Characterizing, Rationalizing, and Reifying Mental Models of Recursion

Shawkat H. Bhuiyan Jim E. Greer Gordon I. McCalla

ARIES Laboratory
Department of Computational Science
University of Saskatchewan
Saskatoon, SK, Canada S7N 0W0
aries@cs.usask.ca

Abstract

Mental models reflect people's knowledge about entities and systems around them. Therefore, knowing and understanding mental models can help in exploring cognitive issues in instruction including why a student takes a certain approach or applies a particular strategy to solve a problem, why a student makes mistakes, and why and how misconceptions are developed. Four different mental models of recursion, used for synthesizing solutions to recursive programming problems, have been identified through students' protocols. Each model has been characterized in a way consistent with the students' protocols. Various problem solving behaviours are rationalized in terms of the models. Suggestions are made as to how the mental models develop and evolve in the course of learning. We also present a learning environment in which these mental models are reified and we show how mental models can be incorporated into an intelligent tutoring system.

Introduction

A *mental model* is a coherent collection of knowledge held by a person about some aspect, entity or concept of the world. People use mental models to interpret the world, and therefore, aspects of human behaviour can be explained through these models (Gentner and Stevens 1983). Since mental models reflect people's knowledge about entities around them, knowing and understanding mental models can help in understanding cognitive issues in instruction such as *why* a student takes a certain approach or applies a particular strategy to solve a problem, *why* a student makes mistakes, and *why* and *how* misconceptions are developed. By understanding the mental models of a student, a tutor or mentor can design appropriate individualized pedagogical guidance.

Most mental model research has been conducted in domains involving physical devices with well-structured physical principles. Working on the Student Computing ENvironmenT (SCENT) LISP programming advisor (McCalla *et al* 1988), an ongoing intelligent tutoring project at the ARIES Laboratory, we became interested in understanding mental models of abstract concepts, especially recursion in computer programming. Recursion is a very interesting domain to study because many students claim to experience a significant cognitive change (often describing this as a radical re-organization of knowledge) as they gain understanding of recursive concepts.

In this regard, an empirical study was carried out to study mental models of recursion with the following goals in mind: 1) What are mental models of recursion in the context of computer programming? 2) How do these models develop and how are they modified? 3) How does migration from one model to another model take place? 4) How do different models relate to one another? 5) When multiple models exist, are they all applicable or are they situation dependent? 6) What are the transformation operators or mapping functions among different mental models? 7) Are changes in mental models evolutionary or revolutionary in nature?

The protocol analysis and preliminary findings of our empirical study were discussed in (Bhuiyan, Greer & McCalla 1989). This paper takes the work further by characterizing mental models of recursion, rationalizing students' problem solving in light of these models, and providing a system for reifying these mental models.

Background

Various researchers have studied students' conceptions of recursion in the context of computer programming. Their findings can be summarized as follows: 1) Novices at early stages of learning about recursive programming possess a *loop model;* that is, they view recursion as some sort of iterative process (Anzai and Uesato 1982; Kahney 1982; Kurland and Pea 1985; Kessler & Anderson 1986). 2) A *syntactic model* of recursion has been suggested by Pirolli (1985) as an ideal novice model of recursion, yet with a *syntactic model* a student cannot explain the behaviour of a recursive procedure (Kahney 1982). 3) First learning iteration may help students learning recursion, but not vice versa (Anzai and Uesato 1982; Kessler and Anderson 1986). 4) Experts' mental models of recursion are different from those of novices (Kahney 1982). 5) There are no significant difference in students' performance with recursive tasks when they were taught with theoretical *vs* syntactic methods, although there is some indication that transfer is enhanced through a theoretical approach (Greer 1987).

Although the literature hints at several different mental models of recursion, no comprehensive work has yet been done in this domain. Therefore, we believe it is important

to investigate mental models of recursion more carefully, especially how the models are used in synthesizing recursive programs, and hence we carried out the empirical study. The intent of the study was not to statistically validate mental models of recursion, but rather to explore the goals mentioned above.

Mental Models of Recursion

Our empirical study involved monitoring the problem solving of six non-major computer science students on a weekly basis to analyse their knowledge of recursion as they learned the concept in a programming course. Four different mental models of recursion: *loop model, syntactic model, analytic model* and *analysis/synthesis (A/S) model* have been identified through analysis of the students' verbal protocols. These four mental models of recursion were used by various students at varying times as they synthesized solutions for recursive problems. Earlier papers (Bhuiyan et al. 1989; Greer 1987) provide evidence justifying these mental models. A number of analysis models were also used by the students for analyzing and tracing recursive programs, but these are not discussed here. Here an attempt is made to characterize students' problem-solving behaviour consistent with each model.

Loop Model

The *loop model* is a flawed model of recursion that many students develop early in their study of recursive programming (Anzai and Uesato 1982; Kahney 1982; Kurland and Pea 1985; Kessler and Anderson 1986). This model comes into existence when the student tries to understand and explain recursion in terms of prior knowledge about iterative programming.

Typically, the beginner has some basic declarative knowledge about recursion either from a text book or from a teacher. This knowledge could include facts such as: a recursive function calls itself, a recursive function has a base case and a recursive case, and so forth. The student may not have acquired any means to apply this declarative knowledge, i.e, he/she may not have the procedural counterpart of the knowledge. On the other hand, his/her knowledge about iteration typically consists of both declarative and procedural knowledge. The student's knowledge about iteration dominates his/her knowledge about recursion, and it is likely that the student tries to understand recursion by drawing analogies to loop or iterative structures.

Usually students develop informal solution plans before writing programs. A student with a *loop model* is likely to develop an iterative solution plan. If he/she attempts to translate this to a recursive algorithm, there is a good chance that the algorithm will be flawed, incorporating such features as declaration of loop index variables, initialization of loop index variables, update (increment/ decrement) of these variables, and a termination test. Manifestation of these features depends upon the initial selection of the loop structure that the student had in mind while developing the algorithm. The student's program usually does not contain an explicit loop, because the student knows that recursions are different from iterations, but instead substitutes the intended loop structure with a recursive call.

Protocol analysis revealed several issues related to students' use of the *loop model*. Learning iteration may assist later learning of recursion, as Kessler and Anderson (1986) and Uesato and Anzai (1982) claim, but we observed that the *loop model* did not help students in understanding recursion; rather students became confused as they tried to apply the *loop model* to synthesizing recursive solutions. By the end of the second week of our investigation, all students who were initially using the loop model seemed to have moved on to more suitable mental models. Greer (1987) made a similar observation in his experiment.

Syntactic Model

Students acquire the *syntactic model* by abstracting structural features of recursion when they have little conceptualization of recursion as a problem-solving technique, but have considerable declarative knowledge about recursion. At this stage, their declarative knowledge focuses on two structural features of recursive functions: a base case and a recursive case; where a process to solve a problem is repeatedly called in the recursive case, and the base case stops the process. When a student transforms such knowledge of recursion into problem solving for simple problems, he/she usually abstracts these two features into a *recursion template*, a schema with slots for base case and recursive case as shown in Figure 1. Again, each case has a condition part and an action part.

```
(defun <function-name> (<arg1> ... <arg n>)
   (cond
      (<condition-1>  <action-1>)   ;; base case
      (<condition-2>  <action-2>)   ;; base / recursive case
        •
        •
        •
      (<condition-k>  <action-k>)))   ;; recursive case
```

Figure 1: *Syntactic model* template

To synthesize a solution to a recursive programming problem using the *syntactic model*, first a template is selected. The selected template can be a 'basic template' like the one in Figure 1 or a template from an earlier problem analogous to the current problem. After selecting the template the student fills in the condition and the action parts in the case slots with programming language specific code chunks. In general, a student makes several attempts to fill in recursive case slots when partial results are to be passed back. The protocols in Figure 2 illustrate the use of the *syntactic model*.

A serious shortcoming of the *syntactic model* is that although a student is aware of the conditions and actions in a template, he/she may not know *how* to derive them. This frequently results in nearly random slot-filling behaviour by novice programmers. Moreover, while synthesizing programs using this model, novices,

generally, think of the solutions at the programming language code level. This may be due to the fact that they are simply concerned with filling in the template slots.

(a) Student K is looking for an arbitrary recursion template

K: .. I don't know. I have to ... I can't remember the "format" that goes here. You showed me a couple of minutes ago.

Tutor: Okay. Why do you have to see a format?

K: Well, the base case and the recursive case - how they are set up, I forgot.

(b) An example of template filling behaviour

K: So, we need a base case. The base case would be: if the list is empty then return 0, else .. (over a minute) .. Else take rest of L. ... return Count + 1. ....This is not how it's done

Figure 2: Protocol demonstrating the *syntactic model*

Although the *syntactic model* is better than the *loop model* it does not constitute a complete understanding of recursion. Nevertheless, we observed that novice students frequently used the *syntactic model* in synthesizing recursive solutions. This corresponds to Escott's (1988) research, where she found 80% of students' programs were structural analogies of earlier programs. The LISP Tutor (Anderson and Reiser 1986) encourages students to use recursion templates which they fill in to arrive at a final program. We have found that for solutions to routine problems, the *syntactic model* is sufficient, although its usefulness diminishes when students are faced with novel or difficult problems. Perhaps for this reason the LISP Tutor augments the problem solving by guiding students through a planning dialog to formulate solutions.

Analytic Model

Although a student may solve simple recursive problems employing the *syntactic model*, complex problems require a deeper understanding in order to map the problem to a recursive construct. This "deep understanding" involves the ability to analyze input-output behaviour of a problem and to determine input conditions and corresponding output actions together with associated transformations. Such analytic cognitive ability to synthesize a recursive solution gives rise to the *analytic model*.

With the *analytic model*, a student does not view recursion simply as a physical construct like the syntactic template; rather he/she views recursion as a problem-solving technique. Therefore, the significance of this model is to analyse the input-output behaviour of the given problem.

Solution synthesis using the *analytic model* has the following three steps: 1) Determine cases: Analyze the various input cases for the given problem. Determine the input cases and the corresponding output strategies. In this step, the student specifies his/her intentions, an informal solution specification, for the problem. 2) Translate input cases to input conditions and translate output strategies to output actions: In this step, the student plans a high-level solution, which can be translated to any programming language. Students in the empirical study were found to use natural language phrases to outline the solution plan. 3) Translate input conditions and output actions into programming language code.

The *analytic model* provides two levels of intermediate representations between the problem statement and the solution (step 1 and step 2 above). The step 2 intermediate representation is particularly interesting because it supports a pseudo-language; a mixture of natural language and programming language primitives, which students use to elaborate the input conditions and the corresponding output actions for the problem. To describe a solution in natural language, a student needs to attain a deeper level of cognition than for the *syntactic model*. The protocol in Figure 3 shows a student's solution plan for the *LIST-B* problem, to *return a list of all top level B's in a list*:

S:....So, these are lists, then? O.K. If Null L then Return the empty list. If First of the list is a B then return B, and work (on) the rest. Else Gather ..

The solution plan consisted of the following three cases:

Case 1:	If NULL L then return an empty list.
Case 2:	If the first of the list is a B then return B, and work on the rest of the input list.
Case 3:	Else (for none of the above conditions), Gather

Figure 3: Protocol demonstrating the *analytic model*

In Figure 3, student S was not working directly with base case(s) or recursive case(s) as in the *syntactic model*. Moreover, she was also not concerned with transforming the solution plan into program code. For example, in Case 2, she did not articulate the exact code for the condition part or for the action part. Interestingly, she made no mention of constructing an output list of B's, which the problem asked for, but she wanted to process ("work on") the rest of the input list. This suggests her intention (or goal) was implicit from the problem statement.

The main difference between the *analytic model* and the *syntactic model* is that with the *syntactic model* a student writes program code directly from the problem statement, whereas with the *analytic model* the student first determines a solution plan from the problem statement using an intermediate language and then transforms the plan into program code.

Analysis/Synthesis Model

The *analysis/synthesis (A/S) model* is the most powerful model of recursion encompassing both structural and functional properties. Expert programmers seem to possess this model. Only one student in our study began to acquire this model and attempted to use it in the solution of a challenging problem in the last week of our study. We have uncovered little direct evidence for this model, although other work (cf. Greer 1987) supports the existence of the *A/S model*. We include a brief description of the *A/S model* in this paper for the sake of completeness.

The *A/S model* gives the ability to reduce a given problem into smaller ones and to synthesize the corresponding solutions into a global solution for the problem. The usual steps of solving a problem using the *A/S model* are as follows: 1) Determine the smallest pieces that can be solved instantaneously. 2) Break the problem into subproblems where solution to a subproblem is the solution to a smallest piece plus solution to a smaller subproblem. 3) Determine stopping case(s).

Even expert programmers do not apply the *A/S model* every time they solve a problem recursively. If the problem is simple then there is a good chance that a similar problem has been previously solved, thus providing a template or a chunk to embody the solution. In other words, for familiar problems, most programmers seem to employ a *syntactic model.*

Evolution of Mental Models

Students acquire (or develop) mental models of recursion during instruction and problem solving. Mental models seem to evolve in a sequential fashion, roughly in the sequence of *loop model*, then *syntactic model, analytic model* and finally the *A/S model.*

Over the five weeks of our study students were asked (among other things) to solve nine problems with recursive solutions. From their protocols, the mental model that they were trying to use for formulating each problem solution was identified. The problems generally increased in difficulty through the first four weeks and a difficult, challenging problem was given in the final week. Table 1 shows the mental models that the six students used while attempting to formulate a solution to the problems. In a number of cases one mental model proved insufficient for the student, and a second attempt using a different mental model was observed. Table 1 demonstrates that in most cases students evolved more sophisticated mental models over time.

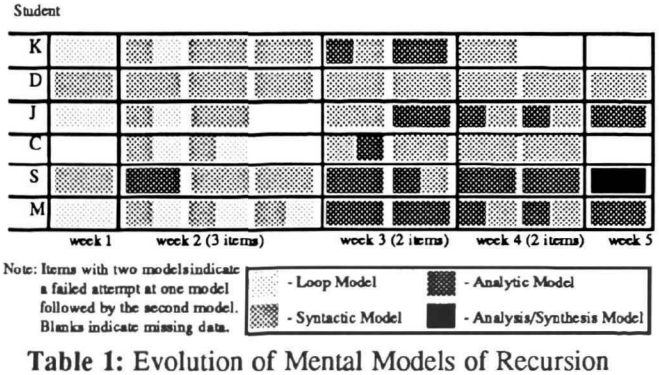

Table 1: Evolution of Mental Models of Recursion

Our empirical study supports the hypotheses that students construct particular mental models, that the models are enhanced by new material that they learn from class lectures or from other sources, that the models become inappropriate for certain problem solving situations, and finally that new models evolve. It seems that mental models, once formed, are persistent. Students frequently return to more established mental models when a first attempt to solve a problem with a new model fails. It also seems that some students will attempt to apply a well established mental model, find that it is not sufficient to conceptualize the solution to the problem, and only then will they apply a new mental model. Students seem to adopt more sophisticated mental models of recursion as they meet with more difficult recursive problems.

A Learning Environment to Support Mental Models

We have determined that mental models play an important role in students' understanding of recursive programming. It follows that mental models can be utilized in better understanding students' intermediate problem solving behaviour and also can help in promoting students' learning of recursion. In essence, mental models, if properly used in intelligent tutoring systems, will increase the diagnostic precision and the appropriateness of pedagogical guidance offered by the system. To this end, we have constructed a prototype learning environment (named PETAL) in which students may choose from a variety of **P**rogramming **E**nvironment **T**ools (PETs) which correspond to various mental models of recursion.

At the present time PETAL is being used to further explore mental models of recursion by observing student problem-solving behaviour when they are constrained by an environment that supports a particular mental model. PETAL does not simulate the execution of explicit runnable mental models; rather it uses what we believe about the existence of these models in the learner's mind to provide scaffolding to support the learner's use of the models. It is important to realize that it is the learner who "runs" the mental model on his/her mind with PETAL merely providing support and constraint. PETAL directly supports specific models of recursion by providing an intelligent computer-based environment in which students formulate solutions to programming problems. The student first selects a problem to work on and then specifies the parameters for the function to be synthesized. Next the student selects among the available PETs to choose an environment corresponding to a mental model in which to solve the problem. Students may switch from one PET to another as they attempt to solve a problem. Since the *loop model* is not a viable model of recursion, there is no corresponding PET. In our prototype implementation of PETAL, only the *syntactic* and the *analytic* PETs have been built. Twenty-three programming tasks have been included so far.

The *syntactic* PET supports solution synthesis from the viewpoint of the *syntactic model.* The student using this PET must construct a recursive template of base case(s) and recursive case(s). Next he/she must fill in the case slots with problem specific code chunks. In order to assist the student, the *syntactic* PET provides a menu of available code chunks specific to the selected problem. The code chunks for a particular problem are created in advance by a domain expert. Code chunks may contain distractors, which can be useful in diagnosing and clarifying potential

misconceptions which the student might possess. Once the code chunks have been filled into the template, corresponding LISP code is automatically generated by the PET.

The *analytic* PET corresponds to the *analytic model* for solution synthesis. With the *analytic* PET, the student views recursion as a problem solving technique based on input/output (I/O) analysis of the problem whereupon these I/O behaviours are mapped to properties of recursion: base case(s) and recursive case(s). Unlike the *syntactic model*, the *analytic model* provides cognitive support to the student to derive the cases through I/O analysis. Program synthesis using the *analytic* PET consists of three stages. **Intention stage**: Determine input cases and corresponding output strategies for the given problem. **Plan stage:** Determine the solution plan by deriving input conditions and output actions corresponding to the input cases and the output strategies respectively. **Code translation stage:** Translate the solution plan into program code.

Figure 4 shows the *analytic* PET initialized with the *count-atoms* problem (counting the number of top level atoms in a list). The *analytic* PET manifests these three stages of solution synthesis. First, at the **intention stage**, the student determines the possible input cases and the corresponding output strategies for the problem. Lists of such choices of the input cases and the output strategies are displayed in Input Case Choices and Output Strategy Choices tables (in the upper left part of the screen display). As input case choices and output strategies are selected, each is displayed in the I-Cases and O-strategies tables (in the lower left part of the screen display). Together the cases and the strategies constitute an informal and coarse-grained solution to the problem. In a broad sense, the student's intentions are captured in this informal solution.

During the **plan stage**, the student develops a solution plan consisting of input conditions and output actions, derived from the input cases and the output strategies respectively. An input condition and its corresponding output action together make a subplan, where the condition is the context of the subplan and the action is the goal of the subplan. The *analytic* PET provides natural language phrases to elaborate the solution plan. These phrases are problem specific and are supported by empirical evidence from our earlier study of problem-solving protocols.

After developing the solution plan, the **code translation stage** begins. The translation is not fully automatic since precise code cannot always be inferred from the student's natural language plans. When the student presses the **Edit Program** button, a LISP Editor appears which shows all the input-output analysis (as commentary) together with a partially filled code template for the solution. It is left to the student to flesh out the program code derived from the solution plan.

The example above, shows how the *analytic* PET provides students support for synthesizing solutions to recursive problems consistent with the *analytic model* of recursion. PETAL currently acts as a stand-alone programming environment, and not as a complete instructional system like the Bridge System (Bonar & Cunningham, 1988) or the LISP Tutor (Anderson & Reiser 1986). PETAL is designed to ease students' transitions through increasingly sophisticated mental models of recursion. Even without knowledgeable feedback to the student, PETAL is proving to be a useful programming environment, since students explore solutions to problems in a rich conceptual environment and can produce LISP code with little effort. After pilot-testing PETAL with a small number of students we are now beginning to consider its potential as an interface to an intelligent tutoring system (specifically the SCENT-3

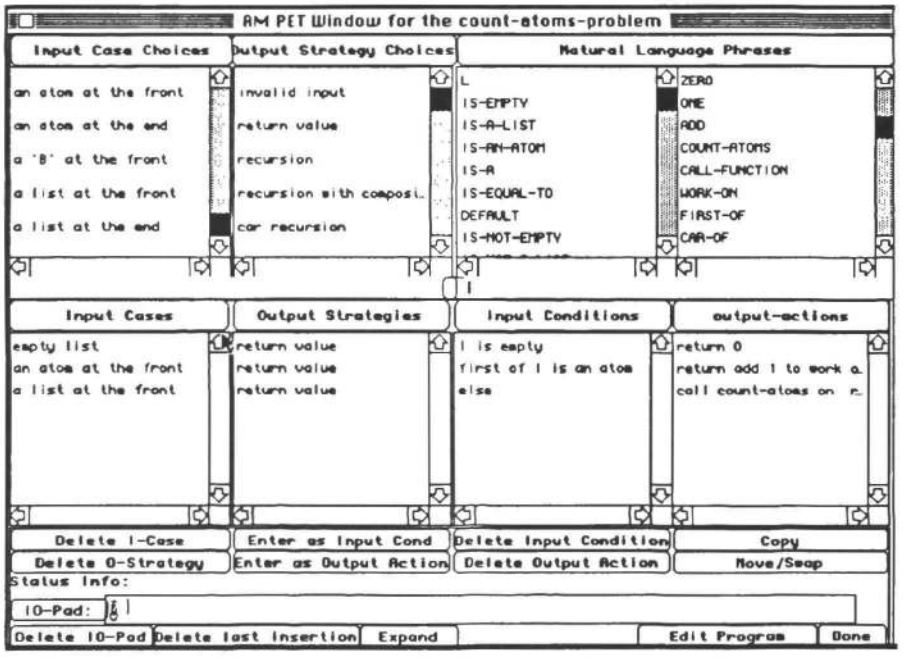

Figure 4: *Analytic* PET for the count-atoms problem

system). Interacting with PETAL, students make explicit the general strategies and specific plans they are considering. This knowledge about strategies and plans is extremely useful and difficult to infer in a standard intelligent tutoring system. The additional knowledge available to a tutoring system that uses a mental model-based interface is considerable. At the same time, the added benefit of knowledgeable feedback to the student using PETAL should prove to be substantial. Although much more research is needed, PETAL promises to be a viable front-end for the SCENT intelligent tutoring system.

Conclusion

Intelligent tutoring systems will achieve their fullest potential when they can support students' learning at the deep "mental modelling" levels, rather than merely support their understanding at a surface level. We have examined how to create such support tools for the domain of programming, in particular for recursive programming concepts. A prerequisite to the success of our endeavour has been to carry out an empirical study delineating the space of mental models actually used by students. Then support tools for each identified mental model have been defined and are currently being tested on students. Student use of these tools is feeding back into our understanding of the mental models themselves. The symbiotic relationship between empirical studies and the system that grows out of these studies is an important lesson of this research.

This research also contributes to the burgeoning body of knowledge about mental models. In contrast to the usual mental models of physical systems, we investigate mental models in an abstract domain. Recursion in computer programming has two features: first, it is a problem solving technique used in synthesizing programs, and second, it is a process or control structure that controls the execution order of the program. Using the synthesis models of recursion, discussed in this paper, students formulate recursive solutions. Our experiments have shown that students do indeed employ mental models in recursive problem solving, that they adopt more sophisticated mental models as the need arises, and that they can switch among mental models as appropriate.

Moreover, our work on PETAL adds to the repertoire of tools to support deep conceptualization by students as they learn. Our PETs are similar to so-called intermediate representations employed in various intelligent tutoring systems such as the Bridge Tutor (Bonar & Cunningham 1988), GIL (Reiser et al 1990), and Angle (Koedinger & Anderson 1990). However, we provide several different representations, rather than just one as in these other systems. These multiple representations are proving to be valuable pedagogically, and are being used to help us refine and clarify our understanding of the structure and use of mental models. They also provide us with a precise lens through which to view the genetic relationships students evolve among several different mental models.

Our future work will continue to refine and elaborate these mental models. We will explore how students actually use the mental models, especially how use of the models is altered as student knowledge evolves. Finally, we will continue to investigate how to build effective support tools for the mental model level of learning.

Acknowledgements

We would like to thank NSERC and the University of Saskatchewan for their financial support.

References

Anderson, J.R. and Reiser, B. 1986. The LISP Tutor, *Byte*, April, 159-175.

Anzai, Y. and Uesato, Y. 1982. Learning recursive procedures by middle school children. *The 4th Cognitive Science Conference*, 100-102, Ann Arbor, MI, USA.

Bhuiyan, S.H., Greer, J.E. and McCalla, G.I. 1989. Mental models of recursion and their use in SCENT. In Ramani et al. (Eds.), *Knowledge-based Computer Systems*, 135-144. Bombay, India.

Bonar, J.G. and Cunningham, R. 1988. Intelligent tutoring with intermediate representations. In *Proceedings of ITS'88*, 25-32. Montreal, Canada..

Escott, J. 1988. *Problem Solving by Analogy in Novice Programming*. ARIES LAB Research Report 88-3. Dept of Comp. Science, University of Saskatchewan, Canada.

Gentner, D. and Stevens, A. Eds. 1983. *Mental Models*. Hillside, NJ: Lawrence Erlbaum.

Greer, J.E. 1987. *Empirical Comparison of Techniques for Teaching Recursion in Introductory Computer Science*. Ph.D. Thesis. The University of Texas at Austin.

Kahney, H. 1982. *An In-depth Study of the Cognitive Behaviour of Novice Programmers*. Tech. Report No. 5. Milton-Keynes, England: The Open University.

Kessler, C. and Anderson, J. 1986. Learning flow of control: Iterative and recursive procedures. *Human-Computer Interaction*, **2**, 135-166. Hillside, NJ: Lawrence Erlbaum.

Koedinger, K.R. and Anderson, J. 1990. Theoretical and empirical motivations for design of ANGLE. *AAAI Spring Symposium* , Stanford University.

Kurland, D.M. and Pea, R.D. 1985. Children's mental models of recursive LOGO programming. *Journal of Educational Computing Research*, 1(2):235-243.

McCalla, G.I., Greer, J.E. et al. 1988. Intelligent advising in problem solving domains: The SCENT-3 architecture. *Proceedings of ITS'88*, 124-131. Montreal, Canada.

Norman, D. 1983. Some observations on mental models. In D. Gentner and A. Stevens (Eds). *Mental Models*. Hillside, NJ: Lawrence Erlbaum.

Pirolli, P. 1988. A Cognitive model of computer tutor for programming recursion. *Human-Computer Interaction*, **2**, 329-355. Hillside, NJ: Lawrence Erlbaum.

Reiser, B.J., Ranney, M. Lovett, M.C. and Kimberg, D.Y. 1990. Facilitating students' reasoning with causal explanations and visual representations. *4th Conference on AI and Education*, 228-235. Amsterdam, Netherlands.