The Education Column

BY

Juraj Hromkovič

Department of Computer Science
ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
juraj.hromkovic@inf.ethz.ch

Demystifying coding for schools — what are we actually trying to teach?

Tim Bell
University of Canterbury, NZ
tim.bell@canterbury.ac.nz

Abstract

As computer science enters school curricula around the world, many teachers are having to teach the topic of computer programming for the first time. For those who have little experience in the area it can be daunting, while those who are experienced programmers may be so familiar with the subject that it is hard to see what is difficult about it! This article explores the reality of what we need to be teaching in schools, and considers what the essence of the topic is.

1 Introduction

The idea of teaching computer science at all levels of the school system is gradually being adopted around the world (see for example [10, 16]), with the subject being introduced in the first year of school [6, 9].

Such changes are not usually driven by simply wanting to teach young children to be programmers, but to address broader issues such as helping students to be informed citizens in a digital world [8], and to reflect this, the subject usually has a broader name such as "computational thinking" or "computing". However, "coding" is becoming an important component in new curricula, and is a catchword for those promoting the discipline [3]. For those new to programming (especially established school teachers who have never programmed themselves), the prospect of teaching "coding" can be daunting. For those who are experienced programmers, it can be tempting to impose their own experience (perhaps as a self-taught programmer in their youth, or working at a professional level) onto the curricula being developed for use in schools.

When teaching a subject the focus can easily end up on details (such as print statement formats and while loop conditions), and these are indeed needed to prescribe a plan for teaching, but we need to stay aware of the big picture — what

are the key concepts that students should take away from having learned about programming? What do we want them to remember a year later, or five years later? These are things that may be obvious to those who are heavily involved in computer science as a profession, but can be a great mystery to those from outside our culture who are having to prepare to teach this new topic.

This article reflects on what we really want students to take away from programming courses, and especially on what teachers will need to be aware of to achieve this. Some of the ideas discussed are misconceptions that we have commonly encountered while helping teachers to deliver the new subject, and others are ideas that may be so obvious to experienced computer scientists that we forget to articulate them to people who are new to our culture.

2 Teaching Programming

This section considers a number of issues that come up when teaching programming in schools, and looks at how they relate to the bigger picture of how programming is used in practice.

Learning to program isn't the same as learning to teach programming. Understanding the subject is clearly important, but to teach effectively there are many ideas that can be brought to bear on teaching programming, which even the most experienced programmer won't necessarily know about. Some ideas about teaching programming are well established, and others are the subject of great debate [14] — one particularly hotly debated issue is whether we teach objects first or later [1]. There are also various ideas about what is the best programming language for learners [17]. There is evidence that students can learn better working in pairs [18], and that it can be useful to use Parson's problems [13] while teaching programming. Ideas around teaching programming to young students are still being developed, as it is relatively recent that this has been done at scale in typical classrooms with typical teachers, rather than specialist clubs and outreach programs.

There are many books on how to program, but relatively few books on how to *teach* programming (although a few are now appearing e.g. [2, 11]). Because programming at all levels in schools is a new phenomenon for most countries, we are still at the early stages of research, and much of the knowledge about this is coming through conferences and journals that aren't where teachers would look for guidance.

Programs are used everywhere. A variety of terms, such as "app", "application", "code", "solution", "software" and "firmware", are used to refer to pro-

grams in various contexts, and a lay person may not realise that these things are all programs. Even simple-looking devices like a burglar alarm, WIFI router, or ticket vending machine may not be recognised as being based on a program, let alone devices like a smartphone or smart watch, which typically run many dozens of programs. This can be overcome to some extent when teaching the basics of programming by mentioning examples from everyday life when simple constructs are taught; for example, "count = count +1" might be used in a fitness tracking device to add one to the number of steps taken, or "if margin < 0" might be used in a word processor to determine if a value is out of range. This will help students to appreciate that what appears to be a meaningless exercise, possibly in an environment that doesn't look like anything used commercially, is actually the basis of many different forms of "program" that people develop for everyday use. And to make things even more interesting, a programming language itself is implemented by a program!

Meaningful experiences can help to understand jargon. A key outcome of learning programming and other computer science concepts early is that students become familiar with technical terms in a meaningful context. The new topics being introduced to schools come with a lot of jargon — words like "algorithm", "binary" and "coding" can make teachers genuinely fearful, and it's important to overcome this. Students may see these words as a lot of unnecessary jargon to memorise, and may be put off by their teacher's confusion with the terminology. We have found that it's effective to engage teachers new to the subject by first *using* the idea (such as showing an algorithm to calculate the checksum in a product barcode), and *then* labelling it (in this case distinguishing the "algorithm" from a program that implements the algorithm, and of course introducing the term "checksum") so that the mysterious terminology appears *after* the concept has been brought down to size.

For students, rather than memorising definitions of jargon like "debug", "algorithm", and so on, teachers can use the terms to label the experiences and tools that students are engaging with. If a student has regularly implemented even the simplest programs, they will have had to track down errors in it (debug), and by implementing even simple algorithms (like drawing a regular polygon in a turtle-based language, which can be acted out away from the computer and then implemented on a digital device) they can start to distinguish between concepts like an algorithm and a program. If the teacher uses this terminology regularly and appropriately when referring to something the student is doing, it will acquire an authentic meaning for them. Once when using this approach a young student asked "Why do you use such big words for such simple ideas?" Every discipline has its jargon, and allowing teachers and students to engage with the ideas first

and *then* labelling them can help to have them seen as approachable concepts.

Programming is more about communicating with humans than with machines. Writing a computer program could be described as giving instructions to a computer, but much of the discipline of programming is around communication with humans via the computer, rather than the computer being the final recipient of what is written. The programmer is communicating to two sets of people: the users, and future programmers.

Almost every program involves human interaction, and the user experience is notoriously poor in many digital systems. Introductory programming is an excellent place to sensitise students to the importance of thinking about the user. Many beginners overlook giving any instructions to the user (e.g. which keys to press for a game, or what range of values is expected for an input), and the output can be jarring or confusing (e.g. a rather blunt "You are wrong" in response to a quiz answer, or an uninformative "Out of range, try again" for an input value).

The other important audience for a program is the next programmer who needs to work on it. Trying to read someone else's program helps a student to appreciate comments that explain what is happening at just the right level of detail, variable names that are unambiguous, and layout that follows conventions, so they can focus on the content and not the form. The future programmer may even be the original programmer, trying to make sense of what they wrote a year ago — or perhaps even a day ago!

In both cases it can be useful for a student to swap their program with a neighbour and see if they can figure out how to use it, or look at the code and work out how it works. Blackwell [4] points out that "many skills of a professional programmer are related to social context rather than the technical one", and even simple introductory programming environments can help students develop skills for this social context.

Programming integrates well with other subjects. Rather than teaching programming in isolation, it can be integrated with other areas of the curriculum [12], and this is particularly natural at primary school where the same teacher may be teaching the different subjects. For example, at a junior level students might learn about odd and even numbers, and writing a program to print the odd numbers forces them to articulate the meaning. At a more senior level, students might learn a definition of prime numbers, and again articulate it as a program that demonstrates their understanding. Turtle-graphics languages such as Scratch or Logo also naturally use mathematical concepts like positive and negative numbers, coordinates, angles, and other concepts from geometry.

Many other subjects can also be integrated: literacy becomes important if the

program is telling a story or providing an understandable textual interface, music can be represented through programming, and physical fitness can be involved in acting out coordinate based instructions. These examples just scrape the surface of many examples of integrated learning that we have observed.

Programming is a skill that demands practice. This is related to the pedagogy of programming: some students (and teachers) have a model of learning a subject that there are some facts to learn, and once you know them you are competent in the subject. This view relates to working at a low level of Bloom's taxonomy, but programming is a very creative activity, where the programmer is generally operating at a high level of the taxonomy. This misconception can be reflected in wanting to learn programming by reading a book, or perhaps attending a short course, and hoping that from then on one is able to program.

A better model is to think of programming more like fitness training or learning a foreign language; regular exercises are far more important that trying to learn it in a hurry and hoping to know it some time later. Robins [15] introduces a "Learning Edge Momentum" model, which highlights that in programming it is particularly important to understand the basic concepts, as later concepts will make no sense if one basic concept is missing (for example, objects won't make sense if the concept of type isn't understood, or *for* loops will be confusing if the role of a variable isn't clear).

To support teachers new to programming, we encourage them to do small regular exercises to keep up their "fitness level"; this also applies to students, as it is much better to do a little regularly with programming, rather than, say, a short segment of a course where they write one large program. Of course, working on such rudiments needs to be balanced with more motivational large projects, but large projects alone can be frustrating if students don't have the skills to draw on.

Technology changes quickly, but the basics don't. A concern that teachers often raise is that the computing is changing so fast that they are worried that even if they learn to teach the subject, their knowledge will go out of date quickly. While it's true that technology keeps changing, the basic ideas around computer science and programming don't change so rapidly. Since teaching is more about laying foundations, focussing on the basics is appropriate.

For example, a new introductory language might become popular, but chances are it has very similar ingredients to ones that already exists. In principle, any language that is Turing-complete is sufficient to fully control any conventional computing device, and so from an educational point of view we don't need to be concerned that learning to program in one language will be of no use for learning to program a new one in the future. In fact, the key is that we are teaching

programming, not a particular language. A cue for students is to call the subject *programming*, not Python, or Java, or Scratch.

The Böhm-Jacopini theorem [5] underpins the idea that teaching programming should cover three basic constructs: sequence, selection and iteration. In principle, these are the only control structures needed to program any computing device. In addition to these control structures, a language needs input, output and storage (variables) to give sufficient scope to program anything that is computable. An important consequence of this observation is that "toy" languages like Scratch are actually just as capable as the most advanced languages, or at least, they capture the key logic needed to make things happen on a computational device, and the differences between languages highlight features that make programming more convenient for particular applications, rather than some fundamental new capability. Also, not all introductory programming systems are Turing-complete — for example, the popular "Beebot" and "ScratchJr" teaching tools focus on sequence, but this can be seem as an important stepping stone that is aimed at an appropriate cognitive level for young students.

Blackwell [4] reflected on what programming is from a cognitive point of view. As well as highlighting the boundaries of programming (for example, writing HTML or setting a microwave oven isn't programming because the system isn't Turing-complete), Blackwell notes that programming involves more than just writing code; the programmer must identify requirements, derive a specification, design how it will work, code the commands, and debug it to be sure that it will function as intended. He highlights that programming reflects a loss of direct manipulation: the programmer must anticipate what will happen in advance (e.g. all combinations of user input), and account for these before the program is run. These skills can be exercised in the simplest of programming systems, and in this light, "coding" is a relatively small part of the whole process of writing a successful program.

Along with this is the need for persistence; writing a program is easy, but debugging it is the real challenge, and persistent work is required to make sure the program works properly, rather than making do with something that is almost correct [7].

Teaching core concepts well is better than covering every possible technique. When designing computing courses, we need to be careful to focus on quality rather than quantity. There are endless programming languages, environments (mobile, web, desktop, server) and toolkits that could be taught, but the goal should be to provide students with a good grounding, and inspire them to learn more, rather than overwhelm them with so many topics that it has the effect of putting them off the subject. The same applies to teachers: if a teacher is pres-

sured to deliver a curriculum that beyond what they have had the professional development for, the students may end up getting a poor experience of learning to program, and may go away with the impression that programming is difficult and confusing.

Computer science is much more than programming. While this article has focussed on programming, it's important that computing courses take a much broader view of the discipline. Programming enables us to make things happen, but there is a lot to know before we can write effective and efficient programs, which is informed by the field of computer science. There are intriguing ideas in algorithm design — some algorithms are staggeringly more efficient than others for solving the same problem, while other problems have no known programmable solution that will work in a reasonable amount of time. Programs operate on data, and how that data is represented has an effect on how effectively it can be processed. Computers need to communicate with each other, and the programs that do this need to follow appropriate protocols to make sure that works well. When computers communicate with humans, the way the operate needs to be informed by a basic understanding of how humans think and perceive. And beyond the basics there are so many more questions: can we imitate human intelligence? Can we simulate processes from the physical world? Or can we create new virtual worlds? Are there things we could implement, but shouldn't?

These are all questions that the discipline of computer science is concerned with, and students can engage with these ideas even before they write programs e.g. using non-computer based activities such as "CS Unplugged" (`csunplugged.org`). In fact, it is valuable that they have such experiences because for some students this will provide the motivation to learn to program; while some may enjoy programming for its own sake, others will be more motivated if they can see how it can be applied, and that there are tools and concepts beyond programming that are exciting and relevant to our human world.

The epigram that "computer science is no more about computers than astronomy is about telescopes" captures this idea when applied to programming; it's a tool that is normally used to make things happen in a digital world, but it is a means, not the end in itself. Computer science courses often start (and sometimes end!) with programming, and this can give an inaccurate message to students of what the discipline is about. By keeping students aware of the bigger picture, we are more likely to capture their interest and give them a balanced view of what matters.

3 Conclusion

We are at an exciting point in education, where many countries are adding a whole new subject to their curriculum that hasn't been taught before. Empowering teachers to deliver this with enthusiasm is important, and the ideas shared above are intended to help us think about approaching this change in a way that neither underplays how significant the change is, but also doesn't make it so overwhelming that the value of the change is lost because schools are unable to deliver it effectively.

References

[1] Owen Astrachan, Kim Bruce, Elliot Koffman, Michael Kölling, and Stuart Reges. Resolved: objects early has failed. *ACM SIGCSE Bulletin*, 37(1):451–452, 2005.

[2] Phil Bagge. *How to Teach Primary Programming Using Scratch*. The University of Buckingham Press, 2015.

[3] Tim Bell. Surprising Computer Science. In Andrej Brodnik and Jan Vahrenhold, editors, *8th International Conference on Informatics in Schools: Situation, Evolution, and Perspective*, pages 1–11. Springer, 2015.

[4] A Blackwell. What is programming. In *14th workshop of the Psychology of Programming Interest Group*, pages 204–218, 2002.

[5] Corrado Böhm and Giuseppe Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–371, may 1966.

[6] Neil C C Brown, Sue Sentance, Tom Crick, and Simon Humphreys. Restart: The Resurgence of Computer Science in UK Schools. *Trans. Comput. Educ.*, 14(2):9:1–9:22, jun 2014.

[7] Quintin Cutts, Emily Cutts, Stephen Draper, Patrick O'Donnell, and Peter Saffrey. Manipulating mindset to positively influence introductory programming performance. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 431–435. ACM, 2010.

[8] Caitlin Duncan, Tim Bell, and Steve Tanimoto. Should your 8-year-old learn coding? In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education - WiPSCE '14*, pages 60–69, New York, New York, USA, nov 2014. ACM Press.

[9] Katrina Falkner, Rebecca Vivian, and Nickolas Falkner. The Australian Digital Technologies Curriculum: Challenge and Opportunity. *Proceedings of the Sixteenth Australasian Computing Education Conference (ACE2014)*, pages 3–12, 2014.

[10] Walter Gander, Antoine Petit, Gérard Berry, Barbara Demo, Jan Vahrenhold, Andrew McGettrick, Roger Boyle, Avi Mendelson, Chris Stephenson, Carlo Ghezzi, and Others. Informatics education: Europe cannot afford to miss the boat. `http://europe.acm.org/iereport/ie.html`, 2013.

[11] Mark Guzdial. Learner-Centered Design of Computing Education: Research on Computing for Everyone. *Synthesis Lectures on Human-Centered Informatics*, 8(6):1–165, nov 2015.

[12] Irene Lee, Fred Martin, and Katie Apone. Integrating Computational Thinking Across the K–8 Curriculum. *ACM Inroads*, 5(4):64–71, dec 2014.

[13] Dale Parsons and Patricia Haden. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*, pages 157–163. Australian Computer Society, Inc., 2006.

[14] Jody Paul. "What First?" Addressing the Critical Initial Weeks of CS-1. In *Proceedings. Frontiers in Education. 36th Annual Conference*, pages 1–5. IEEE, 2006.

[15] Anthony Robins. Learning edge momentum: A new account of outcomes in CS1. *Computer Science Education*, 20:37–71, 2010.

[16] Josh Tenenberg and Robert McCartney. Editorial: Computing Education in (K-12) Schools from a Cross-National Perspective. *Trans. Comput. Educ.*, 14(2):6:1–6:3, jun 2014.

[17] David Weintrop and Uri Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In *Proceedings of the 14th International Conference on Interaction Design and Children*, pages 199–208. ACM, 2015.

[18] Linda Werner and Jill Denning. Pair programming in middle school: What does it look like? *Journal of Research on Technology in Education*, 42(1):29–49, 2009.