Editors: Rubin Landau, rubin@physics.oregonstate.edu | Scott Lathrop, lathrop@illinois.edu

Teaching Scenario-Based Programming: An Additional Paradigm for the High School Computer Science Curriculum, Part 1

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

This article describes a pilot programming course in which high school students were introduced, through the visual programming language of live sequence charts (LSC),[1] to a new paradigm termed scenario-based programming.[2] The rationale underlying this course was teaching high school students a "second," very different programming paradigm.[3] Using LSC for this purpose has other advantages, such as exposing students to high-level programming, dealing with nondeterminism and concurrency, and referring to human-computer interaction (HCI) issues.

Here, we describe in detail a course that realizes this rationale and demonstrates that high school students can successfully learn the principles of scenario-based programming and deal with advanced topics such as nondeterminism and visual programming. This work also contributes to the discussion about guiding principles for curriculum development by highlighting an important principle: the educational objective of a course should include more than mere knowledge enhancement. It should be examined and justified through its contribution to learning fundamental ideas and forming useful habits of mind. This article is divided

 1521-9615/17/$33.00 © 2017 IEEE Copublished by the IEEE CS and the AIP September/October 2017

The rich set of associations between concepts learned in different contexts leads to generalizations and allows the learned concepts to be retrieved and applied outside the specific context in which they were originally introduced.

into two parts that are included in two consecutive issues. Part 1 describes the pedagogic rationale, LSC, and the course structure, whereas Part 2 will look at an evaluation of the course.

Overall Background

Computer science (CS) education at the high school level has received increasing attention in recent years. In the US, Barack Obama called on young Americans to learn programming (https://www.youtube.com/watch?v=6XvmhE1J9PY) and launched a $4 billion program to expand participation in CS education (https://www.whitehouse.gov/blog/2016/01/30/computer-science-all). Organizations such as code.org (https://code.org) and the Khan Academy (https://www.khanacademy.org/computing/computer-programming) offer programming courses for beginners of all ages, an idea that has gained public support from industry leaders like Bill Gates and Mark Zuckerberg, and celebrities such as basketball star Chris Bosh (https://www.youtube.com/watch?v=dU1xS07N-FA). In various European countries, New Zealand, Israel, and elsewhere, CS education in high school is already relatively established.[4,5]

Although high school programs vary considerably among countries, it's commonly agreed among 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). This view of what CS education means is manifested, for example, in the CSTA K-12 CS standards.[6] An example of a high school curriculum that realized these principles is the seminal work of Judith Gal-Ezer and her colleagues (hereafter denoted as GBHY95).[3] The full program GBHY95 suggested includes five units of 90 hours each. The essentials were implemented in an Israeli CS high school program 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 (this issue, commonly referred to as the second programming paradigm, is elaborated on later).

In this article, we describe a course that was successfully delivered to 19 students in a 12th-grade CS programming class. The main objective of the course was to introduce students to an additional programming paradigm, but in addition, by the very nature of the paradigm taught, it dealt with fundamental CS ideas that are central to LSC, such as nondeterminism, abstraction, and system design. The objectives of the pilot were to evaluate the extent to which the course achieved its learning goals and to better understand the various aspects involved in delivering it.

Theoretical Background

The following subsections present the learning theories underlying our pedagogic approach and reviews previous CS education research that relate to the topics on which our course focused.

Meaningful learning and fundamental ideas. The term *meaningful learning* was coined by David Ausubel,[7] who contrasted it with rote learning. Ausubel focused on the cognitive structure and claimed that meaningful learning occurs when new knowledge is significantly related to existing knowledge, yielding a highly interconnected structure. This facilitates the retention of the existing knowledge and supports the learning of new information, which can be tied to the existing knowledge structure. Furthermore, the rich set of associations between concepts learned in different contexts leads to generalizations and allows the learned concepts to be retrieved and applied outside the specific context in which they were originally introduced. Jerome Bruner called this kind of generalization *nonspecific transfer* and considered it as the ultimate goal of the learning process.[8] Another principal component of Bruner's theory was organizing learning around fundamental ideas.

Andreas Schwill extended Bruner's work and applied it to computer science education.[9] He formulated Bruner's notion of what makes an idea fundamental into a set of four criteria, each

These two learning approaches—learning by doing and constructionism—emphasize the role of motivation and engagement in the learning process.

defining an essential dimension of fundamental ideas: the horizontal criterion states that fundamental ideas appear in multiple ways and in different domains; the vertical criterion states that fundamental ideas can be taught on various levels of complexity; the time criterion defines fundamental ideas as those that can be clearly observed in the historical development of the discipline; and the sense criterion states that fundamental ideas are related to everyday language and thinking. Based on this set, Schwill developed a catalog of fundamental ideas of CS, grouped into a tree-like structure under three master ideas: algorithmization, structured dissection, and language.

The same characteristics that make ideas fundamental—their general nature and their applicability to many domains—are also the essence of Colin Corder's notion of soft concepts,[10] which, according to him, are what make such concepts difficult to teach and learn. Teaching fundamental ideas lies at the foundation of our educational approach; thus, promoting meaningful learning of such ideas was the main consideration underlying our pedagogy, which is mainly based on the conceptual recommendations of Bruner and Ausubel.

Promoting meaningful learning. Bruner suggested arranging the curriculum in a spiral manner around fundamental ideas. According to this principle, known as the *spiral curriculum*, concepts should be revisited in different contexts and on different levels of complexity throughout the curriculum. This was based on his belief that "any subject can be taught effectively in some intellectually honest form to any child at any stage of development."[8] The fact that fundamental ideas appear in different contexts and levels of complexity reflects their horizontal and vertical nature, as formalized by Schwill.

Ausubel recommended the introduction of advance relevant organizers to which the new knowledge could be connected. While Ausubel presented this idea in the context of designing a lesson or a learning unit, it's also relevant to the design of a more longitudinal learning experience, such as a curriculum. Each time a concept is revisited, new knowledge, based on previous knowledge, is created. Emphasizing common core, that is, the fundamental idea that connects the previous and the current learning experiences, leads to generalization, which in turn supports nonspecific transfer. Thus, interpreting Burner's ideas through Ausubel's cognitive approach implies that the occurrence of the ideas at the early stages of the spiral produces organizers that help deal with more advanced occurrences of these ideas at later stages. It's thus important to start introducing these ideas early on, as this can affect the ultimate level of dealing with the concept.

The idea that learning should be arranged in a gradual way that matches the level of the learners and their needs is also the essence of the scaffolding theory of learning,[11] which is based on Bruner's ideas and on Lev Vygotsky's concept of the zone of proximal development.[12]

Another strategy for achieving meaningful learning is by making it active. The advantages of learning by doing have been emphasized by various authors, and it's also the essence of Idit Harel and Seymour Papert's constructionism.[13] In addition to the cognitive aspects, these two learning approaches—learning by doing and constructionism—emphasize the role of motivation and engagement in the learning process. Project-based learning[14] is another highly effective approach that emphasizes learning by doing but also highlights the role of collaboration and of solving real-world problems as activities that support the learning process.

A second programming paradigm. GBHY95 suggested that students should be introduced, besides the main language that they learn (currently, in Israel, it's 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."[3] These two sentences present a clear rationale that's in line with Bruner's philosophy of teaching fundamental ideas in a spiral manner: programming is mainly a vehicle for dealing with the fundamental idea of an algorithm; students should be exposed to various manifestations of the concept, with the purpose of enriching their understanding of it.

Following the general concept of the spiral curriculum, we believe that fundamental issues should also be considered at earlier stages, in a way that prepares the ground for dealing with them in the future in a more advanced way.

Following this rationale, GBHY95's high school program devoted a 90-hour unit (out of five units of similar length) to the second paradigm; teachers could choose from a few optional courses, each focusing on a different paradigm. One of these was logic programming with Prolog. The rationale behind this course was exposing students to a high-level, declarative programming style. Several studies showed that learning Prolog supports developing abstraction skills, such as using black boxes.[15] However, studies by Marian Petre[16] and Josie Taylor[17] indicated difficulties related to the lack of a clear operational model in Prolog. LSC is also a language that's radically different from conventional languages used in introductory courses, maybe even more so than Prolog. Besides being declarative and high level, it's also a visual language, and its development environment implements a novel approach for concrete interface programming (the play-in method). This makes LSC a very interesting choice for the second paradigm. Our findings in other work[18] suggested that the relatively clear and accessible underlying operational model of LSC makes it easier for students to cope with its declarative nature.

Nondeterminism (ND). This topic usually isn't included in high school programs,[19] but as a fundamental idea in CS,[20] we believe ND should be included, in line with Bruner's spiral approach for teaching fundamental ideas. ND is also one of the essential characteristics of concurrency, which is another fundamental principle in CS. In the GBHY95 high school program,[3] ND was covered in the computational models course, which was one of the options for the theory module.[21] In that course, ND was 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.[22] As LSC is a nondeterministic programming language, teaching it inherently includes teaching ND, but of the kind that appears in nondeterministic and concurrent programming. In one work,[19] this kind of ND was termed *operative ND*. At the core of this type of ND lies the idea of *true don't care*, which means that there's a priori no preference, and all possible computations are equally good. Thus, operative ND has universal semantics, as opposed to the existential semantics typically presented to students in courses that deal with automata theory. The results reported in the previous work showed that after learning LSC, students can reach a significant understanding of this kind of ND. A possible implication of these findings, yet to be investigated, is that operative ND should be introduced first, and that it might facilitate the understanding of nondeterministic automata and Turing machines.

System design and abstract thinking. Abstraction is a very fundamental CS idea.[23] Introducing students to system design and developing abstract thinking skills are primary objectives of the advanced programming module of the GBHY95 high school program. Because LSC is a high-level, declarative programming language, its learning naturally supplies opportunities for dealing with system design and abstraction. As we reported elsewhere,[18] novice and graduate students who learned LSC presented patterns of desirable high-level thinking. Within the group of graduate students, the use of abstraction with LSC was compared to difficulties that these students had when solving the same task using object-oriented programming languages.

Other issues. Working with LSC raises other important software engineering issues that typically aren't included in the Israeli high school curriculum, which mainly emphasizes algorithmic thinking, or in other K–12 CS curricula in other countries.[24,25] One of these issues is usability and HCI. The IEEE/ACM Joint Task Force on Computing Curricula (CC2013; http://ai.stanford.edu/users/sahami/CS2013/final-draft/CS2013-Final-v0.9-prerelease.pdf) suggests devoting eight core hours (and elective units) to HCI. While CC2013 refers to undergraduates, this reference is an indication that HCI is a central aspect of software engineering. Following the general concept of the spiral curriculum, we believe that fundamental issues

should also be considered at earlier stages, in a way that prepares the ground for dealing with them in the future in a more advanced way. According to findings we reported elsewhere,[26] graduate students' attention to HCI issues was higher when working with LSC, compared to their attention to these issues when solving similar tasks in object-oriented programming languages.

Because LSC might raise students' attention to HCI issues, exposure to it at early stages can serve as the starting point for a spiral teaching of this subject, setting the ground for future, more advanced learning of the subject. Other issues that naturally come up when working with LSC are requirements engineering and verification. This is because LSC is basically a specification language that aims to describe the behavior of reactive systems,[27] which are systems in which the main complication stems from the intricate interactions among users, environments, and system components. Though these issues weren't the focus of our course, we did use this opportunity to briefly discuss them as well. Again, this can serve as organizers for future learning and is in line with a broad introduction of the CS discipline, which is a central goal of the GBHY95 high school program.

Course Structure and Setting

To achieve meaningful learning, our course followed two pedagogic principles: the zipper principle[3] and project-based learning.[14] The former is a pedagogical method that combines ideas of scaffolding and learning by doing. It means that theoretical lectures are interwoven with hands-on experience in the lab, in which the students exercise the learned concepts on a small scale and in a controlled setting. This supports a gradual, bottom-up learning of language constructs and the execution model, making it possible to familiarize students with the development environment. The first half of the course was arranged according to this principle.

In the second half of the course, which was project-based, students chose, designed, and implemented a project in LSC. Project-based learning is basically a top-down learning approach. Students start from what they want to build, and use their knowledge (and if needed, acquire additional knowledge) to realize it. Among other things, this requires that the learners synthesize their knowledge and provides it with a real-world context. As mentioned earlier, the project-based approach emphasizes collaborative work, and studies show that it increases motivation.[14]

In the GBHY95 high school program, the second paradigm unit is planned for 90 hours. Due to external constraints, our course was 45 hours long. A 90-hour course would allow for deeper treatment of concepts that weren't sufficiently discussed, would include concepts that we omitted for lack of time (such as asynchronous execution), and would allow us to devote more time to the projects. Yet, as our research shows, even a 45-hour course enabled students to achieve (project-based) meaningful learning of the principles of scenario-based programming and of advanced topics, such as ND.

Our course was given to high school students, but we believe that the results are also applicable to achieving similar goals in the context of undergraduates. An undergraduate course can delve deeper and connect the learned subjects to advanced topics such as synthesis.

Live Sequence Charts and Scenario-Based Programming

LSC is a visual programming language for reactive system development that was originally introduced by Damm and Harel.[1] It is supported by the Play-Engine[2] development environment, which we used in the course, and the later tool, PlayGo,[28] which is a much more mature environment. PlayGo is one result of an extensive effort toward making LSC and behavioral programming publicly available and accessible. It can be downloaded for free, together with tutorials, demos, and a language reference, at http://wiki.weizmann.ac.il/playgo.

Scenario-Based Programming

LSC introduces a new paradigm called *scenario-based programming*, the main abstraction being a scenario that describes a series of actions constituting a certain functionality of the system and that can include possible, necessary, or forbidden actions. For example, cash withdrawal is a basic functionality of an ATM, so a scenario that describes system behavior in cash withdrawal will detail interactions between the person withdrawing money and the system, and between internal parts of the system. A scenario is implemented by a live sequence chart (hereafter we just use *chart*). Figure 1 shows an example of a chart for cruise control in an automobile.

Syntactically, a chart is composed of two parts: the prechart and the main chart. The prechart is the upper dashed line hexagon in Figure 1; it's the chart's activation condition. Once the chart is activated, execution enters the main chart; this

is the lower rectangle in Figure 1, which contains execution instructions. The vertical lines represent objects, and the horizontal arrows represent interactions between them. Time flows from top to bottom. Execution rules define how one chart is activated and executed, and how multiple charts are synchronized to run simultaneously. The engine's ability to interweave at runtime charts that implement different aspects of the system into a single flow enables (and encourages) an incremental development process, in which the system is built by heaping separate scenarios. In addition to LSC, scenario-based programming is also available as an extension to Java, C++, Erlang, and Google's Blockly. The general approach has been called *behavioral programming*.[29]

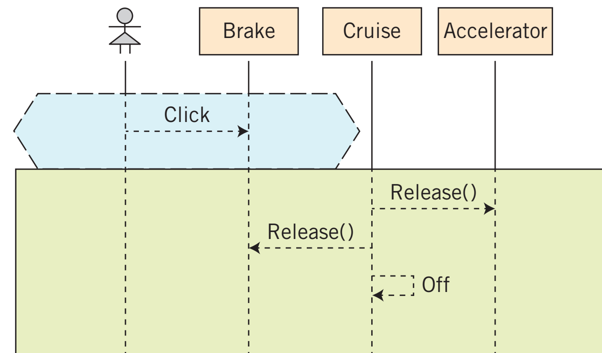

Figure 1. A live sequence chart. This simple scenario is taken from an implementation of cruise control. If and when the user presses (clicks) the brake pedal, the cruise unit releases control of the accelerator and the brake, and then turns itself off.

The Play-In Method

LSC is supplemented with a method for building the scenario-based specification over a real or mockup GUI of the system via the play-in method, which is implemented in the Play-Engine/PlayGo. With play-in, the user specifies scenarios in a way that's close to how real interaction with the system occurs. Figure 2 shows a snapshot of applying play-in on a toy GUI of a cellular phone. The chart on the left implements a scenario that describes what the display and the speaker should do once the user shuts the cellphone cover. It was programmed into the system by actually interacting with the GUI, that is, by playing with its components and through this inserting the scenario's logic. (Inserting visual notations can also be done on the chart and not only through the GUI. This is to support programming of elements that don't have a GUI representation or that have operations that are less convenient to do through the GUI.)

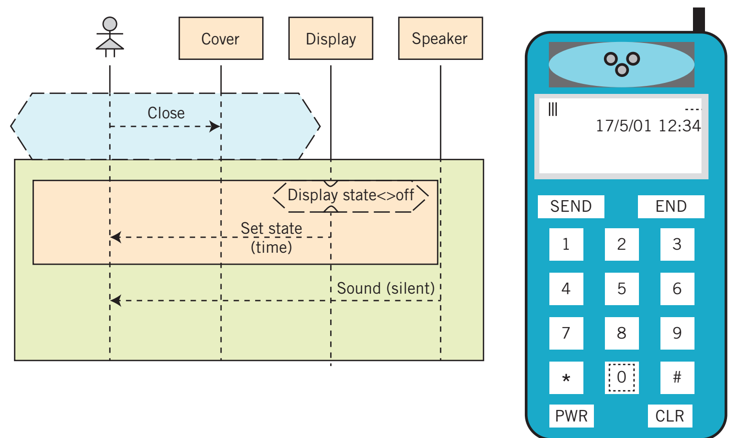

Figure 2. The play-in method. The chart on the left implements a scenario that describes what the display and the speaker should do once the user shuts the cellphone cover.

The Play-Out Method

LSC has an operational semantics that's implemented by the play-out method,[30] which is also included in the Play-Engine/PlayGo tools. Play-out makes the specification directly executable/simulatable, thus enabling the use of LSC as a high-level programming language (other than merely as a specification language).

The Course

The course that we describe here was given to a standard class of 19 (10 girls, 9 boys) 12th-grade high school students majoring in CS. Students' previous experience included two introductory CS courses given in Java (total of 180 hours), one in the 10th grade and the other in the 11th grade; a course on computer organization and assembly language (90 hours), given in the 11th grade; and a shortened version (45 hours instead of 90) of the Computational Models course, which was given in the first half of the 12th grade. This latter course dealt mainly with deterministic finite automata and didn't include the concept of ND (we intended to introduce it through LSC). Our course included 45 hours and replaced the second half of the Computational Models course, meaning it was part of the fifth unit. In parallel with the fifth unit, the students learned software design (the fourth unit), which mainly dealt with (relatively) advanced software issues such as data structures, recursion, API, and performance, presented in an object-based approach.

Our course was developed and executed as a pilot course on scenario-based programming with LSC, under the auspices of the Israeli Ministry of Education. It was mandatory for students in this class, and its final grade was based on a matriculation exam that included a section on LSC and scenario-based programming, together with sections that referred to other CS units that students learned during that year (computational models and software design).

Main Objectives and Teaching Method

The main objective of the course was teaching scenario-based programming as a second paradigm via LSC. The course was divided into two parts. The main purpose of the first was to cover basic concepts and the relevant domains that the course touched on (reactive systems, the software development cycle, and so on). The teaching method followed the zipper principle, meaning that theoretical lectures were interwoven with hands-on exercise in the lab. The second part was project-based. The students were divided into small groups, and each group chose a project and programmed it in LSC and implemented it on the Play-Engine. During this part, we embedded some pauses in which new, more advanced concepts were introduced to the whole class.

Usually, the trigger for pausing the work to discuss a topic was a real need that came from one or more groups. This approach takes advantage of a main characteristic of project-based learning: students have a very strong motivation to learn the concept, as it addresses a genuine need. This is true not only for the specific group from which that need arose but also for the other groups. Because the other students are more or less at the same stage of learning, they naturally feel that this information is very relevant to them. Typically, immediately after the topic is discussed, all the groups discuss it in the context of their project.

The concepts taught this way weren't always totally new. For example, the concept of forbidden scenarios was very briefly mentioned in one of the lectures, with the purpose of delving into it during later stages. One group project dealt with safety requirements (for an elevator), which are commonly about forbidding dangerous scenarios, and recalled that LSC has such syntax. This opportunity was used to discuss in more depth the conceptual issue of forbidding a scenario, and the native support this software concept has in LSC. This illustrates the pedagogic effectiveness of a teaching method that involves organizers given in advance, revisited in a spiral manner in the context of project-based learning.

Course Structure and Projects

The course included 15 lessons of three hours each. Table 1 presents the content of each lesson.

The projects were carried out in five groups of three to four students each. Each group was required to choose a project of reasonable complexity and to implement it in LSC. We devoted to the project 15 hours in class, and the students did some more work at home (the homework that was given required around one more weekly hour, but some of the groups did more work than we required).

Choosing the projects. It was important for us that the students choose a system by themselves, to make them more engaged (indeed, in the post-interviews, students mentioned this as a motivating factor). At this point in the course, the students had a clear enough idea of what they were able to do with LSC. Students' projects included modeling an elevator, a coffee machine, an ATM, and a cellphone, and developing a memory game (Simon).

The work process. After the students chose their projects, we guided them to work according to the following stages: define a set of requirements, implement them, verify that the implementation meets the requirements by simulating it, and decide on the next steps accordingly (either fix the implementation or start a new round). The rationale that underlies this process is of incremental development, which also underlies recent software engineering methodologies such as agile programming. The crux is that LSC naturally supports incremental development, as it lets you build a system by adding separate scenarios that are combined by the underlying engine into a single program.

Implementation. Since the final exam had to be pen and paper, it was important that students also practice drawing the charts by hand. Thus, they were required to implement the project in the Play-Engine but also to submit a handwritten version.

Submissions. Out of five groups, four submitted their projects with quality that varied from satisfactory to very good (especially if we consider the limited time, 15 hours in total). One group didn't submit its project; this group consisted of four students who showed low attendance. According to

Table 1. Course structure.

Lesson	Subject	Hours	Concepts	Details
1	Introduction	3	Visual languages, reactive systems, static vs. dynamic behavior	Rules system as a way for describing legal behaviors; visual representations and formalisms (for example, London's Underground map); reactive systems and the difficulties in modeling dynamic behavior (vs. modeling static relationships)
2	Introduction to LSC and scenario-based programming	3	LSC basics: pre-chart/main chart, play-in/out, lifelines, messages	Introduction to LSC and the Play-Engine; exercise: students program a simple scenario using play-in and run it using play-out
3	Execution engine, and simultaneous execution of multiple scenarios	3	Play-out, monitored vs. executed events, concurrent execution	Theoretical explanation and demonstration of simultaneous execution, followed by practical experience in the lab
4	System specification and the system model	3	The software development cycle, specification, requirements	Using requirements to specify a simple cellphone; implementing the requirements as LSCs
5	Partial order	3	Partial order, synchronization, unification, nondeterminism	Different orders within a chart and between charts; unification rules, synchronization of several charts; using unification rules to synchronize LSCs
6	Conditions and assertions	3	Hot and cold conditions	Conditional execution, assertion vs. graceful abort
7	Small project	3	Variables, loops, subcharts	Pairs work in the lab; build a small system from a set of requirements to introduce students to additional constructs such as loops, variables, and subcharts
8	Symbolic elements	3	Symbolic elements, bindings, existential vs. universal semantics	Using symbolic elements to define general behavior
9	Midterm exam	3		
10-14	Final projects	15		Forbidden elements and scenarios
Exams	Assessment	6		Final exam, matriculation exam

the teacher, this was typical behavior of these students and related to problems not connected to our course.

Assessment

The purpose of the assessment was to evaluate students' understanding of LSC. It served two very different purposes: grading the students, as required in every high school course, and evaluating the course, to examine whether its goals were achieved. Here, we deal with the former; the latter is elaborated on in Part 2. Grading was based on internal midterm and final exams, and on a matriculation exam that included a part written especially for this class. Because the matriculation exam in the context of which the course was given (the fifth unit) is carried out on paper, we built internal exams in the same form. Thus, and as typically happens, the assessment method determined, in a sort of reverse fashion, many aspects of the course. First, we had to give considerable weight during class to actually drawing the charts by hand. While this might be a nonissue in textual languages, it becomes an issue when teaching visual languages that have rich graphical notations. For example, some students have difficulties making accurate drawings; drawing can be time-consuming, especially if different colors are used, or if mistakes are made and the chart needs to be redrawn. To overcome this, we developed with the students a sort of a relaxed version of the visual language. Fortunately, the LSC notations that use colors have other identifying marks. For example, LSC includes conditions, which come in

Table 2. Students' grades.

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

Table 3. The number of questions and the results per metacategory and exam.

Exam	Bloom 1-2		Bloom 3-4		Bloom 5-6	
	No. questions	Success (%)	No. questions	Success (%)	No. questions	Success (%)
Midterm	1	100	2	75	1	93
Final	2	92	3	84	2	83
Matriculation	2	95	3	85	2	84

two flavors: cold and hot. Cold conditions are denoted by blue dashed lines, while hot ones use red solid lines, so they can be distinguished even in black and white.

During the exams, students were required to create, modify, and comprehend LSC systems. For example, one of the exam questions dealt with the specification and implementation of a cellphone. The question included several charts, each implementing a scenario that captures a system requirement, and students had to answer various questions regarding the execution of these charts, which involved, among other things, concurrency and ND; to modify one of the charts to fix a bug; and given a new requirement, add a chart implementing it (https://weizmann.box.com/LSCsExamQuestion).

Table 2 presents average grade, standard deviation, and number of participants for each test.

The rationale that guided the design of the exams was to include questions that measure different levels of understanding, as defined by Bloom's taxonomy (in its revised form[31]). However, we adopted a more relaxed interpretation of the taxonomy, and instead of using six categories, we grouped the categories into three metacategories: one that includes categories 1 and 2 (Remembering and Understanding), one that includes categories 3 and 4 (Applying and Analyzing), and one that refers to categories 5 and 6 (Evaluating and Creating). The rationale is that Bloom's categories can overlap, and the classification of operations into categories can be ambiguous.[32]

Concentrating on three metacategories allows us to consider the categories in a somewhat broader form, which makes the classification less ambiguous. Table 3 describes, per metacategory and exam, the number of questions that were included and the average grade for these questions.

The results of the pilot course indicate that high school students can reach a significant understanding of LSC, and through working with the language, deal with high-level programming and nondeterminism. This is further discussed in Part 2 of this article, which will appear in the next issue. ■

Acknowledgments

We thank Avi Cohen, Ronit Ben-Bassat Levy, Nir Eitan, and Zehava Levin for their help in conducting this research, which 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. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods System Design*, vol. 19, no. 1, 2001, pp. 45–80.
2. D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*, Springer-Verlag, 2003.
3. J. Gal-Ezer et al., "A High School Program in Computer Science," *Computer*, vol. 28, no. 10, 1995, pp. 73–80.
4. T. Bell, P. Andreae, and L. Lambert, "Computer Science in New Zealand High Schools," *Proc. 12th Australasian Conf. Computing Education*, 2010, pp. 15–22.
5. M.E. Caspersen and P. Nowack, "Computational Thinking and Practice: A Generic Approach to Computing in Danish High Schools," *Proc. 15th Australasian Computing Education Conf.*, 2013, pp. 137–143.
6. D. Seehorn et al., "CSTA K–12 Computer Science Standards: Revised 2016," CSTA/ACM 2011; https://www.csteachers.org/resource/resmgr/Docs/Standards/2016StandardsRevision/INTERIM_StandardsFINAL_07222.pdf .
7. D.P. Ausubel, "Cognitive Structure and the Facilitation of Meaningful Verbal Learning," *J. Teacher Education*, vol. 14, no. 2, 1963, pp. 217–222.
8. J.S. Bruner, *The Process of Education*, Harvard Univ. Press, 1960.

9. A. Schwill, “Fundamental Ideas of Computer Science,” *Bull. European Assoc. Theoretical Computer Science*, vol. 53, 1994, pp. 274–295.
10. C. Corder, *Teaching Hard, Teaching Soft: A Structured Approach to Planning and Running Effective Training Courses*, Gower, 1990.
11. D. Wood, J.S. Bruner, and G. Ross, “The Role of Tutoring in Problem Solving,” *J. Child Psychology and Psychiatry*, vol. 17, no. 2, 1976, pp. 89–100.
12. L.S. Vygotsky and M. Cole, *Mind in Society: The Development of Higher Psychological Processes*, Harvard Univ. Press, 1978.
13. I. Harel and S. Papert, eds., *Constructionism*, Ablex, 1991.
14. P.C. Blumenfeld et al., “Motivating Project-Based Learning: Sustaining the Doing, Supporting the Learning,” *Educational Psychologist*, vol. 26, nos. 3–4, 1991, pp. 369–398.
15. B. Haberman and Z. Scherz, “Evolving Boxes as Flexible Tools for Teaching High-School Students Declarative and Procedural Aspects of Logic Programming,” *From Computer Literacy to Informatics Fundamentals*, Springer, 2005, pp. 156–165.
16. M. Petre, “Shifts in Reasoning about Software and Hardware Systems: Must Operational Models Underpin Declarative Ones?” *Proc. 3rd Workshop Psychology of Programming Interest Group,* 1991; www.ppig.org/library/paper/shifts-reasoning-about-software-and-hardware-systems-must-operational-models-underpin.
17. J. Taylor, “Analysing Novices Analysing Prolog: What Stories Do Novices Tell Themselves about Prolog?” *Instructional Science*, vol. 19, no. 4–5, 1990, pp. 283–309.
18. 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.
19. G. Alexandron et al., “On Teaching Programming with Nondeterminism,” *Proc. 8th Workshop Primary and Secondary Computing Education*, 2013, pp. 71–74.
20. 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.
21. J. Gal-Ezer and D. Harel, “Curriculum and Course Syllabi for a High-School Program in Computer Science,” *Computer Science Education*, vol. 9, 1999, pp. 114–147.
22. M. Armoni, N. Lewenstein, and M. Ben-Ari, “Teaching Students to Think Nondeterministically,” *SIGCSE Bull.*, vol. 40, no. 1, 2008, pp. 4–8.
23. M. Armoni, “On Teaching Abstraction in CS to Novices,” *J. Computers in Mathematics and Science Teaching*, vol. 32, no. 3, 2013, pp. 265–284.
24. P. Hubwieser, M. Armoni, and M.N. Giannakos, “How to Implement Rigorous Computer Science Education in K-12 Schools? Some Answers and Many Questions,” *ACM Trans. Computing Education*, vol. 15, no. 2, 2015, article no. 5.
25. P. Hubwieser et al., “Perspectives and Visions of Computer Science Education in Primary and Secondary (K-12) Schools,” *ACM Trans. Computing Education*, vol. 14, no. 2, 2014, article no. 7.
26. 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.
27. D. Harel and A. Pnueli, “On the Development of Reactive Systems,” *Logics and Models of Concurrent Systems*. Springer-Verlag, 1985, pp. 477–498.
28. 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.
29. D. Harel, A. Marron, and G. Weiss, “Behavioral Programming,” *Comm. ACM*, vol. 55, no. 7, 2012, pp. 90–100.
30. 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.
31. 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.
32. U. Fuller et al., “Developing a Computer Science-Specific Learning Taxonomy,” *SIGCSE Bull.*, vol. 39, no. 4, 2007, pp. 152–170.

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.