DYNAMIC STUDENT MODELLING IN AN INTELLIGENT TUTOR FOR LISP PROGRAMMING

Brian J. Reiser
John R. Anderson
Robert G. Farrell

Advanced Computer Tutoring Project
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

We describe an intelligent tutor for LISP programming. This tutor achieves a set of pedagogical objectives derived from Anderson's (1983) learning theory: provide instruction in the context of problem-solving, have the student generate as much of each solution as possible, provide immediate feedback on errors, and represent the goal structure of the problem-solving. The tutorial interface facilitates communication and prevents distracting low-level errors. Field tests of the tutor in college classes demonstrate that it is more effective than conventional classroom instruction.

1. Introduction

Constructing an intelligent tutoring system to teach complex cognitive skills is not only a practical application of cognitive science theory and methodologies, but is also arguably the strongest test of a learning theory. In this paper, we report our work on an intelligent tutor that effectively helps students learn to program in LISP. This tutor was designed according to a set of pedagogical principles derived from Anderson's (1983) ACT* learning theory (Anderson, Boyle, Farrell, & Reiser, 1984; Anderson, Boyle, & Reiser, 1985). We shall describe how the tutor achieves a set of pedagogical objectives, and present the results of recent field tests of the tutor in Carnegie-Mellon University classrooms.

2. Tutorial Goals

Private tutoring is generally found to be the most effective form of instruction. We have found students working with private human tutors to learn material up to four times as quickly as those in the typical classroom situation (i.e., attending lectures, reading texts, and working alone on homework problems). Similarly, Bloom (1984) found students working with private tutors attained a better grasp of the material than a comparable group of students spending the same amount of time in the classroom. We have developed a number of pedagogical principles that explain the effectiveness of private tutoring (Anderson, et al., 1984, 1985), and have guided the design of an intelligent tutor for LISP on those principles.

Most of the learning in acquiring a cognitive skill occurs while the student actually tries to solve problems in the domain. The major role of a tutor is to make the problem-solving episodes more effective learning experiences. Our LISP tutor, called GREATERP (Goal-Restricted Environment for Tutoring and Educational Research on Programming), is a device for structuring student's problem-solving and providing appropriate feedback and guidance to enable the student to effectively learn how to program. Thus, the tutor is an environment for writing LISP programs, and is able to provide instruction in the most effective context -- while the student is trying to solving problems using the target skills.

A second pedagogical objective is that the student should do as much of the work as possible. Students learn more by doing than by being told. The tutor must be able to adapt to the amount of assistance required for the student to be able to solve the problems. Thus, the tutor must be able to monitor the student's problem-solving to determine if and when guidance is needed. Enough guidance must be provided to limit the student's floundering, and therefore enable the student to successfully solve the problem, without leading the student more than necessary.

A third objective is that the tutor should provide immediate feedback. If the tutor is able to point out errors while they are being made rather than after the entire program is written, the student can then correct those errors and avoid large amounts of time wasted in trying to isolate program bugs. Students often spend much of their learning time recovering from errors. These errors can interfere with acquiring the target skills, as students often get lost while trying to track down an error, perhaps misdiagnosing the cause of an error and changing correct parts of the program. Furthermore, students are more likely to correctly debug their knowledge upon immediate feedback, because the rules they used to commit the error are still active in memory and thus more successfully modified than when memory search is required to find the responsible rule. In order for the tutor to provide effective immediate feedback, it must constantly monitor and understand the student's behavior.

A final constraint on the design of the tutor is that it should represent the structure of the problem for the student. Often instruction communicates the final form of an answer (e.g., a program or a geometry proof) without focusing on the types of goals and subgoals generated in the problem-solving in order to produce such an answer (Anderson et al., 1984, 1985). Thus students are left to induce the procedures for obtaining such a solution with insufficient constraints, and in the early stages of learning often fall back upon generate and test strategies. Thus, it is important for the tutor to communicate the goal structure of LISP programming.

We have tried to achieve these goals in a tutor that serves as a helpful "programming environment". Students can compose programs in this environment just as if they were using a smart structured editor. However, whenever the student makes a planning error, a coding error, or asks for assistance, the tutor provides helpful information so that the student can continue. The tutor will also interrupt if necessary to curtail floundering and help the student get back on a correct path to a solution. In addition, this environment is designed to represent the conceptual structure of programming problems more accurately than typical environments (e.g., screen editors).

3. The Model-Tracing Methodology

The key to a tutor's success is the ability to fit each act of the student into a model of correct and incorrect methods for solving problems in the domain. A detailed analysis of each portion of the student's solution is necessary in order to diagnose errors and to provide appropriate guidance. We call the process of understanding the student's behavior as it is generated *model-tracing*. In this methodology, the tutor solves the problem along with a student, tracing the student's reasoning as he or she enters each part of the solution. With each input typed, the tutor tries to figure out what correct rule or misconception would have led to that input being generated. If it is a correct rule, then the tutor stays silent and waits for further input. If, on the other hand, the input is diagnosed as an error, then the tutor interrupts with advice. Thus, to the extent that the student is following a path that will lead to a correct solution, the tutor stays in the background, acting as an intelligent structured editor. Upon request, or when the tutor diagnoses that the student is in trouble, the tutor provides the next step in the solution, enabling the student to continue. In addition, if the student has difficulty writing code, the tutor will assist the student in planning out the solution, and then return the student to writing code.

In order to implement the model-tracing methodology, the tutor draws on three components:

1. *Ideal student model:* The domain knowledge necessary to solve problems.
2. *Bug Catalogue:* Knowledge about the common mistakes and poor strategies of novice programmers.
3. *Tutoring control module:* Pedagogical strategies that structure the interaction with the student.

Ideal Student Model. The tutor must be able to solve problems in the domain so that it can understand the student's behavior and assist in the problem-solving as required. However, an expert system could not adequately serve as the basis for the tutor. Experts will solve problems using more advanced heuristics, macro-rules, and other techniques not yet in the curriculum for the student. Instead, the tutor must not only be able to solve problems, but must be able to solve them as advanced students would do. The rules for reasoning in the domain that we want the student to acquire must be available to the tutor for the purposes of explanation (Clancey, 1983). Thus, the LISP tutor contains an *ideal student model*, a simulation of the programming knowledge ideal students use in solving LISP problems. This ideal model is based on a detailed theory of how students learn to program (Anderson, Farrell, & Sauers, 1984). The ideal model for LISP programming is implemented in GRAPES, a Goal-Restricted Production system (Sauers & Farrell, 1982). The GRAPES architecture is particularly well suited for modelling the type of goal decomposition found in solving programming problems. Each production in the ideal model contains a specification of the goal the rule will achieve and a description of the conditions under which the rule is applicable. Table 1 shows the production rule that applies to code the function *append* in order to concatenate two lists.

Production Rule in Ideal Model:

```
IF   the goal is to combine LIST1 and LIST2
       into a single list
     and LIST1 is a list
     and LIST2 is a list
THEN use the function APPEND
     and set subgoals to code LIST1 and LIST2
```

A Related Buggy Rule:

```
IF   the goal is to combine LIST1 and LIST2
       into a single list
     and LIST1 is a list
     and LIST2 is a list
THEN use the function LIST
     and set subgoals to code LIST1 and LIST2
```

Tutor's Response to the Bug:

You should combine the first list and the second list, but LIST is not the right function. If you LIST together (a b c) and (x y z), for example, you will get ((a b c) (x y z)) instead of (a b c x y z). LIST just wraps parens around its arguments.

Table 1. A correct and buggy production rule in the tutor's model.

The ideal model contains both planning and coding production rules. The planning rules design an algorithm to achieve a particular program specification, and the coding productions then write the code to achieve the algorithm. In many cases, coding productions exist that map directly from the program specification to the code, bypassing the separate planning step. These more complex productions are necessary to handle cases where a more competent or advanced student does not require a separate planning phase, but can go directly to the code. Thus, there must be enough redundancy in the ideal model to follow the many different paths through a problem that students of different backgrounds and abilities will require.

A problem is specified to the tutor by setting a goal to code a function that computes a particular operation on one or more objects, and by specifying a set of facts in working memory that describe the relationships between the conceptual objects in the problem. When the ideal model is given a problem to code a LISP function, it applies a large sequence of production rules to plan and then write the LISP code.

Bug catalogue. Associated with the rules in the ideal model there is also a large set of *buggy* rules which represent misconceptions novice programmers often develop during learning. Buggy rules are incorrect variations of correct rules in the ideal model (Brown & Burton, 1978; Sleeman, 1982). A buggy rule may represent a poor strategy, semantic confusions between the basic LISP functions, misunderstandings about manipulating objects such as variables, erroneous use of syntactic constructs (e.g., missing quotes, misgrouped parentheses), or other common slips. A buggy version of the *append* rule is shown in Table 1.

In order to diagnose errors, the tutor compares the student input against the correct rules the ideal model is considering and the associated buggy rules relevant to the current state in the problem solution. By dynamically modelling the student's path through a problem, the tutor always has a model of the student's intentions. Inferring intentions is necessary for responding appropriately to student's misconceptions about programming (Johnson & Soloway, 1984).

Tutorial rules. A tutorial rule is associated with each production rule in the ideal model and with each rule in the bug catalogue. The tutorial rule is the bridge between the internal representation of the tutor and what the student inputs and sees on the screen. First, each tutorial rule contains one or more patterns that enable the tutor to recognize if the student is executing the associated production rule. For example, the patterns for the correct rule and buggy rule in Table 1 would be "(append" and "(list", respectively. In addition, the tutorial rule specifies how to explain the associated production rule to the student. Tutorial rules associated with correct productions describe why the rule is applicable and what code should be written. Those rules associated with buggy productions describe why the student's answer is wrong, and provide a hint toward the correct solution. An explanation constructed by the tutor is shown with the buggy rule in Table 1. The descriptions are constructed by instantiating english templates with the english descriptions for the various objects in the current problem, and with examples associated with those objects.

Tutoring Control Structure. This module contains the pedagogical strategy of the tutor. It determines when to curtail the student's floundering and interrupt with the next step in the solution, when to invoke a planning mode, and selects remedial problems tailored to the particular student's weaknesses.

4. Feedback and Guidance in the LISP Tutor

The tutor is designed to provide only as much guidance as necessary while encouraging the student to generate as much of the solution as possible. Thus, the tutor generally tries to provide hints rather than actual solutions. There are several types of guidance provided by the tutor. These involve responding to errors, providing hints and reminders for clarification, and helping the student plan a solution before coding.

The bug catalogue enables the tutor to respond effectively to student errors. As soon as the student makes a mistake, the tutor responds with an appropriate diagnostic message. Because students can write their code a small piece at a time with the tutor, the feedback appears as soon as one item of the code is wrong. This is in contrast to the standard learning situation where a student receives feedback only after the entire function has been coded (or perhaps even an entire set of functions), and then tries to run the code. The tutor also must respond to "undiagnosed" errors. These are student answers that fail to match either a correct rule or one of the buggy rules. Although the tutor can say nothing specific about why their code will not work, the tutor responds that it "doesn't understand that answer", and then describes the current goal in the problem solution. Often this reminder clarifies the problem for the student, who is then able to enter the correct code. Because the tutor has all the knowledge the student is expected to have at that point in the course, it is very rare that the student enters code that would actually work but is not recognized as correct by the tutor.

The tutor also provides guidance by hinting toward the correct solution if the student is having difficultly. These hints take the form of queries and reminders about the current goals. The student can also request a clarification of the current goal via a special *Clarify* key. If necessary, the tutor can provide the next small piece of the code so that the student can continue. This is done at the student's request via a special *Explain* key. Such a request causes the tutor to query the ideal model for the best production rule it is currently considering. At that point the tutorial rule associated with the production is accessed to provide an explanation, and the production is executed, updating the code or the current plan.

The tutor will also intervene and cut off the student's attempts at coding when they are no longer fruitful, i.e., when the student has made more than the maximum number of allowed errors for that portion of code. Typically, the student is allowed to continue making errors as long as the errors are correctly diagnosed, on the theory that the error diagnosis and feedback can provide useful discriminations for the student. However, the student is limited to two errors that fail to match bugs in the catalogue. Errors such as entering drastically inappropriate

functions or trying to code the wrong part of the problem indicate the student is confused enough that further attempts at coding that portion would not succeed. If the current portion of code is sufficiently complex, the tutor will initiate a "planning mode" to work out an algorithm to code the problem. That is, the tutor will work through the algorithm with the student, step by step, using an example. Then, after the algorithm is constructed, the student can return to coding, presumably with a better idea of what he or she should be doing in order to get their code to work properly. If the current part of the problem is more straightforward, the tutor provides the next step, setting the student back on one of the correct solution paths. By providing the next portion of code, the tutor enables the student to work through the rest of the problem in cases where the student might otherwise have had to give up. As a consequence, students can tackle more and more difficult problems.

One consequence of this immediate feedback is that students receive less practice debugging their code. In fact, their only debugging is as they are writing the code -- i.e., in trying to generate another portion of code after the tutor has diagnosed an error. However, we do not view this as a limitation. In fact, in the normal learning situation, learning to program necessarily confounds the skills of code generation, code evaluation, and debugging. To the extent possible, this tutor enables students to learn to generate code without the complications of having to simultaneously learn debugging skills. Instead, separate lessons on debugging and evaluation can be included to train those skills independently.

5. The Tutorial Interface

The tutorial interface is designed to facilitate student's learning by providing the environment with the intelligence to structure the code being entered and prevent "low-level" syntactic difficulties. This is achieved by providing the student with an intelligent structured editor with which to enter code. The structured editor automatically balances parentheses and provides placeholders for the arguments of each function. The placeholders are provided by a template associated with each coding rule. For example, consider what happens when the student tries to define a new function. To begin, the student types a left parenthesis and the word *defun*. At that point the tutor recognizes the correct application of the *defun* rule, and redisplays the code as

```
(defun <NAME> <PARAMETERS>
    <PROCESS>
    )
```

The symbols in brackets indicate arguments that must be coded, in this case referring to the name of the function, a parameter list, and the function body. The tutor places the cursor beneath the symbol "<NAME>" and illuminates it to indicate that this symbol must be coded next.

This structured editor relieves students of the burden of balancing parentheses. Furthermore, the editor traps illegal characters and stops students from committing simple syntactic errors, such as forgetting parentheses or quoting function calls. For example, when a function call is required, typing any character other than a left parenthesis will produce a beep. Typing a single quote would produce the message "Function calls should not be quoted", while any other character would evoke the message "You should be typing a function call", followed by "Function calls begin with a left parenthesis" if the error is repeated. This type of response is quickly understood by students with a minimal amount of distraction from what the student was intending to do. Thus, the editor enables students to focus on those aspects of LISP that are conceptually more difficult. Our results demonstrate that enabling students to pay more attention to the central conceptual issues in programming leads to faster learning of these major skills, yet with no deficit in the student's knowledge of syntax.

The structured editor also facilitates communication between the student and the tutor. The student types directly into the code, replacing one of the placeholding symbols, and thus it is always clear what part of the problem is being coded. In the question/answer format of most eductional software, the tutor and student can easily get "out of synch" on complex problems, where the student is not sure what part of the problem the tutor is discussing or querying.

A simple windowing system is used to keep information current on the screen. The Code Window always displays the code written at the current point in the problem. A separate Tutor Message Window is used to display messages from the tutor such as hints or diagnostic error messages. Thus, the student can read these messages while retaining access to the code, including the last (possibly incorrect) student input. A third Goals window reminds the student about the current goal in the problem-solving.

Figure 1 demonstrates the tutor responding to a student error. In this case, the student is writing a function to create a list of numbers from 1 to n. Here the student has forgotten to use the function *return* in order to return a value from the iterative function *prog*. The error message shown the in the Tutor Window (the top window) appears as soon as the student finished typing the atom *result*. Figure 2 demonstrates the planning capabilities of the tutor. Upon having difficulty in coding a recursive function to compute the factorial of a number, the tutor helps the student plan the code with the use of concrete examples.

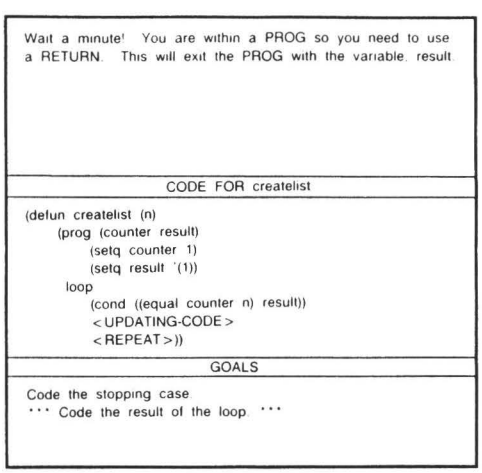

Figure 1. The tutor's response to a student error.

```
In examples A and B what do you have to do to get the result
of fact called with n?
PRESS          IF YOU WANT TO
1      Multiply n by one less than n
2      Multiply n by fact of one less than n.
3      Add n to the result of fact called with one less than n.
4      Have the tutor choose
Menu Choice 2
                    CODE FOR fact
(defun fact (n)
    (cond ((zerop n) t)
          <RECURSIVE-CASE>))
                    EXAMPLES
      fact (n)        fact (n-1)
A   (fact 1) = 1     (fact 0) = 1
B   (fact 3) = 6     (fact 2) = 2
```

Figure 2. The tutor aiding the student in algorithm design for a recursive function.

6. The Goal Structure of LISP Programming

The tutor has been designed to communicate the conceptual structure of programming problems. This is accomplished in part by using the placeholders to provide a template for the rest of the problem solution. For example, when the student types the iterative construct *prog*, the tutor displays the general pattern for iteration:

```
(prog <LOCAL-VARIABLES>
   <INITIALIZATIONS>
   <BODY>
   <REPEAT>
   )
```

This template helps structure the iterative problem for the student into a list of local variables, initializations of those variables, the body of the loop (i.e., the repeated actions), and the call to return to beginning of the loop. In accordance with the top-down nature of the task, many symbols are themselves expanded into more detailed symbols. For example, the iterative <BODY> includes a <TERMINATING-CASE> and <UPDATING-CODE>.

The tutor also communicates the goal structure in its guidance for planning LISP programs. When requested or when the student encounters sufficient difficulty, the tutor initiates a planning mode, where it leads the student through the design of an algorithm to accomplish the current portion of the problem. Thus, the student learns how a complex problem can be broken down into simpler problems to be solved. In both coding and planning modes, special Goals windows remind the student about the current goal in the problem solution.

7. Dynamic versus Post-Hoc Student Modelling

One of the central pedagogical objectives in the tutor's design is the principle of immediate feedback. Because of this principle, we have constructed the tutor so that each symbol of the student's input is processed and interpreted immediately, rather than waiting until the student completes some portion of the code, such as a function call, or perhaps the entire function definition. The goal is to respond to the smallest unit of input that can disambiguate what the student is intending to do. Typically, the grain size for input in the tutor is a single LISP atom. Except for syntactic errors such as missing parentheses, the tutor will respond as soon as the student finishes typing an atom such as the word *append*, *list*, *side1*, etc. If the tutor responded upon individual keystrokes, perhaps diagnosing as soon as the input did not fit with one of the correct rules, there would typically not be enough of an answer to enable the categorization of the error. Furthermore, it would not enable students to correct a typing mistake by deleting the incorrect letters. On the other hand, a larger unit of input would inhibit the immediate feedback of the tutor, and increase the chance that the student would become lost. Thus, the tutor greatly limits the consequences of the student's errors. Students can learn from their errors, without the danger of spending an unproductive amount of time trying to track them down, perhaps even changing correct parts of the program, and finally loosing track of what they were trying to accomplish.

The grain size of student input also has important ramifications for student modelling. By trying to interpret each part of the student's answer as it is typed, the tutor has access to the current state of the ideal model. Thus, the tutor knows the exact state of the problem solution, and the current goal, and is in a better position to diagnosis the student's error. A limitation with the successful PROUST debugging aid (Johnson & Soloway, 1984) that analyzes a program after it has been fully written is that PROUST often finds many alternative interpretations for a particular line of code due to the many possible buggy transformations of the various paths through a problem it is forced to consider. On the other hand, the LISP tutor tracks student's problem-solving as it occurs, and thus can construct a more accurate model of the student's reasoning, because the ambiguous portion of code never is completed. Instead, the LISP tutor diagnoses an error at the first sign, so an entire line of code could never be omitted or misplaced. Many of the complex bug interactions that present such a problem for PROUST's analysis are thus avoided.

8. Generic and Individualized Student Models

The tutor is able to use its ideal model and bug catalogue to respond to the student's behavior during the problem-solving. The set of productions form a *generic student model*, a model of the set of target knowledge and possible misconceptions. Thus, the generic student model contains rules which a particular student may know less well than others, and rules which the student may not posses at all. The generic student model is adequate for the type of immediate feedback and minimal guidance desired in this tutor. The responses and explanations need only to be tailored to the particular problem context and student's answer, but not to the strengths and weaknesses of a particular student.

An individual student model is also kept by the tutor. This is in the form of an overlay of the generic model (Goldstein, 1982). Each production in the ideal model contains a weighting, which is the tutor's measure of how well the student knows that rule. Each time the student performs that rule correctly, the weight is increased, and each time the student exhibits an error concerning that rule, it is decremented. Currently, these weights are used to assign remedial problems to the student. Each remedial problem is considered to see whether it involves production rules on which the student is weak. It may also be possible to use these weights in constructing explanations for errors. For example, an error on a well-learned production is likely to be a careless slip rather than evidence for a serious misconception. It is a question for future research as to how one might want to differentially respond to such an error.

9. Field Tests of the Tutor

The current version of the tutor consists of ten lessons, beginning with the basic functions of LISP, and including the topics Function Definitions, Predicates and Conditionals, Structured Programming, Numeric and List Iteration, and Numeric and List Recursion. The tutor contains 375 production rules in the ideal model and 475 buggy versions of those rules. These rules enable the tutor to diagnose and respond appropriately to between 45% and 80% of the student's errors, depending on the complexity of the lesson and the amount of testing of the lesson we have conducted. That is, our well-worked lessons correctly diagnose 80% of the students' errors, while our newest implemented lessons diagnose only about 45%.

We have completed two major evaluations of the tutor. A first evaluation compared students working with the tutor to a group working with a private human tutor, and to a group working essentially on their own (but with the help of a teaching assistant when necessary). These students were University of Pittsburgh and Carnegie-Mellon undergraduates with no prior programming experience. The material covered six lessons, from basic LISP to recursion. All three groups scored approximately the same on final performance tests, but both tutoring groups learned the material much more quickly than the untutored group. Students learned almost twice as quickly with the computer tutor (15 hours for six lessons) as without a tutor (27 hours), and nearly as quickly as those students with human tutors (12 hours). Furthermore, a greater percentage of students tutored by either human or computer successfully completed the lessons in the time allotted.

In a second evaluation, we again compared students working with the tutor to another group working on their own. This evaluation was a six week mini-course offered to Carnegie-Mellon University undergraduates during the fall 1984 semester. The no-tutor group corresponds to the standard pedagogical situation where students attend lectures, read a text, and do homework unassisted. Students had one previous programming course in Pascal. All students went to the same lectures and read the same text. Here students working with the tutor performed 47% better on the final exam, and learned the material 30% faster than those working without the tutor.

These results demonstrate that the LISP tutor appears to achieve its educational objectives. It is more effective than the standard pedagogical situation. Students learn more quickly and perform better on achievement tests. The benefits of the tutor are greater for the more difficult lessons, and are somewhat greater for less experienced students.

The tutor is currently used by many students learning their first programming language at CMU and to fulfill the university's programming course requirement. This self-paced LISP course for humanities students consists of a 2 hours/week lecture and question and answer sessions, with the majority of the coursework involving students interacting with the LISP tutor to write programs.

10. Current Directions

Our current work is focused on extending the current tutor to teach the skills of debugging and program comprehension. In the evaluation lessons, students will go through the code for a function, guided and monitored by the tutor, specifying the flow of control and the results of the function calls. In the debugging lessons, students will be asked to run functions, determine whether a bug exists, locate the bug, and then correct the code. In these skills, as in code generation, the tutor monitors the student's performance by comparing it with the ideal model, providing feedback upon errors and guidance when necessary. The principles used in the LISP tutor are also currently being explored as the basis for tutoring systems for other problem-solving domains such as algebra and geometry (Boyle & Anderson, 1984).

References

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA: Harvard University Press.

Anderson, J. R., Boyle, C. F., & Reiser, B. J. (1985). Intelligent tutoring systems. *Science*, **228**, 456-462.

Anderson, J. R., Boyle, C. F., Farrell, R. G., & Reiser, B. J. (1984). Cognitive principles in the design of computer tutors. *Proceedings of the Sixth Annual Conference of the Cognitive Science Society*, Boulder, CO.

Anderson, J. R., Farrell, R., & Sauers, R. (1984). Learning to program in LISP. *Cognitive Science*, **8**, 87-129.

Bloom, B.S. (1984). The 2 Sigma Problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, **13**, 3-16.

Boyle, C. F. & Anderson, J. R. (1984). Acquisition and automated instruction of geometry proof skills. Paper presented at the Annual Meeting of the American Educational Research Association, New Orleans.

Brown, J. S. and Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, **2**, 155-192.

Clancey, W. J. (1983). The epistemology of a rule-based expert system -- A framework for explanation. *Artificial Intelligence*, **20**, 215-251.

Goldstein, I. P. (1982). The genetic graph: A reprensentation for the evolution of procedural knowledge. In D. Sleeman & J. S. Brown (Eds.), *Intelligent tutoring systems*. New York: Academic Press.

Johnson, W. L. & Soloway, E. (1984). Intention-based diagnosis of programming errors. *Proceedings of the National Conference on Artificial Intelligence*, Austin, TX.

Sauers, R., & Farrell, R. (1982). *GRAPES user's manual*. ONR Technical Report ONR-82-3, Carnegie-Mellon University.

Sleeman, D. (1982). Assessing aspects of competence in basic algebra. In D. Sleeman and J. S. Brown (Eds.), *Intelligent tutoring systems*. New York: Academic Press.

Acknowledgements. This research is supported by Office of Naval Research under Contract No. N00014-84-0064. We would like to acknowledge the considerable contributions of Albert Corbett, Elliot Jaffe, Beth Marvel, Peter Pirolli, and Ross Thompson to the LISP tutor. Brian Reiser is now at Department of Psychology, Princeton University. Robert Farrell is now at the Department of Computer Science, Yale University.