The Education Column

BY

Juraj Hromkovič

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

Algorithmic Thinking from the Start

Juraj Hromkovič juraj.hromkovic@inf.ethz.ch
Tobias Kohn tobias.kohn@inf.ethz.ch
Dennis Komm dennis.komm@inf.ethz.ch
Giovanni Serafini giovanni.serafini@inf.ethz.ch

Department of Computer Science, ETH Zurich, Switzerland

Abstract

Programming education is about introducing the language and way of thinking of computer science itself, and not only about teaching a specific programming language. We are actively involved in reaching out to teachers at primary, at lower and at higher secondary schools, and in training them to successfully teach programming (and computer science in a broader sense) to students while aiming at getting to the core of programming education as early as possible; i. e., avoiding a lengthy introduction to syntactical details, but teaching algorithmic thinking. In this paper, we describe a few corner stones of our approach towards teaching computer science with the above points in mind.

1 Our Approach in a Nutshell

A sustainable computer science education must by its very definition not solely focus on teaching a specific programming language or using specific software on a computer. A programming language in school is merely a means to teach the basic concepts of computer science such as the modular design of algorithms and algorithmic thinking. In this paper, we outline our approach towards teaching programming in primary school as well as lower and higher secondary school using a spiral curriculum. Our main objective is to point out how to teach algorithmic thinking already at the beginning of programming education. In particular, an introduction to programming is not necessarily a precursor to teaching algorithmic thinking, but rather provides the very means to teach algorithmic thinking.

Algorithmic thinking and spiral curriculum. We consider *algorithmic thinking* and *computational thinking* as two equivalent and interchangeable ways to express the same concept, even though they were first introduced in different eras (*algorithmic thinking* has been in use for several decades, while *computational thinking* is a more recent term). We believe that a sound definition of algorithmic thinking should by rooted in the scientific core of computer science as proposed by Aho:

> *"We consider computational thinking to be the thought process involved in formulating problems so their solutions can be represented as computational steps and algorithms." [1]*

This leaves us with the question of how to successfully teach algorithmic thinking. With the idea of using a spiral curriculum, we actually build upon sound and well-known didactic principles.

In the early 1960s, the Harvard psychologist and father of the cognitive revolution Jerome Bruner argued the value of pedagogically and didactically proper design of early teaching:

> *"The early teaching of science, mathematics, social studies, and literature should be designed to teach these subjects with scrupulous intellectual honesty, but with an emphasis upon the intuitive grasp of ideas and upon the use of these basic ideas." [3]*

Moreover, Bruner pointed out the inherently iterative nature of learning and postulated the need for a so-called *spiral curriculum* that avoids to introduce a large number of new key concepts at once. Learning is, in his eyes, a carefully designed game between intuition, a stepwise rising knowledge, and continuously increasing abstraction skills of the students.

> *"A curriculum as it develops should revisit these basic ideas repeatedly, building upon them until the student has grasped the full formal apparatus that goes with them." [3]*

We aim at introducing students to algorithmic thinking very early while teaching them how to program. Moreover, the students should learn from the beginning to design their programs in a proper and structured way. In the spirit of a spiral curriculum, we start at primary school, continue this process at lower secondary school, and carry it on later at higher secondary school. In this paper, we exemplify our approach by showing how we introduce and gradually extend the notion of a loop.

Implementation. Our classes are all taught in Logo and Python, respectively, since these languages allow a focus on concepts rather than the language itself [7]. In Logo, we can start with simple commands that directly induce a visual feedback (by moving a turtle on the screen), and consequently testing and correcting programs becomes available already for beginners and children. Python is also well-suited for classroom use, especially since it includes *turtle graphics* as well, and thus allows for a smooth transition from Logo to Python. Moreover, `TigerJython` [2] offers a variable-free looping construct as in Logo. This supports our spiral teaching approach, and we have gained very positive experience in that our students take loops as a natural concept and exhibit less problems writing programs with loops [7].

Without having a variable-free looping construct, we can either choose to introduce variables prior to loops, leave the occurring variables as a bit of magic in the code, or introduce loops together with variables at the same time. All three approaches are unsatisfactory, as pointed out earlier [7]: The first contrasts with the desire to use loops as early as possible in the curriculum as one of the starting points of abstraction. The second one comes with the danger of leaving students with the feeling of not being able to fully understand the programs they are to write. Finally, the third one comes with a steep learning curve as the students have to master at least two concepts simultaneously. Having the concept of a loop introduced in its simplest form (i. e., the `repeat`-loop with a fixed number of iterations), even children (ages 9 to 12) can be enabled to grasp this concept, before it is revisited in more and more complex ways.

Programming environments. Our class activities in Logo rely on a redesigned version of the open source Java application `XLogo` [9]. The main objectives of the `XLogo4Schools` programming environment [13] focus on reducing the extraneous cognitive load of the learning environment by rethinking the user interface as well as by considerably improving both the quality of the graphical output and the general performance of the application. Last year alone, some 1200 students of primary schools in Switzerland were introduced to programming with `XLogo4Schools`.

`XLogo online` is a browser-based, single-page programming environment for schools intended to be the successor of `XLogo4Schools`. It relies on modern, state-of-the-art web technologies, offers a reactive design allowing for running it on a broad set of devices, and can be used in both an online and an offline mode. `XLogo online` was conceived and developed as part of a Master's thesis [11]. We plan to make it publicly available within a couple of weeks.

For Python programming, we have also created a dedicated development environment, suited for students and classroom teaching: `TigerJython`. Apart from the simple user interface, it includes a debugger and enhanced compiler messages, both specifically directed at the novice programming student. Like `XLogo4Schools`, the environment is freely available online [2].

Turtle graphics. Turtle graphics provides an excellent model of a programmable machine as the current state and the properties of the turtle are directly observable, which serves as the aforementioned direct feedback. Besides that, the basic instructions for movement fit well into the students' mental models [7], and the turtle serves as a metaphor for introducing various new concepts. Defining a new function, for instance, can be motivated as "teaching the Turtle a new word" [10]. Yet, the students learn that the communication with the computer (the turtle, respectively) needs to be precise. Since computers have no intellect, there is no room for interpretation.

2 Modular Design of Algorithms

Students start out with a very limited set of words, each standing for a specific instruction given to the machine. Assume that the students already know how to move the turtle forward (`fd`) and backward (`back`) on a straight line, how to raise (`pu`) and lower (`pd`) the pen, how to rotate it (`lt` and `rt`) as well as how to iterate over a sequence of instructions for a predefined number of times (the aforementioned `repeat`-loop); these commands constitute the current "vocabulary" of the turtle. Teaching these basic building blocks of Logo does not take more time than four lessons.

In subsequent activities, the students learn how to develop a table that consists of rows of identical objects in a proper modular way [8]. More specifically, while the program for drawing a house is already available in the teaching material [4, 5, 6], the students are expected to consequently apply the concept of modular design, which they have practiced before. The steps are to

- identify the next shape or pattern they can systematically reuse,
- write a sequence of instructions for drawing it,
- give this subprogram a name, and
- test and iteratively improve the code until the solution meets the assignment.

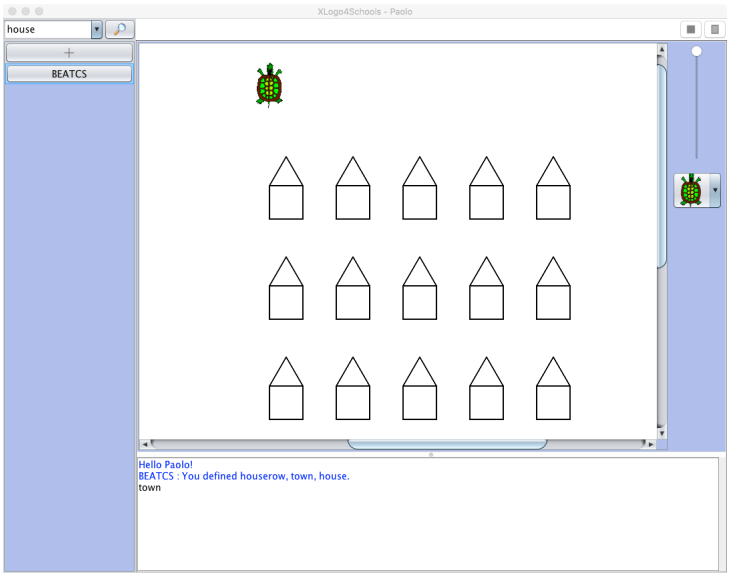

Figure 1: A small town that consists of 15 houses in `XLogo4Schools`.

To this end, the primary-school students are given the following program accompanied by an exercise, which asks them to study the effects of each command in detail.

```
to HOUSE
  rt 90
  repeat 4 [fd 50 rt 90]
  lt 60 fd 50 rt 120 fd 50 lt 150
end
```

Next, they are told how to design the following program `HOUSEROW`.

```
to HOUSEROW
  repeat 5 [HOUSE rt 90 pu fd 50 lt 90 pd]
end
```

This program uses `HOUSE` as a subprogram. Here, the most difficult task is to position the turtle in such a way that, after each iteration, the new house is drawn at the correct coordinates.

Finally, the students are asked to use the modular approach in order to draw the town that consists of multiple streets. This way, the students learn how to extend the vocabulary of the turtle step by step with more complex programs. The crucial observation is that the overall complexity is hidden in the smaller subprograms. The students learn that modular development is a systematic and efficient problem-solving strategy. Moreover, they experience

that subsequent changes in a basic module of a properly developed complex program require no or very limited additional programming effort.

3 Spiral Curriculum

As mentioned above, our goal is to spend as little time as possible on syntactical details of the programming language used, especially at the beginning. In fact, this section exemplifies the notion of repeatedly building on, and expanding the understanding of, one single programming construct throughout the curriculum. Students are not exposed to the full syntax from the beginning, but expand their knowledge of one construct as they progress throughout the curriculum. The focus here is not on the form of a loop, but rather on how it can be used to solve specific problems, and simplify the students' programs.

Both `XLogo4Schools` and `TigerJython` allow us to introduce and apply the concept of a loop without having to even mention variables. The following programs show implementations of drawing squares (as already used as part of the program `HOUSE`) first in `XLogo4Schools` and then `TigerJython`, which is one of the first tasks students are confronted with.

```
repeat 4 [ forward 100 left 90 ]
```

```
repeat 4:
    forward(100)
    left(90)
```

At a later stage, we revisit loops after having introduced the concept of parameters (variables whose values are not changed during runtime); the following two programs give examples.

```
to polygon :n
    repeat :n [ forward 100 left 360/:n ]
end
```

```
def polygon(n):
    repeat n:
        forward(100)
        left(360/n)
```

Yet later, either at lower or at the beginning of higher secondary school, and after having generalized parameters to variables, we can use loops in a more general way, by first using a variable inside a `repeat`-loop, and finally `while`-loops. This way, the concept of loops gets revisited again and again with increasing both complexity and ability of the students.

```
x = 30
repeat 20:
    forward(x)
    left(90)
    x += 10
```

```
x = 30
while x < 250:
    forward(x)
    left(90)
    x += 10
```

Such spiral approaches combined into a spiral curriculum proved to be very precious in order to keep the students interested all along the way while simultaneously focusing on teaching problem-solving strategies of increasing complexity and continuously extending the programming abilities of the students.

4 Conclusions

Algorithmic or computational thinking has become an essential part of any comprehensive general education. To succeed, it is vital that we start with algorithmic thinking early on, and help the students advance towards, ultimately, mastery of the subject.

Instead of a sequential curriculum, starting with programming, and then moving on to algorithmic thinking, we propose an interleaved approach, a spiral curriculum. Programming is not seen as a mere prerequisite for intermediate and advanced topics in computer science, but rather as an opportunity to start teaching the core principles of algorithmic thinking from the beginning on. An example of algorithmic thinking in early programming is modular design as presented in this article.

References

[1] Alfred V. Aho. Computation and Computational Thinking. *The Computer Journal*, Volume 55 Issue 7:832–835, Oxford University Press, 2012.

[2] Jarka Arnold, Tobias Kohn, and Aegidius Plüss. `http://www.tigerjython.ch`. Last visited on January 30th, 2017.

[3] Jerome S. Bruner. *The Process of Education.* Harvard University Press, revised edition, 1976.

[4] Heidi Gebauer, Juraj Hromkovič, Lucia Keller, Ivana Kosírová, Giovanni Serafini, and Björn Steffen. Programmieren mit LOGO. http://abz.inf.ethz.ch/wp-content/uploads/unterrichtsmaterialien/primarschulen/logo_heft_de.pdf.

[5] Heidi Gebauer, Juraj Hromkovič, Lucia Keller, Ivana Kosírová, Giovanni Serafini, and Björn Steffen. Programming in LOGO. http://abz.inf.ethz.ch/wp-content/uploads/unterrichtsmaterialien/primarschulen/logo_heft_en.pdf.

[6] Juraj Hromkovič. *Einführung in die Programmierung mit LOGO – Lehrbuch für Unterricht und Selbststudium*. Springer, 3rd edition, 2014.

[7] Juraj Hromkovič, Tobias Kohn, Dennis Komm, and Giovanni Serafini. *Combining the power of Python with the simplicity of Logo for a sustainable computer science education.* In *Proceedings of the 9th International Conference on Informatics in Secondary Schools (ISSEP 2016)*, volume 9973 of LNCS, pages 155–166, Springer-Verlag 2016.

[8] Juraj Hromkovič, Tobias Kohn, Dennis Komm, and Giovanni Serafini. Examples of algorithmic thinking in programming education. *Olympiads in Informatics* 10:111–124, 2016.

[9] Loïc Le Coq. xLogo. http://xlogo.tuxfamily.org/. Last visited on January 30th, 2017.

[10] Seymour Papert. *Mindstorms*. Basic Books, 2nd edition, 1993.

[11] Jacqueline Staub. *xLogo online – a web-based programming IDE for Logo*. Master's Thesis, ETH Zurich, 2016. https://e-collection.library.ethz.ch/view/eth:49742?lang=en.

[12] John Sweller. Cognitive load theory. Volume 55 of *Psychology of Learning and Motivation*, pages 37–76. Academic Press, 2011.

[13] Marko Zivković. XLogo4Schools. http://sourceforge.net/projects/xlogo4schools/. Last visited on January 30th, 2017.