Teaching Scenario-Based Programming: An Additional Paradigm for the High School Computer Science Curriculum, Part 2

Giora Alexandron and Michal Armoni | Weizmann Institute of Science
Michal Gordon | Holon Institute of Technology
David Harel | Weizmann Institute of Science

Computer science (CS) education at the high school level has received increasing attention in recent years. Although high school programs vary considerably among different countries, it's commonly agreed by CS educators that high school programs, like undergraduate ones, should teach fundamental CS ideas and present CS as a science (of computing), rather than as a technical subject that's mainly about programming (www.computingatschool.org.uk/data/uploads/internationalcomparisons-v5.pdf). An example of a high school curriculum that realized such principles is the seminal work of Judith Gal-Ezer and colleagues,[1] who described the implementation of this program's essentials in an Israeli CS high school course that has been in use, with routine updates, for almost two decades.

One of the underlying principles of this curriculum is that students should be exposed to more than one programming paradigm (a principle referred to as the second programming paradigm). This article describes a course that implements that principle. Here are the core issues upon which the course is based (a detailed description can be found in the previous issue of this magazine[2]):

- *A second programming paradigm*. Gal-Ezer and colleagues suggested that students should be introduced, in addition to the main language that they learn (in Israel, this is currently Java or C#), to "another language, of radically different nature, that suggests alternative ways of algorithmic thinking. This emphasizes the fact that algorithmics is the central subject of study."[1] Live sequence charts (LSC) is a language that's "radically different" from the conventional languages used in the introductory courses. Besides being declarative and high-level, it's also a visual language. These characteristics render LSC a very interesting choice for the second paradigm.
- *Nondeterminism (ND)*. The topic of ND usually isn't included in high school programs,[3] but as a fundamental idea of CS,[4] we believe it should be included. ND is also one of the essential characteristics of concurrency, which is a central issue in modern computing. It is typically introduced through nondeterministic finite automata; however, several studies have indicated that the kind of ND that appears in automata theory is hard to teach and learn.[5] As LSC is a nondeterministic programming language, teaching it inherently involves teaching ND but of the kind that appears in nondeterministic and concurrent programming, not that of automata. We term this kind of ND *operative ND*. At its core lies the idea of "true don't care," which means that there's a priori no preference of which possible continuation of the computation to follow, all of them being equally good.
- *System design and abstract thinking*. Abstraction is a fundamental idea of CS,[6] and introducing students to system design and developing abstract thinking skills are primary objectives of the advanced programming module of the Israeli high school program. As LSC is a high-level, declarative programming language, its learning naturally supplies opportunities for dealing with system design and abstraction.

Course Structure and Setting

The methodology of the course followed two principles: the zipper principle[1] and project-based learning. The former means that theoretical lectures are interwoven with hands-on experience in the lab, in which students exercise the learned concepts on a small scale and in a controlled setting. The first half of the course was arranged according to this principle. The second half was project-based, with students working in small teams, each implementing a project in LSC.

The course was given to a standard class of 19 12th-grade high school students majoring in CS. Their previous experience included mainly programming courses in (180 hours in total). Our course was 45 hours long, and it replaced the second half of a 90-hour course on computational models. The assessment was based on pen-and-paper exams and projects. It served two very different purposes: grading the students, as required for every high school course, and evaluating the course. In the exams, the students were required to create, modify, and comprehend LSC systems.

The course was given to high school students, but we believe the results are also applicable to achieving similar goals in the context of undergraduates. An undergraduate course can delve deeper, for example, by comparing different paradigms and connecting the learned subjects to advanced topics such as synthesis. A description of an advanced course for graduate students on visual languages, which included LSC, can be found elsewhere.[7]

Live Sequence Charts

The language of LSC was originally introduced[8] as an extension of message sequence charts and was later extended significantly[9,10] as a visual programming language for reactive system development. It's supported by the Play-Engine[9] development environment, which we used in the course, and the later tool, PlayGo,[11] which is a more mature environment. LSC introduces a new paradigm, called scenario-based programming. In the abstract sense, a scenario describes a series of actions that constitute a certain system functionality. Collectively, these scenarios define system behavior.

The Play-Out Method

LSC has an operational semantics that defines how the execution engine should combine all the scenarios. This operational semantics is implemented by the play-out method.[9,10] Play-out makes the specification defined by the collection of scenarios directly executable/simulatable, thus making LSC a high-level programming language (other than merely a specification language).

Other findings obtained for graduate students strengthened the connection between abstract thinking and working in LSC and scenario-based programming.

The Play-In Method

LSC is supplemented with a method for building the scenario-based specification over a real or a mock-up GUI of the system called the play-in method.[9,10,12] With play-in, users specify the scenarios in a way that's close to how real interaction with the system occurs.

Course Evaluation

Assessment of the course as a pilot was based on evaluating the pedagogic outcomes that were achieved. This evaluation involves four dimensions: the effect of learning LSC on the use of abstraction, the effect of learning LSC on students' understanding of nondeterminism, meaningful learning of LSC and scenario-based programming, and students' attitudes toward the language and the course.

Developing Abstract Thinking

In previous work,[13] we presented findings suggesting that learning LSC and scenario-based programming scaffolds the development of abstract thinking. In particular, based on an analysis of final projects and interviews with a sample of students, we showed that the students exhibited a high level of abstraction. They worked with black boxes, used symbolic instances (a feature of LSC that makes it possible to define a common behavior for sets of objects belonging to the same class), were able to move between levels of abstraction, and demonstrated metacognitive processes.

Other findings obtained for graduate students strengthened the connection between abstract thinking and working in LSC and scenario-based programming. These findings can be grouped into the following two categories.

Functionality first, integration later. The high-level, declarative, and incremental nature of LSC led graduate students to adopt a kind of programming style in which the order of concerns is different than in object-oriented programming (OOP). In LSC, when adding new functionality to the system under development, the students first concentrated on the new functionality and its implementation, only later considering integration with the rest of the system (we referred to this as "functionality first, integration later"). When working with object-oriented languages, these concerns were executed in reverse order. First, the students considered the integration of the new functionality with the rest of the system (the objects), and only later did they turn to implementing it (referred to as "integration first, functionality later"). In a way, this difference can be thought of as bottom-up programming (with LSC) versus top-down programming (with OOP). Note that by bottom-up programming we aren't referring to a Bricolage.[14] Interestingly, a study of novices working with Scratch found that the students decomposed the system in a "scenario-based" way without any guidance to do so, and used this as evidence that scenario-based, bottom-up decomposition is cognitively easier than top-down decomposition.[15] While this argument requires further study, these studies highlight a substantial way in which LSC leads programmers to practice a different way of programming problem-solving.

Holding a less detailed mental model. The findings classified under this category illustrate how the incremental, declarative programming style that LSC fosters allowed the graduate students to hold a less detailed mental model of the program they developed. We believe that this decreases the working memory load involved in adding a new feature to the system, thus reducing the total cognitive load associated with this task.

Overall, our findings implied the following connection: LSC leads programmers to adopt a kind of programming that requires holding a less detailed mental model of the system. Working with a less detailed mental model allows diverting cognitive resources to high-level tasks, which scaffolds high-level thinking. To some extent, this argument might be applicable to other languages that support incremental development (declarative languages such as Prolog or aspect-oriented programming [AOP] languages such as AspectJ), but this requires further study.

We note that working with a less detailed mental model might interfere with programming tasks

We believe that our relatively encouraging findings on students' ability to create nondeterministic programs are strongly related to the kind of ND that was taught and presented.

such as debugging, which requires working on a lower level of abstraction to understand the lower-level details. Interestingly, this serves as another opportunity to use this course as a platform for discussing higher-level concepts. In LSC, this issue can be addressed in part by a feature called forbidden scenarios, which lets us correct several kinds of program errors (bugs) in a nonintrusive way, by forbidding the scenario that leads to them. In addition to the practical matter of correcting the program, this feature exposes students to a different bug-fixing strategy.

Learning Nondeterminism

ND is inherent to LSC. In a previous work,[3] we used the term operative ND to denote the kind of ND that appears in concurrent and nondeterministic programming. The main difference between operative ND and the kind of ND that appears in automata theory is that the former has universal semantics, whereas the latter has existential semantics (a detailed review of the historical development of the concept of ND in computer science, which also refers to the way that different computer scientist captured the difference between the two types of ND, can be found elsewhere[4]). Thus, this course offers an opportunity to expose the students to another facet of ND, hence to a wider and more general perspective of it, in line with Schwill's vertical characteristic of fundamental ideas. The findings that we presented suggested that learning LSC and scenario-based programming promotes the understanding of operative ND. These findings also show that after learning LSC, high school students were capable of understanding ND on a level that allowed them to mentally simulate systems that included nondeterministic and concurrent behavior, and create systems that included nondeterministic and concurrent characteristics.

The part of the findings that dealt with the creation of ND was the most interesting, first, because it indicated that a high level of learning was achieved (in Bloom's revised taxonomy,[16] creating is considered the highest level of learning), and second, because this result contrasted the findings of several studies that dealt with the learning of nondeterministic automata and reported that students had special difficulties with creating this kind of ND.[17,18]

We believe that our relatively encouraging findings on students' ability to create nondeterministic programs are strongly related to the kind of ND that was taught and presented. Regarding the kind of ND, we believe that operative ND is much more intuitive and easy to understand than the kind of ND that appears in automata theory. Regarding the way that it's presented, this refers mainly to two issues: the project-based learning approach that we took, the advantages of which we already discussed, and the fact that in LSC, ND is an intrinsic part of the language and appears in multiple ways and in different levels of complexity. The fact that ND is present from the beginning helps students get used to it and allows a gradual learning curve.

We found this to be exemplified in the coffee-machine project. In an interview held with the representative of that group, the student referred to the fact that several charts in the project can run simultaneously, yielding a nondeterministic behavior. The student said that the group programmed each of the charts independently, and that they were aware of the fact that there are several possible executions. To verify, the student was requested to enumerate the possible execution paths. She responded promptly, indicating that she already possessed this knowledge (at least partially), demonstrating how the fact that ND is integral to LSC facilitated its use. Among other things, we believe this scaffolds more advanced, deliberated use of the concept. The default nature of ND in LSC is in contrast to the way ND appears in automata theory, as an extension to the basic model, of which it isn't an intrinsic part. This can yield both cognitive and attitudinal issues with regard to the concept. For example, it can explain, at least partially, findings reporting that students learning automata theory perceived ND as a non-legitimate solution and thus abstained from using it.

To conclude, we believe that LSC yields a learning environment that scaffolds the learning and use of operative ND. It's a question for further study whether learning this kind of ND has a positive influence on the learning of ND in automata theory.

Table 1. Students' grades.

Exam	Success (%)	Std	*N*
Midterm	89	10	17
Final	92	12	19
Matriculation	90	6	19

Learning LSC and Scenario-Based Programming

A main objective of the pilot phase was to examine whether the 45-hour course that we developed is enough to achieve meaningful learning of the language and its concepts. To measure this, we evaluated whether the students reached a satisfactory level of understanding of the course topics. The evaluation was based on the results of exams and projects. During the course, students were subjected to three exams (midterm, final, and matriculation) that covered the main concepts of the language, as defined by the syllabus. Each concept was measured by several items, measuring various levels of complexity. The operationalization of complexity level was based on Bloom's taxonomy.

The grading process was conducted as follows. First, a grading scheme that we all agreed on was developed and then tested on a sample of five test forms before being modified to include new kinds of errors. Then, the modified scheme was used to grade all the forms. In the midterm and final exams, the grading (in both the testing and final grading phases) was conducted by the first author. In the matriculation exam, the grading (in both phases) was conducted independently by the first and third authors, with the grade taken to be the mean of the score given by the two graders. The intention was to discuss and resolve substantial differences (defined as 10 points or more) between the two graders in both phases, but since no such differences were found, there was no need for such cross-rate agreement phase.

Table 1 shows the results of the exams. They indicate that the students reached a satisfactory level of understanding with respect to the measures defined by the pedagogic and research team. While exams yield a standard, less subjective measurement, the kind of knowledge that they can measure is limited. The projects complete the picture, capturing another aspect of the learning by illuminating students' ability to apply their knowledge in a real-world context, taking a significant programming task and dealing with all aspects of it. Their evaluation was qualitative, mainly aiming at getting an overall picture of the level of projects that can be created by a typical CS high school class in such a time frame, to identify major difficulties, and so on.

As described in the last issue,[2] out of five groups, four completed their projects and submitted them; one group didn't submit its project due to issues unrelated to the course. The quality of the projects varied, but all of them reached a satisfactory level. By satisfactory, we mean that the students took a core functionality of the system, formalized it as requirements, and implemented these requirements. For example, the core functionality of the Simon project was randomizing a sequence of colors, asking the user to repeat it, and verifying that the user sequence matches the original one.

The projects were also used to evaluate the learning of concepts that weren't included in the exams. One of these concepts was forbidden scenarios, a novel programming concept that LSC introduces that's used mainly for capturing safety properties and that doesn't exist in conventional languages. Thus, we were interested to see how students adopt and use it. Such usage was demonstrated by the project of the group that modeled an elevator, in which forbidden scenarios were used to represent safety requirements. The idea of using forbidden scenarios for this specific purpose came from the students after this concept was very briefly mentioned by the teacher to prepare the ground for delving into it later. Another example of using an advanced concept that wasn't included in the exam was found in the project of the group that modeled the Simon memory game. This group used symbolic elements in its existential semantic mode to achieve randomization (the course covered symbolic instances in their universal semantic mode). This was a very sophisticated use, and in fact, the students got it so well that they even found a bug in the implementation of this feature in the development environment.

Another aspect was the kind of systems and software engineering concepts the students dealt with. The high-level, declarative nature of LSC allowed the students to focus on the high-level behavior of the systems they developed. This enabled them to create interesting systems and to experience dealing with issues related to specification, requirements, and usability. In a previous work,[19] we studied the effect of working with LSC on graduate students' attitudes toward usability issues. Specifically, we analyzed the influence of LSC on the way programmers perceive their role with respect to the system they develop. Our experience with the pilot high school course provided collaborative evidence—for example, in an interview with a student who represented the coffee-machine group, she said that when programming the machine she perceived herself as a user more than as a programmer. However, when asked the same question regarding the project in the Assembly course, her answer then was that she felt more like a programmer.

Attitudes

Revealing students' attitudes served two different yet complementary goals. One was evaluating the course as a pilot and the other was research-oriented, revealing attitudinal issues involved in learning the language. For the former, the focus is the course, whereas for the latter, the focus is the language and its learning. In both aspects, our approach was exploratory in nature, and the focus was mainly on revealing attitudes and classifying them according to various dimensions (positive/negative sentiment is an example of such an important dimension). The findings were based mainly on a qualitative analysis of the postinterviews. The criterion for selecting the students to be interviewed was ensuring one student per project team (total of four students). Within the team, the representative student was picked based on availability (during the 12th-grade final exams period, students have many constraints, so availability became a primary concern).

The findings presented here relied on students' response to both direct, reflective questions and to other parts of the interview that didn't deal directly with attitudes but either revealed interesting attitudes or backed up attitudes that were reported by the students.

In the first part of the interview, students were asked about their attitude regarding the course and the language: "What did you like/dislike about the course?" "What was easy/hard for you?" "What was missing in the course?" "What did you like/dislike about LSC?" "What did you find easy/difficult in LSC?" Of course, subjective feelings should be taken with a grain of salt, but they're valuable when combined with other sources of data, yielding what's called in qualitative research a triangulation of the results. Another issue that should be considered is students' previous programming experience, which they used as a reference point when asked about the course and the language. This experience included mainly Java (3 units, total of 270 hours) and Assembly (1 unit, total of 90 hours). The teaching method of the Java courses was based mainly on the zipper principle, and the teaching method of the Assembly course combined the zipper principle with project-based learning. Below we describe attitudinal categories emerging from the interviews that were found to be significant and were triangulated from several viewpoints.

Visuality. This aspect was mentioned very often as a motivating factor and as something that's very different from the previous languages that the students studied. Indeed, Marian Petre[20] mentioned attractiveness as a prominent characteristic of visual languages. On the other hand, some students complained that it takes a lot of time to draw the charts and that even fixing a small error requires erasing and starting from scratch. This was raised even though we allowed students to make changes on their drawings. However, some students (the issue was raised only by female students) couldn't tolerate the idea of submitting a drawing that isn't neat and clear. One solution was allowing students to use pencils in the matriculation exam, though typically this isn't allowed.

The play-in method. The through-interface programming that the Play-Engine offers was mentioned by students as engaging, and they said it makes programming sort of "playing."

An experimental language. Another issue that students mentioned as intriguing was the fact that the language is in its experimental phase, meaning, according to them, that they're on the cutting edge, maybe gaining some advantage for the future and having the opportunity to affect research (they were told that their feedback will be used to improve the language).

Project-based learning. This was another factor that came up quite often. For example, in the interview, one student said that she liked projects because they allowed her to be creative, which was motivational. In this context, an interesting connection between project-based learning and high-level programming was made by the student who represented the group that modeled the elevator. The student compared the project that he developed in the LSC course with the project that he developed in the Assembly course, and mentioned that the kind of things that LSC deals with (high-, system-level programming) allows him to create more interesting and complex projects. An external indication of the projects as a motivational factor was the fact that two of the groups continued to work on their projects even after the course ended, though they knew that this work wasn't going to be graded.

High-level abstraction. One of the things that was very interesting for us was comparing the high school students' attitudes toward high-level abstraction with attitudes found in the course given to graduate students. In a previous work,[21] we reported that some of the graduate students who had significant experience with low-level languages referred to the high-level, visual programming in LSC as "not really programming," and that in some cases this even led to a negative attitude that affected performance. Such an attitude toward high-level programming wasn't observed in the high school course. We believe that, among other things, it indicates that less experienced students can be more ready to accept new programming concepts. We believe that one way to maintain this desired flexibility is by introducing students to different programming approaches on early stages.

The results of the course indicate that high school students can reach a significant understanding of LSC. Also, students find the visual nature of the language, the through-interface programming (play-in), and the kind of systems that it allows them to develop attractive and motivating. We believe that this renders LSC a good choice for teaching a second programming paradigm, a central pedagogic principle that underlies the CS high school curriculum suggested earlier.[1]

Furthermore, learning the language fosters the development of abstract thinking and the understanding of ND, making it an effective platform for teaching these fundamental CS ideas in a way that follows Jerome Bruner's philosophy of teaching fundamental ideas in a spiral, scaffolded manner.[22]

A new course that introduces a whole new technology obviously presents a substantial overload. Our course is no exception. Some may argue that such an overload (especially on the K–12 level) outweighs the additional knowledge gained by students, which in this particular case includes a new language, a new programming paradigm, and a corresponding technologic platform supporting the two. However, this is a narrow—though not uncommon—view of the kind of knowledge taught by a specific course, and hence of curricular goals. A curriculum design that implements such a point of view is typically organized by units of knowledge that constitute a set of learning objectives. A development process that corresponds to such a curriculum involves the definition of an appropriate set of knowledge units and a list of courses covering the set, such that the courses form a partition thereof, that is, each unit is covered by one course. This approach can be exemplified by early CS curricula.[23,24]

We argue that a more appropriate perspective is to organize any curriculum, including a K–12 one, around habits of mind (as suggested, for example, by Al Cuoco[25]) and fundamental ideas (as suggested by Bruner). By definition, a set of courses corresponding to such a curriculum is not a partition of the set of knowledge units. Rather, this set can be represented by a multilayer structure, in which the units on the higher level are conceptual units, representing ideas. A unit of this sort is necessarily present in several courses, as derived from its horizontal characteristic. It's also present in multiple levels of difficulty and age levels, as derived from its vertical characteristic. From this point of view, the new context isn't a disadvantage. On the contrary, it serves as an opportunity to expose students to another facet of the idea in a different context. This exposure promotes the perception of such an idea in its general sense, which can lead to a meaningful learning of it and to non-specific transfer. Non-specific transfer was considered by Bruner as the ultimate goal of a learning process. Thus, achieving non-specific transfer of a fundamental idea is certainly a goal that justifies such an overload. ■

Acknowledgments

We would like to thank Avi Cohen, Ronit Ben-Bassat Levy, Nir Eitan, and Zehava Levin for their help in conducting this research. This research was partially supported by an Advanced Research Grant to DH from the European Research Council (ERC) under the European Community's FP7 Programme and by the Israel Science Foundation. The work of the first author was supported by a grant from the Azrieli Foundation.

References

1. J. Gal-Ezer et al., "A High School Program in Computer Science," *Computer*, vol. 28, no. 10, 1995, pp. 73–80.

2. G. Alexandron et al., “Teaching Scenario-Based Programming: An Additional Paradigm for the High School Computer Science Curriculum, Part 1,” *Computing in Science & Eng.*, vol. 19, no. 5, 2017, pp. 58–67.
3. G. Alexandron et al., “On Teaching Programming with Nondeterminism,” *Proc. 8th Workshop Primary and Secondary Computing Education*, 2013, pp. 71–74.
4. M. Armoni and M. Ben-Ari, “The Concept of Nondeterminism: Its Development and Implications for Teaching,” *SIGCSE Bull.*, vol. 41, no. 2, 2009, pp. 141–160.
5. M. Armoni, N. Lewenstein, and M. Ben-Ari, “Teaching Students to Think Nondeterministically,” *SIGCSE Bull.*, vol. 40, no. 1, 2008, pp. 4–8.
6. M. Armoni, “On Teaching Abstraction in CS to Novices,” *J. Computers in Mathematics and Science Teaching*, vol. 32, no. 3, 2013, pp. 265–284.
7. D. Harel and M. Gordon-Kiwkowitz, “On Teaching Visual Formalisms, *IEEE Software*, vol. 26, no. 3, 2009, pp. 87–95.
8. W. Damm and D. Harel, “LSCs: Breathing Life into Message Sequence Charts,” *Formal Methods System Design*, vol. 19, no. 1, 2001, pp. 45–80.
9. D. Harel and R. Marelly, *Come, Let’s Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.
10. D. Harel and R. Marelly, “Specifying and Executing Behavioral Requirements: The Play-In/Playout Approach,” *Software and Systems Modeling*, vol. 2, no. 2, 2003, pp. 82–107.
11. D. Harel et al., “PlayGo: Towards a Comprehensive Tool for Scenario Based Programming,” *Proc. IEEE/ACM Int’l Conf. Automated Software Eng.*, 2010, pp. 359–360.
12. D. Harel, “From Play-In Scenarios to Code: An Achievable Dream,” *Fundamental Approaches to Software Engineering*, LNCS 1783, T. Maibaum, ed., Springer, 2000, pp. 22–34.
13. G. Alexandron et al., “Scenario-based Programming: Reducing the Cognitive Load, Fostering Abstract Thinking,” *Proc. 36th Int’l Conf. Software Eng.*, 2014, pp. 311–320.
14. M. Ben-Ari, “Constructivism in Computer Science Education,” *J. Computers in Mathematics and Science Teaching*, vol. 20, no. 1, 2001, pp. 45–73.
15. M. Gordon, A. Marron, and O. Meerbaum-Salant, “Spaghetti for the Main Course? Observations on the Naturalness of Scenario-Based Programming,” *Proc. 17th ACM Ann. Conf. Innovation and Technology in Computer Science Education*, 2012.
16. L.W. Anderson, D.R. Krathwohl, and B.S. Bloom, *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom’s Taxonomy of Educational Objectives*, Longman, 2001.
17. M. Armoni and J. Gal-Ezer, “On the Achievements of High School Students Studying Computational Models,” *SIGCSE Bulletin*, vol. 36, no. 3, 2004, pp. 17–21.
18. M. Armoni and J. Gal-Ezer, “Non-Determinism: An Abstract Concept in Computer Science Studies,” *Computer Science Education*, vol. 17, no. 4, 2007, pp. 243–262.
19. G. Alexandron, M. Armoni, and D. Harel, “Programming with the User in Mind,” *Proc. Psychology of Programming Interest Group Annual Conf.*, 2011; www.ppig.org/papers/23/20%20Alexandron.pdf.
20. M. Petre, “Why Looking Isn’t Always Seeing: Readership Skills and Graphical Programming,” *Comm. ACM*, vol. 38, no. 6, 1995, pp. 33–44.
21. G. Alexandron et al., “The Effect of Previous Programming Experience on the Learning of Scenario-Based Programming,” *Proc. 12th Koli Calling Int’l Conf. Computing Education Research*, 2012, pp. 151–159.
22. J.S. Bruner, *The Process of Education*, Harvard Univ. Press, 1960.
23. W.F. Atchison et al., “Curriculum 68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science,” *Comm. ACM*, vol. 11, no. 3, 1968, pp. 151–197.
24. R.H. Austing et al., “Curriculum ’78: Recommendations for the Undergraduate Program in Computer Science: A Report of the ACM Curriculum Committee on Computer Science,” *Comm. ACM*, vol. 22, no. 3, 1979, pp. 147–166.
25. A. Cuoco, E.P. Goldenberg, and J. Mark, “Habits of Mind: An Organizing Principle for Mathematics Curricula,” *J. Mathematical Behavior*, vol. 15, no. 4, 1996, pp. 375–402.

Giora Alexandron is a principal data scientist at the Center for Educational Technology. Contact him at Gioraa@cet.ac.il.

Michal Armoni is a senior scientist at the Weizmann Institute of Science. Contact her at michal.armoni@weizmann.ac.il.

Michal Gordon is a senior lecturer at Holon Institute of Technology. Contact her at michalgor@hit.ac.il.

David Harel is a professor at the Weizmann Institute of Science and vice president of the Israel Academy of Sciences and Humanities. Contact him at dharel@weizmann.ac.il.