Judith Gal-Ezer and David Harel

What (Else) Should CS Educators Know?

Beyond the mastery of core CS material, good CS educators should also be familiar with a significant body of material that will expand their perspectives on the field, and consequently, enhance the quality of their teaching.

There is a difference between the background required of a practitioner or researcher in a scientific field and an educator thereof. While the work of the former requires extensive knowledge and skills in the field itself, the latter must have the additional ability to convey this knowledge to others correctly and reliably, to teach the said skills, to provide perspective, and to infuse the students with interest, curiosity, and enthusiasm. All this requires the educator to be more of a scientific intellectual, at least as far as the field in question is concerned. We claim that while some of this is a matter of personality and natural aptitude, some can be acquired by being exposed to material that goes beyond the technical core parts of the field.[1]

We take a closer look at these issues as they arise in the CS field. In particular, we identify some of the additional material with which CS educators should become acquainted, over and above that normally covered by an undergraduate CS program. As an interesting upshot of our work, we have constructed an undergraduate course out of this material, and have already had some experience delivering it. Parenthetically, it would be interesting to do the same for other subjects, resulting in

[1] It goes without saying that CS educators must be educators, and not only experts in teaching computing. For example, a high school CS teacher must have studied the various educational material required of any high school teacher.

TRISHA KRAUSS

special educator's courses in fields other than CS.

We must first state in the strongest possible terms that a CS educator must have a thorough background in CS, on a fitting level. While it is reasonably obvious that college-level teachers must be equipped with a doctoral degree in CS, the fact that high school-level teachers must have a Master's degree in CS is not always sufficiently acknowledged. In fact, this requirement is rarely met; in many high schools worldwide, CS is taught by people who don't even have full undergraduate training in the subject they teach. This situation must change, but it is not the subject of this article.[2] Hence, we proceed on the assumption the educator has mastered the field itself as befitting the level being taught.

We now claim that even an excellent formal CS education might not suffice for the generality and perspective required of educators, even when they are involved in teaching only parts of the subject. To address this, we propose that over and above the regular courses, CS educators be exposed to a bird's-eye view of the field, preferably in two parts—the algorithmic side and the systems side.

In addition to an overview, there are several topics that we claim educators should study and become familiar with, and we discuss each of them separately. These topics form the basis of the CS educator's course we recommend. The topics are: Some history of CS—that of the theory as well as that of the machines themselves; the delicate question of the nature of the field and its relationship with other disciplines; the details of various CS curricula and study programs at both high school and college/university levels; a variety of issues concerning the problems of teaching programming; and the use of tools and aids in teaching computer science.

When discussing these topics, we provide some guidelines for covering them in the CS educator's course and list relevant bibliographic items. Some of these items are more central or fundamental than others, and it is these that ought to be at the basis of the CS educator's course we recommend. We might add that in some cases these writings provide a sufficiently broad and general perspective to be of interest outside CS (for example, to philosophers and historians of science).

In addition to becoming knowledgeable in the topics we discuss (by taking a special educator's course, or by other means), a CS educator should also try to keep up to date with relevant developments by reading the professional periodicals in CS education. A sampling of pertinent literature includes:

- *Mathematics and Computer Education*; published three times per year by The MATYC Journal Inc.
- *Computers and Education*; published eight times per year by Pergamon Press.
- *SIGCSE Bulletin*, a quarterly published by ACM's Special Interest Group on Computer Science Education.
- *Journal of Computer Science Education (JCSE)*; a quarterly published by the International Society for Technology in Education.
- *Journal of Computers in Mathematics and Science Teaching (JCMST)*; a quarterly published by the Association for the Advancement of Computing in Education.
- *International Journal of Mathematical Education in Science and Technology*; a quarterly published by Taylor and Francis Ltd.
- *Journal of Educational Computing Research*; published eight times per year by Baywood Publishing Co.
- *Journal of Technology and Teacher Education*; a quarterly published by the Association for the Advancement of Computing in Education.

There are also regular columns on CS education in *Communications of the ACM* and in IEEE's *Computer*, as well as in newsletters like ACM's *SIGACT News* and the *Bulletin of EATCS*.

As noted earlier, the approach taken here could be useful in other fields as well, but clearly we concentrate on CS because of our own background and occupation. Nevertheless, we feel there is something different about tackling the issue of good education in our field. The point is that CS is not only the scientific basis of a major technological revolution, but has at its heart a special and powerful way of thinking—algorithmically—which is required in dealing with the ever-complex modern world, and which is becoming crucial in many other scientific and engineering disciplines, too. Endowing students with the ability to approach a new kind of problem that requires them to think differently is a subtle and difficult educational challenge that requires careful attention.

History

Any history, and that of a scientific discipline in particular, has ramifications beyond the mere chronological listing of events and facts.

Studying the history of a science helps one appreciate the difficulties that faced the pioneering figures and provides a deeper understanding of the forces and considerations that helped form it. This includes lessons that can be learned from false starts, failures, and misconceptions. Moreover, history provides a

[2]For more details, see "A high school program in computer science" in *Computer* (1995).

global perspective of the field and its structure, and often clarifies its relationship with other fields. History also influences a student's thinking about present and future developments.

The history of CS is somewhat unique, for two main reasons. First of all, the discipline itself is young and is developing amazingly fast. Although one of the earliest algorithms (for computing the greatest common divisor) goes back to Euclid, people commonly agree that as a science computing has only been around for something like 65 years. Many of its pioneers are still with us; budding computer scientists can hear them teach and lecture, and can sometimes even work with them.

Second, as we shall argue, CS itself is an unusually dichotomic subject—one facet is more mathematical and the other is a type of engineering—a fact that is reflected interestingly in its history. On the one hand, the mid-1930s witnessed some of the most fundamental work on the nature and boundaries of computing, regardless of the technology that would later be used to implement it. This work was carried out by mathematicians like Turing, Church, and Gödel. On the other hand, the first general-purpose computers were built just a little later, and in the process some of the most basic and lasting principles governing the structure and operation of actual computing devices were formulated.[3] This work was carried out by many people, including physicists like Atanasoff, mathematicians like von Neumann, and engineers like Mauchly and Eckert.

As to incorporating historical aspects into standard CS courses, many possibilities present themselves. For example, when mentioning Cook's theorem or the Turing test, students can be sent off to find out more about the people in question and their work (as is nicely done in Koffman's book). Lee's paper discusses several other such ideas.

As to integrating the history of the field into our CS educator's course, this should be done mainly by guided reading that could lead to the preparation of written reports and papers. These reports could be period-oriented or subject-oriented, or could treat a subject from the historical perspective of a single person.

The bibliographic list for this section is a partial collection of possible starting points, but there are many more. In particular, the list can be supplemented by articles—often lengthy and informative—taken from *Annals of the History of Computing*, an IEEE quarterly, or from the brief reports in *Computer's* "looking.back" column.

Bibliography for History

Ashherst, F.G. *Pioneers of Computing*. Frederick Muller, London, 1983.

Hodges, A. *Alan Turing: The Enigma.* Simon & Schuster, New York, 1983.

Hyman, A. *Charles Babbage, Pioneer of the Computer.* Princeton University Press, Princeton, NJ, 1982.

Katz, K. The present state of historical content in computer science texts: A concern. *SIGCSE Bulletin,* (1995).

Koffman, E.B. *Pascal: Problem Solving and Program Design.* 4th ed. Addison-Wesley, Reading, MA, 1993.

Lee, J.A.N. Those who forget the lessons of history are doomed to repeat it or, why I study the history of computing. *Annals of the History of Computing 13*, 1 (1996).

Lee, J.A.N. *Computer Pioneers.* IEEE Computer Society Press, Los Alamitos, Calif., 1995.

Metropolis, N.C., et al., (Eds). *A History of Computing in the Twentieth Century*. Academic Press, New York, 1980.

Sammet, J.E. Some approaches to and illustrations of programming language history. *Annals of the History of Computing*, (1991).

Wexelblat, R.L. *History of Programming Languages*. Academic Press, New York, 1981.

Wilkes, M.V. *Computing Perspectives.* Morgan Kaufmann, San Francisco, 1995.

What Is CS?

The unique nature of CS, with its special algorithmic way of thinking and extremely short history, has led to a diversity of opinions about its very substance. As an example, here are two strikingly conflicting quotes by two prominent computer scientists:

"Computer science has such intimate relations with so many other subjects that it is hard to see it as a thing in itself."
—M.L. Minsky, 1979

"Computer science differs from the known sciences so deeply that it has to be viewed as a new species among the sciences."
—J. Hartmanis, 1994

So, is CS a science in its own right? We feel that in a strange way both quotes are right on the mark: CS is definitely a new and important science, but its relationships with other fields like mathematics, physics, and electrical engineering are also very significant.[4] However, what is important for us is that a CS educa-

[3] The fact that these technological developments came later—so that we knew about what could be computed by algorithmic devices even before any such devices existed—is in itself a remarkable historical fact.

[4] Interestingly, CS is closely related to the life sciences too, both in taking and in giving. For example, AI research often draws upon brain research, and deep algorithmic ideas are being increasingly used in the human genome project.

tor be exposed to the wide variety of arguments and opinions on these matters, and the bibliographical items listed here are a good place to start.

In fact, there is no clear agreement even on the name of the field. In European universities, the titles of many of the relevant departments revolve around the word "informatics," whereas in the U.S. most departments are "computer science." To avoid using the name of the machine in the title (a problem that prompted Dijkstra to quip that doing so is like referring to surgery as knife science), some use the word "computing" instead. Other department names contain "information systems" or "computer studies." Another possible name for the field, which is not intended to cover the full scope of CS but, rather, its heart and basis, is "algorithmics."

There are several causes of disagreement and confusion regarding the what-is-CS question. One is the dichotomy mentioned earlier between the mathematical and engineering facets of the field. For example, the analysis of algorithms is on the mathematical side of things, whereas software engineering is on the engineering side.

Moreover, there are dichotomies within each of these two facets. The mathematical aspects of CS include not only computability and computational complexity, touching upon logic, combinatorics and probability theory, but also numerical analysis, which can be viewed as a direct outcome of the need for extremely heavy, yet accurate, computations. Interestingly, the engineering facet of the field is also dichotomic, with designing and building hardware being in many ways quite a different kind of endeavor from developing software.

As if all this were not enough, there is another problematic issue, more directly related to education. It is rooted in the dire need to make people computer literate in this age and time; hence the tremendous effort to integrate computers into education on all levels and in a wide variety of ways. This, in turn, causes computer science to be confused with computer literacy, for example, algorithms with spreadsheets, programming with word processing, and the average-case analysis of random walks with Net surfing.

One way to get an idea of what the field is really about is to inspect the various curricula proposed for university-level study. The celebrated ACM 1968 curriculum for undergraduate study, and its descendants that appeared in later years, divide CS into information structures and processes; information processing systems; and methodologies. A different kind of three-way division is given in the *Algorithmics* book, which argues that a beneficial way to view CS is by the kinds of complexity it deals with computational complexity; system, or behavioral, complexity; and cognitive complexity.

How should the what-is-CS topic be integrated into our recommended course? The multitude of opinions and approaches this topic invokes seems to call for basing this part of the course on reading assignments followed by extensive class discussion. It is interesting to get the students to talk about their own background, and the way they feel it did or did not provide a comprehensive treatment of the field. The bibliographic list for this section contains relevant items for such reading, some written by central figures in the debate.

BIBLIOGRAPHY FOR "WHAT IS CS?"

Brooks, F.P. The computer scientist as toolsmith II. *Communications of the ACM 39*, 3 (1996), 61–68.

Curriculum '68: Recommendations for academic programs in computer science. *Communications of the ACM 11*, 3 (1968), 151–157.

Dijkstra, E.W. On a cultural gap. *The Mathematical Intelligencer 8*, 1 (1986), 48–52.

Denning, P.J. et al. Computing as a discipline. *Communications of the ACM 32*, 1 1989, 9–23.

Harel, D. *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, Mass., 1987 (2nd ed., 1992).

Hartmanis, J. About the nature of computer science. *Bulletin of EATCS 53*, (1994), 170–190.

Knuth, D.E. Computer science and its relation to mathematics. *American Mathematical Monthly 81*, (1974), 323–343.

Knuth, D.E. Algorithmic Thinking and Mathematical Thinking. *American Mathematical Monthly 92*, (1985), 170–181.

Minsky, M.L. Form and content in computer science. *Communications of the ACM 17*, 2 (1970), 197–215.

Minsky, M.L. Computer science and the representation of knowledge. *The Computer Age: A Twenty Year View* (Dertouzos, L. and J. Moses, Eds.) MIT Press, Cambridge, Mass. 1979, 392–421.

A Bird's-Eye View

As mentioned earlier, we propose that beyond whatever technical knowledge students have acquired in CS, if they are planning on becoming CS educators they could benefit greatly from a general exposition of the discipline. It should be as comprehensive as possible, with depth and detail being sacrificed for scope and perspective. This overview cannot be part of the CS educator's course proposed here, if only because of its expanse; other ways of delivering it should be found, such as by individual reading or as a separate course.

Many educators have thought long and hard about how best to organize such an overview, and several authors have written books along these lines. The bibliographic list contains some of our favorites. However, we feel that no single book is wholly satisfactory for the present purposes. In fact, it is probably best to divide the desired bird's-eye view into two, in line with the divisions we've already mentioned—algorithmics and systems. The reason is that both facets, with their methodologies and techniques, are just too different to be described together.

For the algorithmic portion of the story, we (not surprisingly) recommend *Algorithmics*. As for the engineering/systems part, we are not familiar with a satisfactory book dedicated exclusively to such a survey, but good chapters on the subject appear, for example, in the books of Brookshear, and Aho and Ullman.

BIBLIOGRAPHY FOR A BIRD'S-EYE VIEW

Aho, A., and Ullman, J.D. *Foundations of Computer Science. Computer Science Press*, New York, 1992.

Biermann, A.W. *Great Ideas in Computer Science*. MIT Press, Cambridge, Mass., 1990.

Brookshear, J.G. *Computer Science: An Overview*, 4th Ed. Addison-Wesley, Reading, Mass., 1994.

Dewdney, A.K. *The New Turing Omnibus*. Computer Science Press, New York, 1993.

Goldshlager, L., and Lister, A. *Computer Science: A Modern Introduction.* Prentice-Hall International, London, 1988.

Harel, D. *Algorithmics: The Spirit of Computing*. Addison-Wesley, Reading, Mass., 1987 (2nd ed., 1992).

Pohl, I., and Shaw, A. *The Nature of Computation: An Introduction to Computer Science*. Computer Science Press, Rockville, Md., 1981.

Curricula

The topics we have discussed so far are usually of interest to a wide range of people. The field's educators, though, should also be interested in the various approaches to teaching CS, which are reflected in proposed curricula. A closer look at how computing manifests itself in the realm of education reveals three totally different directions, only one of which deals with computer *science* itself: disseminating computer literacy; using computers in teaching other subjects; and teaching CS. It is extremely important that CS educators be aware of this partition, taking care not to confuse the spirit and methods of one direction with those of the others.

Since this article concentrates on the third of these directions, we shall not say much about the first two. Still, it can be beneficial to incorporate into the CS educator's course some issues related to educating for literacy or to the use of computers in teaching. For example, a student can be asked to suggest ways of employing computers to help teach a particular topic in a different discipline.

As to CS curricula, we are interested here in both the high school level and the college/university undergraduate level.[5] Over the past years, there has been a steady evolution of undergraduate curricula in CS, the latest of which is ACM's Curriculum '91. These extensive team efforts reflect the accumulated experience and wisdom of many people; they embody interesting ideas regarding the structure and contents of computer science and how to best teach it. Given the fact that the students in our CS educator's course might eventually find themselves participating in curricula design or modification, it is important to get them to take an active role in this part of the course. Students could be asked to study proposed curricula in detail, comparing them in terms of goals, underlying principles, and structure. A class discussion could then take place where students would also report on the extent to which the recommended curricula match their own past CS studies.

On the high school level things have been somewhat slow, due in part to the lack of an adequate separation between CS and general computer literacy in high school teaching. Nevertheless, there have been efforts to establish CS curricula for high school, including one by an ACM team, and a three-year program that we have been involved in. These, too, should be discussed in the course, since some of the participating students could very well become high school teachers.

It is noteworthy that many high school teachers have never studied CS in an orderly fashion at all. As a result, some have no real feel for the algorithmic basis of CS, and are able to teach programming only, and this very often just as the coding of simple algorithms in a simple, fixed language. Consequently, additional teacher training is often needed. A course like the one proposed here could be useful for this purpose too, in which case it could be given as part of a specially tailored in-service training programs for high school teachers. Such programs have indeed been proposed, and could also be discussed in the course.

[5] We do not discuss graduate studies here; they are a topic in themselves. One difference is that teachers of graduate level CS programs typically have the technical background required to do their work (teaching skills is another matter). In any case, even some graduate level teachers could probably benefit from acquainting themselves with the topics discussed here.

BIBLIOGRAPHY FOR CURRICULA

Biermann, A.W. Computer science for the many. *Computer 27*, 2 (1994), 62–73.

Curriculum '68: Recommendations for academic programs in computer science. *Communications of the ACM 11*, 3 (1968), 151–197.

Gal-Ezer J. Computer science teachers' certification program. *Computers and Education 25*, (1995), 163–168.

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

Merrit, S., et al. *ACM Model High School Computer Science Curriculum*. ACM Press, New York, 1994.

Maddux, C.D., Johnson, L. and Harlow, L. The state of the art in computer education: Issues for discussion with teachers-in-training. *Journal of Technology and Teacher Education 1*, (1993), 219–228.

Poirot, J., et al. Proposed curriculum for programs leading to teacher certification in computer science. *Communications of the ACM 28* (1985), 275–279.

Rogers, J., et al. Computer science for secondary schools: Course content. *Communications of the ACM 28* (1985), 270–274.

Tucker, A., et al. Computing curricula 1991: A summary of the ACM/IEEE-CS joint curriculum task force report. *Communications of the ACM 34*, 6 (1991), 69–84.

The Problematics of Teaching Programming

We now get to one of the most difficult parts of all. It concerns questions such as when, to whom, how, and why—indeed, *whether*—to teach programming. This is a broad and a multisided topic, that is highly controversial but also crucial in its long-lasting influence on students. We do not get too deeply into the debate itself, however, we do make a humble attempt to put some order into the issues most widely discussed in the literature.[6]

First, we should state clearly that we take programming here in a rather broad sense, covering not only the coding act itself, but also the design of the algorithms underlying the programs and, to some extent, considerations of correctness and efficiency. To some, this interpretation of programming might be the obvious one to adopt, but experience shows the point ought to be made more explicitly.

One of the interesting questions that arise when people start talking about teaching programming revolves around "to whom." The issue extends beyond the CS boundaries: Should everyone have programming skills? Here "everyone" includes college-level students in other fields (for example, in the natural and social sciences), and perhaps even non-CS-oriented high school students. Do these people have to be able to actually program a computer or perhaps only to be more sophisticated users thereof. Does knowing how to program bring any significant added value in other kinds of problem-solving tasks?

In his 1974 paper in *American Mathematical Monthly*, Knuth writes: "It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer, i.e., express it as an algorithm."

Fifteen years earlier, in a 1959 paper in the same periodical, Forsythe wrote: "The automatic computer really forces the precision of thinking which is alleged to be a product of any study of mathematics."

In any event, most people agree that anyone studying CS ought to know how to program. Programming in the broad sense of the word is the heart of the first portion of virtually all CS programs on both the college/university level and the high school level.[7] There are exceptions—sometimes outspoken ones. Dijkstra, for example, is opposed to teaching actual programming in the first CS course. In his 1989 article, he says: "Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has not been implemented on campus so that students are protected from the temptation to test their programs." Most curricula, however, do not reflect this rather extreme position.

Now comes the widely debated issue of the *mother-tongue*, that is, the first programming language one learns. Most people feel the first language, and especially the programming paradigm students first encounter, has a significant impact on the way they will attack problems later on. Arguments are constantly made regarding the pros and cons of languages of all flavors—procedural, declarational, functional, logical, and object-oriented. The arguments are controversial, and debates on the topic become heated, giving rise to something of a culture war. The jury, however, has not yet decided.

The impact of this first language decision can be

[6] In this section we do not comment on the bibliographic items. In fact, the list itself is extremely incomplete, given the vast amount of literature on the topic.

[7] In our opinion, it is unfortunate that on the high school level most CS programs contain little more than programming, hence the need for more extensive high school curricula, such as that of ACM or the one we have been involved in, both mentioned under Curricula.

softened by familiarizing students with two or more languages from different paradigms in early stages of their CS education (even in high schools). There have even been attempts to design multiparadigmatic languages, with two or three paradigms all rolled up in one. (Due to the importance of the programming language issue, and the huge amount of material published on it, we devote a special bibliographic list to it.)

Bibliography for Programming Language Issues

Abelson, H., and Sussman, G.J. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

Baranauskas, M.C.C. Observational studies about novices' interaction in a Prolog environment based on tools. In *Proceedings of the 7th Interna tional PEG Conference*, 1993, 537–549.

Bayman, P., and Mayer, R.E. Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology 80*, 3 (1988), 291–298.

Cooper, D. *Oh! Pascal!* W.W. Norton, New York and London, (3rd ed.), 1993.

Fleury, A.E. Students beliefs about Pascal programming. *Journal of Educational Computing Research 9,* 3 (1993), 355–271.

Gal-Ezer, J. A pre-programming introduction to algorithmics. *Mathematics and Computer Education 30,* 1 (1996), 61–69.

Joy, M., and Matthews, S. Some experience in teaching functional programming. *International Journal of Mathematical Education in Science and Technology 25*, 2 (1994), 165–172.

Lee, A., and Pennington, N. The effects of paradigm on cognitive activities in design. *International Journal of Human-Computer Studies 40* (1994), 577–601.

Savitch, W.J. *Pascal, An Introduction to the Art and Science of Programming*. (3rd ed.) Benjamin/ Cummings, New York, 1991.

Wexelblat, R.I. The consequences of one's first programming language. *Software—Practice and Experience 14* (1981), 733–740.

Now, there have always been specific concepts within programming that are hard to teach, especially in precollege levels. They include the idea that an algorithm, or program, is fixed yet is supposed to deal with many different inputs of varying sizes; the very notion of an assignment statement; control structures, such as conditionals and repetitions, and so on. In particular, recursion is considered to be one of the universally most difficult concepts to teach. Notions that transcend the programming act itself, and are relevant to algorithms in general, can be difficult to teach too, such as upper and lower bounds on the computational complexity (such as running time) of a program and, by extension, on the inherent complexity of the algorithmic problem at hand.

Besides the language issue and the challenge of getting difficult notions across, the very insistence that programming is more than mere coding has its price, too. Students must understand they are not in this business only to get their programs to run, to paraphrase the title of a well-known paper. They must be taught and coached in the algorithmic way of thinking, and must come to grips with the difficulties and pitfalls it entails. This is a broad and sweeping challenge for an educator, addressed by many authors, and we recommend it be treated in detail in our course.

Bibliography for the Problematics of Teaching Programming

Du Boulay, B. Some difficulties of learning to program. *Journal of Educational Computing Research 21*, 1 (1986), 57–73.

Dijkstra, E.W. On the cruelty of really teaching computing science. *Communications of the ACM 32*, 12 (1989), 1398–1414.

Hancock, C. Context and creation in the learning of computer programming. *For the Learning of Mathematics 8*, 1 (1988), 18–24.

Headrick, R.W. Structured programming complexity revisited. *Computers and Education 20*, 4 (1993), 315–316.

Mayer, R.E. A psychology of learning BASIC. *Communications of the ACM 22,* 11 (1979), 589–593.

Mayer, R.E., Dych, J.L., and Wilberg, W. Learning to program and learning to think. *Communications of the ACM 27*, 9 (1986), 605–610.

Murname, J. To iterate or to recurse? *Computers and Education 19*, 4 (1992), 387–394.

Roberts, E.S. *Thinking Recursively.* John Wiley, New York, 1986.

Saj-Nicole, A.J., and Soloway, E. But my program runs! *Journal of Educational Computing Research 2*, 1 (1986), 95–125.

Wirth, N. Program development by stepwise refinement. *Communications of the ACM 14*, 4 (1971), 221–227.

Tools and Methods for Teaching

In recent years, educational activities are becoming increasingly dependent on computer-based teaching aids. Surprisingly, CS education lags behind in this kind of computerization. While students on all levels will obviously use a computer to practice programming, there is not enough good courseware to help

teach general CS topics (including programming). Nevertheless, there have been several proposals.

For teaching Turing machines and other kinds of automata, there are animation-based software packages, such as *Turing's World* by Barwise and Etchemendy. Actually, a good way of exploiting computers in teaching all kinds of CS topics is to use visualization and graphic animation in their various guises.

Other methods proposed to help teaching computing include programming based on case studies, and induction-based algorithmic design that incorporates correctness considerations. See the articles by Linn and Calancy, and by Manber, respectively. One of the more interesting proposals for teaching precollege programming is Pattis' *Karel the Robot*.

As far as the CS educator's course is concerned, this is the most open-ended of our topics. First, as far as we know, it has not yet been systematically looked into; CS educators are not that concerned these days with using tools. When they are, it is mainly in exploiting the Internet for distance teaching, for distributing homework assignments, and so on. Second, it is far from clear how this topic should be incorporated into our course. Instructors no doubt have their own favorite tools and their own methods and techniques—some of which are no doubt unpublished—and there is no reason why they should not discuss them in the course as they see fit. Students of the course might use these in their own teaching activities later on in their careers, and might even find themselves involved in refining them or developing new ones themselves.

Bibliography for Tools and Methods for Teaching

Barwise, J., and Etchemendy, J. *Turing's World*. CSLI Publications, Stanford, CA, 1993.

Brown, M.H. Zeus: A system for algorithm animation and multi-view editing. In *Proceedings of Visual Languages '91*, (Oct. 1991), 4–9.

Halewood, K., and Woodward, M.R. A uniform graphical view of the program construction process: GRIPSE. *International Journal of Man-Machine Studies 38* (1993), 805–837.

Jackson, D.F., et al. The design of software tools for meaningful learning by experience. *Journal of Educational Computing Research 9* (1993), 413–443.

Linn, M.C., and Calancy, M.J. The case for case studies of programming problems. *Communications of the ACM 35*, 3 (1992), 121–132.

Manber, U. Using induction to design algorithms. *Communications of the ACM 31*, 11 (1988), 1300–1313.

Pattis, R.E. *Karel the Robot: A Gentle Introduction to the Art of Programming.* Wiley, New York, 1981, (2nd ed., 1995).

Conclusion

In the more operational part of this article we have made suggestions about what to include in a course for CS educators (even stating that some of this ought to be of interest outside the CS community). However, we have not been sufficiently explicit about how such a course should be delivered, how the hours ought to be divided, and so forth. This is deliberate, as we do not feel there is enough accumulated experience in these matters to make clear-cut and stringent recommendations, and in any case we feel that instructors should shape the course as they see fit.

We do have some experience in implementing the ideas discussed here. Gal-Ezer has given such courses at Tel-Aviv University and The Open University, both in Israel. The course uses a specially prepared study guide, and a reader containing around 30 papers. We found it very useful to have students prepare lengthy term papers, and we are also planning to have a number of invited lecturers, in the hope that both what they say and how they say it will be of value to the students.

One of the main lessons we learned from teaching the material was that students must have an appropriate CS background. We cannot stress this statement enough. For example, one student in class was from electrical engineering, another's sole connection to computing was via her use of computers in general education, and a third's CS knowledge was 25 years old. These students simply did not fit in.

In summary, although our recommendations are somewhat incomplete, we hope our work will help in initiating such courses and in deciding on their structure and contents. C

Judith Gal-Ezer (galezer@cs.openu.ac.il) is Head of the Department of Mathematics and Computer Science at The Open University of Israel, Tel-Aviv. Part of this work was done during a sabbatical stay at the Weizmann Institute of Science.

David Harel (harel@wisdom.weizmann.ac.il) is Dean of the Faculty of Mathematical Science at the Weizmann Institute of Science, Rehovot, Israel. Part of this work was done during a visit to Lucent Technologies, Bell Labs.