better aid in transfer of knowledge about variable initialization between types of programming languages, they recommend the following for programming in Scratch:

- Initialization should occur at the beginning of the program.
- Initialization should occur in the green flag script.
- Initialization should be hidden from the user.
- Absolute blocks must be used to initialize.

Control and Modularity

Rich and colleagues (2018a, 2018b) ([8, 10]) conducted an extensive literature review in order to create four learning trajectories related to conditionals (Figure 2), repetition (Figure 3), sequence (see Figure 4), and decomposition (Figure 5). Their reviews focused on literature that (a) examined learners in grades K through 8 and (b) focused mainly on block-based programming languages. The resulting trajectories provide a map of student thinking at beginning, intermediate, and advanced stages, as well as possible pathways through which student knowledge might build. In these trajectories, gray boxes relate to unplugged goals and white boxes relate to computer-based goals.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

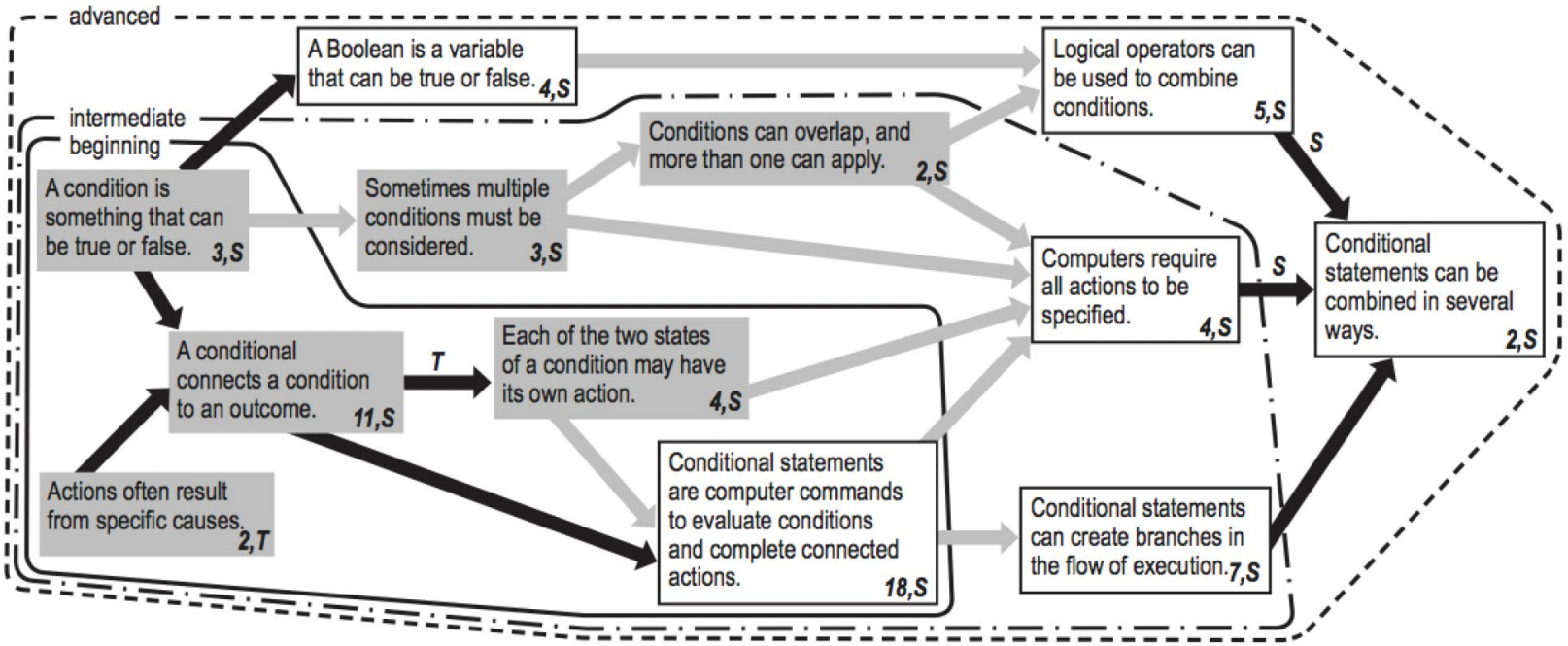

Figure 2. *Rich et al.'s (2018b) Learning Trajectory for Conditionals (Grades K–8)*
Note. This figure is fully described in Appendix A.

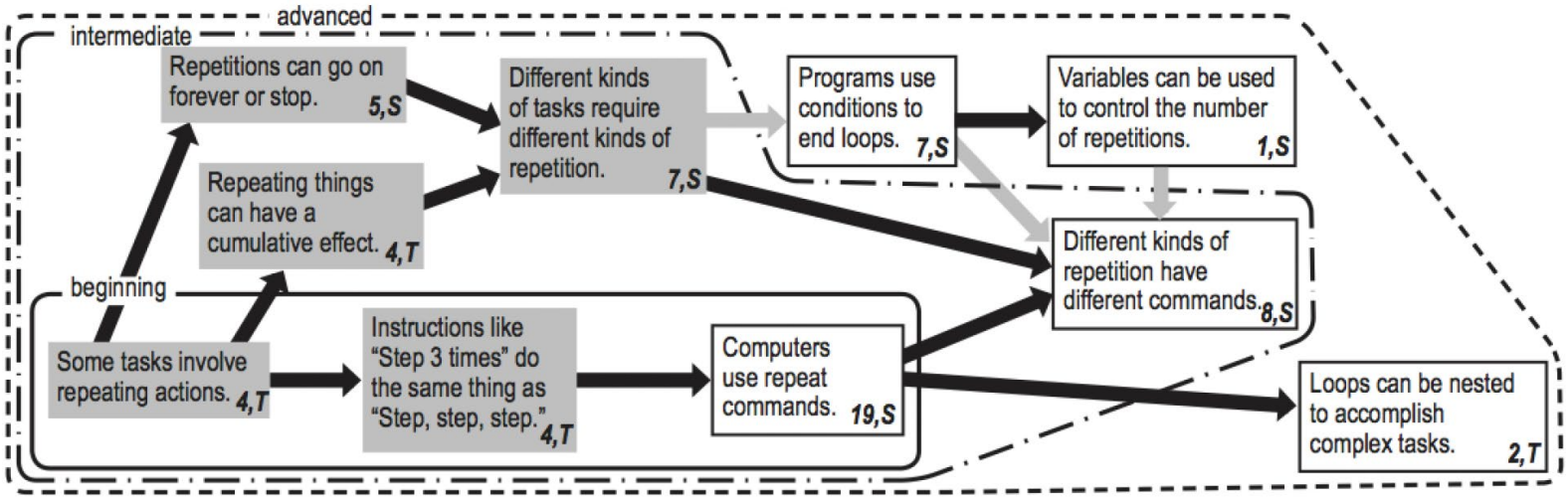

Figure 3. *Rich et al.'s (2018b) Learning Trajectory for Repetition (Grades K–8)*
Note. This figure is fully described in Appendix B.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

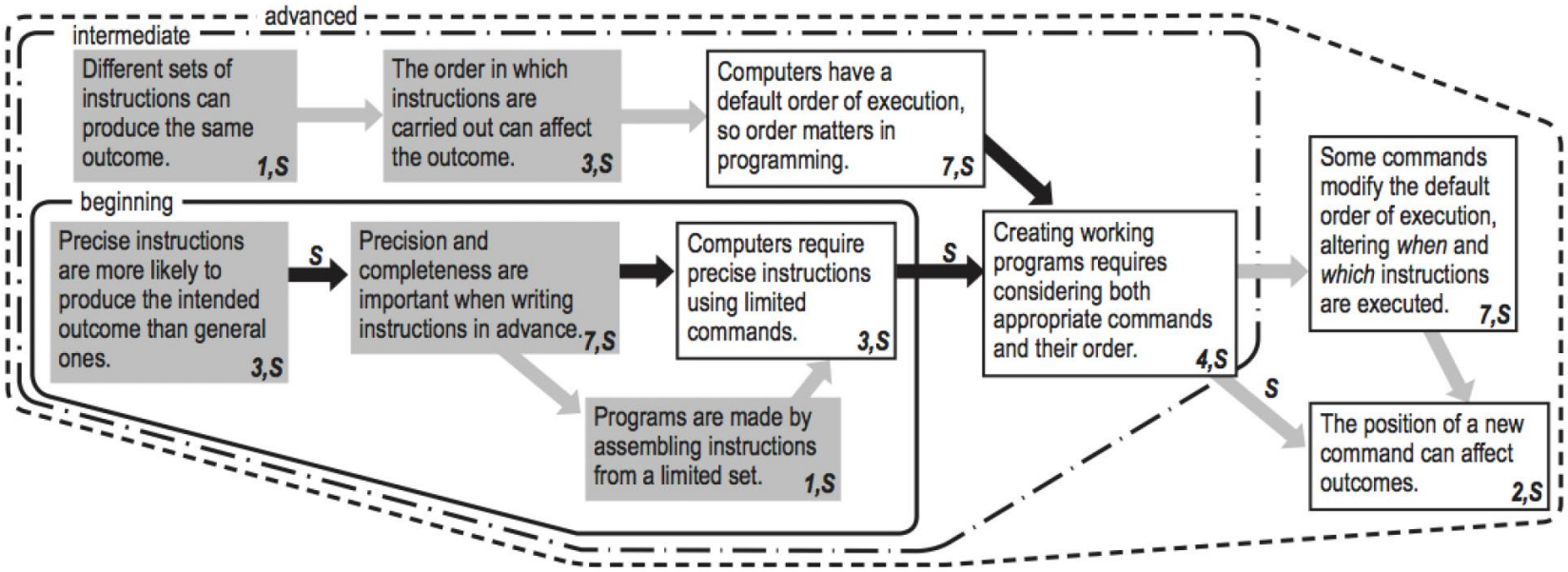

Figure 4. *Rich et al.'s (2018b) Learning Trajectory for Sequence (Grades K–8)*
Note. This figure is fully described in Appendix C.

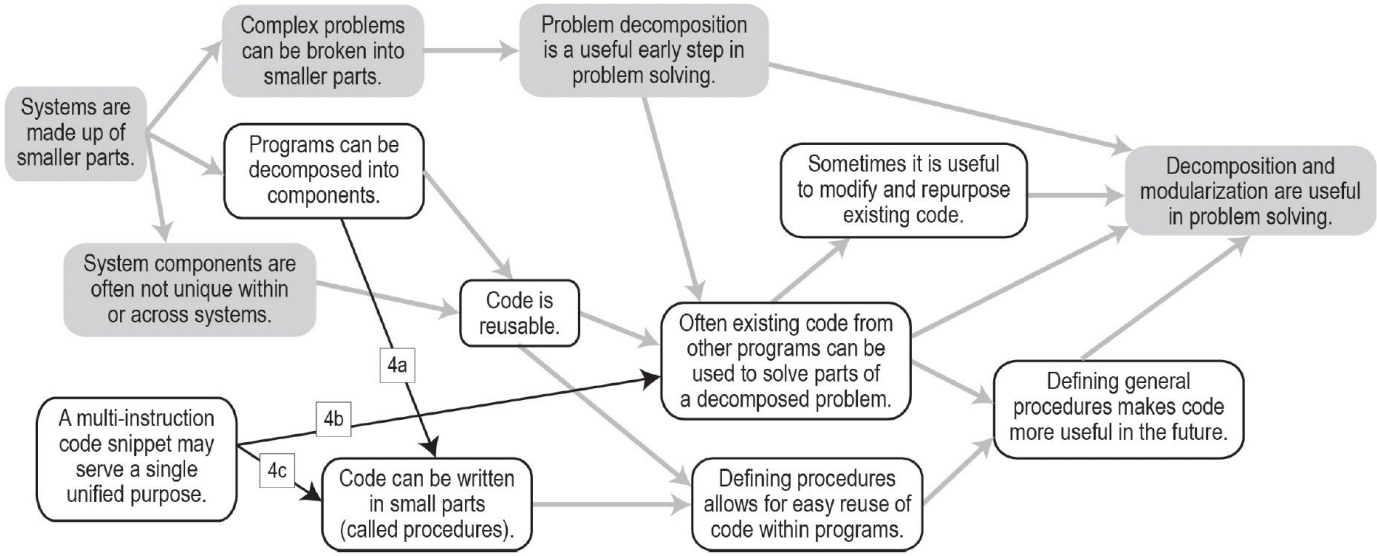

Figure 5. *Rich et al.'s (2018a) Learning Trajectory for Decomposition (Grades K–8)*
Note. This figure is fully described in Appendix D.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Luo et al. (2020) ([7]) drew on these learning trajectories to examine fourth grade students' understanding of control and modularity as they solved assessment items. Their findings include the following:

- **Conditionals:** No students displayed knowledge of the goal *"A condition is something that can be true or false,"* suggesting this needs to be explicitly taught.
- **Conditionals:** Unlike the direction presented in Rich et al.'s (2018b) ([10]) learning trajectory for conditionals (Figure 6), students understood the goal *"A conditional connects a condition to an outcome"* before they understood *"A condition is something that can be true or false."*
- **Repetition:** Students varied in the level of knowledge they reflected: some reflected the beginner level, some reflected the intermediate level, and some did not hold any of the ideas on the trajectory.
- **Sequence:** All students reflected the intermediate level of the learning trajectory.
- **Decomposition:** A subset of students displayed beginner-level knowledge, and a couple of students did not hold any of the ideas on the trajectory.

Program Development and Program Comprehension

In a study with 6-year-old students learning computational thinking while coding robots, Falloon (2024) ([3]) describes significant improvements in student understanding over seven lessons. Students got better at creating sequences and algorithms, fixing mistakes, and recognizing patterns. As the lessons continued, the students became more confident and accurate in writing code. They shifted from using single-block strategies to more efficient modular coding, which suggests an improved understanding of coding processes. Students also evolved in their error correction techniques, moving from restarting their code to editing it, indicating a deeper grasp of how to fix coding issues. However, the concept of efficiency seemed too abstract for some students, indicating there may be limits to what young learners can understand. Overall, the study results suggest that young children may be capable of more complex thinking than previously believed.

Several authors argue that program comprehension is just as important as program development for students' understanding of algorithms and programming (Izu et al., 2019 [6]). Drawing on research that examines code reading and code tracing as well as psychology research on worked examples, Griffin (2016) ([5]) puts forth a sample learning progression for deconstructing code (Table 4). This learning progression, which the author states is applicable to learners across all grade levels, includes reading, tracing, debugging, and writing. Notably, writing code from scratch, typically a major focus in programming education, is positioned at the end of this progression. Rich and colleagues' (2019) ([9]) literature review also informed a learning trajectory that provides detailed learning goals for debugging (Figure 6).

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Table 4. *Griffin's (2016) Learning Progression for Deconstructing Code (All Grade Levels)*

Timing	Skills	Activity	Evidence Statements
Early	Read & Trace	Explore	Students can read and trace well-written code. They can identify, label, or explain features, data, patterns, output, behaviors, etc. They can interact with a code visualizer or watch an instructional video. They can *remix* code by changing small sections of existing code and attempt to explain the consequences.
			Students perform categorization, matching, and choice activities to link terms and concepts with examples. Students compare code segments that accomplish the same task.
	Debug	Explain	Students who are shown the location of a bug can explain it (and/or fix it).
Middle	Read & Trace	Discern differences, dissonances	Students can discern differences, e.g., determine if two code segments perform the same tasks. Students can resolve dissonances, e.g., solve Parsons problems (by reordering misordered code).
Later	Write & Debug	Complete	Students complete supplied code. Sample progression:
			Fill in the blank(s). / Fix a bug (bug location provided).
			Find and fix a bug (bug location not provided).
			Complete skeleton code (where minimal code provided).
		Create	Students write code from scratch.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

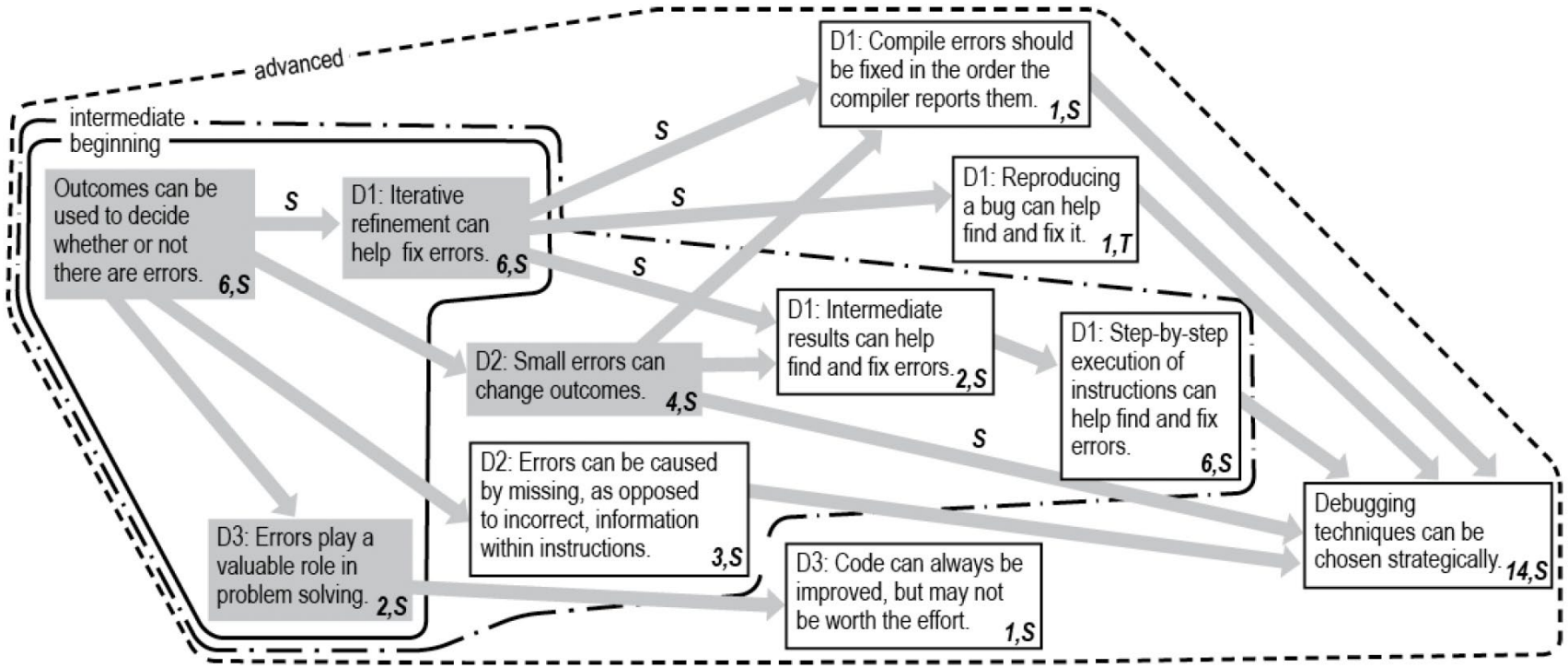

Figure 6. *Rich et al.'s (2019) Learning Trajectory for Debugging (Grades K–8)*
Note. This figure is fully described in Appendix E.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

References

Citation	Subconcept(s) Covered						Grade(s) Covered		
	Algorithms	Variables	Control	Modularity	Program Dev.	Other	K–5	6–8	9–12
[1] Bers, M. U. (2019). Coding as another language: A pedagogical approach for teaching computer science in early childhood. *Journal of Computers in Education, 6*(4), 499–528. https://doi.org/10.1007/s40692-019-00147-3	●	●	●	●	●	●	●		
[2] Dwyer, H., Hill, C., Carpenter, S., Harlow, D., & Franklin, D. (2014). Identifying elementary students' pre-instructional ability to develop algorithms and step-by-step instructions. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE) 2014* (pp. 511–516). Association for Computing Machinery. https://doi.org/10.1145/2538862.2538905	●		●				●		
[3] Falloon, G. (2024). Advancing young students' computational thinking: An investigation of structured curriculum in early years primary schooling. *Computers & Education, 216*, 105045. https://doi.org/10.1016/j.compedu.2024.105045			●		●		●		

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Citation	Subconcept(s) Covered						Grade(s) Covered		
	Algorithms	Variables	Control	Modularity	Program Dev.	Other	K–5	6–8	9–12
[4] Franklin, D., Hill, C., Dwyer, H. A., Hansen, A. K., Iveland, A., & Harlow, D. B. (2016). Initialization in Scratch: Seeking knowledge transfer. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education (SIGCSE) 2016* (pp. 217–222). Association for Computing Machinery. https://doi.org/10.1145/2839509.2844569		●					●	●	
[5] Griffin, J. M. (2016). Learning by taking apart: Deconstructing code by reading, tracing, and debugging. In *Proceedings of the 17th Annual Conference on Information Technology Education (SIGITE) 2016* (pp. 148–153). Association for Computing Machinery. https://doi.org/10.1145/2978192.2978231					●				
[6] Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., & Weeda, R. (2019). Fostering program comprehension in novice programmers—learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR) 2019* (pp. 27–52). Association for Computing Machinery. https://doi.org/10.1145/3344429.3372501						●			

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Citation	Subconcept(s) Covered						Grade(s) Covered		
	Algorithms	Variables	Control	Modularity	Program Dev.	Other	K–5	6–8	9–12
[7] Luo, F., Israel, M., Liu, R., Yan, W., Gane, B., & Hampton, J. (2020). Understanding students' computational thinking through cognitive interviews: A learning trajectory-based analysis. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE) 2020* (pp. 919–925). Association for Computing Machinery. https://doi.org/10.1145/3328778.3366845			●	●			●		
[8] Rich, K. M., Binkowski, T. A., Strickland, C., & Franklin, D. (2018a). Decomposition: A K–8 computational thinking learning trajectory. In *Proceedings of the 2018 ACM Conference on International Computing Education Research (ICER) 2018* (pp. 124–132). Association for Computing Machinery. https://doi.org/10.1145/3230977.3230979				●			●	●	
[9] Rich, K. M., Strickland, C., Binkowski, T. A., & Franklin, D. (2019). A K–8 debugging learning trajectory derived from research literature. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE) 2019* (pp. 745–751). Association for Computing Machinery. https://doi.org/10.1145/3287324.3287396					●		●	●	

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Citation	Subconcept(s) Covered						Grade(s) Covered		
	Algorithms	Variables	Control	Modularity	Program Dev.	Other	K–5	6–8	9–12
[10] Rich, K. M., Strickland, C., Binkowski, T. A., Moran, C., & Franklin, D. (2018b). K–8 learning trajectories derived from research literature: Sequence, repetition, conditionals. *ACM Inroads, 9*(1), 46–55. https://doi.org/10.1145/3183508			●				●	●	
[11] Zhang, L., & Nouri, J. (2019). A systematic review of learning computational thinking through Scratch in K–9. *Computers & Education, 141*, 103607. https://doi.org/10.1016/j.compedu.2019.103607	●	●	●	●	●		●	●	●

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Appendix A. Text Description of Figure 2

Rich et al.'s (2018b) Learning Trajectory for Conditionals (Grades K–8)

This learning trajectory map illustrates the relationships among learning goals related to conditionals. Goals are organized into three levels of advancement in understanding (**beginning**, **intermediate**, and **advanced**) and are connected by arrows that represent relationships between goals as either required (black arrows) or helpful (gray arrows). Goals are either in gray boxes to indicate unplugged learning or white boxes to indicate computer-based learning. Each goal is labeled with an identifier (e.g., "2,T" or "4,S") representing the number of ideas used to form the goal and the goal's strongest type of supporting evidence (S for student support, or T for theoretical support). Some arrows are also labeled with S or T to indicate that the relationship is supported by the literature.

Beginning-Level Learning Goals

1. **A condition is something that can be true or false. (3, S — Unplugged)**
   - Is required for knowing that:
     - » "A conditional connects a condition to an outcome."
     - » "A Boolean is a variable that can be true or false."
   - Is helpful for knowing that:
     - » "Sometimes multiple conditions must be considered."
2. **A conditional connects a condition to an outcome. (11, S — Unplugged)**
   - Is required for knowing that:
     - » "Each of the two states of a condition may have its own action." (This relationship is supported by theoretical evidence.)
     - » "Conditional statements are computer commands to evaluate conditions and complete connected actions."
3. **Actions often result from specific causes. (2, T — Unplugged)**
   - Is required for knowing that:
     - » "A conditional connects a condition to an outcome."
4. **Each of the two states of a condition may have its own action. (4, S — Unplugged)**
   - Is helpful for knowing that:
     - » "Conditional statements are computer commands to evaluate conditions and complete connected actions."
     - » "Computers require all actions to be specified."
5. **Conditional statements are computer commands to evaluate conditions and complete connected actions. (18, S — Computer-based)**
   - Is helpful for knowing that:
     - » "Conditional statements can create branches in the flow of execution."
     - » "Computers require all actions to be specified."

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Intermediate-Level Learning Goals

6. Sometimes multiple conditions must be considered. (3, S — Unplugged)

- Is helpful for knowing that:
  - » "Conditions can overlap, and more than one can apply."
  - » "Computers require all actions to be specified."

7. Conditions can overlap, and more than one can apply. (2, S — Unplugged)

- Is helpful for knowing that:
  - » "Computers require all actions to be specified."
  - » "Logical operators can be used to combine conditions."

8. Computers require all actions to be specified. (4, S — Computer-based)

- Is required for knowing that:
  - » "Conditional statements can be combined in several ways." (This relationship is supported by student evidence.)

9. Conditional statements can create branches in the flow of execution. (7, S — Computer-based)

- Is required for knowing that:
  - » "Conditional statements can be combined in several ways."

Advanced-Level Learning Goals

10. A Boolean is a variable that can be true or false. (4, S — Computer-based)

- Is helpful for knowing that:
  - » "Logical operators can be used to combine conditions."

11. Logical operators can be used to combine conditions. (5, S — Computer-based)

- Is required for knowing that:
  - » "Conditional statements can be combined in several ways." (This relationship is supported by student evidence.)

12. Conditional statements can be combined in several ways. (2, S — Computer-based)

- End point (no outgoing arrows)

See Figure 2. *Rich et al.'s (2018b) Learning Trajectory for Conditionals (Grades K–8)*.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Appendix B. Text Description of Figure 3

Rich et al.'s (2018b) Learning Trajectory for Repetition (Grades K–8)

This learning trajectory map illustrates the relationships among learning goals related to repetition. Goals are organized into three levels of advancement in understanding (**beginning**, **intermediate**, and **advanced**) and are connected by arrows that represent relationships between goals as either required (black arrows) or helpful (gray arrows). Goals are either in gray boxes to indicate unplugged learning or white boxes to indicate computer-based learning. Each goal is labeled with an identifier (e.g., "4,T" or "8,S") representing the number of ideas used to form the goal and the goal's strongest type of supporting evidence (S for student support, or T for theoretical support).

Beginning-Level Learning Goals

1. **Some tasks involve repeating actions. (4,T — Unplugged)**
   - Is required for knowing that:
     - » "Instructions like 'Step 3 times' do the same thing as 'Step, step, step.'"
     - » "Repetitions can go on forever or stop."
     - » "Repeating things can have a cumulative effect."
2. **Instructions like 'Step 3 times' do the same thing as 'Step, step, step.' (4,T — Unplugged)**
   - Is required for knowing that:
     - » "Computers use repeat commands."
3. **Computers use repeat commands. (19,S — Computer-based)**
   - Is required for knowing that:
     - » "Different kinds of repetition have different commands."
     - » "Loops can be nested to accomplish complex tasks."

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Intermediate-Level Learning Goals

4. **Repetitions can go on forever or stop. (5,S — Unplugged)**
   - Is required for knowing that:
     - » "Different kinds of tasks require different kinds of repetition."
5. **Repeating things can have a cumulative effect. (4,T — Unplugged)**
   - Is required for knowing that:
     - » "Different kinds of tasks require different kinds of repetition."
6. **Different kinds of tasks require different kinds of repetition. (7,S — Unplugged)**
   - Is required for knowing that:
     - » "Different kinds of repetition have different commands."
   - Is helpful for knowing that:
     - » "Programs use conditions to end loops."
7. **Different kinds of repetition have different commands. (8,S — Computer-based)**
   - End point (no outgoing arrows)

Advanced-Level Learning Goals

8. **Programs use conditions to end loops. (7,S — Computer-based)**
   - Is required for knowing that:
     - » Variables can be used to control the number of repetitions."
   - Is helpful for knowing that:
     - » "Different kinds of repetition have different commands."
9. **Variables can be used to control the number of repetitions. (1,S — Computer-based)**
   - Is helpful for knowing that:
     - » "Different kinds of repetition have different commands."
10. **Loops can be nested to accomplish complex tasks. (2,T — Computer-based)**
    - End point (no outgoing arrows)

See Figure 3. *Rich et al.'s* (2018b) *Learning Trajectory for Repetition (Grades K–8)*.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Appendix C. Text Description of Figure 4

Rich et al.'s (2018b) Learning Trajectory for Sequence (Grades K–8)

This learning trajectory map illustrates the relationships among learning goals related to sequence. Goals are organized into three levels of advancement in understanding (**beginning**, **intermediate**, and **advanced**) and are connected by arrows that represent relationships between goals as either required (black arrows) or helpful (gray arrows). Goals are either in gray boxes to indicate unplugged learning or white boxes to indicate computer-based learning. Each goal is labeled with an identifier (e.g., "3,S" or "4,S") representing the number of ideas used to form the goal and the goal's strongest type of supporting evidence (S for student support, or T for theoretical support). Some arrows are also labeled with S or T to indicate that the relationship is supported by the literature.

Beginning-Level Learning Goals

1. **Precise instructions are more likely to produce the intended outcome than general ones. (3,S — Unplugged)**
   - Is required for knowing that:
     - » "Precision and completeness are important when writing instructions in advance." (This relationship is supported by student evidence.)
2. **Precision and completeness are important when writing instructions in advance. (7,S — Unplugged)**
   - Is required for knowing that:
     - » "Computers require precise instructions using limited commands."
   - Is helpful for knowing that:
     - » "Programs are made by assembling instructions from a limited set."
3. **Programs are made by assembling instructions from a limited set. (1,S — Unplugged)**
   - Is helpful for knowing that:
     - » "Computers require precise instructions using limited commands."
4. **Computers require precise instructions using limited commands. (3,S — Computer-based)**
   - Is required for knowing:
     - » "Creating working programs requires considering both appropriate commands and their order." (This relationship is supported by student evidence.)

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Intermediate-Level Learning Goals

5. **Different sets of instructions can produce the same outcome. (1,S — Unplugged)**
   - Is helpful for knowing that:
     - » "The order in which instructions are carried out can affect the outcome."
6. **The order in which instructions are carried out can affect the outcome. (3,S — Unplugged)**
   - Is helpful for knowing that:
     - » "Computers have a default order of execution, so order matters in programming."
7. **Computers have a default order of execution, so order matters in programming. (7,S — Computer-based)**
   - Is required for knowing that:
     - » "Creating working programs requires considering both appropriate commands and their order."
8. **Creating working programs requires considering both appropriate commands and their order. (4,S — Computer-based)**
   - Is helpful for knowing that:
     - » "Some commands modify the default order of execution, altering *when* and *which* instructions are executed."
     - » "The position of a new command can affect outcomes." (This relationship is supported by student evidence.)

Advanced-Level Learning Goals

9. **Some commands modify the default order of execution, altering *when* and *which* instructions are executed. (7,S — Computer-based)**
   - Is helpful for knowing that:
     - » "The position of a new command can affect outcomes."
10. **The position of a new command can affect outcomes. (2,S — Computer-based)**
    - End point (no outgoing arrows)

See Figure 4. *Rich et al.'s (2018b) Learning Trajectory for Sequence (Grades K–8)*.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Appendix D. Text Description of Figure 5

Rich et al.'s (2018a) Learning Trajectory for Decomposition (Grades K–8)

This learning trajectory map illustrates the relationships among learning goals related to decomposition. Goals are connected by arrows that represent relationships between goals as either required (black arrows) or helpful (gray arrows). Goals are either in gray boxes to indicate unplugged learning or white boxes to indicate computer-based learning.

Learning Goals

1. **Systems are made up of smaller parts. (Unplugged)**
   - Is helpful for knowing that:
     - » "Complex problems can be broken into smaller parts."
     - » "Programs can be decomposed into components."
     - » "System components are often not unique within or across systems."
2. **Complex problems can be broken into smaller parts. (Unplugged)**
   - Is helpful for knowing that:
     - » "Problem decomposition is a useful early step in problem solving."
3. **Problem decomposition is a useful early step in problem solving. (Unplugged)**
   - Is helpful for knowing that:
     - » "Often existing code from other programs can be used to solve parts of a decomposed problem."
     - » "Decomposition and modularization are useful in problem solving."
4. **Programs can be decomposed into components. (Computer-based)**
   - Is required for knowing that:
     - » "Code can be written in small parts (called procedures)."
   - Is helpful for knowing that:
     - » "Code is reusable."
5. **System components are often not unique within or across systems. (Unplugged)**
   - Is helpful for knowing that:
     - » "Code is reusable."

(continued)

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

6. **Code is reusable. (Computer-based)**
   - Is helpful for knowing that:
     - » "Often existing code from other programs can be used to solve parts of a decomposed problem."
     - » "Defining procedures allows for easy reuse of code within programs."
7. **Code can be written in small parts (called procedures). (Computer-based)**
   - Is helpful for knowing that:
     - » "Defining procedures allows for easy reuse of code within programs."
8. **A multi-instruction code snippet may serve a single unified purpose. (Computer-based)**
   - Is required for knowing that:
     - » "Often existing code from other programs can be used to solve parts of a decomposed problem."
     - » "Code can be written in small parts (called procedures)."
9. **Often existing code from other programs can be used to solve parts of a decomposed problem. (Computer-based)**
   - Is helpful for knowing that:
     - » "Sometimes it is useful to modify and repurpose existing code."
     - » "Defining general procedures makes code more useful in the future."
     - » "Decomposition and modularization are useful in problem solving."
10. **Sometimes it is useful to modify and repurpose existing code. (Computer-based)**
    - Is helpful for knowing that:
      - » "Decomposition and modularization are useful in problem solving."
11. **Defining procedures allows for easy reuse of code within programs. (Computer-based)**
    - Is helpful for knowing that:
      - » "Defining general procedures makes code more useful in the future."
12. **Defining general procedures makes code more useful in the future. (Computer-based)**
    - Is helpful for knowing that:
      - » "Decomposition and modularization are useful in problem solving."
13. **Decomposition and modularization are useful in problem solving. (Unplugged)**
    - End point (no outgoing arrows)

See Figure 5. *Rich et al.'s (2018a) Learning Trajectory for Decomposition (Grades K–8)*.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Appendix E. Text Description of Figure 6

Rich et al.'s (2019) Learning Trajectory for Debugging (Grades K–8)

This learning trajectory map illustrates the relationships among learning goals related to debugging. Goals are organized into three levels of advancement in understanding (**beginning**, **intermediate**, and **advanced**) and are connected by arrows that represent relationships between goals. Goals are either in gray boxes to indicate unplugged learning or white boxes to indicate computer-based learning. Some goals are grouped into one of three dimensions and prefaced by D1 (strategies for finding and fixing errors), D2 (types of errors), or D3 (the role of errors in problem solving). Each goal is labeled with an identifier (e.g., "1,T" or "6,S") representing the number of ideas used to form the goal and the goal's strongest type of supporting evidence (S for student support, or T for theoretical support). Some arrows are also labeled with S or T to indicate that the relationship is supported by the literature.

Beginning-Level Learning Goals

1. **Outcomes can be used to decide whether or not there are errors. (6,S — Unplugged)**
   - Is helpful for knowing that:
     - » "D1: Iterative refinement can help fix errors." (This relationship is supported by student evidence.)
     - » "D2: Small errors can change outcomes."
     - » "D2: Errors can be caused by missing, as opposed to incorrect, information within instructions."
     - » "D3: Errors play a valuable role in problem solving."
2. **D3: Errors play a valuable role in problem solving. (2,S — Unplugged)**
   - Is helpful for knowing that:
     - » "D3: Code can always be improved, but may not be worth the effort."
3. **D1: Iterative refinement can help fix errors. (6,S — Unplugged)**
   - Is helpful for knowing that:
     - » "D1: Compile errors should be fixed in the order the compiler reports them." (This relationship is supported by student evidence.)
     - » "D1: Reproducing a bug can help find and fix it." (This relationship is supported by student evidence.)
     - » "D1: Intermediate results can help find and fix errors." (This relationship is supported by student evidence.)

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0

Intermediate-Level Learning Goals

4. **D2: Small errors can change outcomes. (4,S — Unplugged)**
   - Is helpful for knowing that:
     - » "D1: Compile errors should be fixed in the order they compiler reports them."
     - » "D1: Intermediate results can help find and fix errors."
     - » "Debugging techniques can be chosen strategically." (This relationship is supported by student evidence.)
5. **D2: Errors can be caused by missing, as opposed to incorrect, information within instructions. (3,S — Computer-based)**
   - Is helpful for knowing that:
     - » "Debugging techniques can be chosen strategically."
6. **D1: Intermediate results can help find and fix errors. (2,S — Computer-based)**
   - Is helpful for knowing that:
     - » "D1: Step-by-step execution of instructions can help find and fix errors."
7. **D1: Step-by-step execution of instructions can help find and fix errors. (6,S — Computer-based)**
   - Is helpful for knowing that:
     - » "Debugging techniques can be chosen strategically."

Advanced-Level Learning Goals

8. **D1: Compile errors should be fixed in the order the compiler reports them. (1,S — Computer-based)**
   - Is helpful for knowing that:
     - » "Debugging techniques can be chosen strategically."
9. **D1: Reproducing a bug can help find and fix it. (1,T — Computer-based)**
   - Is helpful for knowing that:
     - » "Debugging techniques can be chosen strategically."
10. **D3: Code can always be improved, but may not be worth the effort. (1,S — Computer-based))**
    - End point (no outgoing arrows)
11. **Debugging techniques can be chosen strategically. (14,S — Computer-based)**
    - End point (no outgoing arrows)

See Figure 6. *Rich et al.'s (2019) Learning Trajectory for Debugging (Grades K–8)*.

© 2026 Computer Science Teachers Association (CSTA). | BY-NC-SA 4.0