Computer Science in High-Schools: Curricula and Research

Judith Gal-Ezer
The Open University of Israel
galezer@cs.openu.ac.il

ABSTRACT
A high-school curriculum in computer science was designed by a committee, appointed by the Ministry of Education. The committee also supervised the preparation of a comprehensive program of studies and materials based on the curriculum, and accompanied its implementation. The program emphasizes the foundations of computer science, algorithmic thinking, the design of solutions to algorithmic problems, and teaches programming as a way to get the computer to carry out algorithms. This paper describes the dissemination and adoption of the curriculum in Israeli high-schools, and two of the studies that were conducted following the implementation of the curriculum.

KEY WORDS: Computer Science, High-school Curriculum, Programming

INTRODUCTION

A committee appointed by the Ministry of Education in 1990 put together a computer science high-school curriculum[1]. The committee included computer scientists who are also involved in various kinds of educational activities, experienced high-school teachers of computer science, and computer science education professionals from the Ministry of Education[2]. The curriculum is described in a comprehensive paper (Gal-Ezer et al. 1995). The program emphasizes the basics of algorithmics, and although it teaches programming it does so only as a means for getting a computer to carry out algorithms. The background and motivation for the program, its general structure and its initial implementation were all discussed in Gal-Ezer et al. (1995); in the following we will very briefly describe the principles that guided the committee's work and the structure of the program. A detailed description of the entire program is given in Gal-Ezer & Harel (1999).

This curriculum that was designed for the Israeli educational system can also be applied elsewhere, with the needed adaptations. Indeed, considerable activity has

[1] In Israel, high-school refers to grades 10 to 12, elementary school to grades 1 to 6, and junior high school to grades 7 to 9. At the end of high-school, students take a battery of matriculation exams. All exams have at least two different versions: a 3-point version and a 5-point version

[2] The author of this paper served on the committee, and currently serves as its head (the committee now includes other new members)

Proceedings of the 3rd Panhellenic Conference «Didactics of Informatics»
A. Jimoyiannis (ed.)
University of Peloponnese
Korinthos, Greece, 7-9 Oct. 2005

surrounded CS curricula at all levels, and the most updated are Curricula 2001 for College and University level (IEEE, 2002) and the K-12 curricula for school level (ACM, 2004).

The Israeli program has been formally approved by the Israel Ministry of Education, its implementation in Israeli high-schools has been completed, and it now replaces all previous programs. As a result, it was possible to conduct research studies relating to the implementation of the program, and to provide recommendations for improving the study materials and the program. Among these were studies relating to efficiency, reduction and non-determinism, and the results are described in detail in Gal-Ezer & Zur (2002) and Armoni & Gal-Ezer (2003).

PRINCIPLES

The principles that guided the committee's work are as follows:

- **Computer Science is a full fledged scientific subject** and should be taught on the same par with Physics, Chemistry and Biology.
- **The program should concentrate on the key concepts and foundations of the field,** emphasizing the notion of an algorithmic problem and its solution the algorithm. The program should focus on lasting concepts of computer science not on changing technology.
- **Two different programs are needed, one for the 3-points and one for the 5-points,** the first for those who have only a general interest in computer science, and the second for those who want to get much more involved in this scientific discipline.
- **Each of the programs should have required units and electives,** to achieve variance and flexibility.
- **The zipper principle:** Conceptual and experimental issues should be interwoven throughout the program.
- **Two different programming paradigms should be taught** to provide different ways of algorithmic thinking or different ways of solving problems.
- **A well-equipped and well-maintained computer laboratory is mandatory,** to support laboratory sessions and individual "screen time" for each student.
- **New course material must be written for all parts of the program,** by different teams in different academic institutions. The teams must have computer scientists on board, as well as high school teachers and researchers in computer science education.
- **Teachers certified to teach computer science must have an adequate formal computer science education,** i.e. at least an undergraduate degree in computer science.

THE PROGRAM

The study material is divided into five modules of 90 hours each, some of which have more than one alternative.

Fundamentals 1 and **2** (double-length; 180 hours): These introductory units provide the foundation for the entire program. They introduce the central concepts of algorithmics, and in parallel teach how algorithms are implemented in a programming language (at first a procedural approach was taken).
Second Paradigm or Applications (90 hours): This unit introduces the student either to a second programming paradigm or to some general kind of application area, such as information systems or computer graphics, and teaches both its principles and its practice. The second paradigm is to be conceptually quite different from the procedural approach adopted in the Fundamentals.
Software Design (90 hours): This unit is a continuation of Fundamentals. It concentrates on data structures, introducing abstract data-types in the process. It also goes beyond stand-alone algorithms by discussing the design of complete systems.
Theory (90 hours): This unit is intended to expose the student to selected topics in theoretical computer science. Currently approved alternatives are a full 90-hour unit on models of computation, or a two-part unit consisting of 45 hours on models of computation and 45 hours on numerical analysis.

There are three possible study versions that can be constructed from these modules:

A low-level 90-hour version, an intermediate 270-hour version, and a full 450-hour version intended for scientifically-oriented students interested in continuing their studies in computer science or related fields.

The first program (called the "one-unit program") consists only of the 90-hour Fundamentals 1 unit. It provides a brief "taste" of the material and is mandatory for all students. Depending on the school, the program is taught in the 10th or 11th grade.

The second program (the "three-unit program") consists of Fundamentals 1 and 2, and an additional 90-hour unit, which can be taken from the Second Paradigm or Applications modules. This program also may be taught beginning in the 10th or 11th grade, over one school year.

The third program (the "five-unit program") consists of Fundamentals 1and 2, one 90-hour unit taken from the Second Paradigm or the Applications modules, Software Design and Theory. This program is intended for students majoring in Computer Science, and is taught in the 11th and 12th grades.

DISSEMINATION AND DIFFICULTIES

In the fall of 1991 we began a limited field test with eight study groups in five schools. In the fall of 1994, this was expanded to forty groups in eight schools. By the end of 2000, the entire program was successfully adopted and implemented in all high schools.

One major obstacle we had to overcome was convincing the Ministry of Education that it is essential that CS teachers have adequate formal CS education. The Ministry of Education's official regulations state that in order to teach a full scientific program in the 11th and 12th grade, a second degree – M.Sc. or equivalent - is required. However, in practice, this regulation was not complied with. In view of the burgeoning hi-tech job market at that time, we were realistic and required only a B.Sc. or equivalent. We succeeded in convincing the Ministry. However, at that time, many teachers had already been teaching CS or programming in high schools without any formal training. Hence we designed a crash training program that consisted of six university courses, which teachers with no formal education were required to complete.

In addition, even the best teachers needed training in order to become familiar with the new material and the new emphasis on long-standing concepts rather than changing technologies, like pseudo code in designing algorithms instead of the flowcharts they were used to teaching. Thus, a massive instructional program was designed and many in-service and pre-service teachers participated in this course. The Ministry of Education which budgeted the entire new program (development of material and field tests) also budgeted the in-service and pre-service teacher training.

Another problem we encountered was the shortage of computers in schools. This has changed considerably in most of the schools, yet there are still inadequately maintained school labs.

The main obstacle that we face today is the need to update the program. We need to switch to an object-oriented paradigm in the first two modules, and in the fourth one; to prepare new written materials and, of course, to provide more in-service teacher training. Severe budget cuts in the budget of the Ministry of Education in Israel have affected the entire educational system and inevitably also the implementation of the computer science program.

The course materials for the program were written by development teams from academic institutions. Some members of these teams were also involved in the teaching the material, and accompanied the implementation of the program. This enabled us to conduct many studies relating to various units of the program. Due to space limitations, we will summarize here only two of these studies that examined how students perceive the notions of efficiency and of non-determinism. The former is introduced in the first module and the latter in the fifth theoretical module.

Efficiency: How well is it conceived?

There is no need to explain the importance of teaching algorithm efficiency in any CS study program. We advocate teaching it at relatively early stages of the curriculum, to help expand the range of algorithmic methods, plan algorithms, evaluate alternative methods and perform analysis. Following this recommendation, Fundamentals 1 indeed introduces the notion of efficiency at an early stage. However, our initial observations of the implementation of the new CS curriculum revealed that high school students find it difficult to correctly perceive efficiency. This observation motivated the research related to misconceptions of efficiency, which was part of a wider study of the implementation of the new CS curriculum.

As a consequence of surveying many studies, we were already aware that students' conceptions of scientific issues are often not in line with accepted scientific thinking; that is, they have misconceptions regarding various notions. Relating to misconceptions is crucial since being aware of students' misconceptions can help teachers assist their students and decrease the effect of the misconception on their understanding. Understanding or realizing that misconceptions exist, is especially important in relation to the new CS curriculum, since it rests on interweaving conceptual and practical aspects, and students' misconceptions would comprise a serious stumbling block to understanding the concepts and therefore to their ability to write an efficient program in practice.

For the past 25 years or so, scholars have studied students' misconceptions regarding mathematics and science. In their study of mathematics and science education, Stavy and Tirosh (1996a, 1996b, 2000) examined common thinking patterns underlying misconceptions in different content areas. They found that many misconceptions in mathematics and science stemmed from a small number of intuitive rules. They identified three different intuitive rules: 'more of A, more of B'; 'everything can be divided by two' and 'same A, same B'. Students consider these rules to be self-evident, and apply them with great confidence. By creating logical environmental relations, they appear to enable students to understand a given situation to a certain extent without having a real understanding of the concept.

After teaching the *Fundamentals* unit for the first time, which was while we were still developing the material, we assumed that students' intuitive perceptions of the concept of algorithm efficiency could be framed in four categories:

- a shorter program (in terms of number of lines) is more efficient
- the fewer variables there are, the more efficient the program
- two programs containing the same statements are equally efficient (even if the order of the statements is different)
- two algorithms that perform the same task are equally efficient.

The findings of our study (described in detail in Gal-Ezer & Zur, 2002) confirmed these assumptions. We also realized that the misconceptions were in line with the results of Stavy & Tirosh (1996a, b), The first two misconceptions can be interpreted as being in line with the intuitive rule 'more of A, more of B' (a longer program, in terms of code lines, needs more execution time, or, in other words, is less efficient; and the more variables a program includes, the greater the execution time), while the other two misconceptions are in line with 'same A, same B' (same statements or same task, same efficiency).

Having discerned these misconceptions, we suggested indirect intervention, which has been found to be effective in helping students to better understand complex terminology. Indeed, in a separate study relating to the perception of efficiency, interviews with students were conducted to determine whether indirect intervention had an effect even in the limited time of an interview, and we found a significant effect on outstanding students (unpublished report). We recommend integrating such intervention into the written materials of the first unit of the CS study program or, at least, drawing teachers' attention to the need for it and recommending that they implement it in their teaching.

We also found that it was very difficult for 10th grade students who are not necessarily science-oriented to study this material as part of the mandatory program. This led us to the conclusion that it may not be useful to teach efficiency in the 10th grade, not even to a limited extent, at least not to the extent that it is currently taught. Not all 10th graders have enough mathematical background or sufficient algorithmic thinking to deal with this concept. This conclusion may yield radical rewriting of the materials; however, this could only follow a more focused investigation of the issue.

The second study dealt with the concept of non-determinism which is taught to advanced high school students, and in this case, there was no option of eliminating the material.

Computational Models: Didactic guidelines

The concept of non-determinism is introduced in the CM (Computational Models) unit, one of the alternatives of the fifth module. This unit – a theoretical unit – was very carefully designed from the didactic aspect. In order to fit into the 90 hour limit, not all classical models, traditionally introduced in a college or university computational models course, could be introduced. A didactic decision had to be taken, regarding the choice of models. One decision taken was to focus on models which are automata-based, and not to introduce other common models (such as regular expressions or grammars). This restriction yields “concrete” models, as they resemble real automatic machines. In addition and most important, these models have very similar definitions, enabling the introduction of each new model as based on the preceding one, minimizing the conceptual adaptation that the student needs to go through when a new model is introduced. For example, the Turing machine model is presented as a generalization of a pushdown automaton, achieved by permitting access to cells beneath the top cell of the stack.

Yet, the variety of models introduced is rich enough to enable meaningful theoretical discussion, through which major concepts in computer science are introduced: Some models are equivalent and some are not, some are non-deterministic, some differ in their closure properties and in the contribution of non-determinism to their computational power, and one of the models is equivalent to a computer program (or an algorithm). The CM unit consists of three parts: Finite automata, pushdown automata and Turing machines. We will concentrate here on the first part.

The concept of non-determinism is introduced somewhat differently from the traditional introduction of this concept in the computer science curriculum. The non-deterministic finite automaton (NFA) model is usually defined as a straightforward version of the deterministic finite automaton (DFA) (for example, Hopcroft & Ullman, 1979). In a DFA, the transition function maps each pair of a state and an input letter to a single state, while in an NFA, the transition function maps each pair of a state and an input letter to a set of states (which may also be empty). The definition of NFA is a generalization of the definition of DFA, since it enables the range of the transition functions to include any sets of states and not just singletons.

However, this generalization encapsulates two important differences between a DFA and an NFA, only one of which relates to "real" non-determinism: The *first difference* is that an NFA permits omitting transitions (that is, a state and an input letter can be mapped to an empty set), but this option still preserves the deterministic nature of the model (that is, when a mapping of a state and an input letter is given, it is to only one state). The *second difference*, which introduces real non-determinism, is that in an NFA, a state and an input letter can be mapped to a set of two or more states.

Since the concept of non-determinism (expressed by the second difference) is an abstract one and it is reasonable to assume that students will have difficulty understanding it, we thought it might be better to introduce it as a stand-alone feature. Therefore, the introduction of NFA in the CM unit is preceded by the introduction of an intermediate additional model: the Non-Complete Deterministic Finite Automaton (NCDFA). In this model, the transition function maps each state and input letter to a single state or to an empty set of states. Thus, the first difference between a DFA and an NFA, which permits omitting transitions while preserving the deterministic nature of the model, is shown in the definition of the NCDFA; while the second difference, which permits non-determinism, is expressed only in the definition of the next model -- the NFA. The abstract concept of non-determinism is thus isolated, and is introduced by itself. The addition of the NCDFA model also enriches the variety of models introduced in the CM unit, and enables practicing comparison of models. The desire to focus on non-determinism led to another change in the traditional definition of an NFA. The common definition of NFA permits e-transitions (transitions that can be made without reading any input symbol). In order to avoid additional perceptional difficulties, and since e-transitions do not enhance the computational power of NFAs, it was decided that e-transitions would not be included in the CM definition of NFA.

In order to clarify how non-deterministic models work, mainly their acceptance mechanism, the CM unit uses the magic coin metaphor, introduced by (Harel, 1987): Students are told that they should think of a non-deterministic automaton as if it were equipped with a coin which it flips before it decides which transition to choose. However, this is an unbalanced magic coin – it will always tend to accepting paths. That is, each time the automaton has to select one of several possible transitions, flipping the coin will cause it to choose a transition which can lead to a final state, while reading the suffix of the input word (if such a transition exists). Thus, if there is an accepting path for a given input word, the magic coin will guide the automaton to such a path. Non-Determinism is again discussed in the CM unit when introducing Pushdown Automata (PDA), but we will not refer to it in this paper.

Non-determinism: How well is it conceived?

Despite the efforts devoted to designing the unit, implementing it was somewhat discouraging at the beginning. When first implementing the unit (while developing it), we found that students did not fully understand the concept of non-determinism, and did not use it properly. This observation motivated the study we conducted, described in detail in Armoni and Gal-Ezer (2003). The population was high-school students, from the 11th grade and the 12th grade. In questions we integrated into their exams, we asked the students to design automata, without guiding them as to which model to use.

We will not describe the results of the study in detail here, but give only a general view of the results. Generally, the solutions given by the students can be framed in five categories:

- Fully deterministic solutions
- Solutions in which the students used decomposition to two or three sublanguages. The automata built for these sublanguages were fully deterministic, and only the use of construction algorithms introduced non-determinism into the process. We categorized this as a deterministically-based solution since when independent thinking was required, the student used the deterministic model.
- Almost deterministic solutions, with only few local non-deterministic behaviors.
- Almost non-deterministic solutions, with only a few instances in which the student ignored the freedom of the non-deterministic model and used redundant transitions.
- Fully non-deterministic solutions.

About half of the students solved the questions deterministically, or almost deterministically. No significant differences were found for grade (11th and 12th) or level of mathematics. In analyzing the students' solutions, we recognized four patterns which seem to indicate the existence of a problem in the perception of non-determinism. We therefore believe these patterns deserve close attention.

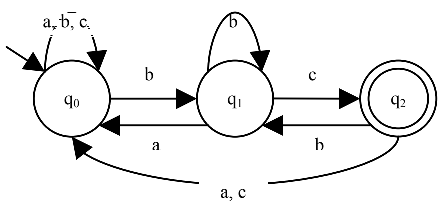

Figure 1: Local non-determinism in an almost deterministic automation

The first pattern was found in most of the solutions in the category "almost deterministic". These solutions include automata which are basically deterministic, but contain local non-determinism, expressed in a few non-deterministic transitions. In most cases, due to the deterministic character of the automaton, these transitions are redundant. An example of such an automaton can be seen in Figure 1 which shows an automaton which is deterministic in nature, except for the initial state, in which there are two transitions with the letter b. The self loop with b in the initial state is redundant, though it does not violate the correctness of the automaton (An automaton which accepts the language containing exactly the words over the alphabet {a,b,c} which end with the string "bc". This was part of one of the relevant questions). However, its existence indicates an only partial understanding of the non-deterministic mechanism. Interestingly enough, most of the redundant non-deterministic transitions that we found were self loops in the initial state. Indeed, a self loop transition, with all the alphabet letters, can be found in many of the examples of NFAs in the CM textbook, and some of the students may identify the non-deterministic model with such a transition.

The second pattern is "symmetric" to the first. It can be found in most of the solutions in the category “almost non-deterministic”. These solutions include automata which are basically non-deterministic, but contain a few transitions which would be found in the deterministic version of this automaton, but are redundant in the non-deterministic automaton. An example of such an automaton is shown in Figure 2. This automaton is non-deterministic in nature, except for the state q_1, in which there is a redundant self loop transition with the letter b. Again, this transition doesn’t violate the correctness of the automaton. However, its existence may indicate only a partial understanding of the non-deterministic mechanism.

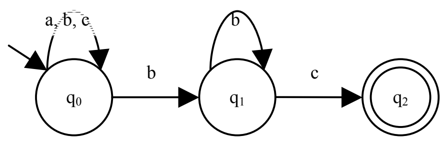

Figure 2: Redundant transitions in a non-deterministic automation

In some cases, we could identify traces of the solution process in a student’s answer. Sometimes students wrote a few preliminary versions which they chose not to complete. In the few such cases we encountered in this data, the process indicates a change from a non-deterministic model to a deterministic one. That is, the early versions are non-deterministic or almost non-deterministic, but as the students “improve” the solution they construct almost deterministic or fully deterministic automata. In some of these cases, the preliminary versions were indeed incorrect, but usually only simple and local corrections were necessary. It seems that after recognizing a problem in the automaton, the students preferred to shift to the deterministic, and perhaps more familiar, model, instead of correcting the mistake within the non-deterministic model.

The fourth and last pattern was found among the fully deterministic solutions, or the solutions in which some or all of the automata for the sublanguages were deterministic. In these cases, the students constructed incorrect NCDFAs (incorrect in the sense that they don’t accept the required language). The students utilized the freedom of omitting necessary transitions, which is characteristic of the non-deterministic model, without introducing the non-deterministic transitions that enable this. An example of such a solution is shown in Figure 3. In this automaton, there are no transitions with a and b in

q_1, and no transitions in q_2. These transitions cannot be omitted in any correct automaton that accepts that language, unless it contains non-deterministic behavior with b in the initial state. So, even though the automata constructed in this pattern were deterministic in nature, the error stemmed from a partial understanding of the non-deterministic mechanism.

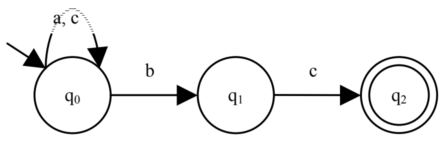

Figure 3: Incorrect non-complete deterministic automation

We emphasize again that a solution that matches one of these patterns may indicate a problem in the perception of non-determinism, even when the solution is correct. When we considered the data for each teacher separately, we found that for about half of the teachers (5 of 11), the ratio between students using the deterministic approach and those using the non-deterministic approach was about 50-50. For 3 teachers the ratio is about 60%, 70% and 85%, in favor of the deterministic approach, while for the other three teachers, the ratio is about 70%, 85% and 90% in favor of the non-deterministic approach. Thus, it seems reasonable to hypothesize that the teacher factor is significant. If the teacher emphasizes the non-deterministic model, even more than it is emphasized in the textbook, and demonstrates its advantages (for example, by using the non-deterministic model whenever s/he constructs an automaton, both when teaching the material and thereafter), this may affect the students' tendency to use the model.

DISCUSSION AND CONCLUSIONS

The two studies we briefly described pointed out difficulties in the perception of efficiency of algorithms, introduced in early stages of the study program, and non-determinism introduced in the last unit of the study program. In a third study not describe here, (Armoni & Gal-Ezer 2005), we also found difficulties in the perception of reduction, also introduced in the fifth unit. Our main concern is how prevent misconceptions and how to disseminate more successfully the basic concepts of the theoretical foundations of computer science.

Regarding efficiency, as mentioned, we suggested indirect intervention. In relation to non-determinism our results clearly show a significant tendency towards the deterministic model. Specifically, students may not realize that the deterministic and non-deterministic models are equivalent. Therefore, the teaching process should emphasize the theoretical aspects and not only the technical aspects. Indeed, studies dealing with the perception of non-determinism that focus on the teaching process, using classroom observations and interviews with teachers, might be helpful.

Despite the fact that our results show that the concept of non-determinism is a difficult one for students to cope with, and since it is one of the basic topics of CM, it is important for students to understand it properly. Full understanding of the non-deterministic model can affect students' comprehension of other topics in CM, such as pushdown automata and context free languages, since the pushdown automata model is also a non-deterministic model, which is introduced in a later part of the CM unit. Thus, special effort should be made to prepare teachers for this unit, to ensure that the teaching process in class properly emphasizes the non-deterministic model in its theoretical and technical aspects.

In conclusion, the high-school curriculum we designed yielded a rigorous and sound program of studies, whose adoption was relatively smooth and successful. However, there is more to be done, in updating and improving the materials, following the consequences of the various studies that were conducted, and continuing with intensive pre-service and in-service teacher training based on the results of the research.

REFERENCES

ACM, (2004), *A model curriculum for K-12 computer science*, K-12 Task Force, http://www.acm.org/education/k12/k12final1022.pdf

Armoni, M. & J. Gal-Ezer (2003), Non-determinism in CS high-school curricula, http://fie.engrng.pitt.edu/fie2003/papers/1251.pdf

Armoni, M. & J. Gal-Ezer (2005), Teaching reductive thinking", *Mathematics and Computer Education.* (accepted for publication)

Gal-Ezer, J., C. Beeri, D. Harel, &d A. Yehudai. (1995), A high school program in computer science, *Computer*, 28(10), 73-80

Gal-Ezer, J. & D. Harel (1999), Curriculum and course syllabi for a high-school pogram in computer sience, *Computer Science Education*, 9(2), 114-147

Gal-Ezer, J. & E. Zur (2002), The concept of 'algorithm efficiency' in the high school CS curriculum, FIE 2002, http://fie.engrng.pitt.edu/ fie2002/index.htm

Harel, D. (1987), *Algorithmics: The Spirit of Computing*, Reading, MA: Addison-Wesley

Hopcroft, J. E. & J. D. Ullman (1979), *Introduction to Automata Theory, Languages and Computations,* Reading, MA: Addison-Wesley

IEEE (2002), *Year 2001 model curricula for computing (CC-2001)*, IEEE Computer Society/ACM Task Force, http://www.computer.org/ education/cc2001/report/

Stavy, R. & D. Tirosh (1996a), Intuitive rules in science and mathematics: The case of 'more of A - more of B', *International Journal of Science Education*, 18(6), 653-667

Stavy, R. and D. Tirosh (1996b), Intuitive rules in science and mathematics: The case of ‘everything can be divided by two’, *International Journal of Science Education*, 18(6), 669-683

Stavy, R., & D. Tirosh (2000), *How students (mis-)understand science and mathematics: Intuitive rules*, New York: Teachers College Press