Engaging Software Engineering Students with Natural Numbers

Amitrajit Sarkar
Department of Computing
Christchurch Polytechnic Institute of Technology
Christchurch, New Zealand

Mike Lopez
Department of Computing
Christchurch Polytechnic Institute of Technology
Christchurch, New Zealand

ABSTRACT
According to Kronecker, a famous European mathematician, only natural numbers, i.e. positive integers like 1, 2, and 3… are given by God or belong to nature. All other numbers, like negative numbers, fractional numbers, irrational numbers, complex numbers, etc., are creations of the human mind. It is important to notice that all these other numbers are created using the natural numbers. Natural numbers have very interesting patterns and those patterns are elegantly simple and hence simply beautiful. The idea of this paper is to explore different patterns that are created with natural numbers, to demystify the connection of the natural numbers with nature, and then to use them to teach important concepts of software engineering. We will take various examples, discuss the teaching methodology used to teach them, and uncover different software engineering concepts and best practices. The examples that we will use are the Fibonacci sequence and other natural number patterns, and we will connect them with software engineering concepts like loop patterns, recursion, refactoring and decomposition. For the last few years we have used this in our software engineering classes with much success, particularly in relation to student engagement and helping students to think creatively. We are confident that this type of teaching approach can be seamlessly integrated in tertiary as well as in high school software engineering curricula and has no geographical boundaries. This novel teaching approach is ready to be tested in different cultural settings. Finally, we conclude the paper with a desire for future research in cross-cultural, multi-institutional and multi-national settings.

General Terms
Pattern Recognition, Algorithms, Software Engineering Education.

Keywords
Software Engineering, Introductory Programming, Recursion, Iteration, Natural Numbers.

1. INTRODUCTION
According to Kronecker, a famous European mathematician, only natural numbers, i.e. positive integers like 1, 2, and 3… are given by God or belong to nature. All other numbers, like negative numbers, fractional numbers, irrational numbers, complex numbers, etc., are creations of the human mind. It is important to notice that all these other numbers are created using the natural numbers. Natural numbers have very interesting patterns and those patterns are elegantly simple and hence simply beautiful.

1.1 Natural Numbers
People normally introduce natural numbers via enumeration: 0, 1, 2, ... The dots at the end say that the series continues in this manner. Mathematicians consider 0 as a natural number but computer scientists do not. Rather than entering in to that debate, let us exclude 0 as a natural number as we are discussing computer science in this paper. If n is a natural number, then one more than n is a natural number too. While this description is still not quite rigorous, it is a good starting point for describing natural numbers:

A natural-number is either 1 or the result of adding 1 to a natural number. Let us suppose that an operation add1() adds 1 to a natural number. Although we are familiar with natural numbers from school, it is instructive to construct examples from the data definition. Clearly, 1 is the first natural number. It follows that:

(add1 1) is the next one.

From here, it is easy to see the pattern:

(add1 (add1 1))

(add1 (add1 (add1 1)))

(add1 (add1 (add1 (add1 1))))

This example should remind us of the list construction process. We built lists by starting with an empty list and by adding more items. Now, we build natural numbers by starting with 1 and by adding on 1. In addition, natural numbers come with century-old abbreviations. For example, (add1 1) is abbreviated as 2, (add1 (add1 1)) as 3, and so on.

As identified by Mallik [1], we are so used to natural numbers that may fail to notice some interesting patterns in them. For example, let us observe the simple yet beautiful regularity of appearance of all the consecutive natural numbers in the following equations:

1+2 = 3

4+5+6 = 7+8

9+10+11+12 = 13+14+15,

And it continues in this fashion up to infinity. In the next section, let us explore some more examples involving natural numbers.

1.2 Natural Numbers in Nature
There are many situations in nature that involve the Fibonacci series: the original problem about rabbits where the series first appears, the family trees of cows and bees, the golden ratio and the Fibonacci series, the Fibonacci Spiral and sea shell shapes, branching plants, flower petal and seeds, leaves and

petal arrangements, on pineapples and in apples, pine cones and leaf arrangements. All involve the Fibonacci numbers.

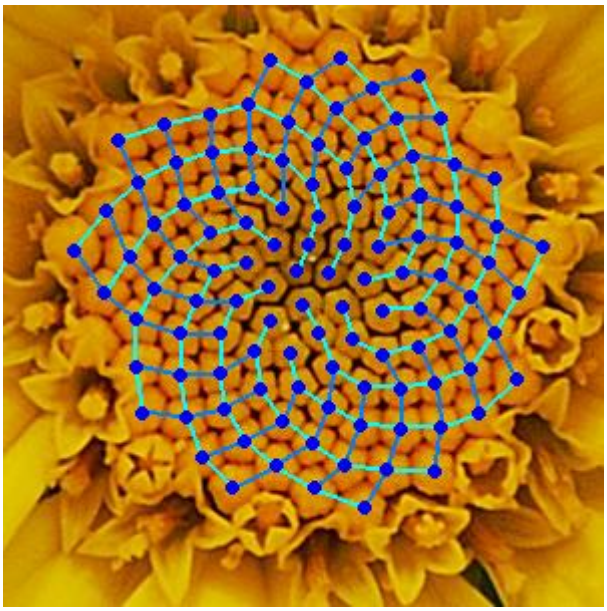

Figure 1: Yellow Chamomile head showing the Fibonacci sequence[1]

The Yellow Chamomile head in Figure 1 shows the arrangement in 21 (blue) and 13 (aqua) spirals. Such arrangements involving consecutive Fibonacci numbers appear in a wide variety of plants [2].

Fibonacci sequences appear in biological settings, in two consecutive Fibonacci numbers, such as branching in trees, arrangement of leaves on a stem, the fruitlets of a pineapple, the flowering of artichoke, an uncurling fern and the arrangement of a pine cone. In addition, numerous poorly substantiated claims of Fibonacci numbers or golden sections in nature are found in popular sources, e.g., relating to the breeding of rabbits, the seeds on a sunflower, the spirals of shells, and the curve of waves [2].

The Fibonacci numbers are also found in the family tree of honeybees. In the *Bee Ancestry Code*, Fibonacci numbers appear in the description of the reproduction of a population of idealized honeybees, according to the following rules:

- If an egg is laid by an unmated female, it hatches a male or drone bee.
- If, however, an egg was fertilized by a male, it hatches a female.

Thus, a male bee will always have one parent, and a female bee will have two. If one traces the ancestry of any male bee (1 bee), he has 1 parent (1 bee), 2 grandparents, 3 great-grandparents, 5 great-great-grandparents, and so on. This sequence of numbers of parents is the Fibonacci sequence. The number of ancestors at each level (Fn) is the number of female ancestors (which is Fn-1) plus the number of male ancestors (Fn-2).

Fibonacci numbers are intimately connected with the golden ratio, for example the ratios of two consecutive numbers in the Fibonacci sequence are seen to generate the following sequence: 1, 0.5, 0.666…, 0.6, 0.625, 0.615…, 0.619…, 0.61818.... This sequence converges to the golden ratio.

We recognise good proportion in the same way as we know how to divide a line in half or erect a perpendicular. We easily settle that an object of art has good or bad proportion, or that this face looks too long, or too short and out of proportion. This magical connecting thread of proportion is none other than the Golden Proportion, a phenomenon related to beauty. To the Greeks, who were predominantly geometers, this Golden Section was a harmonious, almost mystical, constant of nature [2].

1.3 Iteration

In a landmark paper in 1966 [3], Böhm and Jacopini proved that any algorithm capable of executing on a Turing machine could be expressed with just three constructs: sequence, selection and repetition. The use of these constructs led to what is now known as structured programming, in contrast to programs which used a GOTO statement. However, just because a program could be so expressed does not necessarily mean it should be. Edsger Dijkstra [4] argued that the GOTO was harmful, whereas Hopkins [5] made a plea for the sensible use of GOTO. Using an argument based on graph isomorphism, Maurer [6] demonstrated that any unstructured program could be transformed to a structured form. Nowadays, programmers generally prefer to avoid GOTO statements in favour of structured programming techniques.

1.4 Recursion

However, although only sequence, selection and repetition are needed, another widely used technique is that of recursion. Recursion encapsulates decomposition of a problem into sub-problems of the same kind [7]. It allows for elegant definitions and mathematical proofs, but has long been regarded as difficult for beginning programmers to master [7, 8, 9, 10, 11]. However, Mirolo [10] argues that there is no empirical evidence for this and that teaching recursion first should be investigated. Part of the difficulty may be associated with the choice of language. In functional languages, such as Haskell or Scheme, recursion is a natural way of expressing algorithms, whereas in procedural languages, such as Java or C, it may be less natural. This is because procedural languages are much closer to the natural computational model of most computers. In mathematics and computer science, a class of objects or methods exhibit recursive behaviour when they can be defined by two properties:

1. A simple base case (or cases), and
2. A set of rules which reduce all other cases toward the base case.

For example, the following is a recursive definition of a person's ancestors:

- One's parents are one's ancestors (base case).
- The parents of one's ancestors are also one's ancestors (recursion step).

[1] This image is from the Wikimedia Commons. The original image is from Joaquim Alves Gaspar, licensed under the Creative Commons Attribution-Share Alike 3.0 Unported license. It was retouched by RD Bury.

The Fibonacci sequence is a classic example of recursion:

- Fib(0) is 0 [base case]
- Fib(1) is 1 [base case]
- For all integers n > 1: Fib(n) is (Fib(n-1) + Fib(n-2))

Error! Reference source not found. presents a recursion example in Scratch.

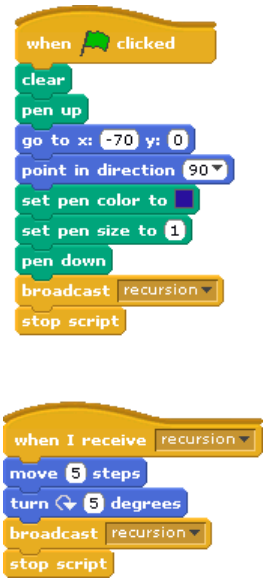

Figure 2: Recursion Example in Scratch

1.5 Recursion versus Iteration

However, a more fundamental issue relates to the conceptual models needed to understand recursion. Kurland and Pea [13] observed programmers who viewed recursion as iteration, and several educators [14, 15] studied transfer abilities from iteration to recursion and vice-versa and concluded that it is more sensible, pedagogically, to base understanding of recursion on iteration than iteration on recursion. Kahney and Eisenstadt [16] examined novices' judgments of given recursive programs and concluded that they developed one of several mental models of recursion, which they named: copies, loop, odd, null, and syntactic magic. Of these, only the *copies model* is regarded is a correct model of recursion. In the copies model, a recursive procedure can be understood as a procedure looping over a stack of function calls [15]. With the *syntactic magic* model the student has no clear idea of how recursion works, but is able to match on syntactic elements. As Sanders and colleagues [17] point out, "It is interesting to note that a large number of students in both classes manifest the magic model of recursion" [p. 141].

From the literature, it would appear that the key problem for students in understanding recursion is the lack of an appropriate conceptual model. Whereas iterative conceptual models can be reinforced by techniques such as tracing, recursive models require the student to work at a higher layer of abstraction. Starting from this insight, Wu and colleagues [18] mapped students' understanding of recursion to a classification of the student as concrete or abstract learners. Although abstract learners generally performed better than concrete learners, they found that concrete models worked better than abstract models for all learners, and that abstract learners did not necessarily benefit more from abstract conceptual models, nor did concrete learners necessarily benefit more from concrete conceptual models. Moreover, although recursion was introduced to students in formal terms, Levi and Talendot [19] found that students did not naturally use any of the formal terms in their discussions about recursion. This suggests a disconnection between recursion as presented by lecturers and recursion as understood by students.

To illustrate some of these issues, we present two implementations of a factorial function in Figure **3: Two Factorial function implementations in C#**

3.

```
Static long iterativeFactorial (long n) {
      long result=1;
      for (long i=n; i > 1; i--) {
            result *= i;
      }
      return result;
}

static long recursiveFactorial (long n) {
      if (n < 2)
            return 1;
      else
            return n * recursiveFactorial(n-1);
}
```

Figure 3: Two Factorial function implementations in C#

However, there have been only a few attempts to compare learners' ability to deal with recursion and with iteration. Among these, Bhuiyan et al. [20] point out the influence of a *looping* mental model on the understanding of recursive computations. Kessler & Anderson [21], and to some extent Wiedenbeck [11], infer from their analysis that iteration should be taught before recursion in order to facilitate the development of a computation model. In contrast, Turbak et al. [22] have observed an improvement of students' performance in exams after changing the course structure and teaching loops as a special case of tail-recursion. Moreover, according to Benander et al. [23], recursive code does not seem to impair program comprehension. Recently, Mirolo [10] conducted a recursion test and an iteration test on first year computer science major students and observed that students find recursion and iteration equally difficult to comprehend. He concluded that, from a teacher's perspective, more effort should be spent to foster problem solving, plan-composition and abstraction skills.

2. WHAT WE ARE TRYING TO ACHIEVE

The objective is to master different looping techniques and to learn and apply recursion. The introductory programming students were exposed to interesting problems related to recursion and looping constructs and as educators we tried to formulate engaging techniques and thus introduced the natural number examples. First, we introduced them to two types of iteration in the form of while or for loops. After that we exposed them to the technique of recursion. Recursion is a basic technique that all programmers must understand. Once they completely understand iteration, replacing iteration with recursion is not that big a step.

Recursion was certainly a lot more intuitive for the task the introductory programming students tried to solve, despite their lack of experience with this style of programming.

As a programmer, learning more techniques give them more tools to choose from, which means that they can choose the

approach which is most intuitive given the problem. Whether it is easy or hard to understand the technique the first time they encounter it does not really matter.

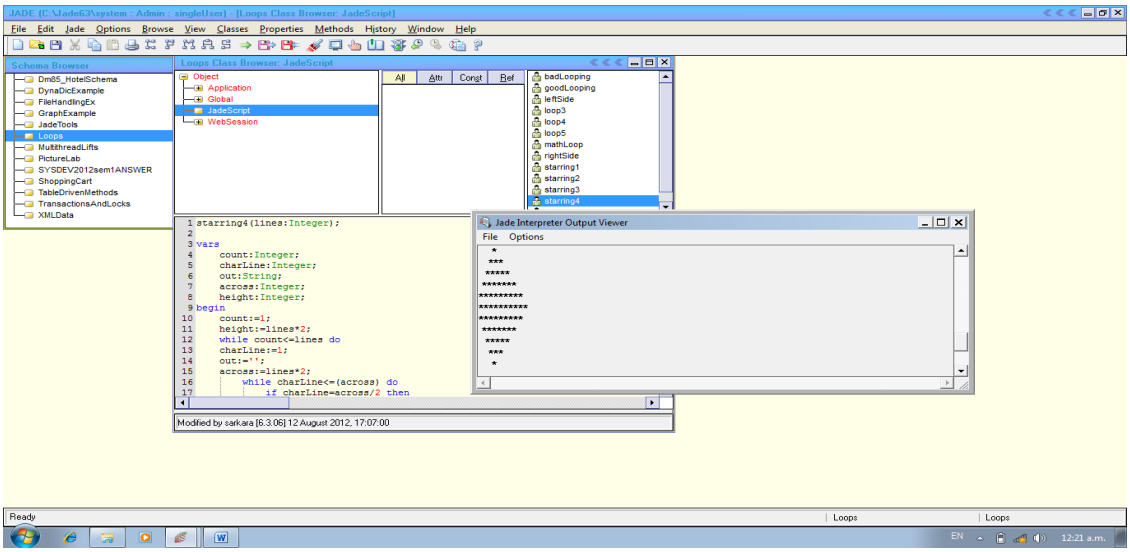

Figure 4: The in class exercise to create a Diamond Shape

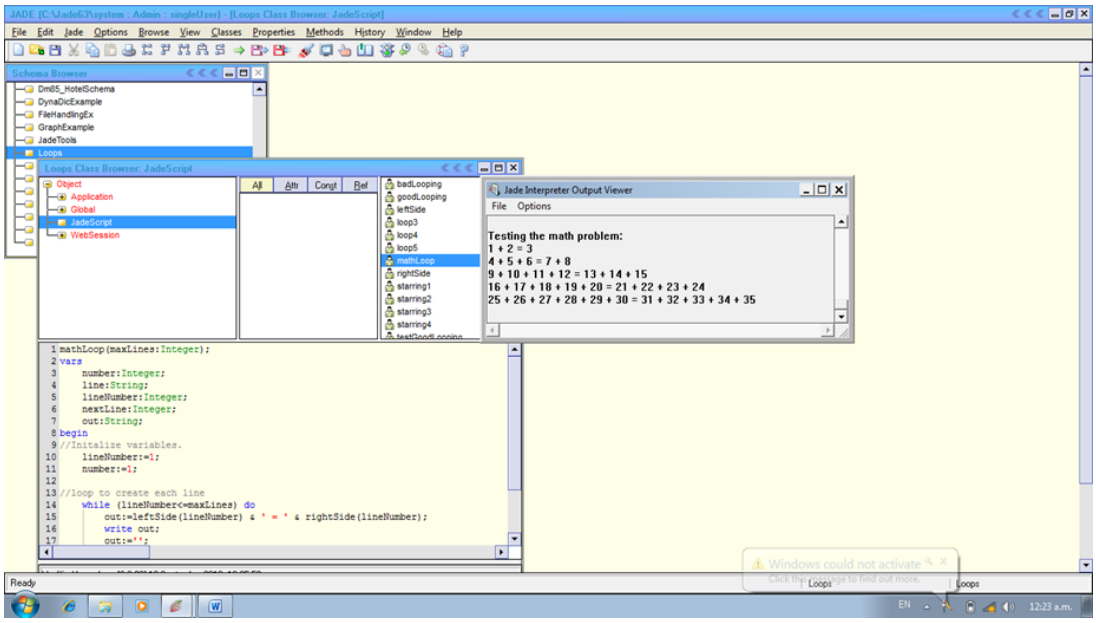

Figure 5: The in class exercise to create the natural number sequence

As part of this learning journey we constantly expose them to known, not very known and absolutely new problems, and encourage them to craft creative solutions to those problems and to understand when to use iteration and when they can use recursion.

3. HOW WE DID IT

To achieve the learning objectives we have selected a set of interesting examples that we will discuss in this section. Some of these examples were used as in-class examples and others can be a good teaching resource for loops and recursion.

The first example is the Fibonacci sequence. The Fibonacci numbers are defined as follows: the first Fibonacci number is 1, the second Fibonacci number is 1, and each subsequent Fibonacci number is the sum of the previous two. Thus the sequence starts 1, 1, 2, 3, 5, 8, and so on. Define a recursive function ***fib*** that takes a natural number n and returns the nth Fibonacci number. (Hint: fib needs two base cases.) Find the 7^{th} number in the Fibonacci sequence.

The second example is the Factorial. Define a recursive function ***factorial*** that takes a natural number n and returns the product of the first n positive numbers. That is: factorial n = 1 * 2 * … * (n-1) *n. For example, factorial 6 is 720.

The third example is to create a diamond shape (see Figure 4).

As identified by Mallik [1], we are so used to natural numbers that we may fail to notice some interesting patterns in them.

For the fourth example, let us observe the simple yet beautiful regularity of appearance of all the consecutive natural numbers in the following equations:

- 1+2 = 3
- 4+5+6 = 7+8
- 9+10+11+12 = 13+14+15,

It continues in this fashion up to infinity. Based on this natural number pattern we formulated the question:

- Find the sequence of the 5th row (See Figure 5).

A fifth example was Sierpinski's Triangle. Students were asked to describe the recursive structure of a Sierpinski's triangle. In particular, students had to identify the base case(s) and which operations could be applied to increase the recursion depth by 1.

A sixth interesting example could be Lucas Numbers or Lucas Series. Like the Fibonacci numbers, each Lucas number is defined to be the sum of its two immediate previous terms, i.e. it is a Fibonacci integer sequence. However, the first two Lucas numbers are L0 = 2 and L1 = 1 instead of 0 and 1, and the properties of Lucas numbers are therefore somewhat different from those of Fibonacci numbers. A Lucas number may thus be defined as follows: The sequence of Lucas numbers begins: 2,1,3,4,7,11,18,29,47, ... Also like all Fibonacci integer sequences, the ratio between two consecutive Lucas numbers converges to the golden ratio.

The last example (Figure 6) we want to share is the Collatz conjecture, named after Lothar Collatz, who first proposed it in 1937. The conjecture is also known as the 3n + 1 conjecture. It states:

- Take any natural number n.
- If n is even, divide it by 2 to get n / 2.
- If n is odd, multiply it by 3 and add 1 to obtain 3n + 1.
- Repeat the process indefinitely.

The Collatz conjecture

Consider the following operation on an arbitrary positive integer:

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

Examples
Starting with n = 6, one gets the sequence
6, 3, 10, 5, 16, 8, 4, 2, 1.
With n = 11, for example, takes longer to reach 1:
11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.

This program halts when the sequence reaches 1. If the Collatz conjecture is true, the program will always halt (stop) no matter what positive starting integer is given to it.

Figure 6: The Collatz conjecture

The conjecture is that no matter what number you start with, you will always eventually reach 1. The property has also been called oneness.

A pseudo code solution of the Collatz conjecture is shown in Figure 6.

Finally, we note that computing compound interest and The Tower of Hanoi game are also good exercises that can be used.

4. HOW WELL IT WENT

Students enjoyed the examples and the challenges of identifying patterns and crafting creative solutions. They were fully engaged during and learnt that:

- A "while" loop has two visually distinct parts: a body and a termination condition.
- A "for" loop has more parts: initialization, increment, termination condition, body.

A tail-recursive function does not have a standard, visually distinct way of code organization. It is functionally equivalent, and crystal clear in simple cases (like factorial or Fibonacci numbers), but not as clear in more general cases.

For example, if we are iterating over a list, a tail-recursive function is fine. But when we need to increment a counter along the way, we need to change the code in a non-obvious way. That suggests the requirements of more structured forms of looping. They also learnt about a common computer programming tactic that is to divide a problem into sub-problems of the same type as the original, solve those problems, and combine the results. This is often referred to as the divide-and-conquer method.

5. CONCLUSION AND FUTURE RESEARCH

There are basically two types of iteration. There is the *foreach* style of iteration which just iterates over a collection of things, and then there is the type of iteration that has many conditions that can affect the next step in the iteration. This latter form of iteration fits very nicely with use of pattern matching and recursion. As students started getting used to this way of thinking, they found that the code became surprisingly easy to write. The divide and conquer method that it naturally leads to is very powerful.

For the last few years we have used this in our software engineering classes with much success, particularly in relation to student engagement and helping students to think creatively. We are confident that this type of teaching approach can seamlessly be integrated in tertiary as well as in high school software engineering curriculum and has no geographical boundary. This novel teaching approach is ready to be tested in different cultural settings.

Further work is needed to explore the conceptual models students actually use to understand recursion. It is still an open question whether providing students from the outset with an expert conceptualisation is the best way for them to learn, or whether there are intermediate conceptual models that could provide useful scaffolding as they develop their understanding. Finally, we conclude the paper with a desire for future research in cross-cultural, multi-institutional and multi-national settings.

6. REFERENCES

[1] Mallik, A. K. 2004. From Natural Numbers to Numbers and Curves in Nature-I. RESONANCE, September 2004.

[2] Mallik, A. K. 2004. From Natural Numbers to Numbers and Curves in Nature-II. RESONANCE, October 2004.

[3] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," Communications of the ACM, vol. 9, no. 5, pp. 344-371, 1966.

[4] E. Dijkstra, "Go to statement considered harmful," Communications of the ACM, vol. 11, no. 3, pp. 147-148, 1968.

[5] M. Hopkins, "A case for the GOTO," ACM SIGPLAN Notices - Special issue on control structures in programming languages, vol. 7, no. 11, pp. 59-62, 1972.

[6] W. Maurer, "Generalized structured programs and loop trees," Science of Computer Programming, vol. 67, no. 3, pp. 223-246, 2007.

[7] D. Ginat and E. Shifroni, "Teaching recursion in a procedural environment: How much should we...?," in Proceedings of the 30th SIGCSE conference, 1999.

[8] B. Haberman and H. Averbuch, in Proceedings of the 7th ITiCSE conference, 2002.

[9] H. Kahney, "What do novice programmers know about recursion," in Proceedings of the SIGCHI conference, 1983.

[10] R. Sooriamurthi, "Problems in comprehending recursion and suggested solutions," in Proceedings of the 6th ITiCSE conference, 2001.

[11] S. Wiedenbeck, "Learning recursion as a concept and as a programming technique," in Proceedings of the 19th SIGCSE conference, 1988.

[12] C. Mirolo, "Is iteration really easier to learn than recursion for CS1 students?," in Proceedings of the ICER conference 2012, Auckland, 2012.

[13] D. Kurland and R. Pea, "Children's mental models of recursive logo programs," in Proceedings of the Fifth Annual Conference of the Cognitive Science Society, 1983.

[14] Y. Anazi and Y. Uesato, "Learning recursive procedures by middle-school children," in Proceedings of the fourth annual conference of the Cognitive Science Society, 1982.

[15] A. Kessler and J. Anderson, "Learning flow of control: Recursive and iterative procedures," Human-Computer Interaction, vol. 2, pp. 135-166, 1986.

[16] H. Kahney and M. Eisenstadt, "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, 1982.

[17] I. Sanders, V. Galpin and T. Götschi, "Mental Models of Recursion Revisited," in Procceedings of the ITiCSE conference '06, Bologna, Italy., 2006.

[18] C. Wu, N. Dale and L. Bethel, "Conceptual models and cognitive learning styles in teaching recursion," in Proceedings of the SIGCSE Conference '98, Atlanta, GA, 1998.

[19] D. Levi and T. Lapidot, "Recursively speaking: Analyzing students' discourse of recursive phenomena," in Proceedings of the SIGCSE conference, Austin, TX, 2000.

[20] S. Bhuiyan, J. E. Greer, and G. I. Mccalla. Supporting the learning of recursive problem solving. Interactive Learning Environments, 4(2):115{139, 1994.

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

[22] F. Turbak, C. Royden, J. Stephan, and J. Herbst. Teaching recursion before loops in CS1. J. Computing in Small Colleges, 14(4):86{101, 1999.

[23] A. Benander, B. Benander, and H. Pu. Recursion vs. iteration: An empirical study of comprehension. J. of Systems and Software, 32(1):73{82, 199