Informatics in Education, 2014, Vol. 13, No. 1, 87–119

A Survey on Teaching and Learning Recursive Programming

Christian RINDERKNECHT
Department of Programming Languages and Compilers, Eötvös Loránd University
Budapest, Hungary
E-mail: rinderkn@caesar.elte.hu

Received: July 2013

Abstract. We survey the literature about the teaching and learning of recursive programming. After a short history of the advent of recursion in programming languages and its adoption by programmers, we present curricular approaches to recursion, including a review of textbooks and some programming methodology, as well as the functional and imperative paradigms and the distinction between control flow vs. data flow. We follow the researchers in stating the problem with base cases, noting the similarity with induction in mathematics, making concrete analogies for recursion, using games, visualizations, animations, multimedia environments, intelligent tutoring systems and visual programming. We cover the usage in schools of the Logo programming language and the associated theoretical didactics, including a brief overview of the constructivist and constructionist theories of learning; we also sketch the learners' mental models which have been identified so far, and non-classical remedial strategies, such as kinesthesis and syntonicity. We append an extensive and carefully collated bibliography, which we hope will facilitate new research.

Key words: computer science education, didactics of programming, recursion, tail recursion, embedded recursion, iteration, loop, mental models.

Foreword

In this article, we survey how practitioners and educators have been teaching recursion, both as a concept and a programming technique, and how pupils have been learning it. After a brief historical account, we opt for a thematic presentation with cross-references, and we append an extensive bibliography which was very carefully collated. The bibliography is the foundation of our contribution in the sense that we started from it, instead of gathering it in support of our own ideas, as is usual in research papers.

In writing this survey, we committed ourself to several guidelines which the reader is advised to keep in mind while reading.

1. We restricted ourself exclusively to the published literature on teaching and learning recursive programming, not computer programming in general. While it may

be argued that, for example, articles and books about functional programming almost constantly make use of recursion, we preferred to focus on the papers presenting didactical issues explicitly and exclusively related to recursion.
2. We did not review the emergence of the concept of recursion from its mathematical roots, and we paint a historical account with our fingers, just enough to address the main issues of this survey with a minimal background information.
3. We did not want to mix our personal opinion and assessment of the literature with its description, because we wanted this article to be in effect a *thematic index to the literature*, even though it is not possible to cite all references in the text, for room's sake. The only places where we explicitly express our own ideas are in our own publications, in the definitions found in the introduction (in the absence of bibliographic reference) and in the conclusion.
4. We did not cover topics like the teaching of recursion and co-recursion in the context of lazy evaluation, or programming languages based on process algebras or dataflow, because they have not been addressed specifically in didactics publications, perhaps because they are advanced topics usually best suited for postgraduate students, who are expected to master recursion, and most publications deal with undergraduates or younger learners.
5. Despite our best efforts in structuring the ideas found in the literature, the following presentation contains some measure of repetition because papers often cover mutually related topics, so a printed survey cannot capture exactly what is actually a semantic graph, and such a graph would better support a meta-analysis of the literature (where cross-referencing, publication timelines, experimental protocols and statistics would be in scope) rather than a survey.

Introduction

In abstract terms, a definition is *recursive* if it is self-referential. For instance, in programming languages, function definitions may be recursive, and type definitions as well. Give'on (1990) provided an insightful discussion of the didactical issues involved in the different meanings ascribed to the word, which appeared first in print by Robert Boyle in 1660 (*New Experiments Physico-Mechanicall*, chap. XXVI, p. 203) to qualify the movement of a moving pendulum, which returns or "runs back". (Beware the incorrect and ominous "to recurse".)

A short history. Formal definitions based on recursion played an important role in the foundation of arithmetic (Peano, 1976) and constructive mathematics (Skolem, 1976, Robinson, 1947, 1948), as well as in the nascent theories of computability (Soare, 1996, Oudheusden, 2009, Daylight, 2010, Lobina Bona, 2012), with the caveat that recursion theory is only named so for historical reasons. The first computers were programmed in assembly languages and machine codes (Knuth, 1996), but the first step towards recursion is the advent of labelled subroutines and hardware stacks, by the end of the 1950s. `BASIC` epitomises an early attempt at lifting these features into a language more abstract than assembly: recursion is simulated by explicitly pushing (`GOSUB`) program pointers

(line numbers) on the (implicit) control stack, and by popping (`RETURN`) them. According to the definition above, this is not recursion, which is to be understood as being purely syntactic (a function definition), not semantic (the evaluation of an expression). Moreover, the lack of local scoping precludes passing parameters recursively. Nevertheless, "recursion" has been taught with `BASIC` by Daykin (1974).

With even more abstract programming languages, fully-fledged recursion became a design option, first advocated in print by Dijkstra (1960) and McCarthy (1960), and implemented in `LISP`, `ALGOL`, `PL/I` and `Logo` (Martin, 1985, Lavallade, 1985), with the notable exceptions of Fortran and COBOL. Formal logic was then used to ascribe meanings to programs, some semantics relying on recursion, like rewrite systems, some others not, like set theory or λ-calculus (where recursion is simulated with fixed-point combinators). Even though the opinion of Dijkstra (1974) (1975) varied, recursion proved a powerful means for expressing algorithms (Dijkstra, 1999) (Reingold, 2012), especially on recursive data structures like lists, *i.e.*, stacks, and trees. With the legacy of `LISP` and `ALGOL`, together with the rise and spread of personal computers, recursion became a common feature of modern programming languages, and arguably an essential one (Papert, 1980, Ford, 1982, Astrachan, 1994).

1. Recursion, Iteration and Loops

Recursion is often not clearly understood, as demonstrated by the frequent heated or misguided discussions on internet forums, in particular about the optimisation of tail calls. Moreover, some researchers implicitly equate loop and iteration, use the expression "iterative loop", or call recursion a process implemented by means of a control stack, whilst others use a syntactic criterion.We must define recursion in order to relate it to other concepts, like loops, iteration, tail recursion, embedded recursion, structural recursion etc.

Definitions. Give'on (1990) remarked: "the concept of recursion is being vaguely and inconsistently constructed from some syntactical properties of the program, from its associated semantics and from features borrowed from models of execution of programs". Indeed, broadly speaking, there are two angles to approach the question: the static (syntactic) approach and the dynamic approach to recursion. Sometimes the dichotomy is put in terms of *programs* (structured texts or abstract syntax trees) versus *processes* (autonomous agents or stateful actors). In fact, to address the vast literature about the teaching and learning of recursion, it is essential to understand both views and their relationship.

In the static comprehension, recursion is restricted to the general definition we gave at the start of this introduction: the occurrence of the symbol being defined inside its definition (what is called impredicativity in logic). Implicitly, this means of course that it must be clear what the denotation of the occurring symbol is, in order to determine whether it is an instance of the definition. For example, in the language `OCaml`, the function definition

```
let rec f x = (fun _ -> x) f
```

is recursive because the name `f` in the right-hand side refers to the `f` in the left-hand side. Note that there is no call to `f` in the definition of `f`, so the concept of *recursive call* is actually irrelevant: recursion in this context is a property of definitions based on lexical scoping rules, not of the objects potentially computed (values), nor the way they are computed (semantics). In particular, the recursive definition of a function does not necessarily entails that it is total, hence terminates for all inputs. (A type system may enforce that property, as in `Coq` or `Agda`.) Sometimes, an examination of the program cannot determine whether the occurrence of a symbol refers to the definition at hand. For instance, let us consider the following fragment of `Java`:

```
public class T { public void g(T t) { ... t.g(t); ... } }
```

The occurrence of `g` in the expression `t.g(t)` does not necessarily refers to the current definition of the method `g`, because that method may be overridden in subclasses. Therefore, here, we cannot conclude that `g` is recursive according to the static criterion. (It can be argued, though, that the class `T` is recursive because its definition includes the declaration (type) of its method `g`, where `T` occurs – Interfaces perhaps illustrate this better.)

The dynamic comprehension of recursive functions can be expressed abstractly as a property about *dynamic call graphs*: recursion is a reachable cycle, which means, in operational terms, that the control flow of calls returns to a vertex (a function) which was previously called. Here, the notion of *recursive definition* is not central, and it makes sense to speak of *recursive call* (a back edge closing a path). Another, less general, approach to a dynamic definition of recursion relies on a particular execution model, often based on stack frames allocated to function calls and their lexical context. Anyway, as a property about the control flow, recursion in that sense becomes undecidable in general for Turing-complete languages.

It should be noted that the static and dynamic definitions of recursion may overlap, but are different in general, that is, if a function is recursive according to the syntactic criterion, it may not be recursive according to the dynamic criterion (as the above `OCaml` function `f` illustrates), and vice versa. Consider the following `OCaml` program implementing the factorial function:

```
# let pre self n = if n = 0 then 1 else n * self(n-1);;
val pre : (int -> int) -> int -> int = <fun>
# let rec fact n = pre fact n;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

The syntactic criterion decides that `pre` is not recursive and `fact` is; the dynamic criterion sees these two functions as *mutually recursive*, that is, the control flow goes from one to the other, and vice versa. Furthermore, there are different techniques to achieve dynamic recursion without static recursion at all. For example, using fixed-point combinators in `OCaml` with the command-line option `-rectypes`:

```
# let pre self n = if n = 0 then 1 else n * self(n-1);;
val pre : (int -> int) -> int -> int = <fun>
# let y f = (fun x a -> f (x x) a) (fun x a -> f (x x) a);;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# let fact = y pre;;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

Here, neither the higher-order function `y` (called the *call-by-value* `Y` *combinator*), nor the function `pre` are statically recursive (as the absence of the keyword `rec` shows well), but they are mutually recursive in the dynamic sense. (The rationale behind the definition of `y` is obscure, but relies on the fact that `(y f) x` yields the computation of `(f(y f)) x`, showing that `y f` is the fixed point of `f`.) It is even possible to define the factorial function without recursion, loops or jumps (`goto`) in `C`, but the program is cryptic:

```
#include<stdio.h>
#include<stdlib.h>

typedef int (*fp)();

int fact(fp f, int n) {
return n? n * ((int (*)(fp,int))f)(f,n-1) : 1; }

int read(int dec, char arg[]) {
return ('0' <= *arg && *arg <= '9')?
read(10*dec+(*arg - '0'),arg+1) : dec; }

int main(int argc, char** argv) {
if (argc == 2) printf("%u\n",fact(&fact,read(0,argv[1])));
else printf("Only one integer allowed.\n");
return 0; }
```

(See Goldberg and Wiener (2009) for a practical use of such a simulated recursion in `Erlang`.) References can also be used to define the factorial function without static recursion, with a technique called *Landin's knot*:

```
# let g = ref (fun n -> 42);;
val g : ('_a -> int) ref = {contents = <fun>}
# let f n = if n = 0 then 1 else n * !g(n-1);;
val f : int -> int = <fun>
# let fact = g := f; fun n -> !g(n);;
val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

Here, none of the definitions are statically recursive, although `f` is dynamically recursive.

Finally, it is perhaps worth insisting on the case where there are more than one definition, like $f(x) := g(x-1)$ and $g(x) := h(x+1, f(x-1))$. Neither definition is statically recursive, although they are mutually recursive according to the dynamic interpretation. Furthermore, it is clear that these definitions are equivalent to $f(x) := h(x, f(x-2))$, which is statically recursive. This shows that the concept of mutual recursion is dynamic, but the static criterion could be extended to apply transitively to the *static call graph*, which is an over-approximation of the dynamic call graph, so we can speak of mutual recursion in a static sense as well, but keeping in mind that there can be mutual recursion statically when there is none dynamically.

Tail recursion, iteration and loops. The concept of *tail recursion* is difficult to apprehend because it is built upon both the dynamic call graph and the *data flow*. We have already seen that recursion can be defined as a cycle in the dynamic call graph. Here, we define the *dataflow graph* as the dynamic call graph with an additional kind of edges oriented according to the direction where the data flows (it is a multigraph): if a caller passes arguments to the callee, there is a *data edge* doubling the *control edge*; if the value of a function call is needed to further compute an expression or complete an instruction, there is a data edge from the callee to the caller, that is, a backward data edge with respect to the control edge. Since, in the absence of run-time errors, the result of a call is needed, at the very least, to stand for the result of the caller itself, there is always a back edge. Therefore, we could make those edges implicit and only retain them when the value of the call is needed in a strictly embedding expression, not just to be returned in turn. Tail recursion is then a cycle along the control edges, which is not a retrograde cycle following the data edges. In other words, the data flows solely in the same direction as the control flows. (Note that, in general, there may be no data flow between two calls.)

For instance, the value of the recursive call in $f(x, y) := f(x, g(y))$ is the value of $f(x, y)$ being defined, so the call is tail recursive. On the other hand, the value of the call $f(x-1)$ in $f(x) := x \times f(x-1)$ is not the value of the call $f(x)$ being defined because a multiplication by x is pending, so it is not tail recursive. The same holds for $f(x) := g(x, f(x-1))$. Note that, within the dynamic interpretation of recursion, the concept of tail recursion applies to function calls, not to function definitions as a whole, so it is technically incorrect to say that a function definition is tail recursive.

Within the static understanding of recursion, it is not possible to define tail recursion in general because only definitions may be recursive and only calls may be in *tail position*. The latter refers to a syntactic criterion which implies that the value of a call is only used to become the value of the current function being called. In practice, however, it is possible to speak of a tail recursive call when the static and dynamic interpretations agree, that is, when a definition includes non-ambiguously a call to the function defined (a special case of static recursion) and that call is in tail position. Nevertheless, since the very reason to distinguish tail recursive calls is that they can often be compiled as efficiently as loops are (a technique known as *tail call optimisation*), the interaction between the control flow and the data flow must be made explicit anyway, even within a static framework, and this proves challenging to students and professors alike. Even

more puzzling is the fact that the optimisation applies to non-recursive calls as well, as long as they are in tail position.

When a recursive call is not tail recursive, it is sometimes called an instance of *embedded recursion*. In theory, it is always possible to rewrite any embedded recursion into tail recursion, but the result can be rather hard to understand, hence difficult to design directly. Moreover, in programming languages featuring conditional loops (`while`), recursion can be avoided in theory, but, in practice, many algorithms are expressed more compactly or more legibly if recursive. A *loop* is a segment of code syntactically distinguished and whose evaluation is repeated until a condition on the state of the memory is met. The syntactic condition, *e.g.,* a keyword and markers for a block, is meant to differentiate loops from source code whose control flow relies on jumps (`goto`) and could actually be an unstructured implementation of loops (using backward jumps), but are not loops. *Iteration* is none other than the concept of repetition applied to a piece of source code, therefore, from a theoretical standpoint, it should include recursion and loops, but, in practice, iteration is often used as a synonym for the execution of a loop in an imperative language (*looping*); in a purely functional language, iteration is tail recursion. Conditional loops (`while`) and recursion have the same expressive power, so using one form or the other is a matter of style as long as side-effects are allowed, because loops require a model of computation where data is mutable.

As we mentioned earlier, some researchers prefer to define recursion not on programs, but on *processes*, that is, on the dynamic interpretation of programs. For instance, Kahney (1983) defines recursion as a process "that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones." Of course, one data structure suitable for implementing this mechanism is the *control stack*, which we already mentioned about "recursion in `BASIC`" (Daykin, 1974). It is perhaps interesting to notice the use of the "forward" and "backward" terminology about the control flow on the call graph, although that graph is oriented from callers to callees and there are no back edges because these would not denote calls but returns. (Our own definition of dynamic recursion is a cycle in the dynamic call graph, where "backward" qualifies the data flow superimposed on the call graph.) We will see in a forthcoming section that this operational interpretation of recursion can be suitably exploited by kinesthetic teaching. The sections on analogies and mental models also revisit this choice. Finally, when contrasting the static (syntactic) and dynamic (control stack) definitions, it is worth keeping in mind that it is possible to compile recursive definitions of functions in such a way that the size of the control stack is statically bounded; in other words, recursion can always be transformed into iteration.

Teaching. Clearly, recursion and loops are not mutually exclusive and may serve the same purpose, which often bewilders the beginner. Consequently, a simple attempt at a remedy consists in clearly separating the different concepts at stake in the evaluation of a program (Velázquez-Iturbide, 2000), so that side-effects, for instance, do not get in the way of learning recursion declaratively. To teach the difference between iteration and embedded recursion, some researchers have proposed to teach how to translate an embedded recursive definition into an iteration, while remaining in the same programming language (Augenstein and Tenenbaum, 1976, Rubio-Sánchez and Velázquez-Iturbide, 2009,

Rubio-Sánchez, 2010, Rinderknecht, 2012). Foltynowicz (2007) went even further by deriving loops from embedded recursion, and vice versa, which is of great theoretical and practical interest, in particular for understanding compilers and interpreters. By exhibiting a systematic way to move back and forth from recursion to loops, while maintaining the meaning invariant, these didactic approaches aim at demystifying recursion without resorting to a low-level view of evaluation with the control stack.

Finite iteration is unidirectional in the sense that the control flow does not return to a previous program location where the environment, *i.e.*, the bindings of the variables to their values, is the same. Embedded recursion is often called bidirectional when it is based on the strict interpretation of the composition of functions, as opposed to a non-strict semantics, like lazy evaluation, which is perhaps better explained by graph rewriting. Consider for instance $f(g(x))$, where x is a value. First, the value of $g(x)$ is computed (control and data flow forward), that value is bound to an implicit variable y (control and data flow back) and then the call $f(y)$ is evaluated (control and data flow forward).

Finally, let us take note of a radical and contrarian view: to avoid recursion as much as possible (Anonymous, 1977, Buneman and Levy, 1980). For instance, Harvey (1992) advocates the use of a functional style where recursion is hidden inside higher-order functions like maps and folds. This is indeed the approach often taken when teaching purely functional programming languages, especially those with a non-strict semantics like `Miranda` or `Haskell`.

2. Functional Programming

Segal (1994) notes that, in the context of the functional programming language `Miranda`, "by using the library of functions as a toolbox, recursion, the underlying structure of many of the functions and the only repetitive construct provided by the language, can remain largely hidden." Er (1984) argued that recursion is made difficult by block-structured programming languages, which suggests that one way of encouraging the use of desirable constructs, like recursion, would be to employ or develop domain-specific languages (Sinha and Vessey, 1992); *cf.* Brooks *et al.* (1992). It would then make sense to teach recursion with functional languages, because these feature prominently mathematical functions and immutable data, forcing the programmer to think recursively (Henderson and Romero, 1989, Howland, 1998).

Because it is possible, for the purpose of teaching, to define a semantics for functional languages based on term or graph rewriting, Velázquez-Iturbide (1999), Pareja-Flores *et al.* (2007) and Rinderknecht (2012) can ask learners to trace by hand the evaluation of their small programs. Segal (1994) remarks that "we would argue [...] that the ability to be able to evaluate a recursive function mentally or 'by hand' (that is, independent of a machine), is an essential component of recursive knowledge for both learners and experts." In the case of teaching higher-order functions, using manual reductions is also a recommendation of Clack and Myers (1995), who also list a long series of typical errors and their analysis. Furthermore, Burton (1995) observes that

> *perhaps students are puzzled, unnecessarily, by the the language (I refer to natural language here) with which we talk to them about recursion. Peter Landin is fond of pointing out the numerous inconsistencies with which such language is riddled (the phrase "calls itself ", for instance, probably elides all kinds of different semantic levels). An advantage of teaching via reduction sequences is that it enables us to take the (natural) language out – just reduce, reduce, reduce (perhaps with the aid of a machine).*

He also recommends what he calls a "separation of concerns" in teaching at first list processing, pattern matching and recursion in isolation: this avoids the issue for the students to assimilate recursion at the same time as other imperfectly understood concepts. Velázquez-Iturbide (1999) also relies on term rewriting to teach recursion before moving to recursion in an imperative language with recursive data types. By writing down the rewrite system in the exact order of a top-down design, students become accustomed to laying out calls to functions yet to be defined; by also asking them to write down all the left-hand sides of the rules (patterns) before proceeding to the right-hand sides in random order, not only completeness is improved, but also the conception of a program as a text written in one pass is undermined, and the model of a form or a blueprint is proposed instead. This twofold method seems to defuse a bit the typical question of a recursive call (right-hand side) to the current function "still under construction", because at least all the configurations of the input (left-hand side) have been already laid out and it is also normal to call yet undefined functions, just like it is normal to have pending references in a map being drawn to other parts yet to be filled. This view seems to be one of the conclusions of Vitale (1989), when he writes, in abstract terms:

> *It is proposed that a restricted notion of "recursion" could be usefully defined, entailing:*
>
> 1. *that the attitude of the subject, with respect to the definition of a notion, the solution of a problem, the answer to a question, etc., should contain a measure of* suspended attention*, deferring in a way the final restructuring of the definition solution, answer, etc., to the completion of a downward and then upward spiralling path;*
> 2. *that the spiralling path should be describable by the dialectical coexistence of* permanence *(the path, global because relying on the various steps) and* change *(the pitch of the spiral, local because defined – and possibly changing – at every turn).*

(For some technical corrections on the article of Vitale (1989) and some context on the relevance of recursion in the cognitive sciences and artificial intelligence, see the follow-ups by Trautteur (1989) and Apostel (1991), as well as Kieren (1989) in the context of `Logo`.) Furthermore, by using directed acyclic graphs to represent programs and data, instead of abstract syntax trees, *aliasing* (data sharing) becomes visible and the control stack and heap can arise from this model without resorting to low-level descriptions (Rinderknecht, 2012).

Another approach, advocated by Felleisen *et al.* (2001), consists in systematically starting with the definition of recursive data types, because such types already suggest

the recursive structure of the function definition to process their values. We will revisit this method when presenting *structural recursion*. Pirolli (1986) showed that focusing the teaching of recursion on the structure of the function definition is more effective than insisting on the evaluation process, with traces of the control and data flows.

When loops are taught after recursion in a functional language, no *transfer of skills* seems to be observed, undermining the idea that iteration is inherently simpler than recursion (Mirolo, 2011). For an equivalent study with logic programming in `Prolog`, see Haberman (2004). Moreover, simple functional programs on lists can be translated systematically in `Java` (Rinderknecht, 2012), following design patterns similar to those by Felleisen and Friedman (1997), Bloch (2003) and Sher (2004). The programs which are derived are in static single assignment form and eschew the `null` from Pandora's vase (Cobbe, 2008, Hoare, 2009). However, Segal (1994), Clack and Myers (1995) noted that inducing students to think recursively with functional languages may yield some of the problems encountered with imperative languages, and Paz and Lapidot (2004) showed how prior experience with imperative programming influences the learning of functional programming. This brings us to examine when is recursion taught.

3. Curricular Approaches

The scheduling of the teaching of recursion in school curriculums has long been debated (Olson, 1987, Barfurth and Retschitzki, 1987, Greer, 1989). For example, Zmuda and Hatch (2007) compare two approaches: the scheduling of consecutive units of teaching on recursion versus the intermittent teaching of recursion, whereby two units about recursion are separated by a different topic.

Secondary schools. In many countries, programming literacy, as opposed to vocational training on software products (*e.g.*, ICT in the United Kingdom since the 1990s), is still absent in the secondary schools curriculums. For instance, the French government officially introduced it only in July 2011, as an option for science majors, and recursion is not even mentioned in the new regulation, whose implementation started in September 2012. (The mathematics curriculum contains only one paragraph about algorithms, which must be explicitly iterative (Modeste, 2012).) Wherever programming is featured in introductory courses, recursion is usually avoided, even though it is present in mathematics courses, usually in the guise of numerical progressions, Euclid's algorithm, Newton-Raphson approximation method, and proofs by mathematical induction (Buck, 1963). Therefore, because university students often experience significant difficulties in grasping recursive programming (Sooriamurthi, 2001, Ginat, 2004), some educators have insisted on a better articulation between secondary and post-secondary curriculums. For instance, some researchers have been promoting a greater presence of discrete mathematics and proof techniques in secondary schools (Abramovich and Pieper, 1996, da Rosa, 2002, Rosenstein *et al.*, 1997, Kaiser, 2004*a*,*b*), as well as the creation of computing clubs with activities about recursion (Gunion *et al.*, 2009*a*). Others have emphasised the duality between recursive programming and mathematical induction (Peelle, 1976, Ford, 1984, Leron and Zazkis, 1986, Anderson, 1992, Brandt and

Richey, 2004, Polycarpou, 2006), which may be used as means to a transfer of skills from secondary mathematics, as is, into college informatics. Even a reverse transfer of skills, from recursive programming to problem solving in mathematics, has been envisaged by Hausmann (1985).

The teachers gleaning recursive definitions in the fields of secondary mathematics often come up with numerical progressions, including the versatile Fibonacci numbers (Rubio-Sánchez and Pająk, 2006, Rubio-Sánchez and Hernán-Losada, 2007, Rubio-Sánchez, 2008), combinatorial identities from Pascal's triangle, the pervasive factorial or the game known as "The Tower(s) of Hanoi (or Brahma)." (Buneman and Levy, 1980, Anderson, 1992, Benander and Benander, 2008) Unfortunately, the pertinence of such examples is undermined by the fact that they frequently enjoy closed forms (like $1 + 2 + \ldots + n = n(n + 1)/2$) or they are computationally inefficient (Er, 1984, Knight, 1988, Costello, 1990, Robertson, 1999, Stojmenovic, 2000, Manolopoulos, 2005), which may not be an issue for a mathematician. Furthermore, to university students interested in programming or professional training, these contrived exercises may appear useless and fail to match their expectations, tainting recursion by association. The same reaction is likely when outbidding with functions defined by more complex recurrent equations, like McCarthy's "91 function," Takeuchi's function (Knuth, 2000) or the simplified form of Ackermann's function (Robinson, 1947, 1948).

Fortunately, most textbooks avoid these pitfalls.

Textbooks. Since the aim of a textbook is to cover a given curriculum, it should not come as a surprise that there are no textbook exclusively devoted to recursive programming, but there have been some companion books, at least up to the 1990s, when computer programming entered mainstream education with the spread of personal computers. (As mentioned before, during the same period, hardware architectures and programming languages widely enabled recursion.)

In university education, from about 1965 to 1975, computer science emerged as a discipline independent from mathematics, which explains the rigorous approach of the books and the interest in theoretical explanations, as well as low-level implementations of recursion. This didactic choice was enabled by the mathematical savvy of the students and the few abstraction layers between programming languages and the hardware of the time. For example, Barron (1968) is concerned with the pragmatics of recursion, its implementation in run-time environments, the comparison with iteration, the natural application to sorting, the mechanisms for recursion in compilers and numerical algorithms – all this with `ALGOL`. Burge (1975) starts with λ-calculus and combinatory logic, and proceeds with the evaluation of mathematical expressions, the definition and traversal of recursive data structures (lists and trees), parsing, sorting algorithms – also in `ALGOL`.

The following period, from about 1975 to 1985, saw recursion uprooted from theoretical grounds and presented both as a method and a programming technique for solving problems whose data structures are recursive (structural recursion), making plain the benefit because the program structure itself then matches that of the data it processes. For instance, Rohl (1984) begins with linked lists and binary trees, explains the solving strategy "divide and conquer" (The input is split, each non-atomic part is recursively processed and the partial solutions are finally combined to form the complete solution.)

and widens the scope to include mutual recursion (Rubio-Sánchez *et al.*, 2008) and recursion on graphs – all with `Pascal`. Roberts (1986) (2006) wrote the most enduring book, first using `Pascal` and now `Java`, where the main difference with previous volumes lies in recursion being illustrated by drawing fractals and backtracking when stuck in a labyrinth, whereas implementation issues make up the last chapter only.

Methodology. To tackle the understanding of the control flow, it is useful to work on design methodology (Kessler and Anderson, 1986). Indeed, embedded recursion is wrongly conceived as an expression of the familiar counting or accumulation technique within loops, not the consequence of the analysis of the original problem. As a remedy, students could be taught to think declaratively when programming recursively in imperative languages (Give'on, 1989, Ginat and Shifroni, 1999), that is, to distinguish specification (what) from evaluation (how) (Ford, 1984). Equivalently, this means that recursion could be taught first as a method for solving problems (analysis and synthesis, familiar to mathematicians since Antiquity), before showing it to be also a programming technique (McKavanagh, 1992, 2004). In the same vein, Ginat (2005), Ginat and Armoni (2006) follow a principle of Pólya distinguishing *working forwards*, which is a heuristics consisting in approaching the solution by stepwise deductions, and *working backwards*, which supposes the goal attained and concentrating on the inductive chain, back to the problem. In the context of functional programming, Rinderknecht (2012) calls the first method *small-step design* because the programmer focuses on the least that can be done in one evaluation step towards the solution, and the second *big-step design* because they assume that the final value is obtained in one step and it has to be (recursively) decomposed in terms of the input. In general, the first way leads to iteration, whereas the second yields embedded recursion. These two methods should be taught as complementary heuristics, because, for the same problem, they may not bear definitions of commensurable efficiency.

Curriculum. To overcome students' reluctance to use recursion within a course on procedural or object-oriented programming, it has been proposed to teach singly-linked lists before arrays and loops (Turbak *et al.*, 1999, Bruce *et al.*, 2005, Goldwasser and Letscher, 2007), which makes recursion appear as a rather natural way to move to and fro inside a unidirectional data structure. It is not surprising that this proposal, where recursion in data types comes before recursion in functions, often originates from the context of object-oriented programming languages (Felleisen and Friedman, 1997, Levine, 2000, Bloch, 2003, Sher, 2004), but is also prominent in statically typed functional languages – refer to the book by Felleisen *et al.* (2001). Indeed, when generalised to other recursive data types, like trees, this kind of recursion is called *structural recursion* and, as mentioned earlier, it yields programs reflecting the structure of the data type, which is helpful since the latter is designed first. For instance, a binary tree is either empty or made of a root and two subtrees, thus the complete traversal of such a tree is expected to require a test for the tree emptiness and two recursive calls.

Didactics. Some researchers have been tackling the issue of teaching and learning recursion through the lenses of cognitive sciences and psychology, inferring the *mental*

models of recursion (Sanders *et al.*, 2006, Mirolo, 2009), in particular the faulty ones that novices construct by interacting with experts and the problem to solve. As explained by Bhuiyan *et al.* (1994), a mental model is twofold: "(1) a knowledge structure in a person's mind that incorporates *descriptive knowledge* and *functional knowledge* about a concept or device; (2) a control mechanism that determines how this knowledge is used in problem solving." Many of the references we gave in previous sections already contain significant discussions and analyses of mental models, as they are used as a rationale for guiding the design, for example, of a tutoring system or a curriculum. In the introduction, we also have mentioned Give'on (1990), who discusses some pedagogical issues with the different meanings of the word *recursion*, and it is fitting now to cite as well Lobina and García-Albea (2009) and Lobina (2011), Lobina Bona (2012), who bring forth a thoughtful analysis of the usages of the same word in the cognitive sciences, with an emphasis on linguistics and psychology. Indeed, these disciplines are essential to the didactics of programming. Lobina and García-Albea (2009) write: "In the 1950's, linguists correctly employed recursion in reference to specific rewrite rules, but ever since their elimination from linguistic theory, most linguists have used recursion, rather puzzlingly, to refer to those structures that recursive rewrite rules were used to generate. This may well be the unfortunate legacy of employing rewrite rules." Consequently, they recommend to reserve the term recursion for processes, not the products of these, because not all hierarchy (self-embedding) is generated by recursive processes. With these distinctions in mind, which will be touched upon again in the section about analogies, it is further worth reading Kilpatrick (1985), who discusses the analogical use of the words *reflection* and *recursion* in the didactics of mathematics.

According to the *constructivist theory of learning* (Wu, Dale and Bethel, 1998, Ben-Ari, 2001) promoted by Jean Piaget, learners construct mental models to understand the world and act proactively, instead of passively reproducing a series of facts and being enjoined belief in a theory, as happens with too many traditional lectures. Inhelder and Piaget (1963) write: "the source of thinking making possible to design recursive solutions to problems lies in elemental forms of reasoning arising from students' comprehension of the relations between the elements to which his/her *actions* are applied when attempting to solve instances of problems." (The emphasis is ours.) The study of da Rosa (2005) argues that the role of the teacher is to help the student to transform this instrumental knowledge into a conceptual knowledge, and finally into formalisation, that is, program writing. Some researchers speak of "misconceptions", others do not because they consider that these are simply transient stages, non-viable conceptions – a *viable* conception allowing to predict the outcome of new experiments. Götschi *et al.* (2003) explain: "Teachers should generate perturbations in the students' existing conceptual structures and hence foster new combinations of concepts. This means that lecturers should present students with problems and examples that challenge their current understanding and reveal non-viable constructions."

In the same vein, a *constructionist theory*, developed by Papert (1980), goes further by insisting that learning is best or truly achieved by making tangible objects in interaction with the environment, which includes the educator. These approaches do not diminish in any way the role of the teacher, who is simply encouraged to engage constructively

with the pupils, and not to act as an oracle or a judge. It is assumed that the learners build their knowledge themselves, based upon previous idiosyncratic conceptions, which they reassess by means of interactive experiments under the benevolent supervision of an expert. Within this framework, where reassessment entails either reinforcement or refutation, the self-referential nature of recursive definitions may seem a priori a cognitive challenge, which Papert (1960*b*) expresses as "the property of recursion being not the repetition of the same act as such, but the repetition of an act that is at the same time the same act and a different one." In fact, the interest in mental models of recursion did not wait for the personal computers to reach homes and classrooms, as it can be traced (in the context of the psychology of mathematics first, and then computing) back to Papert (1960*a*) (1960*b*) and Piaget (Inhelder and Piaget, 1963, Piaget and Stratz, 1974). (See Matalon (1963), Eliot *et al.* (1979) also for early research on children.) Children and adolescents were at the centre of pedagogical investigations with the programming language `Logo`. One hypothesis of Papert is that the *syntonicity* enabled by `Logo` helps the children to learn: "Turtle geometry is learnable because it is syntonic." (Papert, 1980, p. 68) Roughly speaking, syntonicity is a psychological feeling of identification with a putative external agent, in this case the cursor on the screen, called the turtle. This feeling, supported by the fact that the movements of the turtle are relative to its current position (*cf.* `PostScript` below, in the section about `Logo`), entices the children to engage and enjoy what they make, which is more than a drawing since it involves a (projected) whole body experience.

Mental models. According to Kahney (1983), Kahney and Eisenstadt (1982), the mental model of experts, called *copies model*, is based on dynamic instances of procedures, *i.e.*, processes, either passing ("forwards") the control to newly created instances, or, if terminated, returning it ("backwards") to the instance who passed it – George (2000*a*) called the former *active flow*, and the latter *passive flow*. The copies model is the only one viable, that is, consistent with the operational semantics of recursive definitions. Students, on the other hand, seem to often build the *looping model* of recursion, whereby embedded recursion is wrongly understood as a kind of iteration and, typically will consider the base cases as halting conditions (Haberman and Averbuch, 2002). To reduce the risk of confusion, McDougall (1985) recommended that, when teaching `Logo`, the "use of tail recursion for iterative situations be deliberately avoided. [...] Avoidance of early use in programming of tail recursion for repetition might avoid confusion with iteration in children's mental models of recursion." Indeed, according to Tempel (1985), "other flavors of recursion may not be encountered at all" by the learners.

Experiments with experts and novices were set up by Kahney to validate or refute the hypothesis that students had a looping mental model. With high probability, it appeared that most of the students held the looping model instead of the copies model, and some of them had idiosyncratic models in mind, like the *null model* (when recursion is rejected), the *syntactic model* (when the structure of the program is used to predict its outcome, or, when writing it, the necessity of base and recursive cases is understood, but not the derivation of the actions), and the *odd model* (when the meaning of some keywords, *e.g.*, `EXIT` and `CONTINUE`, is taken from their English usage).

To explain the odd model, Paz and Lapidot (2004) suggest to consider the interference of natural language in learning recursion, in the context of learning `DrScheme`:

> *It may be that some students attribute to the function the ability to change the parameter's value, because of the association they create between the programming language and natural language. It is possible that students [...] interpret the expression* `(first L)` *as, for example, 'take the first element'. The meaning of taking the first element, for them, is to extract it and drop the remaining elements, so that* `L` *is left only with the first element.*

In the same vein, some researchers insist on bringing to the fore and qualifying the linguistic aspect of the relationship between learners and teachers. They set up experiments, record all interactions with software and video, then analyse the transcripts to pinpoint the misunderstandings, trace them back to plausible causes and try to capture the mental model at work (Anderson *et al.*, 1984, Levy and Lapidot, 2000, Levy *et al.*, 2001, Levy, 2001, Murnane and Warner, 2001, Levy and Lapidot, 2002). Furthermore, these verbal exchanges can be conducted not solely to infer a mental model of the student and reach a diagnostic and remedy, but even to become a maieutic process on its own right (Chang *et al.*, 1999, 2000).

Götschi and some collaborators (Götschi, 2003, Götschi *et al.*, 2003, Sanders *et al.*, 2006) refined and extended Kahney's classification of mental models; for instance, they identified amongst their university students an *active model*, when they understand the instantiations of recursive calls with smaller arguments and the reaching of the base cases, but they nevertheless fail to grasp the backward, or passive, flow of control from the completed instances to the current, pending one. They also proposed the *step model*, whereby students have not a complete concept of recursive flow of control and execute only one recursive call yielding a base case. There is also the *return value model*, which stems from misconceptions about when the values of function calls are constructed. The two last models are linked to some confusion about the evaluation of function calls in general, like parameter passing and making a function's return value.

Bhuiyan *et al.* (1989) (1991) prefer the expression *mental method* instead of *mental model* and proposed a more detailed classification where *generative methods* comprise the *loop method*, the *syntactic method*, the *analytic method*, and the *analysis/synthesis method*; moreover, *trace methods* (Bhuiyan, 1992, Scholtz and Sanders, 2010) are used by students to verify the correctness of their solutions. (Götschi *et al.,* (2003) define a trace as "a student's representation of the flow of control and the calculation of the solution of a recursive program.") The loop method is the obvious consequence of the flawed loop mental model. The syntactic method is frequently used by novices who have little understanding of recursion as a problem-solving method, but a good declarative knowledge about it. They know how to lie out a recursive template with base cases and recursive cases fitting into simple categories, and it works well for a wide variety of simple problems, but they have difficulties for more complex ones, for example when *generative recursion* (Felleisen *et al.*, 2004) is needed, that is, when a recursive call does not apply directly to a substructure of the input, but to a transformed substructure. This issue is linked to a lack of understanding of recursion as a design method, there-

fore, the next method, *i.e.,* the analytic method, applies to slightly complex problems and proceeds from input and output requirements to an intermediary solution, before writing the code. The analysis/synthesis method goes further by dividing the problem into subproblems whose solutions must be combined: this is the most general method. (See earlier paragraph on methodology.) Dicheva and Close (1996) (1997) focused on misconceptions. Wu (1993) and Wu, Dale and Bethel (1998) explored the learning of recursion in the framework of David Kolb's model (*experiential learning theory*), which we cannot detail here. For yet other angles, like programming competences, concrete vs. abstract models, static vs. dynamic copies model, classes of recursive functions etc. see Er (1995), Burton (1995), Chen (1998) and Mirolo (2010).

Anzai and Uesato (1982) found that children's understanding of a recursive definition in the context of mathematics is eased by prior experience with iteration, although they added that it may be the case that *writing* recursive definitions in a programming language requires different, additional skills. Kessler and Anderson (1986) worked in the context of programming languages and searched for transfer of skills between tail recursion and iteration for novices and they confirmed the conclusion of Anzai and Uesato (1982): both studies found a positive transfer from writing loops to writing recursive definitions, but not vice versa (although tail recursion is arguably too similar to iteration). Moreover, it seems that the incorrect looping model of recursion, previously acquired on loops, is more helpful than learning recursion directly. By contrast, Wiedenbeck (1988) found that previous knowledge of iterative examples does not seem to facilitate subsequent learning on similar recursive problems, although *comprehension* was slightly improved. Furthermore, Kurland and Pea (1985) studied how 12 year old subjects understood recursive definitions and iterations in `Logo`. They found that previous familiarity with iteration helps understanding tail recursion but hampers the correct grasping of embedded recursion, in accord with later work by Murnane (1992). Note that this is not a direct contradiction of Kessler and Anderson (1986), because the latter used tail recursion, and, for Wiedenbeck (1988), the transfer of skills is about comprehension, not design.

The role of examples in learning recursion has been investigated by Pirolli and Anderson (1985), Wiedenbeck (1989), Pirolli (1991) and Tascón-Vidarte *et al.* (2010). Examples should be used to develop *analogical* problem-solving mechanisms, but care must be taken not to rely too much on them too early, lest the learners get stuck in the syntactic model of Kahney, and *knowledge compilation* mechanisms should also be built from past experiences.

4. Visualisation and Animation

Many educators try to capitalise on the fact that vision plays an important role in acquiring concepts and informing their composition to build new ones. This opens different lines of inquiry: visual analogies of recursion, animating the evaluation of programs, visual programming languages, integrated development environments, virtual worlds and games.

4.1. *Analogies Objects*

It is often claimed that everyday life lacks analogies for the concept of recursion (Pirolli and Anderson, 1985), so it is no surprise that most authors come up with the same objects, such as cauliflowers, including the healthy broccoli, ringed targets, tree branches, reflections on facing mirrors, tilings (Chu and Johnsonbaugh, 1987), ladders (Levy and Lapidot, 2002) and Russian dolls (Bowman and Seagraves, 1985). Typical geometric figures are fractals (Riordon, 1984*b*, Elenbogen and O'Kennon, 1988, Wakin, 1989, Bruce *et al.*, 2005, Ammari-Allahyari, 2005, Stephenson, 2009*b*, Gordon, 2006) and certain kinds of artwork, most notably by the Dutch graphic artist M. C. Escher (Gunion *et al.*, 2009*b*). Their structures are characterised by self-replication with self-embedding (also called *nesting*), but, unfortunately, these examples are perhaps more likely to suggest infinity than recursion (whose evaluation must terminate to be useful and, in the case of embedded recursion, may require backtracking), and this involuntary association of infinity and recursion may explain the avoidance of the latter by novices (Wiedenbeck, 1989). By contrast, and with a more optimistic tone, Papert (1980) (p. 71) wrote the following about an exercise with Logo aimed at demonstrating recursion:

> *Thus we have a trick called "recursion" for setting up a never-ending process whose initial steps are shown [...]. Of all ideas I have introduced to children, recursion stands out as the one idea that is particularly able to evoke an excited response. I think this is partly because the idea of going on forever touches on every child's fantasies and partly because recursion itself has roots in popular culture. For example, there is the recursion riddle: If you have two wishes what is the second? (Two more wishes.) And there is the evocative picture of a label with a picture of itself. By opening the rich opportunities of playing with infinity the cluster of ideas represented by the [...] procedure puts a child in touch with something of what it is like to be a mathematician.*

But it seems difficult to generalise this observation, as Turkle (1984) reports that "Matthew, a good-natured and precocious child of five, was eagerly learning to write computer programs to make graphic designs on the screen. His mood changed abruptly and he left the computer in tears when he understood how to make a recursive program: a program whose action includes setting in motion an exactly similar program whose action includes setting in motion an exactly similar program, and so on." McDougall (1991), also using Logo, reported that recursion in objects (figures produced by Logo processes) is firmly conceived by her daughter as different from recursion in processes or programs, but claimed that this conception is nevertheless useful because her daughter, who did not confuse iteration and embedded recursion, used it for teaching a peer. Earlier, Thompson (1985), also observing the dichotomy, asked the students to describe verbally the recursive structure and move towards the recursive Logo program. Murnane (1991) discussed the demerits and merits of various models of recursion, and also uses Logo. (Section 4.5 will be devoted to Logo.)

Processes. Researchers have also looked at everyday examples of recursive processes, instead of objects, for example, the fall of dominoes aligned in a row, which seems to suggest recurrent reasoning to children who are 12 years old or so, in the sense that they almost express the fall of *any* domino by the fall of the previous one (a local property), but descriptions by younger children are of the iterative type: the first domino falls and lets the second fall, and so the third will fall etc. (Piaget and Stratz, 1974) Of course, the recursion suggested here by the experiment is tail recursion, because it ends with the fall of the last domino. Nevertheless, Yang (2004) (2008) went further and claimed that such series of dominoes are an analogy for *linear recursion*, which is an embedded recursion with one recursive call. More accurate are the processes which use backtracking, a distinctive feature of embedded recursion, to model the behaviour of an avatar or robot stuck in a labyrinth (Liss and McMillan, 1988, Dorf, 1992, Roberts, 2006).

Wirth (2008) provided an entertaining recursive method to randomly park cars in a street, and Brown (1972) tried to familiarise social scientists with recursion through examples in Logo.

Schiemenz (2002) came up with an application of recursion to business management, with more examples of recursive objects and recursive problem-solving. Kimura (1977) used businesses too as a framework for explaining the notions of program, processor and process. Embedded recursion is then expressed as the delegation of a task to a group of assistants working on complementary sets of the input. The same analogy is found in a paper by Edgington (2007), and, if enacted by the students as theatrical roles, it becomes *kinesthesis* and a multi-sensory experience for learning recursion in the classroom (Dorf, 1992, Levine, 2000, Begel *et al.*, 2004, Kátai, 2009). In particular, Ben-Ari (1996) proposed to dramatise recursive algorithms, that is, to associate the solution to a real-world task with an algorithm having the same recursive structure, *e.g.*, eating a chocolate bar and searching an array. The students enact the solution and later write the program. These playful activities can be considered as kinds of parlour games, which leads us now to review computer games dedicated to recursion.

4.2. *Computer Games*

There is an increasing interest for games, or game-like features (*e.g.*, achievement badges), for supporting educational purposes (so-called *gamification* of education), even in higher education. Although it was mentioned in section 3 that "The Tower(s) of Hanoi" has long been quite popular, very few studies have been carried out specifically about teaching recursion with video games. Amongst them, the setting of Rossiou and Papadakis (2007) is a virtual classroom, and Chaffin *et al.* (2009) designed a game to facilitate the transfer of skills to writing recursive programs. While very limited in time and number of participants, both studies support the use of computer games for teaching and learning recursion as a concept. As an alternative to games, recursive processes can also be merely illustrated by a series of snapshots or by an animation. The simplest form of visualisation consists in augmenting the text of a program with semantic annotations and pictures.

4.3. *Augmented Text*

Since the 1970s, many graphical notations for inputs and *activation trees*, sometimes called *recursion graphs*, have been proposed, allowing novices to record and follow the evaluation of function calls (Jackson, 1976, Kruse, 1982, Haynes, 1995, Hsin, 2008). For example, if the input is a binary tree to be traversed, the activation tree is also a binary tree because each node is a call to the same function but with different subtrees as arguments. The teacher can show on the blackboard the location in the data diagram at the same moment that the activation tree is extended. Wei and Murray (2008) draw activation trees within a hyperbolic geometry. Moreover, memory allocation and variable assignments decorate the corresponding `Java` program. Kurtz and Johnson (1985) animated the data diagram only.

The syntax of many imperative languages, like `Pascal`, is based on blocks, which makes it hard to trace the execution of function calls, particularly in the presence of recursion (Er, 1984). This leads some instructors to recur to a low-level simulation of the execution, reifying the otherwise invisible control stack (Lee and Mitchell, 1985, Dupuis and Guin, 1989), but, according to Ginat and Shifroni (1999), this puts too much emphasis on the computing model (see also the paragraph about methodology). See also Pirolli (1986), whom we mentioned earlier in the section about functional programming.

Bell and Gilbert (1974) proposed to use Wirth's syntax diagrams, designed for specifying grammars of programming languages like `Pascal`. The usefulness of Backus-Naur Forms (defining context-free languages) and Lindenmayer systems (L-systems) to teach recursion has also been noted by Er (1984), Proulx (1997) and Velázquez-Iturbide (1999) (2000). Zelenski (1999) proposed the generation of random sentences to experiment recursion and Levine (2000) then commented that students have no trouble at all, perhaps because the textual expansion of a non-terminal is hardly seen as a procedure call, let alone calling itself.

For other closely related approaches, also based on annotations and pictures, see Er (1995), Hui and Iverson (1995), Jehng *et al.* (1999), George (1995) (1996) (2000*a*) (2000*b*) and Tung *et al.* (2001), some of whom we mentioned earlier about rewrite systems and functional languages.

4.4. *Multimedia Environments*

Animation has been more widely implemented by means of dedicated multimedia environments (Rosenthal, 2005), either in isolation (for didactic purposes only), or in connection with programming environments (Wilcocks and Sanders, 1994). Here, we will only review briefly those systems designed specifically to teach recursion.

Stern and Naish (2002*a*) (2002*b*) proposed a classification based on an analysis of recursive algorithms for sorting arrays and updating dictionaries: the first category groups searches, the second sorts and the last insertions. They claim that such distinctions enable the tailoring of better animations, aimed at reinforcing the understanding of recursion. Fernández-Muñoz *et al.* (2007) proposed and implemented an automated classification based on source code inspection from which a dedicated animation is gen-

erated. That system was developed extensively (Velázquez-Iturbide *et al.*, 2008) (2009*a*) (2009*b*) (Velázquez-Iturbide and Pérez-Carrasco, 2010). The approach is practical and eclectic, with animations not only of the activation tree, but also of the data structure, the trace and the control stack.

Another direction is open by *intelligent tutoring systems* (Pirolli, 1986) (or *interactive learning environments*), which are multimedia environments that provide interactive feedback and advice to the programmer. The system contains typical beginners' strategies so it can comment upon the code being written (McCalla and Greer, 1993). These strategies are based on mental models of the learners. It seems that the system designed by Greer (1987) became a reference for Bhuiyan *et al.* (1989) (1992) (1994), Bhuiyan (1992) and Greer *et al.* (1994).

As miscellanea, see Moreno-Armella (1992), Wu *et al.* (1996), Wu, Lee and Lin (1998). For a structured text editor guaranteeing the termination of recursive predicates in `Prolog`, see Bundy *et al.* (1991).

Visual programming. Visual programming languages enable the composition of program constructs by manipulating graphical representations instead of writing text. Good and Brna (1996) were the first to investigate whether these languages provided a better support for learning recursion than textual languages, and concluded negatively. Spreadsheet languages are sometimes considered as visual programming languages or even functional languages, and Burnett *et al.* (2001) focused on testing recursive programs with them. Kim (2003) proposed a string of classroom exercises to learn recursion with `Excel`.

Virtual worlds. Tascón-Vidarte *et al.* (2010) designed an interactive interface based on a tangible block-world with augmented reality to learn iteration on lists and aiming at the transfer of skills to directly write tail recursive definitions in `Erlang`. An earlier, three-dimensional virtual world was designed by Dann *et al.* (2001). For two-dimensional geometry considered as a virtual world, we have `Logo`.

4.5. *The `Logo` Years*

We would be remiss not to devote a whole section to `Logo`. The first thing that strikes the reader of the abundant literature about `Logo` is the enthusiasm that blows, page after page. Microcomputers were arriving in the classrooms and everyone was excited and deeply interested in their programming: teachers, of course, but also psychologists, didacticians, mathematicians, computer scientists, software companies, and even the children themselves, whose education was the focal point of attention. The geometric figures produced by the execution of `Logo` programs, the design underpinnings of the language, like its recursive, functional programming style and its grounding in developmental psychology, all this put `Logo` at the confluence of almost all the streams surveyed here: dynamic and geometric analogies for recursion, virtual worlds (in a more abstract sense, they are called *microworlds* in the context of `Logo`, like the microworld of the turtle, the microworld of words, of lists etc.), integrated environments, functional programming, games and theory of learning. Papert (1980) was a pioneer of this movement, taking part to the design of `Logo` at the end of the 1960s, and we quoted him in section 4.1 about recursion.

McDougall (1985) (1988) (1989) (1990*a*) (1990*b*) (1991) has used `Logo` to teach her nine-year-old daughter, who ended mastering embedded recursion by age eleven. According to her, this result confirmed what Papert conjectured, namely that young children in a computer-rich environment can learn abstract or formal thinking – In passing, Papert never attributed this capability to `Logo` alone. Unfortunately, the size of the study group makes it hard to generalise the findings. Rouchier (1986*b*) (1986*a*) (1987) observed adolescents' difficulties in learning embedded recursion after understanding loops and iteration, and proposed to start teaching embedded recursion first. See also the articles by Barfurth (1987), Barfurth and Retschitzki (1987).

Following in the same footsteps, others (Gobet *et al.*, 1989, Retschitzki *et al.*, 1989, Gurtner *et al.*, 1990, Retschitzki *et al.*, 1991) noted that, in the geometric microworld of `Logo`, it is difficult to come up with exercises which show the superiority of embedded recursion over iteration, whereas the microworld of lists is more pertinent. Perhaps the reason is that drawing is inherently a side-effect thereby it empowers loops. In the case of `PostScript`, a concatenative programming language dedicated to graphics, the implicit evaluation stack is used for all computations, including delaying the side-effect of drawing, which is triggered by an explicit `showpage` instruction, so programming remains declarative. Unfortunately, once embedded recursion has been wrongly understood as an iteration in the turtle microworld, the misunderstanding is carried over to the microworlds of words and lists. Moreover, these researchers observed the same difficulties in recognising the base cases (`STOP` rule) as with any other programming language. Give'on (1991) presented a variant of `Logo` with multiple turtles that can move concurrently, and advocated that this paradigm yields simpler recursive programs than traditional (singly threaded) dialects of `Logo`.

Conclusion

The teaching and learning of recursion in computer programming courses has long been a subject of inquiry, attracting a wide range of researchers from many fields of knowledge. It is not possible to isolate a current trend of investigation, as the hallmark of the newest papers can already be found in the early 1990s, although there seems to be a recent decline in the number of publications and a concentration around a few researchers. Here are a few points that may deserve some attention.

- Perhaps the common weakness of many experimental protocols lies in the small number of students (usually, one class), the short span of time (usually, one semester) and the difficulty to define a control group. Consequently, it may help to bring on board statisticians in order to design larger and longer experiments (at least a three-year period).
- Many studies lump all novices, whereas it seems useful to distinguish different profiles and cater them with different learning strategies, as some have proposed. But since the identification of the student mental model can only be achieved by teaching, this begets the question of *adaptive teaching strategies*, once the student has been classified.

- The approaches based on text rewriting (grammars, L-systems, rewrite systems) do not seem to raise issues with learners as far as recursion is concerned. It would be interesting to confirm this and explore whether the purported skills can be transferred to block-structured programming languages.
- Mutual recursion has been studied by Rubio-Sánchez and his colleagues (Rubio-Sánchez and Pająk, 2006, Rubio-Sánchez *et al.*, 2008), who deemed it sometimes easier to teach than direct recursion. If confirmed, this would open a new way to teach direct recursion by program transformation (inlining) (Kaser *et al.*, 1993). Examples of mutual recursion arise naturally in parsers, which were a favourite example in early textbooks, and it was noted above that the derivation of sentences from formal grammars (that is, the reverse function of parsing) usually does not raise problems with recursion. Another use case is finite automata, as found in telecommunication protocols, vending machines, automatic teller machines etc. (One state is implemented by one function whose argument is any of the labels on the outgoing transitions.)
- Kinesthesis and syntonicity seem to be helpful and should be compared with animation, as it may be that watching or imagining the execution of a recursive function (in other words, tracing) is cognitively different from involving one's own body, or a psychological representation of it. Perhaps augmented reality may help too, by creating an immersion (Tascón-Vidarte *et al.*, 2010).
- It should be impressed upon students that the control flow of recursion which many authors qualify as being "bidirectional", is actually not specific to recursion by explaining the evaluation of arithmetic expressions with function compositions (Burge, 1975). (In imperative languages where instructions are separated by semicolons, an instruction can be shown to be an implicit function – an assignment is indeed an operator in the `C` family – and a semi-colon denotes composition.)
- Many educators teaching recursion focus on the control flow, except perhaps if the language is object-oriented, because, in that case, the *data flow* becomes more relevant, and the design is more likely to be bottom-up. (An algorithm ends up being scattered amongst several methods in different classes, so recursion is obscured by the amount of code to be read and mutual recursion is more likely.) That difference may explain why the professors teaching structural recursion on lists before arrays and loops are using some object-oriented language or a functional language. Those teaching a top-down design may end up reordering the definitions in the program to have them compiled incrementally for testing purposes, and also because this corresponds to the order of synthesis. (See the analysis/synthesis method.) By strictly lying down the top-down design in the code, which requires, for example, to use prototypes in `C`, or `forward` declarations in `Pascal`, the students get used to read incomplete programs. (The same can be said about using modules, of course.) Perhaps that skill is correlated with a better understanding of recursion.
- Tail call optimisation should be explained without resorting to low-level concepts (Rinderknecht, 2012).

Acknowledgements

The research was carried out as part of the EITKIC_12-1-2012-0001 project, which is supported by the Hungarian Government, managed by the National Development Agency, financed by the Research and Technology Innovation Fund and is performed in cooperation with the EIT ICT Labs Budapest Associate Partner Group (ictlabs.elte.hu).

The author thanks the following researchers for providing preprints, hard copies and otherwise helpful information: Nell Dale, C. Mitchell Dayton, Carlisle George, David Ginat, Shafee Give'on, Fernand Gobet, Jean-Luc Gurtner, Kátai Zoltán, Anne McDougall, John Murnane, Peter Pirolli François Pottier, Ian Sanders, Manuel Rubio Sánchez, Guiseppe Trautteur, Ángel Velázquez and Michael Zmuda. The anonymous reviewers helped improve the introduction.

References

Abramovich, S., Pieper, A. (1996). Fostering recursive thinking in combinatorics through the use of manipulatives and computing technology. *The Mathematics Educator*, 7(1).

Ammari-Allahyari, M. (2005). *Exploring Students' Understanding of the Relationship Between Iteration and Recursion*. Institute of Education University of Warwick, United Kingdom.

Anderson, J.R., Pirolli, P.L., Farrell, R. (1984). Learning to program recursion. In: *Proceedings of the Annual Conference of the Cognitive Science Society*. Boulder, Colorado, USA, 277–280.

Anderson, J.R., Pirolli, P.L., Farrell, R. (1988). In: Chi, M.T.H., Glaser, R., Farr, M.J. (Eds.), *The Nature of Expertise*. Lawrence Erlbaum, Hillsdale, New Jersey, USA, 153–183.

Anderson, O. D. (1992). Induction, recursion, and the Towers of Hanoi. *International Journal of Mathematical Education in Science and Technology*, 3, 339–343.

Anonymous. (1977). Depth-first digraph algorithms without recursion. In: *Proceedings of the Seventh International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Atlanta, Georgia, USA, 151–153.

Anzai, Y., Uesato, Y. (1982). Learning recursive procedures by middle-school children. In: *Proceedings of the Annual Conference of the Cognitive Science Society*. Ann Arbor, Michigan, USA, 100–102.

Apostel, L. (1991). Elusive recursiveness – The necessity of a dynamic and pragmatic approach: a response to Vitale. *New Ideas in Psychology*, 9(3), 367–373.

Astrachan, O. (1994). Self-reference is an illustrative essential. In: *Proceedings of the Twenty-fifth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Phoenix, Arizona, USA, 238–242.

Augenstein, M., Tenenbaum, A. (1976). A lesson in recursion and structured programming. In: *Proceedings of the Sixth International Technical Symposium on Computer Science and Education*. ACM SIGCSE-SIGCUE, ACM Press Anaheim, California, USA, 17–23.

Barfurth, M. (1987). *Recursion: What is it? Memorandum No.31*. Institut de Psychologie, Université de Fribourg, Fribourg, Switzerland.

Barfurth, M., Retschitzki, J. (1987). The pedagogical relevance of children working with recursion. In: *Proceedings of the Third Conference* "*Logo and Mathematical Education*". Montréal, Canada, 3, 164–172.

Barron, D.W. (1968). *Recursive techniques in programming*. Computer Monographs, MacDonald/Elsevier.

Begel, A., Garcia, D.D., Wolfman, S.A. (2004). Kinesthetic learning in the classroom. In*: Proceedings of the Thirty-fifth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press Norfolk, Virginia, USA, 183–184.

Bell, S., Gilbert, E.J. (1974). Learning recursion with syntax diagrams. *The SIGCSE Bulletin*, 6(3), 44–45.

Ben-Ari, M. (1996). Recursion: from drama to program. *Aspects of Teaching Computer Science,* 7, 45–47.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching,* 20(1), 45–73.

Benander, A.C., Benander, B.A. (2008). Student monks – Teaching recursion in an IS or CS programming course using the Towers of Hanoi. *Journal of Information Systems Education,* 19(4), 455–468.

Bhuiyan, S., Greer, J.E., McCalla, G.I. (1989). Mental models of recursion and their use in the SCENT programming advisor. In: *Proceedings of the International Conference on Knowledge-based Computer Systems* (*LNCS*, *444*). Springer-Verlag, Bombay, India, 135–144.

Bhuiyan, S., Greer, J.E., McCalla, G.I. (1991). Characterizing rationalizing, and reifying mental models of recursion. In: *Proceedings of the Thirteenth Annual Meeting of the Cognitive Science Society*. Psychology Press, Hillsdale, New Jersey, USA, 120–125.

Bhuiyan, S., Greer, J.E., McCalla, G.I. (1992). Learning recursion through the use of a mental model-based programming environment. In: *Proceedings of the Second International Conference on Intelligent Tutoring Systems* (*LNCS*, *608*). Springer, Montréal, Canada, 50–57.

Bhuiyan, S., Greer, J.E., McCalla, G.I. (1994). Supporting the learning of recursive problem solving. *Interactive Learning Environments*, 4(2), 115–139.

Bhuiyan, S.H. (1992). *Identifying and Supporting Mental Methods of Recursion in a Learning Environment*. PhD thesis, University of Saskatchewan, Saskatoon Canada.

Bloch, S. (2003). Teaching linked lists and recursion without conditionals or null. *Journal of Computing Sciences in Colleges*, 18(5), 96–108.

Bowman, B.C., Seagraves, K. (1985). Picturing recursion. *The Computing Teacher*, 12(7), 28–32.

Brandt, K., Richey, M. (2004). Studying mathematical induction and recursive programming together. *Journal of Computing Sciences in Colleges*, 19(4),. 108–114.

Brooks, A., Miller, J., Roper, M., Wood, M. (1992). Criticisms of an empirical study of recursion and iteration. *Technical Report EFoCS-1-92. Empirical Foundations of Computer Science*. Department of Computer Science, University of Strathclyde, Glasgow, United Kingdom.

Brown, J.S. (1972). Recursive functional programming as a conceptual tool for social scientists. In: *Proceedings of the ACM Annual Conference, 1*. ACM SIGCUE, ACM Press, Boston, Massachusetts, USA, 320–320.

Bruce, K.B., Danyluk, A., Murtagh, T. (2005). Why structural recursion should be taught before arrays in CS1. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, St. Louis, Missouri, USA, 246–250.

Buck, R.C. (1963). Mathematical induction and recursive definitions. *American Mathematical Monthly*, 70(2), 128–135.

Bundy, A., Grosse, G., Brna, P. (1991). A recursive techniques editor for Prolog. *Instructional Science,* 20(2–3), 135–172.

Buneman, P., Levy, L. (1980). The towers of Hanoi problem. *Information Processing Letters,* 10(4–5), 243–244.

Burge, W.H. (1975). *Recursive Programming Techniques: The Systems Programming*. Addison-Wesley.

Burnett, M., Ren, B., Ko, A., Cook, C., Rothermel, G. (2001). Visually testing recursive programs in spreadsheet languages. In: *Proceedings of the IEEE Symposia on Human-centric Computing Languages and Environments*. Stresa, Italy.

Burton, C.T.P. (1995). Conceptual structures for recursion. In: *Proceedings of the First International Symposium on Functional Programming Languages in Education* (*LNCS*, *1022*). Springer, Nijmegen, The Netherlands, 179–193.

Chaffin, A., Doran, K., Hicks, D., Barnes, T. (2009). Experimental evaluation of teaching recursion in a video game. In: *Proceedings of the Symposium on Video Games*. ACM SIGGRAPH, ACM Press, New Orleans Louisiana, USA, 79–86.

Chang, K.-E., Lin, P.-C., Sung, Y.-T., Chen, S.-W. (2000). Socratic-dialectic learning system of recursion programming. *Journal of Educational Computing Research*, 23(2), 133–150.

Chang, K.-E., Wang, K.-Y., Dai, C.-Y., Sung, T.-C. (1999). Learning recursion through a collaborative Socratic dialectic process. *Journal of Computers in Mathematics and Science Teaching,* 18(3), 303–315.

Chen, M.-P. (1998). The effect of dynamic copies model in teaching recursive programming. *Journal of Taiwan Normal University,* 43(1), 63–78.

Chu, I.-P., Johnsonbaugh, R. (1987). Tiling and recursion. In: *Proceedings of the Eighteenth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, St. Louis, Missouri, USA, 261–263.

Clack, C.D., Myers, C. (1995). The dys-functional student. In: *Proceedings of the First International Symposium on Functional Programming Languages in Education. Number* (*LNCS*, *1022*). Springer, Nijmegen, The Netherlands, 289–309.

Close, J., Dicheva, D. (1997). Misconceptions in recursion: diagnostic teaching. In: *Proceedings of the European LOGO Conference* (*Mental Models of Recursion*). Budapest, Hungary.

Cobbe, R. C. (2008). *Much Ado about Nothing: Putting Java's Null in Its Place*. PhD thesis, College of Com-

puter and Information Science, Northeastern University, Boston, Massachusetts, USA.
Costello, P. (1990). Analysis of a recursive algorithm for computing binomial coefficients. *Computer Science Education*, 1(4), 317–329.
da Rosa, S. (2002). The role of discrete mathematics and programming in education. In: *Proceedings of the Workshop on Functional and Declarative Programming in Education*. Technical Report 0210, University of Kiel Pittsburgh, Pennsylvania, USA.
da Rosa, S. (2005). *The Learning Of Recursive Algorithms And Their Functional Formalization*. PhD thesis, Instituto de Computación, Facultad de Ingeniería Universidad de la República, Montevideo, Uruguay.
da Rosa, S. (2007). The learning of recursive algorithms from a psychogenetic perspective. In: *Proceedings of the Workshop of the Psychology of Programming Interest Group*. Joensuu, Finland.
Dann, W., Cooper, S., Pausch, R. (2001). Using visualization to teach novices recursion. In: *Innovation and Technology in Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Canterbury, United Kingdom, 109–112.
Daykin, P.N. (1974). Teaching recursive programming using BASIC. *SIGCUE Outlook*, 8(1), 11–13.
Daylight, E.G. (2010). The advent of recursion in programming (1950s–1960s). In: *Computability in Europe* (*Programs, Proofs, Processes*). Ponta Delgada Azores, Portugal.
Denman, R.T. (1996). Derivation of recursive algorithms for CS2. In: *Proceedings of the Twenty-seventh International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Philadelphia Pennsylvania, USA, 9–13.
Dicheva, D., Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research,* 14(1), 1–23.
Dijkstra, E.W. (1960). Recursive programming. *Numerische Mathematik*, 2(1), 312–318.
Dijkstra, E.W. (1974). Determinism and recursion versus non-determinism and the transitive closure. (Dijkstra archive EWD456).
Dijkstra, E. W. (1975). Correctness concerns and, among other things, why they are resented. In: *Proceedings of the International Conference on Reliable Software*. Los Angeles, California, USA, 546–550.
Dijkstra, E.W. (1999). Computing Science: Achievements and Challenges. (Dijkstra archive EWD1284).
Dorf, M.L. (1992). Backtracking the rat way. In: *Proceedings of the Twenty-third International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Kansas City, Missouri, USA, 272–275.
Dupuis, C., Guin, D. (1989). Représentations du fonctionnement d'une procédure récursive en Logo. In: *Proceedings of the 13th Conference "Psychology of Mathematics Education"*. Paris, France, 220–227.
Early, G.G., Stanat, D.F. (1985). Chinese Rings and Recursion. *The SIGSE Bulletin,* 17(4), 69–82.
Edgington, J. (2007). Teaching and viewing recursion as delegation. *Journal of Computing Sciences in Colleges,* 23(1), 241–246.
Elenbogen, B.S., O'Kennon, M.R. (1988). Teaching recursion using fractals in Prolog. In: *Proceedings of the Nineteenth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press Atlanta, Georgia, USA, 263–266.
Eliot, J., Lovell, K., Dayton, C.M., McGrady, B.F. (1979). A further investigation of children's understanding of recursive thinking. *Journal of Experimental Child Psychology,* 28(1), 149–157.
Er, M.C. (1984). On the complexity of recursion in problem-solving. *International Journal of Man-Machine Studies,* 20(6), 537–544.
Er, M.C. (1995). Process frame: a cognitive device for recursion comprehension. *Computers & Education,* 24(1), 31–36.
Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S. (2001). *How to Design Programs*. The MIT Press.
Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S. (2004). The Structure and Interpretation of the Computer Science Curriculum. *Journal of Functional Programming,* 14(4), 365–378.
Felleisen, M., Friedman, D.P. (1997). *A little Java, a Few Patterns*. The MIT Press.
Fernández-Muñoz, L., Pérez-Carrasco, A., Velázquez-Iturbide, J.A.,Urquiza-Fuentes, J. (2007). A framework for the automatic generation of algorithm animations based on design techniques. In: *Creating new learning experiences on a global scale* (*LNCS, 4753*). Springer, 475–480.
Foltynowicz, I. (2007). Recursion versus iteration with the list as a data structure. *Informatics in Education*, 6(2), 283–306.
Ford, G. (1982). A framework for teaching recursion. *The SIGCSE Bulletin,* 14(2), 32–39.
Ford, G. (1984). An implementation-independent approach to teaching recursion. *The SIGCSE Bulletin,* 16(1), 213–216.
George, C.E. (1995). Supporting the learning of recursion. In: *Proceedings of the Conference on the Teaching of Computing*. Dublin City University, Ireland.

George, C.E. (1996). *Investigating the Effectiveness of a Software-Reinforced Approach to Understanding Recursion*. PhD thesis, University of London London, United Kingdom.

George, C.E. (2000*a*). EROSI – Visualizing recursion and discovering new errors. In: *Proceedings of the International Technical Symposium on Computer Science Education.* ACM Press, Austin, Texas, USA, 32, 305–309.

George, C.E. (2000*b*). Experiences with novices: the importance of graphical representations in supporting mental models. In: Blackwell, A.F., E. Bilotta,E. (Eds.), *Proceedings of the Workshop of the Psychology of Programming Interest Group*. Cosenza, Italy, 33–44.

Ginat, D. (2004). Do senior CS students capitalize on recursion?. In: *Proceedings of the Ninth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Leeds, United Kingdom, 82–86.

Ginat, D. (2005). The suitable way is backwards, but they work forward. *Journal of Computers in Mathematics and Science Teaching,* 24(1). 73–88.

Ginat, D., Armoni, M. (2006). Reversing: an essential heuristic in program and proof design. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Houston, Texas USA, 469–473.

Ginat, D., Shifroni, E. (1999). Teaching recursion in a procedural environment – How much should we emphasize the computing model?. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, New Orleans, Louisiana, USA, 127–131.

Give'on, Y.S. (1989). Teaching recursive program composition in procedural environments. *Machine-Mediated Learning,* 3, 125–145.

Give'on, Y.S. (1990). Is recursion well defined??. *Computers & Education*, 14(1), 35–41.

Give'on, Y.S. (1991). Teaching recursive programming using parallel multi-turtle graphics. *Computers & Education,* 16(3), 267–280.

Gobet, F., Núñez, R., Retschitzki, J. (1989). Learning recursion with LOGO: adolescents' difficulties. In: *Proceedings of the Second European LOGO Conference*. Gent, Belgique, 398–409.

Goldberg, M., Wiener, G. (2009). Anonymity in Erlang. In: *Erlang User Conference*. Stockholm.

Goldwasser, M., Letscher, D. (2007). Teaching strategies for reinforcing structural recursion with lists. In: *Proceedings of the Conference on Object Oriented Programming, Systems, Languages and Applications*. ACM SIGPLAN, ACM Press, Montréal, Québec, Canada, 889–896

Good, J., Brna, P. (1996). Scaffolding for recursion; can visual languages help?. *IEE Seminar Digests,* 10(7), 1–3.

Gordon, A. (2006). Teaching recursion using recursively-generated geometric designs. *Journal of Computing Sciences in Colleges*, 22(1), 124–130.

Götschi, T. (2003). *Mental models of recursion*. Technical report, Faculty of Science, University of the Witwatersand, Johannesburg, South Africa.

Götschi, T., Sanders, I., Galpin, V. (2003). Mental models of recursion. In: *Proceedings of the Internationl Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Reno, Nevada, USA, 346–350.

Greer, J.E. (1986). *Techniques for teaching recursion*. Research Seminar presented to the Department of Computer Sciences, University of Texas at Austin, USA.

Greer, J.E. (1987). *An Empirical Comparison of Techniques for Teaching Recursion in Introductory Computer Sciences*. PhD thesis, University of Texas at Austin USA, Department of Mathematics and Computer Science Education.

Greer, J.E. (1989). A comparison of instructional treatments for introducing recursion. *Computer Science Education*, 1(2), 111–128.

Greer, J.E., McCalla, G.I., Price, B., Holt, P. (1994). Supporting the learning of recursion at a distance. In: *Proceedings of the World Conference on Educational Multimedia and Hypermedia* (*ED-MEDIA*). Vancouver, British Columbia, Canada, 652.

Gunion, K., Milford, T., Stege, U. (2009*a*). Curing recursion aversion. In: *Proceedings of the Fourteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Paris, France, 124–128.

Gunion, K., Milford, T., Stege, U. (2009*b*). The paradigm recursion: is it more accessible when introduced in middle school?. *Journal of Problem Solving*, 2(2), 142–172.

Gurtner, J.-L., Gex, C., Gobet, F., Núñez, R., Retschitzki, J. (1990). La récursivité rend-elle l'intelligence artificielle?. *Revue Suisse de Psychologie*, 49(1), 17–26.

Haberman, B. (2004). How learning logic programming affects recursion comprehension. *Computer Science Education,* 14(1), 37–53.

Haberman, B., Averbuch, H. (2002). The case of base cases: why are they so difficult to recognize? Student difficulties with recursion. In: *Proceedings of the Seventh Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Aarhus, Denmark, 84–88.

Harvey, B. (1992). Avoiding recursion. In: Hoyles, C., Noss, R. *Learning Mathematics and LOGO*. The MIT Press Cambridge, Massachusetts, USA, 393–426.

Hausmann, K. (1985). Iterative and recursive modes of thinking in mathematical problem solving processes. In: *Proceedings of the Ninth Conference "Psychology of Mathematics Education"*. Noordwijkerhout, The Netherlands, 1, 18–23.

Haynes, S.M. (1995). Explaining recursion to the unsophisticated. *The SIGCSE Bulletin,* 27(3), 3–6.

Henderson, P.B., Romero, F.J. (1989). Teaching recursion as a problem-solving tool using Standard ML. In: *Proceedings of the Twentieth International Technical Symposium on Computer Science Education*. ACM SIGCSE/IEEE-CS, ACM Press, Louisville, Kentucky, USA, 27–31.

Hoare, T. (2009). Null references: The billion dollar mistake. In: *The Annual International Software Development Conference*. London, England, United Kingdom.
`http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare`

Howland, J.E. (1998). Recursion, iteration and functional languages. *Journal for Computing in Small Colleges*, 13(4).

Hsin, W.-J. (2008). Teaching recursion using recursion graphs. *Journal of Computing Sciences in Colleges,* 23(4), 217–222.

Hui, R. K. W., Iverson, K. E. (1995). Representations of recursion. In: *Proceedings of the Conference on Applied Programming Languages*. ACM SIGAPL, ACM Press, San Antonio, Texas, USA, 91–97.

Hulsizer, A. (2011). *Teaching Recursion through Interactive Media*. Master's thesis, School of Computer Science, College of Engineering, University of Oklahoma, USA.

Inhelder, B., Piaget, J. (1963). De l'itération des actions ą la récurrence élémentaire. In: Gréco, P., Inhelder, B., Matalon, B., Si Piaget, J. (Eds.), *La Formation des Raisonnements Récurrentiels* (*Études d'épistémologie génétique*, *XVII*), Presses Universitaires de France, 47–120.

Jackson, G.A. (1976). A graphical technique for describing recursion. In: *Proceedings of the Sixth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Williamsburg, Virginia, USA, 30–32.

Jehng, J.-C.J., Tung, S.-H., Chang, C.-T. (1999). A visualisation approach to learning the concept of recursion. *Journal of Computer Assisted Learning,* 15(4), 279–290.

Kahney, H. (1983). What do novice programmers know about recursion. In: *Proceedings of the Conference on Human Factors in Computing Systems*. ACM SIGCHI, ACM Press, Boston, Massachusetts, USA, 235–239.

Kahney, H., Eisenstadt, M. (1982). Programmers' mental models of their programming tasks: the interaction of real world knowledge and programming knowledge. In: *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*. Ann Arbor, Michigan, USA, 143–145.

Kaiser, G. (Ed.). (2004*a*). Proof and Discrete Mathematics: Part A. *ZDM: The International Journal on Mathematics Education*, 36(2).

Kaiser, G. (Ed.). (2004*b*). Proof and Discrete Mathematics: Part B. *ZDM: The International Journal on Mathematics Education*, 36(3).

Kaser, O., Ramakrishnan, C.R., Pawagi, S. (1993). On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1–4), 151–164.

Kátai, Z. (2009). Multi-sensory method for teaching-learning recursion. *Computer Applications in Engineering Education*, 19(2), 234–243.

Kay, J.S. (2000). Using the force: how Star Wars can help you teach recursion. *Journal of Computing Sciences in Small Colleges*, 15(5), 277–288.

Kessler, C.M., Anderson, J.R. (1986). Learning flow of control: recursive and iterative procedures. *Human-Computer Interaction,* 2(2), 135–166.

Kieren, T.E. (1989). Observation and recursion in LOGO mathematics: a response to Vitale. *New Ideas in Psychology*, 7(3), 277–281.

Kilpatrick, J. (1985). Reflection and recursion. *Educational Studies in Mathematics*, 16(1), 1–26.

Kim, M. K. (2003). Recursive thinking and solving methods. *Journal of the Korea Society of Mathematical Education, series D* (*Research in Mathematical Education*), 7(4), 211–222.

Kimura, T. (1977). Recursive programming in English for freshmen. In: *Proceedings of the Seventh International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Atlanta, Georgia, USA, 129–132.

Knight, D.G. (1988). Determinants and recursion. *International Journal of Mathematical Education in Science and Technology,* 19(1), 67–71.
Knuth, D.E. (1996). *Selected papers on Computer Science*. CSLI Publications, Stanford University, California, USA, (CSLI Lecture Notes, 59) 205–226.
Knuth, D.E. (2000). *Selected Papers on the Analysis of Algorithms*. CSLI Publications, (CSLI Lecture Notes, 102) 391–414.
Kruse, R.L. (1982). On teaching recursion. In: *Proceedings of the Thirteenth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Indianapolis, Indiana, USA, 92–96.
Kurland, D.M., Pea, R.D. (1985). Children's mental models of recursive LOGO programs. *Journal of Educational Computing Research*, 1(2), 235–243.
Kurtz, B.L., Johnson, D. (1985). Using simulation to teach recursion and binary tree traversals. In: *Proceedings of the Sixteenth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press New Orleans, Louisiana, USA, 49–54.
Lavallade, D. (1985). In search of recursion. In: Hoyles, C., Noss, R. (Eds), *Proceedings of the Conference "Logo and Mathematics Education"*. University of London, Institute of Education, London, United Kingdom.
Lee, P.C., Mitchell, M.A. (1985). Demystifying LOGO recursion: a storage process model of embedded recursion. *Computers in the Schools*, 2(2), 197–208.
Leonard, M. (1991). Learning the structure of recursive programs in Boxer. *Journal of Mathematical Behavior,* 10(1), 17–53.
Leron, U. (1988). What makes recursion hard?. In: *Proceedings of the Sixth International Congress on Mathematics Education*. Budapest, Hungary.
Leron, U., Zazkis, R. (1986). Computational recursion and mathematical induction. *For the Learning of Mathematics*, 6(2), 25–28.
Levine, D. B. (2000). Helping students through multiplicities. *Journal of Computing Sciences in Colleges,* 15(5), 285–291.
Levy, D. (2001). Insights and conflicts in discussing recursion: a case study. *Computer Science Education,* 11(4), 305–322.
Levy, D., Lapidot, T. (2000). Recursively speaking: analyzing students. discourse of recursive phenomena. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Austin, Texas, USA, 315–319.
Levy, D., Lapidot, T. (2002). Shared terminology, private syntax: the case of recursive descriptions. In: Proceedings of the Seventh Annual Conference on *Innovation and Technology in Computer Science Education*. ACM SIGCSE ACM Press, Aarhus, Denmark, 89–93.
Levy, D., Lapidot, T., Paz, T. (2001). It's just like the whole picture, but smaller': expressions of gradualism, self-similarity, and other pre-conceptions while classifying recursive phenomena. In: G. Kadoda (Ed.), *Proceedings of the Workshop of the Psychology of Programming Interest Group*. Bournemouth United Kingdom, 249–262.
Liss, I.B., McMillan, T.C. (1988). An amazing exercise in recursion for CS1 and CS2. *The SIGCSE Bulletin,* 20(1), 270–274.
Lobina Bona, D.J. (2012). *Recursion in Cognition: A Computational Investigation into the Representation and Processing of Language*. PhD thesis University Rovira i Virgili, Department of Psychology, Tarragona, Spain.
Lobina, D.J. (2011). "A running back" and forth: a review of recursion and human language. *Biolinguistics,* 5(1–2), 151–169.
Lobina, D.J., García-Albea, J.E. (2009). Recursion and cognitive science: data structures and mechanisms. In: van Rijn, N.A. (Ed.), *Proceedings of the 31st Annual Conference of the Cognitive Science Society*. Austin Texas, USA, 1347–1352.
Manolopoulos, Y. (2005). On the number of recursive calls of recursive functions. *The SIGCSE Bulletin,* 37(2), 61–64.
Marti, E. (1987). *Différentes Approches (Théoriques et Expérimentales) de la Récursivité*. Institut de Psychologie, Université de Fribourg, Fribourg, Switzerland. (Memorandum, 32).
Martin, M.R. (1985). Recursion – a powerful but often difficult idea. *Computers in the Schools*, 2(2–3),. 209–217.
Matalon, B. (1963). Étude du raisonnement par récurrence sur un modèle physique. In: Gréco, P., Inhelder, B., Matalon, B., Si Piaget, J. (Eds.), *La Formation des Raisonnements Récurrentiels* (*Études d'épistémologie génétique*, *XVII*), Presses Universitaires de France, 283–316.
McCalla, G.I., Greer, J.E. (1992). Helping novices learn recursion: giving granularity-based advice on strategies

and providing support at the mental model level. In: *Proceedings of the NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming*. Genoa, Italy, 57–71.
McCalla, G.I., Greer, J.E. (1993). Two and one-half approaches to helping novices learn recursion. In: Lemut, E., Duboulay, B., Dettori, G. (Eds.), *Cognitive Models and Intelligent Environments for Learning Programming*. Springer Verlag, 185–197.
McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine (part I). *Communications of the ACM*, 3(4), 184–195.
McDougall, A. (1985). Teaching and learning about recursion. In: *Proceedings of the International Conference on LOGO*. Cambridge, Massachussets, USA.
McDougall, A. (1988). *Children, Recursion and Logo Programming*. PhD thesis, Monash University, Melbourne, Australia.
McDougall, A. (1989). Teaching about recursion in Logo: a review. In: Dupe T. (Ed.), *Proceedings of the Australian Conference Computers in Education Conference*. Computer Education Group of the A.C.T., Canberra, Australia.
McDougall, A. (1990*a*). Children, recursion and Logo programming: an investigation of Papert's conjecture about the variability of Piagetian stages in computer-rich culture. In: McDougall, A., Dowling, C. (Eds.), *Proceedings of the IFIP TC 3 Fifth World Conference on Computers in Education (WCCE)*. Sydney, Australia, 415–418.
McDougall, A. (1990*b*). Student difficulties in programming with recursive Logo. In: McDougall, A. (Ed.), *Back to the Future, Forward to the Past*. Computer Education Group of Victoria, Melbourne, Australia, 108–115.
McDougall, A. (1991). Structure and Process Microviews: Partial Understandings of Recursion in Logo Programming. In: *Proceedings of the Fifth Logo and Mathematics Education Conference (LME)*. Lake Tinaroo Queensland, Australia.
McKavanagh, C.W. (1992). Recursion in problem solving. In: *Proceedings of the Joint Conference of the Australian Association for Research in Education and the New Zealand Association for Research in Education*. Deakin University Victoria, Australia.
McKavanagh, C.W. (2004). Recursion in everyday problem-solving. *Australian Vocational Education Review,* 11(1), 35–50.
Mendelsohn, P. (1985). Learning recursive procedures through Logo. In: *Proceedings of the First conference "Logo and Mathematics Education"*. Institute of Education, University of London, London, United Kingdom.
Mirolo, C. (2009). Mental models of recursive computations vs. recursive analysis in the problem domain. In: *Proceedings of the Fourteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Paris, France, 397–397.
Mirolo, C. (2010). Learning (through) recursion: a multidimensional analysis of the competences achieved by CS1 students. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Bilkent University, Ankara, Turkey, 160–164.
Mirolo, C. (2011). Is iteration really easier to master than recursion? Investigation in a functional-first CS1 context. In: *Proceedings of the 16th annual joint onference on Innovation and Technology in Computer Science Education*. Darmstadt, Germany, 362.
Modeste, S. (2012). La pensée algorithmique: apports d'un point de vue extérieur aux mathématiques. In: *Actes du colloque Espace Mathématique Francophone*. Geneva, Switzerland.
Moor, M. (n.d.). A recursion excursion with a surprising discovery. *The Computing Teacher,* 11(5), 49–52.
Moreno-Armella, L. (1992). Visualización y recursividad: un enfoque computacional. In: *Congreso Iberoamericano de Informática Educativa*. CYTED/RIBIE, Santo Domingo, República Dominicana.
Murnane, J. (1991). Models of recursion. *Computers & Education,* 16(2), 197–201.
Murnane, J. (1992). To iterate or to recurse?. *Computers & Education,* 19(4), 387–394.
Murnane, J.S., Warner, J.W. (2001). An empirical study of junior secondary students. expression of algorithms in natural language. In: McDougall, A., Murnane, J., Chambers, D. (Eds.), *Proceedings of the Seventh World Conference on Computers in Education*. Australian Computer Society, Copenhagen, Denmark, 8, 81–85.
Olson, A.T. (1987). The curricular implications of recursion. In: *Proceedings of the Third International Conference for LOGO and Mathematics Education*. Concordia University, Montréal, Canada.
Oudheusden, K.V. (2009). *The Advent of Recursion & Logic in Computer Science*. Master's thesis, Institute for Logic, Language and Computation University of Amsterdam, Amsterdam, The Netherlands.
Papert, S. (1960*a*). Problèmes épistémologiques et génétiques de la récurrence. In: *Problèmes de la construction du nombre*, Presses Universitaires de France, 117–148.
Papert, S. (1960*b*). Problèmes de la construction du nombre. In: Greco, P., Grize, J.B., Papert, S., Piaget, J.,

Problèmes de la construction du nombre, (Études d'épistémologie génétique, *XI*), Presses Universitaires de France.
Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. First edn, Basic Books.
Pareja-Flores, C., Urquiza-Fuentes, J., Rubio-Sánchez, M. (2007). WinHIP: An IDE for functional programming based on rewriting and visualization. *ACM SIGPLAN Notices,* 42(3), 14–23.
Paz, T., Lapidot, T. (2004). Emergence of automated assignment conceptions in a functional programming course. In: *Proceedings of the Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE ACM Press, Leeds, United Kingdom, 181–185.
Peano, G. (1976). The principles of arithmetic, presented by a new method. In: Heijenoort, J. Van, *From Frege to Gödel :A Source Book in Mathematical Logic, 1879–1931*. Third edn, Harvard University Press, 83–97.
Peelle, H.A. (1976). Learning mathematics with recursive computer programs. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, 116–130.
Piaget, J., Stratz, C. (1974). La chute récurrentielle de dominos alignés. In: *Réussir et comprendre*. Presses Universitaires de France, 21–33.
Pirolli, P.L. (1986). A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction*, 2(4), 319–355.
Pirolli, P.L. (1991). Effects of examples and their explanations in a lesson on recursion: a production system analysis. *Cognition and Instruction,* 8(3), 207–259.
Pirolli, P.L., Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology,* 39(2), 240–272.
Polycarpou, I. (2006). Computer science students. difficulties with proofs by induction: an exploratory study. In: *Proceedings of the Southeast Regional Conference*. ACM, Melbourne, Florida, USA, 601–606.
Proulx, V.K. (1997). Recursion and grammars for CS2. In: *Proceedings of the Second Conference on Integrating Technology into Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Uppsala, Sweden, 74–76.
Reingold, E.M. (2012). Four apt elementary examples of recursion. In: Dershowitz, N., Nissan, E. (Eds.), *Language, Culture, Computation: Essays Dedicated to Yaacov Choueka (Lecture Notes in Computer Science)*. Springer-Verlag, Berlin, Germany.
Retschitzki, J. (1986). La récursivité comme méthode générale de résolution de problèmes. *Bulletin du Cours des Animateurs en Informatique,* 15, 4–9.
Retschitzki, J., Gex, C., Gobet, F., Gurtner, J., Núñez, R. (1991). Pensée récursive et enseignement. In: Gurtner, J., Retschitzki, J., *LOGO et Apprentissages*. Delachaux et Niestlé, Neuchatel, Switzerland, 229–240.
Retschitzki, J., Gobet, F., Núñez, R. (1989). *Apprentissage de la Récursivité en LOGO*. Technical Report 75, Département de Psychologie, Université de Fribourg, Fribourg, Switzerland.
Rinderknecht, C. (2012). *Design and Analysis of Purely Functional Programs* (*Texts in Computing, 15*), second edn, College Publications, United Kingdom.
Riordon, T. (1984*a*). Helping students with recursion: teaching strategies (Part II). *The Computing Teacher,* 11(6), 59–64.
Riordon, T. (1984*b*). Helping students with recursion: teaching strategies (Part III: Teaching students about embedded recursion). *The Computing Teacher,* 11(7), 64–69.
Riordon, T. (n.d.). Helping students with recursion: teaching strategies (Part I). *The Computing Teacher,* 11(5), 59–64.
Roberts, E.S. (1986). *Thinking Recursively*. John Wiley & Sons.
Roberts, E.S. (2006). *Thinking Recursively with Java*. John Wiley & Sons.
Robertson, J.S. (1999). How many recursive calls does a recursive function make?. *The SIGCSE Bulletin,* 31(2).
Robinson, R.M. (1947). Primitive recursive functions. *Bulletin of the American Mathematical Society,* 53(10), 925–942.
Robinson, R.M. (1948). Recursion and double recursion. *Bulletin of the American Mathematical Society,* 54(10), 987–993.
Rohl, J.S. (1984). *Recursion via Pascal*. Cambridge Computer Science Texts Cambridge University Press.
Rosenstein, J.G., Franzblau, D.S., Roberts, F.S. (Eds.). (1997). *Discrete Mathematics in the Schools*, *(DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 36)*. American Mathematical Society and National Council of Teachers of Mathematics.
Rosenthal, T. (2005). Introducing recursion by using multimedia. In: *Proceedings of the Tenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Caparica Portugal, 374–374.

Rossiou, E., Papadakis, S. (2007). Educational games in higher education: a case study in teaching recursive algorithms. In: O'Doherty, E. (Ed.), *Proceedings of the Fourth International Conference on Education in a Changing Environment*. University of Salford, Informing Science Press, Salford, United Kingdom, 149–157.

Rouchier, A. (1986*a*). Central recursive calls and nesting in learning Logo programming. In: *Proceedings of the Second Conference "Logo and Mathematical Education"*, *2*. London, United Kingdom.

Rouchier, A. (1986*b*). Learning recursive calls in building up LOGO procedures. In: *Proceedings of the Tenth Conference "Psychology of Mathematics Education", 10*. University of London, Institute of Education, London United Kingdom.

Rouchier, A. (1987). The writing and interpretation of recursive procedures in LOGO. *Psychologie Française,* 32(4), 281–285.

Rubio-Sánchez, M. (2008). An introduction to problem equivalence with combinatorics. In: *Proceedings of the Thirteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE ACM Press, Madrid, Spain, 313–313.

Rubio-Sánchez, M. (2010). Tail recursive programming by applying generalization. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE ACM Press, Bilkent University, Ankara, Turkey, 98–102.

Rubio-Sánchez, M., Hernán-Losada, I. (2007). Exploring recursion with Fibonacci numbers. In: *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Dundee, Scotland, United Kingdom, 359–359.

Rubio-Sánchez, M., Pająk, B. (2006). Fibonacci numbers using mutual recursion. In: Salakoski, T., Mäntylä, Mikko, L. (Eds.), *Proceedings of the Fifth Annual Finnish/Baltic Sea Conference on Computer Science Education, 41*. TUCS General Publications, Finland, 174–177.

Rubio-Sánchez, M., Urquiza-Fuentes, J., Pareja-Flores, C. (2008). A gentle introduction to mutual recursion. In: *Proceedings of the Thirteenth Annual Conference* on *Innovation and Technology in Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press, Madrid, Spain, 235–239.

Rubio-Sánchez, M., Velázquez-Iturbide, J.A. (2009). Tail recursion by using function generalization. In: *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. Paris, France, 394.

Samurçay, R. (1986). Initial representations of students in using recursive Logo procedures. In: *Proceedings of the Tenth Conference "Psychology of Mathematics Education", 10*. University of London, Institute of Education, London, United Kingdom.

Sanders, I.D., Galpin, V.C. (2007). Students. mental models of recursion at Wits. In: *Proceedings of the Twelfth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press Dundee, Scotland, United Kingdom, 317–317.

Sanders, I., Galpin, V., Götschi, T. (2006). Mental models of recursion revisited. In: *Proceedings of the Eleventh Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press Bologna, Italy, 138–142.

Schiemenz, B. (2002). Managing complexity by recursion. In: *Proceedings of the Symposium on Management and Organizational Change. European Meeting on Cybernetics and Systems Research.* Vienna, Austria.

Scholtz, T.L., Sanders, I. (2010). Mental models of recursion: investigating students. understanding of recursion. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Bilkent University, Ankara, Turkey, 103–107.

Segal, J. (1994). Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science,* 22(5), 385–411.

Settle, A. (2013). Reaching the 'Aha!' moment: Web development as a motivator for recursion. In: *Proceedings of the 14th Annual ACM SIGITE Conference on Information Technology Education*. Orlando, Florida, USA, 69–70.

Sher, D.B. (2004). Recursive objects: an object oriented presentation of recursion. *Mathematics and Computer Education*, Winter issue.

Sinha, A.P., Vessey, I. (1992). Cognitive fit: an empirical study of recursion and iteration. *IEEE Transactions on Software Engineering,* 18(5), 368–379.

Skolem, T. (1976). The foundation of elementary arithmetic established by means of the recursive mode of thought without the use of apparent variables ranging over infinite domains. In: Heijenoort, J. Van, *From Frege to Gödel :A Source Book in Mathematical Logic, 1879–1931*. Third edn, Harvard University Press, 302–333.

Soare, R.I. (1996). Computability and recursion. *The Bulletin of Symbolic Logic*, 2(3), 284–321.

Sooriamurthi, R. (2001). Problems in comprehending recursion and suggested solutions. In: *Proceedings of*

the Conference on Innovation and Technology in Computer Science Education. ACM SIGCSE-SIGCUE, ACM Press Canterbury, United Kingdom, 25–28.
Stephenson, B. (2009*a*). Using graphical examples to motivate the study of recursion. *Journal of Computing Sciences in Colleges,* 25(1), 42–50.
Stephenson, B. (2009*b*). Visual examples of recursion. In: *Proceedings of the Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Paris, France, 400–400.
Stern, L., Naish, L. (2002*a*). Animating recursive algorithms. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 4(2).
`http://imej.wfu.edu/articles/2002/2/02/index.asp`.
Stern, L., Naish, L. (2002*b*). Visual representations for recursive algorithms. In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Cincinnati, Kentucky, USA, 196–200.
Stockmeyer, P.K. (2005). *The Tower of Hanoi: A Bibliography.*
`http://www.cs.wm.edu/~pkstoc/biblio2.pdf`
Stojmenovic, I. (2000). Recursive algorithms in computer science courses: Fibonacci numbers and binomial coefficients. *IEEE Transactions on Education,* 43(3), 273–276.
Tascón-Vidarte, J.D., Rinderknecht, C., Kim, J.-I., Kim, H. (2010). A tangible interface for learning recursion and functional programming. In: *IEEE Symposium on Ubiquitous Virtual Reality*. Gwangju Institute of Science and Technology, Gwangju, Republic of Korea.
Tempel, M. (1985). What's so hard about recursion?. In: *Proceedings of the International Conference on LOGO*. Cambridge, Massachussets, USA.
Tessler, J., Beth, B., Lin, C. (2013). Using Cargo-Bot to provide contextualized learning of recursion. In: *Proceedings of Ninth Annual International ACM Conference on International Computing Education Research (ICER)*. San Diego, California, USA, 161–168.
Thompson, P.W. (1985). Understanding recursion: process ≈ object. In: Damarin, S. (Ed.), *Proceedings of the Seventh Annual Meeting of the North American Chapter of the International Group for the Psychology of Mathematics Education*. Ohio State University, Colombus, Ohio, USA, 357–362.
Trautteur, G. (1989). Remarks on recursion: a response to Vitale. *New Ideas in Psychology*, 7(3), 376–378.
Tung, S.-H., Chang, C.-T., Wong, W.-K., Jehng, J.-C. (2001). Visual representations for recursion. *International Journal of Human-Computer Studies,* 54(3), 285–300.
Turbak, F., Royden, C., Stephan, J., Herbst, J. (1999). Teaching recursion before loops in CS1. *Journal of Computing in Small Colleges,* 14(4), 86–101.
Turkle, S. (1984). *The Second Self: Computers and the Human Spirit*, Simon & Schuster, New York, USA.
Velázquez-Iturbide, J.A. (1999). A progressive approach to recursion. In: *Proceedings of the Twenty-ninth Conference on the Frontiers in Education., 1*. IEEE, San Juan, Puerto Rico, 34–38.
Velázquez-Iturbide, J.A. (2000). Recursion in gradual steps (Is recursion really that difficult?). In: *Proceedings of the International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Austin, Texas USA, 310–314.
Velázquez-Iturbide, J.A., Pérez-Carrasco, A. (2010). InfoVis interaction techniques in animation of recursive programs. *Algorithms*, 3(1), 76–91.
Velázquez-Iturbide, J.A., Pérez-Carrasco, A., Urquiza-Fuentes, J. (2008) SRec: an animation system of recursion for algorithm courses. In: *Proceedings of the Thirteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Madrid, Spain, 225–229.
Velázquez-Iturbide, J.A., Pérez-Carrasco, A., Urquiza-Fuentes, J. (2009*a*). A design of automatic visualizations for divide-and-conquer algorithms. In: *Proceedings of the Program Visualization Workshop* (*Electronic Notes in Theoretical Computer Science, 224*), 159–167.
Velázquez-Iturbide, J.A., Pérez-Carrasco, A., Urquiza-Fuentes, J. (2009*b*) Interactive visualization of recursion with SRec. In: *Proceedings of the Fourteenth Annual Conference on Innovation and Technology in Computer Science Education*. ACM SIGCSE, ACM Press, Paris, France, 339–339.
Vitale, B. (1989). Elusive recursion: a trip in recursive land. *New Ideas in Psychology,* 7(3), 253–276.
Wakin, S. (1989). Proof without words: recursion. *Mathematics Magazine,* 62(3), 172.
Wei, X., Murray, K. (2008). A detail+context approach to visualize function calls. *Journal of Computing Sciences in Colleges,* 23(3), 162–167.
Wiedenbeck, S. (1988). Learning recursion as a concept and as a programming technique. In: *Proceedings of the Nineteenth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press Atlanta, Georgia, USA, 275–278.
Wiedenbeck, S. (1989). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies,* 30(1), 1–22.

Wilcocks, D., Sanders, I. (1994). Animating recursion as an aid to instruction. *Computers & Education,* 23(3), 221–226.
Wirth, M. (2008). Introducing recursion by parking cars. *The SIGCSE Bulletin,* 40(4), 52–55.
Wu, C.-C. (1993). *Conceptual Models and Individual Cognitive Learning Styles in Teaching Recursion to Novices.* PhD thesis, University of Texas, Austin, Texas USA.
Wu, C.-C., Dale, N.B., Bethel, L.J. (1998). Conceptual models and cognitive learning styles in teaching recursion. In: *Proceedings of the Twenty-ninth International Technical Symposium on Computer Science Education*. ACM SIGCSE, ACM Press, Atlanta, Georgia, USA, 292–296.
Wu, C.-C., Lee, G.C., Lin, J.M.-C. (1998). Visualizing programming in recursion and linked lists. In: *Proceedings of the Australasian Conference on Computer Science Education, 3*. ACM SIGCSE, ACM Press, University of Queensland, Queensland, Australia, 180–186.
Wu, C.-C., Lin, J.M.-C., Chiou, G.-F. (1996). Visualizing recursion and linked lists. In: *Proceedings of the First Conference on Integrating Technology into Computer Science Education*. ACM SIGCSE-SIGCUE, ACM Press Barcelona, Spain, 232–232.
Yang, F.-J. (2004). The domino effect and linear recursion. In: *Proceedings of the International Conference on Modeling, Simulation, and Visualization Methods*. Las Vegas, Nevada, USA, 201–206.
Yang, F.-J. (2008). Another outlook on linear recursion. *The SIGCSE Bulletin,* 40(4), 38–41.
Zelenski, J. (1999). Nifty Assignments. In: *Proceedings of the Thirtieth SIGCSE Technical Symposium on Computer Science Education*. p. 355.
Zmuda, M., Hatch, M. (2007). Scheduling topics for improved student comprehension of recursion. *Computers & Education,* 48(2), 318–328.

C. Rinderknecht. Christian Rinderknecht's doctoral research at INRIA (French National Institute for Research in Computer Science and Control) dealt with the application of formal methods to telecommunication protocols and was defended in 1998, at Université Pierre et Marie Curie. From 2003 to 2012, he was an Assistant Professor at École Supérieure d'Ingénieurs Léonard de Vinci (France) and Konkuk University (Republic of Korea). Since 2013, he works at Eötvös Loránd University (Budapest, Hungary) in the EIT ICT Labs (`ictlabs.elte.hu`).

Rekursinio programavimo mokymo ir mokymosi apžvalga

Christian RINDERKNECHT

Straipsnyje pateikiama rekursinio programavimo mokymo ir mokymosi literatūros apžvalga. Trumpai apžvelgęs rekursijos pradmenis programavimo kalbose ir kaip ją priima programuotojai, autorius pristato rekursijos mokymo metodus, įskaitant vadovėlių apžvalgą ir keletą programavimo metodologijų, taip pat funkcinę ir imperatyvinę paradigmas bei skirtumą tarp valdymo ir duomenų struktūrų. Autorius pritaria kitiems tyrėjams, teigiantiems, kad problema turi būti nagrinėjama remiantis bendraisiais atvejais, pažymint panašumą su indukcija matematikoje, pateikiant konkrečias rekursijos analogijas, naudojant žaidimus, animaciją, multimediją, virtualiąsias mokymosi aplinkas ir vizualųjį programavimą. Straipsnyje aptariami Logo programavimo kalbos taikymo mokyklose didaktiniai aspektai, taip pat apžvelgiamos konstruktyvistinio ir konstrukcionistinio mokymosi teorijos. Straipsnyje pateikiami anksčiau identifikuoti mokinių mentaliniai modeliai, kurie modifikuoti praplečiant juos kinestetiniu ir sintoniniu modeliais.