If you have parents, you can learn recursion.

Matthias Hauswirth
Università della Svizzera italiana
matthias.hauswirth@usi.ch

Abstract

Recursion is a common phenomenon in nature and a basic building block of computation, however, recursion is rarely taught in schools. We argue that, when approached the right way, recursion can and should be taught to every child.

1 Not Motivating Recursion

If we want to teach recursion in schools, we need to not only explain how recursion works, but we also need to motivate its purpose as a computational concept. We need to explain recursion in a way that is meaningful to students, and in a way that relates to students' concrete experiences.

1.1 Recursion devoid of purpose

A prevalent example used when introducing novices to recursion is the Russian Matryoshka doll (Figure 1), where each hollow wooden doll contains a similar but

Figure 1: Playful recursion: matryoshka dolls
Photo in the public domain (CC0 Creative Commons)

smaller version of itself. Other motivating examples are more poetic in nature, such as the Swiss nursery rhyme "Äs isch ämal ä Ma gsi" about a man with a hollow tooth containing a box containing a letter describing a man with a hollow tooth, or Swiss singer-songwriter Mani Matter's song "Bim Coiffeur" about his head being infinitely reflected between the mirrors at the hairdresser's.

Figure 2: Recursion in nature: broccoli, cauliflower, romenesco, and fern
All photos in the public domain (CC0 Creative Commons)

Another common way to present recursion is to show recursive phenomena in nature, such as the fractal beauty of broccoli, cauliflower, romanesco, or fern (Figure 2).

All these examples *illustrate* the concept of recursion. They can leave students with the impression that recursion is natural, artistic, and fun. This is great! However, these examples do not really explain the deeper *purpose* of recursion.

1.2 Recursion coming out of nothing

The above illustrative examples usually are followed by simple formal examples, such as computing the factorial or the Fibonacci sequence. While these examples crisply and concisely present recursive computations, they often do not succeed in motivating the need for recursion. As Michaelson [13] writes: "many students may come to believe that the greatest common divisor, and the factorial and Fibonacci sequences, were dreamed up solely to illustrate an otherwise pointless technique."

A more motivating class of examples involves fractals, finite subdivision rules, and Lindenmayer systems. These computations produce visual results (Figure 3). They thus are not just more attractive, but they also allow students to immediately see the recursive nature of the computation.

All these examples create something out of nothing. They are *generative*. This makes it harder to motivate them—why would I ever need to compute this number/picture?—and it makes them harder to understand—how could I ever come up with such a magical algorithm?

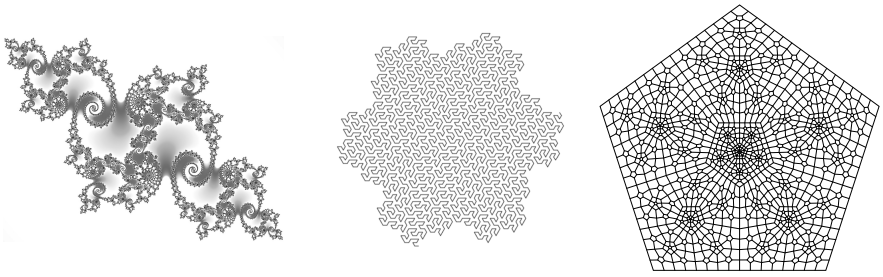

Figure 3: Visual recursion: Julia set[a], Lindenmayer system[b], polygon subdivision[c]

[a] Generated with Frax, `http://fract.al/`

[a] Generated with Lindenmayer System Generator by Nolan Caroll, `http://nolandc.com/sandbox/fractals/`

[c] Generated with Processing script by Christopher M. Overton, `https://www.youtube.com/watch?v=zqB2jN3arWc`

2 The Big Picture

When discussing whether or not recursion should be taught in school, it is helpful to put the idea of recursion in context. Most school-level programming courses introduce students to concepts such as sequences of instructions, variables, conditions, and loops. In this perspective, recursion is considered an optional topic for advanced students: while recursion allows concise and clear expressions of computation, it is not essential in that world view.

However, this first perspective is not the only one. There is a well-known alternative perspective, a dual to the first one, where recursion plays a central role. Many computer science educators are less familiar with that dual, and thus they design their curricula and their teaching around the original. Consequently, recursion may end up in a negligible supporting role, instead of being cast as the star of the show.

2.1 Dual perspectives on computing

The first perspective, based on sequences of instructions, variables, conditions, and loops, is of course known as *imperative*[1] programming. The second perspective, the dual to the first, is based on conditions, functions, and recursions, and is called *functional*[2] programming.

These two perspectives are not just two arbitrary programming paradigms. They are *duals* in many ways, as Table 1 shows. The two paradigms have com-

[1] With "imperative" we include object-oriented programming languages (such as Java).

[2] With "functional" we mean pure functional programming languages (such as Haskell).

	Imperative	Functional
Computational approach	change state	rewrite code
Theoretical foundation	Turing machine	lambda calculus
Repetition in behavior	loop	**recursive** function
Repetition in data	array	**recursive** data structure

Table 1: Dual perspectives on computing

pletely different approaches to computing: In imperative programming, a program is executed by executing statements that change some memory (which holds the state of the computation; usually in terms of variables). In pure functional languages, a program is executed by rewriting it until it represents the solution. This approach should be intimately familiar to students: it is the exact same idea they know from algebra. The two paradigms stand on two different but provenly equivalent theoretical foundations: while the imperative paradigm is inspired by Turing machines, the functional paradigm is inspired by the lambda calculus.

The bottom two rows in the table represent one of the most essential aspects of computation: the support for *repetition*, both in the form of repeated *behavior* and repeated *data*. Note that the functional perspective depends on *recursion* to describe both repeated behavior (via recursive functions) and repeated state (via recursive data structures). Recursion is the fundamental concept enabling repetition in the functional world.

3 Motivating Recursion

Based on the above discussion, one approach to motivate recursion would be to teach functional programming. There, recursion simply is the way to express repetitive computation.[3]

However, one does not need to use functional programming to teach recursion in a more motivated and intuitive way. Independent of the programming paradigm, there is a simple way to provide a more intuitive approach to recursion. The key insight is that *recursion naturally arises due to self-references in data* [7]. Instead of using *generative* examples, where recursion comes out of nowhere, one should start with *structural* examples, where the recursive computation simply traverses a given recursive data structure.

[3]Although one could use a fixed-point combinator for repetition without named functions.

3.1 Start with Structural Recursion

There are many recursive structures in the everyday environment. The following hypothetical conversation, adapted from Shriram Krishnamurthi's tips for teaching recursion [9], highlights an obvious example:

> Teacher: Do you have parents?
>
> Student: Yes.
>
> Teacher: How many?
>
> Student: Two, a mother and a father.
>
> Teacher: Does the mother have parents?
>
> Student: Yes.
>
> Teacher: How many?
>
> Student: Two, the mother's mother and the mother's father.
>
> Teacher: Does the father...

It shouldn't take students long to realize the recursive nature of this structure. But *parents* are not the only such example. Another one is that of *friends*: you have friends, and those friends also have friends. An example for adult learners is that of *managers*: an employee has a manager, and a manager is an employee. An example familiar to computer users is the recursive structure of their *file system*, where folders can contain files and other folders. After introducing these concrete examples, the idea can be generalized to the more abstract concepts of lists, trees, and graphs.

3.2 Recursive Computations Over Recursive Structures

Once students understand the recursive nature of these structures, recursive computations become more natural. Teachers can provide a certain recursive data structure and ask students to write code that answers a given question about the data. The recursive structure of the data encourages recursive formulations of computations. Many computations boil down to a recursive traversal of the structure which performs the same operation for each element.

Once *lists* are understood as an abstract structure, recursive computations that map, filter, or reduce lists become relatively straightforward. Example tasks over lists include the following: given a list of students, produce a list of their grades; given a list of students, produce the list of all students with a passing grade; or given a list of grades, compute the average grade. The same applies to computations over *trees*, e.g., given a file system tree, find the largest file; given a USB

stick, enumerate all the video files it contains; or given a manager, determine the number of employees who directly or indirectly report to her. Similar computations over *graphs* include: given facebook, determine how many friends separate you from Kevin Bacon; or if you wanted to raise a million dollars from your social network, and you started by asking all your friends, and they asked their friends, and so on, and if each person who was asked donated a dollar, through how many levels of friends would you have to go?

Students with prior programming experience, who have learned about arrays, might find it unnatural to look at lists as recursive data structures. Consequently, they might be tempted to resort to using loops instead of recursion when describing computations over lists. For example, if we asked them to determine the number of people in front of them in a line waiting to enter the movies, they might not consider a solution where they would simply ask the person in front of them, and where they could simply add one to the answer they got from that person (recursion). Instead, they might consider a solution where they, or someone else, would have to walk up the line and count all the participants (loop). However, non-linear data structures, such as trees and graphs, might provide an advantage for a recursive approach, because students may not have a prior non-recursive mental model of such structures.

4 Hypotheses

The previous sections outlined how we think recursion should be taught. However, they only represent an opinion based on our intuition. To promote a scientific discourse around that opinion, we now formulate four hypotheses underlying our thoughts:

Hypothesis 1. *It is possible to successfully teach a recursion-based programming course in middle or high school, without additional effort compared to a traditional course based on loops and variables.*

Given the equivalence in power of the functional and imperative paradigms, and given that functional languages use recursion where imperative languages use loops, it is not unimaginable that this hypothesis holds.

Hypothesis 2. *Such a recursion-based course has, compared to a loop-based course, a positive effect (transfer) on other school subjects (e.g., mathematics).*

At least for recursion-based courses that use a pure functional paradigm, where program execution corresponds to evaluation by rewriting, the close similarity to algebra raises the chance for transfer of learning to mathematics.

Hypothesis 3. *Preconceptions about imperative programming (sequence, selection, repetition, and the variables necessary for repetition) make learning recursion more difficult.*

Students' preconceptions affect learning. If students' mental model is informed by array-based structures and loops, chances are that there is a certain cognitive dissonance when they first encounter recursive structures and computations. Introductory programming material for schools that does include recursion often introduces it only after introducing loops (e.g., "Beauty and Joy of Computing" from Berkeley [21] introduces recursion only in the optional units 7 and 8, and "Programming Concepts in Python with TigerJython" [2, Section 2.9] from ETH Zürich introduces recursion after introducing repeat and while loops).

Hypothesis 4. *Recursion is easier to learn when beginning with recursive data structures and the* structural *recursion needed to process them, instead of starting with* generative *recursion.*

Recursive structures occur in concrete forms in the everyday environment. They are often visible and tangible. Recursive behaviors, however, are less obvious. This may be also because behavior, in general, is ephemeral and intangible, and thus is less amenable to analysis and reflection in everyday life.

4.1 Consequences

The validity of our four hypotheses has significant consequences for the teaching of introductory programming. If Hypotheses 1 (Can Be Taught) and 2 (Transfer) hold, it might be a mistake to teach loops instead of recursion. If Hypothesis 3 (Preconceptions) holds, it might be a mistake to teach loops before teaching recursion. If Hypothesis 4 (Structural First) holds, it might be a mistake to not teach data structures early on.

4.2 Initial Evidence

There is a long history of research on teaching recursion, and some of that work provides preliminary evidence related to our four hypotheses.

Hypothesis 1 (Can Be Taught). We did not find any studies directly supporting or refuting our first hypothesis. However, we found several works that present techniques for teaching recursion to novices in intuitive and effective ways. For example, Michaelson [13] teaches recursion by teasing apart children's counting songs (e.g., "ten green bottles hanging on the wall"), intuitively explaining how a song consists of a verse and a shorter similar song. Chaffin et al. [5] teach recursion through a game called "EleMental: The Recurrence", where students

need to implement a depth-first traversal of a binary tree. Lee et al. [11] replace a traditional lecture sequence on recursion with a learning experience that includes solving nine puzzles in the "Cargo-Bot" video game. Their controlled experiment indicates that their approach is effective, especially for learning to write recursive code. The above works are complemented by a long line of research on students' mental models of recursion [10, 3, 8, 16, 17, 14, 19]. These studies helped to understand the difficulties students have in learning recursion. They lead to the development of an intelligent tutoring system for recursion [3], and they provide a basis for improving the teaching of recursion in general.

Hypothesis 2 (Transfer). Schanzer et al. [18] showed that 8th and 9th grade middle school students who went through the Boostrap curriculum, where they learned to build a video game using a functional programming language, significantly outperformed other students on standard math tests on word problems, function application, and function composition. Note that the Bootstrap curriculum used in the study did not involve recursion, and thus this study does not directly support our hypothesis. However, it shows that functional languages, which are particularly amenable to teaching recursion, can enable transfer of learning from programming to mathematics.

Hypothesis 3 (Preconceptions). There is conflicting evidence on this issue. In a study of learning an iteratively and a recursively described version of the factorial function, Anzai [1] found that learning the recursive version after the iterative version was more effective than learning the recursive version before the iterative version. While this could be interpreted as evidence against Hypothesis 3, Anzai's study is limited to a very specific situation and did not involve any programming. Conversely, in a setting where children learned recursion after iteration, they interpreted recursive code as iteration, which lead to correct answers for tail recursive code, but caused mistakes otherwise [10]. Joshua Paley, a high school computer science teacher with extensive experience teaching recursion using three different textbooks, had the following to say [15]: "I find it very difficult to teach recursion in a language that has assignment blocks staring students in the face from the beginning."

Hypothesis 4 (Structural First). Bruce et al. [4] argue for teaching structural recursion and for doing so before teaching arrays. They report on an introductory undergraduate Java programming course. After moving the teaching of recursion before the introduction of arrays and strings, and by focusing on structural recursion, they found that students produced better designed and better encapsulated object-oriented code. Moreover, the students rated the new course as less difficult than the original course. Thompson [20] brings up an argument that in some sense also can be interpreted as support for structural recursion. He distinguishes between a recursive process and a recursive object, and he argues that understanding a recursive object is helpful in developing a recursive process. More precisely,

Thomson's hypothesis is that "to be able to recognize a problem solution as one requiring a recursive process, students must formulate their solution as a recursive object." In Thompson's setting, the recursive object is the output of generative recursion. In our hypothesis, the recursive object is the input of structural recursion. In a different line of work, Levy and Lapidot analyzed students' discourse about recursion [12]. The probably most remarkable finding from their study is the absence of the property of *self-reference* from the students' discussions. This might have been due to the mostly generative examples of recursive phenomena presented to the students. Examples of structural recursion might have the potential to make self-reference more explicit, which could help in better comprehending the essence of recursion.

The above discussions present some initial evidence for at least some of our hypotheses. We hope that further research will be conducted to complement and strengthen that evidence, and to provide empirically supported answers to the questions of whether and how to teach recursion in school.

5 Conclusions

Recursion is a common phenomenon in nature and a basic building block of computation, however, recursion is rarely taught in schools. Is this because recursion is fundamentally hard, or because it is *perceived* to be hard?

Let us close with a quote from Dijkstra: [6]

> A few years later a five-year old son would show me how smoothly the idea of recursion comes to the unspoilt mind. Walking with me in the middle of town he suddenly remarked to me, "Daddy, not every boat has a lifeboat, has it?" I said "How come?" "Well, the lifeboat could have a smaller lifeboat, but then that would be without one."

We hope Dijkstra's five-year old son did not discover self-similarity only because he was so similar to his prodigious father. We believe that *any child* can learn about recursion. And not only because every child has a parent.

References

[1] Y. Anzai and Y. Uesato. Learning recursive procedures by middleschool children. In *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, pages 100–102, August 1982.

[2] J. Arnold, T. Kohn, and A. Plüss. *Programming Concepts in Python with TigerJython*.

[3] S. Bhuiyan, J. Greer, and G. McCalla. Characterizing, rationalizing, and reifying mental models of recursion. In *Proceedings of Thirteenth Cognitive Science Society Conference*, pages 120–126, 1991.

[4] K. B. Bruce, A. Danyluk, and T. Murtagh. Why structural recursion should be taught before arrays in cs 1. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '05, pages 246–250, New York, NY, USA, 2005. ACM.

[5] A. Chaffin, K. Doran, D. Hicks, and T. Barnes. Experimental evaluation of teaching recursion in a video game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 79–86, New York, NY, USA, 2009. ACM.

[6] E. W. Dijkstra. Ewd1298: Under the spell of leibniz's dream. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1298.html.

[7] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA, 2001.

[8] T. Götschi, I. Sanders, and V. Galpin. Mental models of recursion. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '03, pages 346–350, New York, NY, USA, 2003. ACM.

[9] S. Krishnamurthi. How does professor shriram krishnamurthi teach the concept of recursion? https://www.quora.com/How-does-professor-Shriram-Krishnamurthi-teach-the-concept-of-recursion.

[10] D. M. Kurland and R. D. Pea. Children's mental models of recursive logo programs. *Journal of Educational Computing Research*, 1(2):235–243, 1985.

[11] E. Lee, V. Shan, B. Beth, and C. Lin. A structured approach to teaching recursion using cargo-bot. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 59–66, New York, NY, USA, 2014. ACM.

[12] D. Levy and T. Lapidot. Recursively speaking: Analyzing students' discourse of recursive phenomena. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 315–319, New York, NY, USA, 2000. ACM.

[13] G. Michaelson. Teaching recursion with counting songs. In *IDC'15 "Every Child a Coder?" workshop*, June 2015.

[14] C. Mirolo. Mental models of recursive computations vs. recursive analysis in the problem domain. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '09, pages 397–397, New York, NY, USA, 2009. ACM.

[15] J. Paley. Personal communication, May 2016.

[16] I. Sanders, V. Galpin, and T. Götschi. Mental models of recursion revisited. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITICSE '06, pages 138–142, New York, NY, USA, 2006. ACM.

[17] I. D. Sanders and V. C. Galpin. Students' mental models of recursion at wits. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 317–317, New York, NY, USA, 2007. ACM.

[18] E. Schanzer, K. Fisler, S. Krishnamurthi, and M. Felleisen. Transferring skills at solving word problems from computing to algebra through bootstrap. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 616–621, New York, NY, USA, 2015. ACM.

[19] T. L. Scholtz and I. Sanders. Mental models of recursion: Investigating students' understanding of recursion. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 103–107, New York, NY, USA, 2010. ACM.

[20] P. W. Thompson. Understanding recursion: Process approximates object. In *Proceedings of the 7th Annual Meeting of the North American Group for the Psychology of Mathematics Education*, pages 357–362, Columbus, OH, USA, 1985.

[21] University of California, Berkeley. The beauty and joy of computing (bjc). http://bjc.edc.org/bjc-r/course/bjc4nyc.html.