Teaching and learning recursive programming: a review of the research literature

Renée McCauleya, Scott Grissomb, Sue Fitzgeraldc & Laurie Murphyd

a Department of Computer Science, College of Charleston, Charleston, SC, USA
b School of Computing and Information Systems, Grand Valley State University, Allendale, MI, USA
c Department of Information and Computer Sciences, Metropolitan State University, St. Paul, MN, USA
d Department of Computer Science and Computer Engineering, Pacific Lutheran University, Tacoma, WA, USA

Published online: 05 May 2015.

To cite this article: Renée McCauley, Scott Grissom, Sue Fitzgerald & Laurie Murphy (2015) Teaching and learning recursive programming: a review of the research literature, Computer Science Education, 25:1, 37-66, DOI: 10.1080/08993408.2015.1033205

To link to this article: http://dx.doi.org/10.1080/08993408.2015.1033205

PLEASE SCROLL DOWN FOR ARTICLE

Taylor & Francis makes every effort to ensure the accuracy of all the information (the “Content”) contained in the publications on our platform. However, Taylor & Francis, our agents, and our licensors make no representations or warranties whatsoever as to the accuracy, completeness, or suitability for any purpose of the Content. Any opinions and views expressed in this publication are the opinions and views of the authors, and are not the views of or endorsed by Taylor & Francis. The accuracy of the Content should not be relied upon and should be independently verified with primary sources of information. Taylor and Francis shall not be liable for any losses, actions, claims, proceedings, demands, costs, expenses, damages, and other liabilities whatsoever or howsoever caused arising directly or indirectly in connection with, in relation to or arising out of the use of the Content.

This article may be used for research, teaching, and private study purposes. Any substantial or systematic reproduction, redistribution, reselling, loan, sub-licensing,
Teaching and learning recursive programming: a review of the research literature

Renée McCauleya*, Scott Grissomb, Sue Fitzgeraldc and Laurie Murphyd

aDepartment of Computer Science, College of Charleston, Charleston, SC, USA; bSchool of Computing and Information Systems, Grand Valley State University, Allendale, MI, USA; cDepartment of Information and Computer Sciences, Metropolitan State University, St. Paul, MN, USA; dDepartment of Computer Science and Computer Engineering, Pacific Lutheran University, Tacoma, WA, USA

(Received 9 September 2014; accepted 11 March 2015)

Hundreds of articles have been published on the topics of teaching and learning recursion, yet fewer than 50 of them have published research results. This article surveys the computing education research literature and presents findings on challenges students encounter in learning recursion, mental models students develop as they learn recursion, and best practices in introducing recursion. Effective strategies for introducing the topic include using different contexts such as recurrence relations, programming examples, fractal images, and a description of how recursive methods are processed using a call stack. Several studies compared the efficacy of introducing iteration before recursion and vice versa. The paper concludes with suggestions for future research into how students learn and understand recursion, including a look at the possible impact of instructor attitude and newer pedagogies.

Keywords: recursion; programming; research; teaching; learning; mental models; student misconceptions; pedagogy

Introduction

Since recursive programming was introduced with ALGOL 60 in 1960 (Daylight, 2011), educators have been investigating, sharing, and publishing ways in which to most effectively teach this topic. Many consider recursive programming challenging for novices. Gal-Ezer and Harel (1998) suggest recursion is “one of the most universally difficult concepts to teach” [p. 83]. Similarly, Roberts (1986) says students perceive recursion as “obscure, difficult and mystical” when they are first introduced to it [p. 1]. McCracken (1987), however, counters that when students first encounter recursion, they don’t know it is “hopelessly difficult” [p. 4] and warns instructors not to suggest it is.

*Corresponding author. Email: MccauleyR@cofc.edu

© 2015 Taylor & Francis
Is recursion really so difficult to learn? If so, what makes it such a challenging topic? And what have we discovered about teaching it for more than 50 years since recursive programming came into being? While many papers have been published on the topic of recursion, relatively few present empirical evidence on its teaching and learning. In this manuscript, we summarize the empirical research specifically focused on teaching and learning recursive programming. Over 35 publications documenting research results related to teaching and learning recursive programming are reviewed here. We explore investigations into the challenges students face when learning recursion, the common misconceptions they form and the mental models they use as they learn to read and write recursive programs. We highlight research useful to educators interested in how to improve the way they teach recursion, including ways to introduce recursion and the use of frameworks and examples. The learner populations involved in these studies range from elementary school children through university-level students. The programming languages used include LISP, LOGO, BASIC, C, Python, SIMPLE, Scheme, Miranda, SOLO, Pascal, and pseudo-code. Research methods varied from quantitative methods applied to large subject populations to qualitative techniques applied to smaller groups. Moreover, in these studies, student acquisition of knowledge was assessed in several dimensions related to recursive thinking:

- **comprehension**, an indication the learner can describe the concept of recursion as well as the intent of a recursive solution;
- **evaluation**, the ability to trace the flow of a recursive invocation and correctly predict the results; and
- **construction** (or **generation**), the ability to write or create a recursive solution.

Our interest in this work stems from many years of teaching introductory-level computer programming courses, including data structures, where recursion is an essential topic for students to grasp and use effectively. Our aim is to present the landscape of previous research on recursion as an aid to educators who wish to apply these results in their own teaching and for researchers who wish to further investigate this topic.

Many papers on recursion lie outside the scope of this review. Much of the early research on recursion focused on the development of tools to teach recursion in the context of specific languages and environments. Research into tools and visualizations continues today, although few individual tools have been widely adopted and thus we limit our discussion to those research studies focused on the teaching and learning of recursive programming that are likely to be of general interest, regardless of the language or environment used. Recursion theory, mathematical recursion, and recursive phenomena also lie outside the scope of this review.
We begin with an overview of challenges and misconceptions students face when learning recursion, followed by a discussion of the mental models students appear to form when studying recursion. We then discuss research into various approaches to introducing recursion, followed by a summary of the implications for teaching suggested by these studies. We conclude with recommendations for future research.

**Challenges to learning recursion**

While many authors have espoused the difficulty of teaching and learning recursion, only a few empirical research studies have been directly focused on identifying issues contributing to the difficulty of learning recursion or comprehending recursive code. In this section, we consider these studies and explore common misconceptions held by students.

Several studies provide evidence to suggest recursion is inherently difficult. Anderson, Pirolli, and Farrel (1988), in a thorough study of LISP programmers, identified several characteristics of recursion that make it difficult for novice programmers to learn. They observed that students are forced to learn from code samples since physical world examples of recursion are rare and since there are few physical examples, humans do not intuitively think recursively. They suggest the dual meanings of self-referencing procedure calls as both operations to be carried out repeatedly and as something that returns a result, contribute to the difficulty for students. Further, that recursion can take many forms, with recursive calls appearing anywhere within the code of a method, makes learning more difficult since students cannot simply use or recognize a single pattern. Also, verification and debugging requires tracing and the tracing itself involves recursion. Finally, they noted the textbooks available at the time of the study in the late 1980s did not provide adequate guidance on how to turn a problem specification into a recursive function.

Two studies by Lee and Lehrer suggest previous programming instruction may negatively impact students’ ability to learn recursion. In these two studies, Lee and Lehrer (1988) studied the influence of previous programming experience in BASIC on learning recursion. Their first study was conducted with seven graduate students, four of whom had taken at least one course in BASIC and three with no programming experience. All received 1.5 h of instruction each week for eight weeks and completed three programming assignments requiring non-trivial use of recursive procedures: drawing a series of concentric circles; printing all possible pairwise combinations of elements of two lists; and a rock-scissors-paper game. Student programming protocols were collected and in-depth interviews were conducted to elicit specific misconceptions. Lee and Lehrer identified errors common to both the experienced and the inexperienced groups including:
• use of infix notation resulting in negative transfer from graphics to list
processing;
• unnecessary repetition of statements;
• inappropriate use of conditionals;
• inefficient use of logical operations; and
• inappropriate variable initialization.

Errors and misconceptions which were exhibited only by the students
with previous BASIC experience included:

• use of “goto” for recursion;
• inappropriate use of syntax to initialize variables;
• inefficient use of syntax to update variables; and
• inefficient overall structure (unnecessary nesting).

To investigate whether pedagogical approaches might help to mitigate
the negative influences of previous programming experiences, in their sec-
ond study, Lee and Lehrer (1988) altered instructional practices to

• introduce all operations with prefix notation to avoid confusion with
infix;
• introduce recursion using Russian dolls, pseudo-code, and no instances
of tail recursion; and
• use TEST rather than IF constructs to reduce working memory
demands.

Subjects were 24 adult students. Of those, 17 had prior experience with
BASIC. Results showed BASIC students were still more likely to use “goto”
but the incidence was substantially and statistically significantly lower in
this second study than the first study. Use of the TEST construct reduced
unnecessary repeated statements from 100% of the protocols in the first
study to 54% of protocols in this study. The use of unnecessary repeated
statements was higher in the inexperienced group than in the experienced
group, suggesting positive knowledge transfer from prior experience.

While earlier research examined programming artifacts to assess stu-
dents’ difficulties learning recursion, Booth (1993) used an innovative,
qualitative, phenomenograpic approach to explore the challenges students
face when asked to write a recursive solution. Booth asked 14 students to
teach her about recursion by solving a textbook problem to search a list.
She examined student solutions to a searching problem where students
were asked to return the position of a specified element. She categorized
the solutions as:
• solutions using a counter variable (doomed to fail);
• solutions using support functions to count (cheating by using built in language functions to get around a straightforward recursive solution); and
• solutions using direct recursion (headed in the right direction).

Booth notes, “… the most remarkable observation was on the one hand how many ‘wrong’ solutions were being produced and on the other hand what a great deal of ingenuity was going into some of the others.” Booth concluded learners construct knowledge based on preconceived notions or experience, and their knowledge is fragile. She observed learners using problems to challenge their preconceptions, eventually leading to mastery as well as learners using problems to reinforce incorrect conceptions, which in turn led to failure until a new teaching or learning intervention was made.

Another approach used to investigate students’ difficulties with learning recursion is to compare them with students’ understanding of iteration, which some suggest is more straightforward and intuitive. Benander, Benander, and Pu (1996) studied students’ ability to comprehend recursive and iterative algorithms. Participants were 275 computer science majors enrolled in data structures courses, over three years. All subjects were presented with two code segments that manipulated linked lists. The students were not told what the code segments did and the segments were given meaningless names. One code segment searched the list; the other made a copy of the list. Each year, the participants were randomly divided into two groups; one group received the search algorithm implemented recursively and the copy algorithm implemented iteratively; the other group received the search algorithm implemented iteratively and the copy algorithm implemented recursively. Participants were asked to write an English-language description of what each algorithm did. Participants were more likely to comprehend the recursive search algorithm than the iterative version (this result was statistically significant), yet they were more likely to comprehend the iterative version of the copy algorithm than the recursive version (this result was not statistically significant). In years two and three, completion times for the experiment task were tracked. Results showed the rate of comprehension of the recursive code for both tasks was faster than that of the iterative code, with statistical significance. The authors could not explain their results, but suggested additional study. They conjectured the search task might be “more naturally recursive than the copy task; searching continues on the basis of what occurred in the previous step, whereas copying continues independently of what occurred in the previous step” [p. 80].

Another means of demonstrating understanding is the ability to find and fix bugs in incorrect code. Benander, Benander, and Sang (2000) followed up on the Benander et al. (1996) study of program comprehension with a
study that examined debugging of recursive programs. Again they compared recursion to iteration and found students were more successful at finding and fixing bugs in recursive programs than in non-recursive solutions. Students took about the same amount of time to find and fix bugs in both types of algorithms. The authors offered no explanation for their findings.

McCauley, Hanks, Fitzgerald, and Murphy (2015) replicated the study of Benander et al. (1996). Their aim was to determine if results from the first study, which presented students with Pascal code, were consistent in a more modern context using code that was written in Java. In both studies, subjects were students who had previously used recursion to process binary trees, but not to process lists. While students from the initial study found the recursive search algorithm easier to comprehend than the iterative version, in the replication study students could just as easily comprehend the recursive and iterative search methods. McCauley, Hanks, Fitzgerald, and Murphy suggest the difficulty students in the earlier study had with the iterative search method may have been due to the way the method was written in Pascal. As Pascal lacked short-circuit evaluation for Boolean expressions, the iterative code was written in a style that was less easily comprehended than the Java iterative version. Additionally, Java students in the replication study found it significantly more challenging to comprehend a recursive copy method for a linked list than an iterative copy method. This result agrees with what Benander, Benander, and Pu found about the copy routine, although their results were not statistically significant.

In a study of third-year computer science students, Ginat (2005) suggested recursion is difficult because it requires a different and unintuitive way of thinking. He found students struggled to write recursive routines, because they have not been taught to use backward reasoning—working from a goal state back to an initial state. He conjectures most of the problems students have solved before encountering recursion require forward reasoning—working from the initial state to the goal state—and thus recursion requires a new way of thinking. This supports the work of Anderson et al. (1988) who conjecture recursive thinking is not intuitive for most learners.

In a two-part study designed to both guide and better understand students’ understanding of recursion, da Rosa and Chmiel (2012) examined high school students’ solutions to recursive non-programming problems worked on paper. The students had not had any formal instruction on induction or recursion. In part one of the study, students were asked a series of questions about a recursively defined language over three meetings. The questions were intended to lead students to identify the base case (Question: What’s the initial element?) and to see the recursive pattern in the language definition (Question: Given a generic element of the group, is it possible to construct another element from it?). At the second meeting, students were given a recursive rule, and went backward and forward generating strings to complete a table. The students also completed general statements about the
rules and properties of the language. For each of these statements and rules, the authors noted some of the incorrect responses and confusion exhibited by the students. The most notable problems were difficulty constructing the relationship between an element and its predecessor, confusion between a specific element and its immediate predecessor, using particular cases for the generic element, and starting with or constructing an element that was not part of the language.

After observing the problems students experienced with the simple recursive language, da Rosa and Chmiel conducted a second study with natural numbers. Their goal was to see what difficulties students exhibited with the generation of a series of natural numbers using recursive rules. da Rosa and Chmiel asked about even natural numbers \( \geq 4 \) using the rules:

1. 4 is in the set and
2. if \( \alpha \) is in the set then \( \alpha + 2 \) is in the set.

Students were asked to list elements that were and were not in the set, to find predecessors of given numbers and to complete a table. da Rosa and Chmiel also asked about a different set of non-consecutive odd natural numbers with specific questions and a table. They gave a description of students’ responses and difficulties. The authors observed students exhibiting similar types of errors with both the language definition and with natural numbers, although specific errors for natural numbers were not presented.

**Misconceptions about base cases**

Several studies have identified base cases as a source of misconceptions. For example, Close and Dicheva (1997) identified 16 misconceptions in their study of LOGO programmers: three related to the result of executing a STOP command. One misconception was the STOP command was not needed at all. Segal (1995) also found students struggled with base cases, often interpreting a base case as a stopping condition and neither returning values to the caller nor completing suspended routines. Following their earlier study of students’ mental models (Dicheva & Close, 1996), Close and Dicheva (1997) identified this same misconception about base cases: students assumed the base case stopped all calls. The language used in the Dicheva and Close study, LOGO, used a STOP statement as the action for the base case; the use of a STOP statement rather than a return statement might have contributed to the problem by suggesting the cessation of code execution rather than a return to a previous context. Dicheva and Close, however, suggest the misconception resulted from students’ previous experience with only tail-recursive routines, where no unrolling of the recursive calls is necessary to reach a correct result. In a 2003 study,
Götschi, Sanders and Galpin identified the same problem in a group of LISP programmers.

Base cases can also impede the comprehension of recursive code. Haberman and Averbuch (2002) focused on the role of base cases in the comprehension of recursive code. In a two-stage study, they first collected students’ algorithmic solutions to seven problems. A subset of these “solutions” – some correct and simple, some correct and overly complex, and some incorrect – were chosen to make up the questions for the second stage of the study. The second stage involved a six-item questionnaire where each question included a problem description and two recursive solutions, at least one of which was correct. The base case(s) of the solutions differed. Participants, 42 pre-college beginner students and 74 advanced college students, were asked to assess the correctness, readability, and generality of each solution, to identify their preferred solution and to justify their choice. Following completion of the questionnaire, participants were interviewed. The researchers identified four difficulties students had with recursion attributable to problems or misconceptions associated with base cases: (1) ignorance of boundary cases such as empty lists or trees, (2) not handling out-of-range values – they are illegal so students often ignore them, (3) lack of base cases – no terminating condition, and (4) redundancy of base cases.

Students’ understanding of language constructs, or the lack of explicit constructs available in a particular language, may also impede their understanding of recursion. Götschi, Sanders, and Galpin (2003) studied LISP programmers and confirmed the findings of Segal (1995), Dicheva and Close (1996), and Close and Dicheva (1997): students expected computation to stop at the base case. Götschi, Sanders, and Galpin referred to this thought process as the active model. Neither the LISP nor LOGO language includes an explicit return statement, which may contribute to this base case misconception.

Tracing exercises have been used to assess students’ mental models of recursion, but do artifacts of student traces accurately represent their understanding? Scholtz and Sanders (2010) sought to determine if students were able to trace recursive methods proficiently without actually understanding recursion. The study involved 123 first-year university students who had been introduced to recursion. Students were presented questions to test their ability to trace recursive methods that manipulated lists. The study was intended to determine if students understood that a recursive function reaches the base case then completes the actions performed as the stack is popped (the passive flow) before it terminates. Answers were categorized based on whether the students identified the active flow (the actions performed until the base case is encountered), the limiting case, and the passive flow of the given recursive functions (Table 1). Fifteen students who answered all questions correctly volunteered for the next phase of the study.
Scholtz and Sanders interviewed and observed the 15 students as they solved two additional questions. The first question required a trace of a method to draw a figure using turtle graphic operations. The second question required students to write a recursive method to draw a right triangle of asterisks. Once again, responses were scored based on whether they exhibited an understanding of active flow, limiting case, and passive flow. Of the 15 students who had already demonstrated an ability to trace recursive methods, only two correctly answered both additional questions. Seven students demonstrated a poor understanding of the passive flow (actions performed as the stack is popped). Scholtz and Sanders (2010) concluded most students don’t understand the passive flow of recursion. Students did not understand the recursive call must finish before the additional statements are executed.

Another approach to investigating student difficulties with recursion is to examine the plans they use when writing recursive code. In a study of students’ recursive solutions to a binary tree problem, Murphy, Fitzgerald, Grissom, and McCauley (2015) found a majority of students tested for base cases early in order to avoid making recursive calls. Use of this “arms-length recursion” resulted in the development of overly complex and error-prone code. Eighteen participants developed a single method to count the number of binary tree nodes with exactly one child. These 18 solutions included a total of 55 errors that fell into 15 different error categories; ten solutions included errors related to base cases (eight had missing base cases and two had malformed base cases).

Regardless of the learning context or study approach, the main take away from all of this work is many novice programmers do in fact have difficulties writing, tracing, and understanding recursion.

### Mental models

The difficulties students have with recursion suggest it is worthwhile to investigate their mental representations as they comprehend, trace, and write recursive code. Much of the educational research on recursion has been focused on identification of students’ mental models. Bhuiyan, Greer and McCalla (1994) define a mental model as a knowledge structure integrating descriptive and functional knowledge about a concept, along with a control mechanism that determines how this knowledge is used in problem-solving. Mental models are formed as one learns a concept. For recursion, a mental model is the learner’s beliefs/understandings/views of how recursion works.
Viable models allow someone to correctly understand a process. Non-viable models contain misconceptions and do not support an accurate understanding. One viable mental model is to suppose a recursive method makes copies of itself with separate unique instances of local variables. Kahney (1982) named this the copies model. In addition, he identified several incomplete models that predict correct results in some situations but not in others. These models are described in the next section. Other researchers also identified mental models for understanding recursion (Bhuiyan, Greer, & McCalla, 1991; Götschi et al., 2003; Kurland & Pea, 1985; Lewis, 2014). The models may have different names but share characteristics.

One of the more common misconceptions is the notion that a recursive method stops execution as soon as the base case is encountered (Close & Dicheva, 1997; Haberman & Averbuch, 2002; Scholtz & Sanders, 2010; Segal, 1995). This misunderstanding does not cause a problem when tracing tail recursive solutions since there is no additional execution. However, with other forms of recursion where the recursive call is not the final instruction in the method, additional execution is necessary as the call stack is popped. Learners with this incomplete mental model may not correctly predict the complete execution.

Seminal work: Kahney

Some of the earliest work on mental models in recursion is attributed to Kahney (1982, 1983) and Kahney and Eisenstadt (1982). Kahney’s goals were to discover a model of novice programming behavior and to test the hypothesis that novices and experts differ in the models they hold. While Kahney’s work was not limited to the study of recursion, we focus here on his findings related to recursion. In a study of 30 students and 9 experts, several mental models were identified. In the copies model, recursive procedures are seen to generate new instantiations of themselves, passing control, and possibly data forward to successive instantiations and back from terminated ones. Kahney hypothesized this was the model held by experts. In the incomplete looping model, a recursive procedure is viewed as a single instantiation having an entry point and an iterative action part until the base case is encountered. This model does not consider multiple instantiations of a procedure and does not account for control passing back from a sequence of recursive calls. Variables are treated as global values rather than independent variables within each successive function instantiation. This leads to confusion. The model is viable for tail recursion with only passive (forward) flow because, after encountering the base case, no further operations are performed and values are not modified. However, the looping model does not correctly predict the behavior of head recursion where the recursive call occurs at the beginning of the function followed by additional actions.
With the looping model, execution stops and control is not returned after the base case is encountered, as it should be.

Three non-viable models were also identified: the null model, where students had virtually no understanding of recursion; the odd model, where students were found to hold some model with aspects of copies or looping models but the model was not accurate enough to be able to predict the behavior of recursive code; and the syntactic or magic model, where students recognized a recursive procedure as having a particular syntactic structure. Kahney presented subjects with a written description of a logical problem that involved inferring new information from information given and three code segments that were possible solutions to the problem. The first solution was incorrect, both the second and third code segments were correct. The second code segment used tail recursion. Subjects were asked to study each code segment and indicate if the code segment did or did not solve the problem and to use their own words to justify each answer. Kahney hypothesized participants who held the correct copies model would recognize both the second and third solutions as correct, while participants holding a looping model would recognize only the second, tail-recursive segment as correct. Since the second segment employed tail recursion, one does not need to “unroll” the recursion to obtain a correct result; thus, a looping model would suffice. Written justifications for answers were examined to verify the model possessed. Eight of nine “experts” exhibited the copies model. (Kahney’s “experts” included tutors in an Artificial Intelligence summer school program and a research assistant with more than a year of programming experience in LISP, Pascal, and SOLO, a LOGO-like language, used in the study). Results gleaned from participant questionnaires indicated just over half of the participants held the looping model, 10% held the copies model and the rest of the students either had no understanding (believing no code segment was a solution) or some other non-obvious understanding. However, these findings in student understanding did not hold up to scrutiny when compared to participants’ written justifications of their answers: only one student was found to clearly hold the copies model and only four were found to hold the looping model. Unfortunately, Kahney did not interview the students and his results are based only on his interpretations of participants’ written answers.

Table 2 summarizes Kahney’s mental models. Viable models are those that lead to correct understanding. Flawed models work in some circumstances, for example with tail recursion. Non-viable models are deeply flawed and do not lead to correct understanding.

Building on Kahney’s models
Götschi et al. (2003) studied mental models by examining three years of students’ traces of recursive algorithms presented in pseudo-code. They
observed various characteristics or models of student understanding including Kahney’s correct *copies* and incomplete *looping* models, the *active flow* (where students understand recursive computation until a base case is reached, but do not understand the computation “unrolls” itself), a non-viable *step* model (where students appear to have no model of recursion at all and only comprehend a small portion of what is going on such as evaluating the if and else clauses only once), a *return value* model (where it is assumed a return value is evaluated at each call and the collection of return values are combined to provide a solution), Kahney’s *magic* or *syntactic* model (where the surface structure of a recursive routine is recognized, but there is no real understanding of how the computation progresses), the *algebraic* model (where the implementation of a recurrence relation is viewed as an implementation of an algebraic problem), and Kahney’s *odd* model. A weakness of this work, as that of Kahney, is that researchers inferred findings based only on students’ written work; no interviews or follow-up studies to confirm findings were carried out (Table 3).

Mirolo (2010) extended the work of Götschi et al. (2003) by reconceptualizing and organizing mental models into “a few broader classes” [p. 162].

As part of a broader study of recursion comprehension, Haberman (2004) partially replicated Kahney’s study and found students who had studied logic programming previously were more likely to hold the *copies* mental model than those who had only studied recursion in a procedural paradigm.

Booth (1993) used a phenomenographic research approach to look at “what does it mean and what does it take to learn about recursion meaningfully and use it successfully?” After asking 14 students to teach her about recursion by solving a textbook problem to search a list, she identified three conceptions of recursion: (1) recursion as a programming construct (fill in

<table>
<thead>
<tr>
<th>Model</th>
<th>Usefulness</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copies</td>
<td>Viable,</td>
<td>Recursive procedures are seen to generate new instantiations of themselves, passing control, and possibly data forward to successive instantiations and back from terminated ones</td>
</tr>
<tr>
<td></td>
<td>expert view</td>
<td></td>
</tr>
<tr>
<td>Looping</td>
<td>Flawed</td>
<td>Recursive procedure is viewed as a single object having an entry point and an iterative action part. Function variables are treated as global. This erroneous model does not consider multiple instantiations of a procedure and does not account for the undoing of a sequence of recursive calls</td>
</tr>
<tr>
<td>Null</td>
<td>Not viable</td>
<td>Model indicates virtually no understanding of recursion</td>
</tr>
<tr>
<td>Odd</td>
<td>Not viable</td>
<td>Model has aspects of copies or looping models but not accurately enough to be able to predict the behavior of recursive code</td>
</tr>
<tr>
<td>Syntactic or magic</td>
<td>Not viable</td>
<td>Students view recursion as consisting of a specific surface structure, a template including base and recursive cases, with little realization of recursion as a problem-solving technique</td>
</tr>
</tbody>
</table>
the template), (2) recursion as repetition (looping model), and (3) recursion as self-reference (function calls itself; there is a base case). Booth posits these understandings are organized in a hierarchy – as students’ understandings progress they move up the hierarchy.

Kurland and Pea (1985) studied novice programmers’ mental models of recursion by interviewing seven 11- and 12-year-old children who had one-year’s instruction in LOGO. The children were given four programs to trace manually – two iterative and two recursive. None of the children were able to correctly trace the recursive problem with an embedded recursive call despite previous success with writing recursive solutions. The children appeared to think of recursion as looping as reported by Kahney (1982). An embedded recursive call was seen as sending control “back up” to the top of the method. The return from the recursive call was not perceived as causing an unwinding of the nested procedure calls but rather as stopping execution. Kurland and Pea contend that keywords such as STOP and EXIT are misleading for students. In LOGO, EXIT, and STOP do not exit or stop, but rather cause the recursion to start unrolling itself. They suggest programming constructs such as these, which have one natural language meaning and another programming language meaning, lead novice programmers to

<table>
<thead>
<tr>
<th>Model</th>
<th>Usefulness</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copies</td>
<td>Viable, expert view</td>
<td>Recursive procedures are seen to generate new instantiations of themselves, passing control, and possibly data forward to successive instantiations and back from terminated ones (same as Kahney)</td>
</tr>
<tr>
<td>Looping</td>
<td>Flawed</td>
<td>Recursive procedure is viewed as a single object having an entry point and an iterative action part. Function variables are treated as global. This erroneous model does not consider multiple instantiations of a procedure and does not account for the undoing of a sequence of recursive calls (same as Kahney)</td>
</tr>
<tr>
<td>Active flow</td>
<td>Flawed</td>
<td>Students understand recursive computation until a base case is reached, but do not understand the computation “unrolls” itself</td>
</tr>
<tr>
<td>Step</td>
<td>Not viable</td>
<td>Students have no model of recursion and only comprehend a small portion of what is going on such as evaluating the if and else clauses only once</td>
</tr>
<tr>
<td>Return value</td>
<td>Flawed</td>
<td>It is assumed a return value is evaluated at each call and the collection of return values are combined to provide a solution</td>
</tr>
<tr>
<td>Syntactic or magic</td>
<td>Not viable</td>
<td>Students view recursion as consisting of a specific surface structure, a template including base and recursive cases, with little realization of recursion as a problem-solving technique (same as Kahney)</td>
</tr>
<tr>
<td>Algebraic model</td>
<td>Flawed</td>
<td>Implementation of a recurrence relation is viewed as an implementation of an algebraic problem</td>
</tr>
<tr>
<td>Odd</td>
<td>Not viable</td>
<td>Model has aspects of copies or looping models but not accurately enough to be able to predict the behavior of recursive code (same as Kahney)</td>
</tr>
</tbody>
</table>
wrong assumptions about the execution of the code. Kurland and Pea also noted the idealist assumption attributed to Papert (1980) among others, that children can learn recursion through guided exploration was not supported by their observations. They assert, “most [of our students] avoid all but simple iterative programs, which do not require the deep understanding of control structure prerequisite for an understanding of recursion.” They recommended supplementing self-guided exploration by instruction.

**Models for comprehension, evaluation, construction**

In a body of work, Bhuiyan et al. (1991, 1994) and Bhuiyan, Greer, and McCalla (1992) distinguished between generative models (i.e. construction models) required to write recursive routines and trace methods (i.e. evaluation methods) required to trace flow of execution and predict results. By 1994, Bhuiyan, Greer, and McCalla had switched from using the term mental model to using mental method to emphasize the way the models are used to develop and explain recursive programming. Bhuiyan et al. (1991) performed an exploratory study monitoring verbal protocols of six non-major computer science students on a weekly basis over five weeks while they solved recursive problems using LISP. They found evidence of four generative mental methods (the loop, syntactic, analytic, and analysis/synthesis methods) and three trace methods (the stack, tree, and staircase methods) used by various students at varying times (see Table 4). Like Kahney (1982); they, too, observed the flawed loop model, where learners view recursion as a loop and don’t consider multiple instantiations and the need to “unroll” calls. Bhuiyan, Greer, and McCalla observed students initially using a loop model moved on to more suitable models after two weeks. Students holding the syntactic model for generating routines viewed recursion as consisting of a specific surface structure, a template including base and recursive cases, with little realization of recursion as a problem-solving technique. With the more effective analytic model for generating routines, recursion was viewed as a problem-solving technique that involved analyzing the input–output requirements of a problem, which include (1) determining input cases and output strategies, (2) translating input cases into conditions and output strategies into actions, and finally (3) translating conditions and actions into programming language code. The analysis/synthesis method considers problem solving by breaking a big problem into smaller subproblems. This method, which is considered the most powerful and possessed by experts, involves both functional and structural aspects, including (1) breaking a problem into smaller and smaller subproblems, until primitive subproblems which are immediately solvable are reached, (2) building up solutions to the reduced subproblems from the immediately solvable ones and determining the relationship among the reduced solutions and the overall solution, and (3) finally translating results of (2) into programming code. The trace methods included the stack...
method (Kahney’s copies model) and two variations on the stack method named for the way the computation is drawn on paper during a trace: the staircase method and the tree method. A strength of this body of work is its basis on observations of and interviews with students.

Dicheva and Close (1996) studied mental models of 10- to 14-year old LOGO programmers with different levels of programming experience. Participants completed paper worksheets on which they explained what a given recursive code segment would compute and wrote recursive solutions on paper. Some think-alouds followed. Sixteen mental models were identified through analysis of the collected data: eight interpretation (comprehension and evaluation) models, one of which was correct and eight construction models, including two correct models. The correct comprehension model detected was the copies model (see Kahney above); the seven incorrect models fell into the categories of loop models (with four loop-related models identified), do-it-once-more models (with two specific models identified) and the simple model that involved executing the code only once. The loop model identified is similar to Kahney’s looping model. A recursive procedure is viewed as a loop; the recursive calls modify values of parameters and control returns to the beginning of the loop. The do-it-once-more model interprets a recursive call as requiring execution of the code that appears before the recursive call, one more time, once the recursive call is

<table>
<thead>
<tr>
<th>Methods</th>
<th>Type</th>
<th>Usefulness</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Generative</td>
<td>Flawed</td>
<td>Learners view recursion as a loop and don’t consider multiple instantiations and the need to “unroll” calls (same as Kahney)</td>
</tr>
<tr>
<td>Syntactic</td>
<td>Generative</td>
<td>Not viable</td>
<td>Students view recursion as consisting of a specific surface structure, a template including base and recursive cases, with little realization of recursion as a problem-solving technique (same as Kahney)</td>
</tr>
<tr>
<td>Analytic</td>
<td>Generative</td>
<td>Viable</td>
<td>Recursion is viewed as a problem-solving technique that involves analyzing the input–output requirements of a problem</td>
</tr>
<tr>
<td>Analysis/synthesis</td>
<td>Generative</td>
<td>Viable, expert</td>
<td>Problem-solving by breaking a problem into smaller and smaller subproblems, until primitive subproblems which are immediately solvable are reached. Solutions to reduced subproblems are built from the immediately solvable ones; the relationships among the reduced solutions and the overall solution are determined and results are translated into programming code</td>
</tr>
<tr>
<td>Stack</td>
<td>Trace</td>
<td>Viable</td>
<td>Same as Kahney’s copies model</td>
</tr>
<tr>
<td>Stair case</td>
<td>Trace</td>
<td>Viable</td>
<td>Variation on the stack method named for the way the computation is drawn on paper during a trace</td>
</tr>
<tr>
<td>Tree</td>
<td>Trace</td>
<td>Viable</td>
<td>Variation on the stack method named for the way the computation is drawn on paper during a trace</td>
</tr>
</tbody>
</table>
reached. The details of the subcategories are beyond the scope of this review, but may be of interest as the authors were able to align the subcategories with specific misconceptions they detected.

For construction tasks, Dicheva and Close considered the output of two problems, both of which involved drawing a collection of rectangles, adjacent or embedded, that formed a complete figure with bilateral symmetry where the left half was a mirror image of the right half. They identified two correct construction models: the embedded recursion model which they contend is a model held by experts and the tail recursion model which involves two tail-recursive procedures, one drawing the left half and the other drawing the right half of the figure. They also identified six additional incorrect models. The researchers studied the incorrect models in more detail, identified specific misconceptions and suggested ways of teaching to circumvent developing these misconceptions (Table 5).

Lewis (2014) conducted one-on-one interviews with 30 undergraduate students as they completed two multiple-choice problems involving recursion. In the first problem, students were asked to study a recursive function and determine which of five given values was computed by the function for given parameters. In the second problem, the students were given an incomplete function and assigned the task of multiplying its parameters. Students were asked to complete the function definition by choosing, from a selection of five pairs of statements, which pair of statements would complete the function in such a way as to accomplish the multiplication task. The interviewer observed students as they answered each question, kept their written scribbles, and asked questions about what they were thinking as they proceeded. Lewis detected four substitution techniques students used to trace

Table 5. Mental models (Dicheva & Close, 1996).

<table>
<thead>
<tr>
<th>Task</th>
<th>Model</th>
<th>Usefulness</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpretation models</td>
<td>Copies</td>
<td>Viable</td>
<td>Same as Kahney’s copies model</td>
</tr>
<tr>
<td>(comprehension and evaluation)</td>
<td>Loop models</td>
<td></td>
<td>In general, similar to Kahney’s looping model, but with specific submodels differing based on what/how code around the recursive call is executed</td>
</tr>
<tr>
<td></td>
<td>· Truncated procedure</td>
<td>Flawed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>· Proper</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· Embedded</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· Two loops</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Do-it-once-more</td>
<td>Not viable</td>
<td></td>
<td>Subset of looping</td>
</tr>
<tr>
<td></td>
<td>· Proper</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>· Embedded</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>Not viable</td>
<td></td>
<td>Executes the code only once</td>
</tr>
<tr>
<td>Construction models</td>
<td>Embedded</td>
<td>Viable, expert</td>
<td>Recursive call among or followed-by non-recursive statements</td>
</tr>
<tr>
<td></td>
<td>recursion</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Tail recursion</td>
<td>Viable</td>
<td>Two tail-recursive procedures</td>
</tr>
<tr>
<td></td>
<td>Other (6)</td>
<td>Flawed</td>
<td></td>
</tr>
</tbody>
</table>
the computation of each function. The techniques included simulating execution (Kahney’s copies model), accumulating pending calculations where students did all calculations at once rather than as each instantiation terminated, dynamic programming where students worked from base cases back up to the general case rather than the other way around, and predicting the result where students assumed the result of a recursive call and computed the result for the current case based on that assumption (Table 6).

While there are some differences in the numbers and characteristics of the mental models revealed by this body of research, regardless of the learning context or the method of investigation, all of these studies provide further evidence to suggest students struggle to understand recursion. The viable “copies” model and flawed “looping” model appear throughout, as do models that suggest a complete lack of understanding of recursion. Several models – such as the “algebraic” model (Götschi et al., 2003), the “staircase” and “tree” models (Bhuiyan et al., 1994), and the “dynamic programming” model (Lewis, 2014) – were specific to those investigations and were likely influenced by the type of problem students were asked to solve, a technique they had learned to use when tracing recursion, or the alternative problem solving approaches they had been taught in their classes.

### How to introduce recursion

This section describes empirical studies about best practices for teaching recursion. A common theme of these studies is to determine the most effective way to provide initial exposure to the topic. Greer (1989) compared three contexts within traditional instruction including an emphasis on how recursion works, why it works and an applied approach that provides a template for writing recursive methods. He found no differences between the approaches. In contrast, Ginat and Shifroni (1999) found students did better by first learning an applied approach. Kessler and Anderson (1988) identified positive learning transfer when students first experienced iteration

<table>
<thead>
<tr>
<th>Substitution technique</th>
<th>Usefulness</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution simulation</td>
<td>Viable</td>
<td>Same as Kahney’s copies model</td>
</tr>
<tr>
<td>Accumulating pending</td>
<td>Flawed</td>
<td>Students did all calculations at once rather than as each instantiation terminated</td>
</tr>
<tr>
<td>calculations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dynamic programming</td>
<td>Flawed</td>
<td>Students worked from base cases back up to the general case rather than the other way around</td>
</tr>
<tr>
<td>Predicting the result</td>
<td>Flawed</td>
<td>Students assumed the result of a recursive call and computed the result for the current case based on that assumption</td>
</tr>
</tbody>
</table>
followed by recursion. The reverse was not detected. In fact, it appeared students were confused by their understanding of recursion when trying to learn iteration. Wiedenbeck (1989) also identified an advantage when introducing iteration before recursion. Segal (1995) determined students acquired a better mental model if first introduced to mathematical equations such as recurrence relations. Sanders, Galpin, and Götschi (2006) suggested using a variety of examples requiring passive and active flow as an effective strategy. Lee, Shan, Beth, and Lin (2014) used a commercial game to introduce recursion in a compelling context before traditional instruction.

**Start with a variety of analogies**

Greer (1989) compared three instructional analogies to introducing recursion: architecture-oriented, theory-oriented, and task-oriented. The architecture analogy describes how recursion is implemented using a system stack and activation records, and provides detailed program traces. This analogy is based on “how recursion works.” The theory-oriented approach focuses on results produced by recursive procedures and why the procedures produce the results correctly. The theory analogy is based on mathematical induction as a logical rule of inference. This mathematical approach minimizes the need to implement routines, but depends on an understanding of induction. The focus is on “why recursion works.” The task analogy uses production rules and programming strategies rather than a notional machine model or mathematical explanation. Recursion is described as a set of rules and facts and the task of organizing these rules and facts into programs is left to the learner. Students practice writing recursive methods with base cases and recursive method calls (Table 7).

Greer (1989) studied 84 college students in a second course for computer science majors who were randomly assigned to three groups. The study lasted two weeks including three days of lecture and a fourth day for an exam. Each group watched three video-taped lectures during the regularly

<table>
<thead>
<tr>
<th>Approach</th>
<th>Philosophy</th>
<th>Process</th>
<th>Student activities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture-oriented</td>
<td>How recursion works</td>
<td>How recursion is implemented using a system stack and activation records</td>
<td>Program traces</td>
</tr>
<tr>
<td>Theory-oriented</td>
<td>Why recursion works</td>
<td>Results produced by recursive procedures and why the procedures produce the results correctly</td>
<td>Mathematical induction as a logical rule of inference</td>
</tr>
<tr>
<td>Task-oriented</td>
<td>Recursion as a set of rules and facts</td>
<td>Uses production rules and programming strategies</td>
<td>Students practice writing recursive methods with base cases and recursive method calls</td>
</tr>
</tbody>
</table>
scheduled class time. Each video lecture was created by the same instructor for this experiment and introduced recursion using one of the architecture-, theory-, or task-oriented approaches. Students completed a quiz on the fourth day to assess their ability to read, understand, and write recursive methods. Additional questions were included in an exam later in the term as well as the course final exam. A thorough statistical analysis did not reveal a significant difference in student performance across the three approaches.

The authors also hypothesized students with greater computing ability would perform better than other students regardless of the instructional approach. This conjecture was supported by the data. For the analysis, participants were divided into two groups based on their performance on the first exam of the term which did not include recursion. Students above the median were considered to have more ability than the lower half. Students with more initial ability significantly outperformed other students on the recursion quiz as well as on questions on subsequent exams. A post hoc analysis used student mathematics SAT scores to compare performance. Results indicated that students with higher mathematical ability were better suited to learning recursion using the theory-oriented approach.

**Start with iteration before recursion**

Anzai and Uesato (1982) investigated the acquisition of recursion by 88 middle school children (14 years old). The children were given worksheets describing one of two methods to calculate factorial. One method, labeled “white”, demonstrated the calculation as a sequence of multiplications from 1 to N with simple examples. For example, “WHITE (3) is calculated by multiplying $1 \times 2 \times 3$.” Students were then required to use the method to calculate a new value, such as WHITE (4). The recursive strategy, labeled “black”, demonstrated the calculation using examples such as “BLACK (3) is calculated by multiplying 3 times BLACK (2).” Half of the students worked through iterative worksheets before working recursive worksheets. The order was reversed for the remaining students. Results suggested students did better with the recursive calculations after first being introduced to the sequential strategy. No statistical analysis was provided.

Wiedenbeck (1988, 1989) performed experiments similar to those of Anzai and Uesato (1982) to study the acquisition of iteration and recursion concepts when novices learn from example rather than by instruction. The first study included 116 college students in an introductory programming course. The study was conducted early during the course before students were introduced to iteration or recursion. During the first phase, students were given an example for computing the factorial mathematical function, not a computer program. Half were shown an iterative strategy and half were shown a recursive strategy. After reviewing the example, they were asked to compute the factorial of three additional values. A second phase
was similar to the first except the calculation performed was for the summation mathematical function instead of factorial. Once again, students were first given an example and then asked to compute three additional values. Subjects were divided into four groups. Half saw an iterative example in the first phase and half were given a recursive example. Both groups were further divided during the second phase with half given an iterative example and half given a recursive example. This resulted in four groups: iterative–iterative, iterative–recursive, recursive–iterative, and recursive–recursive. All groups had more correct answers during the second phase than the first. Learning occurred regardless of approach and this behavior was fairly uniform across groups. Unlike the findings of earlier researchers, the order of learning iteration and recursion had no observed effect. Students performed equally well whether introduced to recursion or iteration first. The investigator noticed two strategies for solving the problems — direct substitution of numbers and extrapolation of a rule. Students who were able to deduce a rule were more likely to correctly solve more difficult problems.

A second, similar experiment is reported in Wiedenbeck (1988, 1989). In the second study, students studied iterative and recursive computer programs rather than mathematical functions. Forty-four new subjects were again divided into four groups: iterative–iterative, iterative–recursive, recursive–iterative, and recursive–recursive. They were given study materials and a sample Pascal program to calculate factorial. They were then asked to trace the program with several values and to describe its purpose. Unlike the first experiment, a marginal advantage was observed for those who learned iteration before recursion.

Kessler and Anderson (1988) investigated the relationship between writing iterative code and writing recursive code. The 32 participants were undergraduate students with little or no programming experience. There was a training phase and a transfer phase. In each phase, a student completed a self-paced tutorial starting with a description of the construct (iteration or recursion), a template for writing a function and a sample solution. They were then asked to write four functions using the construct. Subjects were randomly assigned to four training-transfer groups: iterative–iterative, iterative–recursive, recursive–recursive, and recursive–iterative. The goal was to determine if there was transfer from learning one programming construct to another. The amount of time to write the four functions was recorded. Three of the four groups completed the second set of tasks faster than the first set. The exception was the recursive–iterative group. These students wrote the iterative solutions slower after first learning about recursion. Kessler and Anderson concluded their subjects were able to write the recursive functions correctly despite having an inadequate mental model. The incorrect mental models became a distraction when learning iteration. The difference in ability to write correct iterative or recursive solutions when iteration was learned
first was significant \((p < .05)\). Kessler and Anderson suggest it is better to introduce iteration before recursion for novice programmers.

Mirolo (2012) also considered iteration and recursion. His research questions were:

1. Do students who learn functional programming and, hence, recursion, before iteration exhibit a stronger ability to deal with recursion?
2. Do students who learn imperative programming and, hence, iteration, before recursion master iteration better than those who study functional programming first?

Mirolo performed a multi-year study of students in two different computing majors: computer science and web technologies. The computer science students first learned programming in a year long course beginning with a recursion-heavy functional programming language; later in the year, the course switched to an imperative/object-oriented programming language with iteration. The web technologies students only studied imperative programming where iteration was studied in depth and recursion was introduced, but not emphasized.

Mirolo used two tests, a recursion test and an iteration test, to assess students’ ability to apply, analyze, and synthesize (in Bloom’s terminology (Bloom, Engelhart, Furst, Hill, & Krathwohl, 1956)) recursion and iteration. The tests also assessed student ability to respond to recursion and iteration questions multistructurally or relationally (in SOLO terminology (Biggs & Collis, 1982)). The computer science group took both the recursion and the iteration tests. Results showed computer science students did not favor one approach (recursion or iteration) over the other; their ability to master recursion and iteration was comparable. The web technologies group took only the iteration test. In comparison to the web technologies group, the computer science group was found to perform significantly better on the iteration test. The authors caution the reader not to assume that these results imply teaching recursion before iteration guarantees a better grasp of iterative programming skills. They suggest the web technologies group may have been less motivated and less interested in programming than the computer science group. Finally, Mirolo found no evidence students perform better with iteration than recursion.

**Start with mathematical functions**

Segal (1995) completed studies with university students nearing the end of a first course in Miranda, a functional programming language. The subjects had already completed a course in Pascal. Students were asked to evaluate three recursive formulas in different contexts:
• Context 1 included natural numbers within recurrence relations
• Context 2 involved lists of numbers using Miranda notation
• Context 3 manipulated strings of asterisks in a programming context

All functions shared the same structure and included embedded recursion where the function performed work both before and after the recursive call. This embedded recursion structure is considered more difficult to understand than more common examples such as factorial and summation which use tail recursion.

In Segel’s first study, eight participants wrote code to manipulate strings (Context 3) and were encouraged to think out loud. The recorded verbal protocols were used as qualitative data to reveal problem-solving processes. A second study was designed to identify any differences in the order of writing functions within each context. Seventeen of the thirty-three participants completed Context 1, Context 2, and then Context 3 in order. The remainder completed Context 3, Context 2, and then Context 1. Students wrote their answers on paper and were encouraged to show all work. The completed worksheets provided qualitative and quantitative data. Participants who demonstrated misconceptions were interviewed.

Quantitative data was not analyzed due to a variety of confounding factors. However, verbal protocols and student interviews provided useful qualitative data. Segal (1995) observed a misconception also reported by Kahney (1995); students commonly considered the base case as a stopping condition and did not complete any suspended processes or return results to the calling methods. This misconception was observed for the list processing (Context 2) and the asterisks programming (Context 3) tasks but was never observed when students solved recurrence relations (Context 1). In addition, participants wrote better answers when encountering recurrence relations before the other two contexts. Segal suggested this is because numbers within recurrence formulas are a more familiar context than computer code to manipulate lists or strings.

**Start with an applied approach**

Ginat and Shifroni (1999) compared two strategies for teaching recursion. Participants included 42 college students in a introductory data structures CS2 course. The participants were randomly assigned to two groups. Groups received different instruction during two labs per week for five weeks. The control group was given instruction with an emphasis on the computing model, system stack, and activation records, similar to the architecture “how recursion works” approach described by Greer (1989). The experimental group was instructed to use a specific three-step divide-and-conquer strategy. The three steps included identifying a base case, decomposing the problem
into a simpler instance, and incorporating results from the simpler instances into a final solution. A pre-test/post-test design was used to assess student abilities to write recursive functions.

Students wrote a recursive method as the post-test after five weeks of instruction. Eighteen percent of the control group answered the question correctly compared to 40% of the experimental group. In addition, incorrect answers provided by the experimental group suggested a better understanding of recursion compared to the control group. Unfortunately, no statistical analysis was reported so it is not known if the differences were significant.

Previous research indicated instruction with an emphasis on the computing model helped students trace recursive methods, but Ginat and Shifroni concluded their applied problem solving approach was better at helping students write recursive methods.

**Start with concrete conceptual models**

Wu (1993) explained it is a teacher’s responsibility to present instructor-defined conceptual models to help students develop their own viable mental models. He designed a study to compare the effectiveness of concrete and abstract conceptual models. Participants included 237 volunteer college students in a CS 1 course. The concrete treatment introduced recursion through a technique for counting using nested Russian dolls. This was followed by instruction on how to design recursive algorithms and verify them through the use of a block diagram tracing method, where overlapping rectangles showing the code and values for variables for each invocation are drawn until a base case is reached. The abstract treatment introduced recursion as a recurrence relation for factorial. Emphasis was placed on identifying the base and recursive cases, and using the mathematical equation to trace and verify the algorithm for a specific value. The remainder of the abstract treatment involved instruction on how to design recursive algorithms and verify them using induction (Table 8).

Wu’s study involved pre-test, post-test, and retention tests a few weeks after the treatment. Wu divided students into abstract and concrete learners based on results from a Kolb Learning Style Inventory (Kolb & Kolb, 2005). He found the impact of treatment was independent of learning-style:

<table>
<thead>
<tr>
<th>Concrete conceptual model</th>
<th>Abstract conceptual model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counting using nested Russian dolls</td>
<td>Recurrence relations using factorial as an example</td>
</tr>
<tr>
<td>Design recursive algorithms using a block diagram tracing technique</td>
<td>Identify base and recursive cases</td>
</tr>
<tr>
<td>Verify recursive algorithms using a block diagram tracing technique</td>
<td>Design and verify recursive algorithms using induction</td>
</tr>
</tbody>
</table>
concrete learners did not benefit more from the concrete treatment and abstract learners did not benefit more from the abstract treatment. On post-test, he found students in the concrete model treatment group significantly outperformed students in the abstract model treatment group. This effect was only weakly supported with retention tests administered two and six weeks after the treatment.

Start with a compelling context
Lee et al. (2014) studied the use of a commercial iPad video game, Cargo-Bot, to help students learn recursion. In Cargo-Bot, the goal is to teach robotic arms to move crates. Players control robots by creating short pieces of visual code. Recursive procedure calls are the only construct for repetition. This makes the game a natural context in which to use recursion. The investigators built on previous work that used the game to introduce recursion to AP high school students (Tessler, Beth, & Lin, 2013). Results of the original study indicated students who first played the game for an hour followed by an hour of traditional instruction were better able to write recursive methods in a post-test compared to students who received instruction followed by game play. However, there was no difference observed in the ability to trace recursive methods.

For the subsequent study, approximately two hundred college students in CS 2 used Cargo-Bot in a more structured way. An experimental group played the game for 250-min sessions followed by 250-min lectures that included examples using the Cargo-Bot language. Students were assigned nine Cargo-Bot exercises that map to course learning objectives and were allowed to continue play outside of class. A control group received 450-min lectures and was not introduced to Cargo-Bot. The control group lectures included examples in Java and were identical to the content used for several years.

Assessment was done with pre-test and post-test quizzes. One question required students to trace the result of a recursive method and a second question to write a recursive method. Similar to the original study, students who played the game were statistically better at writing recursive methods than the control group ($p < .05$). No advantage was detected for tracing recursive methods. Further analysis revealed the performance gain was only achieved by students who completed at least eight of the nine exercises during game play. Students who completed fewer than eight exercises had no observed advantage compared to the control group. The investigators concluded the improved performance was due to the additional practice of writing recursive methods compared to the control group.
**Use a variety of examples**

Pirolli and Anderson (1985) conducted a careful analysis of how three novice programmers learned how to write recursive functions. The participants were given brief instruction and then asked to think aloud as they wrote recursive solutions to new problems. The researchers analyzed several hours of verbal protocols for each participant. The analysis provided insight as to how learners acquire information using sample programs as a guide. Students relied heavily on code examples in the early stages of learning to write recursive functions. They modified the examples as if they were templates. The experience of modifying the examples led to a more abstract understanding of recursion which eventually was used to solve new problems. The authors recommended the use of a variety of code examples early in the introduction to recursion.

Wiedenbeck (1989) compared subjects trained by example with students presented with abstract concepts and no examples. Forty students in an introductory programming course were given worked examples of mathematical functions. For example, showing recursive calculations for factorial (3) and factorial (4) prepared students to correctly calculate factorial (5). The students were not given any additional instruction. Two days later, they were given abstract definitions of three mathematical functions with no examples and asked to solve each with a specific value such as summation (4). The functions included factorial, summation, and Fibonacci. The factorial function, introduced with examples two days prior but not labeled as factorial, was solved correctly significantly more often than the summation and Fibonacci functions stated abstractly with no examples (p < 0.05). The experiment was then repeated with a new cohort of students not trained with examples but rather given the same abstract definitions of summation and Fibonacci. The “no examples” group was compared to the group trained with examples. Students given examples correctly solved more problems than those who did not receive examples during training (p < 0.05). Wiedenbach concluded that providing worked out examples helps learners transfer knowledge to similar problem domains.

Sanders et al. (2006) extended their earlier work (Götschi et al., 2003) by making changes to the way they introduced recursion to first-year computer science students. Initial exposure emphasized recursive methods that require active flow as well as passive flow, in which instructions are performed after the recursive call. Examples included list manipulation, binary search tree algorithms, and mathematical functions. This was in contrast to their previous approach of introducing basis methods such as summation and factorial that only require an understanding of active flow, instructions are performed before the recursive call.

Student performance was assessed with two final exam questions to trace recursive methods. Data were collected in 2003–2005 with an average of
140 students each year. A qualitative analysis of student mental models revealed more students had an accurate understanding of recursion (e.g. a copies model) compared to previous years. The authors attributed improvement to changes in the teaching method but acknowledged additional exposure to similar recursive methods during the course may have also influenced the observed improvement.

**Implications for teaching**

Results sometimes conflict but several teaching strategies have emerged as particularly effective. A commonly reported strategy is to provide students with a variety of analogies, examples, and contexts (Greer, 1989). For example, mathematical functions are a unique context that can introduce recursion without the overhead of coding structures (Segal, 1995). An alternative is to introduce recursion in a more natural context such as a carefully designed game that requires players to solve puzzles using recursion (Lee et al., 2014). Another successful approach is to introduce recursion with concrete examples such as Russian nesting dolls, fractal images, or a description of the stack framework that implements recursion (Wu, 1993). With respect to programming, it is often best to introduce repetition structures such as loops before recursion (Anzai & Uesato, 1982; Kessler & Anderson, 1988; Wiedenbeck, 1988). Concepts should then be reinforced with a variety of programming example solutions that include head recursion, embedded recursion, and tail recursion (Sanders et al., 2006). Students also benefit from tracing methods and predicting the outcome of recursive procedures. This could be numerical results or figures that are drawn using recursive methods.

Students progress from comprehending and tracing solutions to creating their own. However, teaching students to write recursive solutions is more challenging than teaching them to trace solutions. A successful strategy is to teach an applied problem-solving strategy that includes clear step-by-step actions such as: (1) identifying a base case, (2) decomposing the problem into a simpler instance, and (3) incorporating results from the simpler instances into a final solution (Ginat & Shifroni, 1999). Pirolli and Anderson (1985) recommended the use of a variety of code examples early in the introduction to recursion. Students relied heavily on code examples in the early stages of learning to write recursive functions. They modified the examples as if they were templates. The experience of modifying the examples led to a more abstract understanding of recursion which eventually was used to solve new problems.
Research directions

Much of the research on teaching and learning recursion uses static artifacts, such as code solutions or drawings, to infer students’ difficulties, misconceptions, and mental models. Additional qualitative research using observations and interviews could shed light on why these challenges exist and how mental models are formed. Additional research into how best to help students form viable mental models would also be useful. We have noticed our students are often reluctant to draw representations of their code, recursive or not, on paper. Evidence that doing so helps them write, trace, and debug may help to convince them that creating physical representations of their code is valuable, particularly with recursion.

There has been little work exploring the role debugging might play in helping students understand and trace recursive solutions. Presenting incorrect or overly complex solutions and asking students to find and fix the bugs or simplify the code could be helpful. Research into the role recent pedagogies – such as pair programming (McDowell, Werner, Bullock, & Fernald, 2006), peer instruction (Zingaro & Porter, 2014), and problem-based learning (O’Grady, 2012) – might play in facilitating students’ understanding of recursion would also be valuable.

Finally, almost every paper published on recursion, whether it is research-oriented or not, mentions the difficulty students have learning and using recursion. One wonders if this bias might not unwittingly be passed from instructor to student. As McCracken said “If recursion is presented as one powerful tool among others, through many examples and with opportunities to practice using it, sophomores need not have any problems with it” (McCracken, 1987). If we could determine if we are convincing our students learning and using recursion is hard, perhaps we could convince ourselves to teach it differently.

Acknowledgments

We thank Brian Hanks for comments that helped to shape this manuscript.

Disclosure statement

No potential conflict of interest was reported by the authors.

Note

1. See Rinderknecht (2014) for commentary on more than 250 writings related to recursive programming.
References


