PROBLEM COMPILATION AND TUTORING FLEXIBILITY IN THE LISP TUTOR

Albert T. Corbett, John R. Anderson, Eric J. Patterson

Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA 15213

Abstract. The Lisp Intelligent Tutoring System provides assistance to students as they do Lisp coding exercises. The tutor monitors the student's performance and provides feedback when the student moves off a known solution path. The tutor employs a model tracing paradigm to do this, in which the coding task is modelled in the background and used to evaluate the student's behavior. In this paper we discuss the relation between the underlying student model and two of the tutor's characteristics: (1) students are constrained to type code top-down and left-to-right and (2) the tutor provides immediate feedback on a symbol-by-symbol basis. We then describe a new version of the tutor in which students can enter code in any order and control when they get feedback. Finally, results of an initial study are reported comparing the standard version and the new version of the tutor.

The Tutor: An Overview

The Lisp Intelligent Tutoring System is a program that provides assistance to students as they work on Lisp coding exercises (Anderson & Reiser, 1985). The program presents problem descriptions and as the students type answers, the tutor monitors the solutions and stands ready to provide assistance at any point. The tutor has been in use in an introductory Lisp course at Carnegie-Mellon University each term since the fall of 1984. While the lesson material has been extended over the intervening three years[1] the basic architecture of the tutor and the nature of the tutorial interaction have remained essentially unchanged. Thus, the Lisp tutor represents a relatively large and stable intelligent tutoring sytem.

The tutor was initially developed for two purposes, (1) to teach Lisp and (2) to collect data concerning the acquisition of a formal skill in a "real-life" setting, and has proved successful in both endeavors. Two evaluation studies, for example, have indicated that working with the tutor is more effective than doing the same exercises "on your own" in a Lisp environment. In both studies, students covered the exercises more quickly with the tutor and in one of the two studies performed substantially better on a final test (Anderson, Boyle & Reiser, 1985; Anderson & Reiser, 1985). In addition to this evaluation data, the tutor provides data on the course of skill acquisition. As students interact with the tutor, it generates log files containing a time-stamped record of the students' overt responses and the abstract coding rule that governs each response. Initial analyses of these files reveal a large speed-up in application from the first to second use of a coding rule and a subsequent more gradual speed-up. These data are consistent with Anderson's ACT* model of skill acquisition, which suggests that knowledge becomes proceduralized with inital usage and that the procedural knowledge becomes strengthened with use (Anderson, 1987; in press). Finally, the tutor can be used to study parameters of the tutorial process and research is currently underway that manipulates degree of practice and the content of feedback.

While the tutor has reached a stable configuration and is being used productively both in teaching and research, there are several reasons why we would like to create substantially different tutorial interactions that would require significant modifications in various components of the tutor's architecture. First, the theory underlying the tutor has undergone revision since the tutor was first developed. Second, while the tutor is more effective than "learning on your own," it is not as effective as a human tutor (Anderson & Reiser, 1985; Bloom, 1984). Third, feedback from students indicates that some believe they would be happier with some modifications in the tutorial interaction.

We have begun to tackle the task of implementing tutorial changes and in this paper would like to address issues involved in such a task. In particular, we would like to discuss the impact of one component of the tutor, the student model, on the tutor's behavior and implications for modifying the tutor's behavior. We will begin with a brief description of the tutor's current behavior and then will discuss both the role of the student model in governing that behavior and the implications for modifying the tutor. Finally, we will describe some data from one such modification: a version of the tutor in which the student and not the tutor controls when feedback is given.

Model Tracing and the Tutor's Interface

The tutor employs a *model tracing* paradigm to provide assistance to students. In this approach the tutor attempts to model the steps that a student might take in solving a problem and uses the information to evaluate the students' responses. Thus, while the student is working, the tutor in lock-step simulates the steps that a knowledgeable student could take in writing the code. In

[1] The tutor currently includes approximately 240 exercises which cover the first twelve chapters of an introductory Lisp text (Anderson, Corbett & Reiser, 1987).

addition, it models errors that students make at each step on the basis of known misconceptions. By comparing the students' response to the set of possible legal actions and the set of known erroneous actions, the tutor is able to recognize whether the student is on a correct solution path, appears to be suffering from a known misconception, or has typed something unrecognizable.

Figure 1 provides some "snapshots" of what it is like to work with the tutor. Figure 1a depicts the terminal screen shortly after the student has begun working on an exercise in which a function called ***ends*** is to be defined. At the beginning of each exercise, the problem description appears in the *tutor window* at the top of the screen, while the *code window* at the bottom of the screen is blank. In this figure the student has already begun by typing ***(defun***. Once the student has typed a delimiter, in this case a space, the tutor recognizes the response is correct and creates a template on the screen for a call to the operator ***defun***. As shown in Figure 1b, the tutor fills in a right parenthesis that balances the left parenthesis and puts three subgoal symbols on the screen that represent arguments which must be expanded (replaced with code). The student is constrained to write the code in a left-to-right and top-down fashion, so the tutor immediately highlights the next symbol which must be expanded and the student continues typing.

The tutor continues to monitor the student's responses, essentially on a symbol-by-symbol basis. As long as each symbol lies on a known solution path, the tutor continues highlighting nodes and creating function templates where appropriate, and the student continues along without interruption. However, if the student types a symbol that does not fall on a known solution path, the tutor interrupts with feedback. For example, Figure 1c depicts the screen after the student has typed the function name, ***ends***, and the parameter list and has begun working on the body of the function. This figure shows the reaction of the tutor when the student makes a mistake, in typing ***car***. This error suggests that the student has not decomposed the task correctly and the tutor attempts to describe what the current goal should be.

As long as the student makes errors that the tutor can recognize, it will let the student continue trying to expand a goal symbol. However, if the student repeatedly makes errors that the tutor cannot recognize or if the student repeatedly makes the same type of error, the tutor will tell the student what code would work in that step, explain why and fill in the code for the student. The student also has the option at any point of asking the tutor for two types of information. First, the student can ask the tutor to provide a hint about the current goal. In this case the tutor provides a description of what needs to be done, but not how to do it. Second, the student can ask the tutor what to do next, in which case the tutor will tell the student what the next step is, explain why and fill in the code for the student.

After completing each exercise, the student enters a Lisp environment, in which he or she tries the code that was generated and is also free to explore. Then when ready, the student proceeds to the next exercise.

One important feature of this tutorial interaction is that it exemplifies the principle of immediate feedback. As soon as a unit of code is typed, the student knows if it is correct or not. This immediate feedback principle is logically independent of the formulation of the student model, but in practice the realization of the principle is strongly influenced by the specifics of the student model. The relationship between immediate feedback and the

```
Define a function called ends that takes one argument,
which must be a list, and returns a new list containing
the first and last items in the argument. For example:

    (ends '(a b c d)) = (a d)

                    CODE for ends
(defun
                                              [Fig. 1a]
```

```
Define a function called ends that takes one argument,
which must be a list, and returns a new list containing
the first and last items in the argument. For example:

    (ends '(a b c d)) = (a d)

                    CODE for ends
(defun <function> <parameters>
    <process>)
                                              [Fig. 1b]
```

```
You will need to call the function CAR, but not yet
You need to construct a list containing the first item
in the argument and the last item in the argument so
you need to call a list combining function here

                    CODE for ends
(defun ends (lis)
    (car )
                                              [Fig. 1c]
```

Figure 1. Three "snapshots" of the terminal screen as a student codes the function *ends* with the tutor.

student model is an issue we will focus on below. (For a more general discussion of the pros and cons of immediate feedback, see Anderson, Boyle, Corbett & Lewis, in press).

The Student Model

The student model that underlies the tutor is derived from observations of students learning Lisp (Anderson, Farrell & Sauers, 1984; Pirolli & Anderson, 1985) and is partly descriptive and partly prescriptive. In this model, procedural knowledge of how to write Lisp code is modelled by a set of productions. Each production is essentially an IF-THEN rule. An English translation of a typical production rule that students learn in the first lesson would be:

> **IF the goal is to form a list**
> **by inserting** *newitem*
> **at the beginning of**
> **an existing list** *oldlist*
> **THEN code a function call to CONS**
> **and set subgoals to code**
> *newitem* **and** *oldlist***.**

The complete set of correct rules for writing code is referred to as the *ideal student model* and represents the instructional objectives of the text and tutor. The student model also includes a *bug catalog* - a set of incorrect rules that reflect known misconceptions.[2] In actually modelling student behavior, the production system is given a specification of the function to be written (analogous to the English description provided the student) and a goal is set to code the function. The set of production rules is matched to the current goal and problem description and ultimately a rule is selected that satisfies the goal in the context of the problem description. When that production "fires" it generates code, possibly sets new goals (i.e., adds them to a list of goals that remain to be satisfied) and may add information to the problem description.[3]At that point, a new goal is activated (drawn from the list of unsatisfied goals) and the cycle is repeated. This process continues until the exercise is completed.

There are several general characteristics of the student model that have a direct bearing on the nature of the tutorial interaction. First, the productions generate code in a top-down fashion; that is, they generate code by decomposing high-level goals into subgoals. For example, when the system is given the goal of defining a function, a production fires that codes a call to ***defun*** and sets subgoals to code the name of the function, the parameter list for the function and the body of the function. The function name and parameter list do not require subgoaling, but when the goal to code the function body is active, a production will fire that codes a Lisp operator and sets goals to code arguments to that operator. The complete goal decomposition for the function ***ends*** is depicted in Figure 2.

Second, the student model is implemented in GRAPES a production system that embodies the state of the ACT* theory when work on the tutor began. GRAPES assumes that a goal *stack* is maintained and that a single "active" goal is popped off the stack in each cycle. The consequence of this assumption is that not only is the problem solved in a top-down fashion, but that the traversal of the goal tree in generating code is strictly depth-first. The model makes an additional simplifying assumption, generally supported by the data, that sibling goals are tackled in a left-to-right order. This set of assumptions concerning goal satisfaction gives rise to a distinctive characteristic of the tutor mentioned earlier: students are constrained to type their solutions in a top-down, left-to-right fashion.

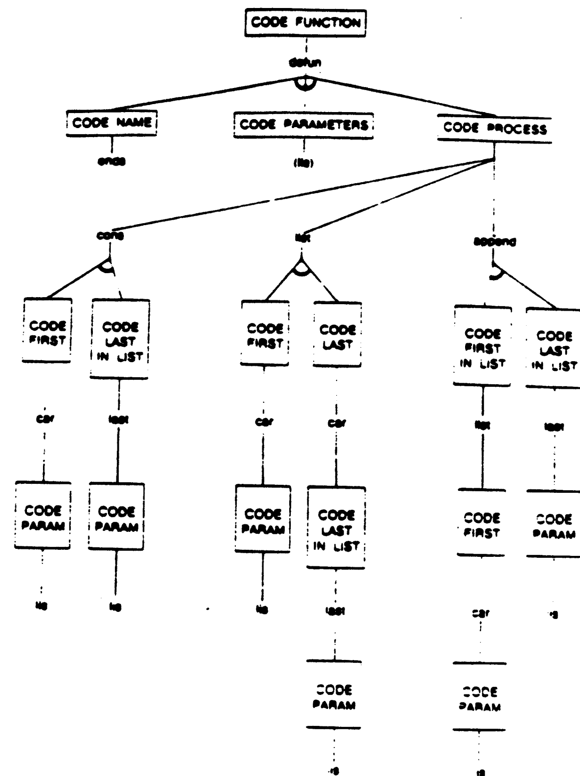

Figure 2. The goal structure of the function *ends*. Goals are represented as boxed nodes. Branches are labelled with code symbols that are generated in satisfying the goals. (Circular arcs indicate points at which multiple subgoals are created, each of which must be satisfied).

A further characteristic of the student model has a strong impact on the nature of the tutorial interaction. The student's knowledge of Lisp is represented at about the finest grain size that has functional meaning in Lisp. Roughly speaking, it models performance at the level of the individual symbol (modelling at a finer level of analysis would essentially be of typing rather than Lisp coding). This has a direct impact on the tutor's behavior, because in model tracing the immediate feedback principle actually specifies that feedback should be given after each production firing. Thus, the tutor's symbol-by-symbol feedback is a consequence of applying immediate feedback to a student model of minimum grain size.

---

[2]The papers in Part III of Sleeman and Brown (1982) describe similar approaches to student modelling that strongly influenced the development of this tutor.

[3]This is an oversimplification since not all productions generate code. Some only modify the the goal structure or the problem description. Another class of productions models a planning process for certain difficult algorithms, rather than generating code. However, these productions are not directly relevant to topic of modifying the tutorial interaction.

In summary, important aspects of the tutorial interaction depend directly on features that are built into the student model. In principle, then, modifications in the tutor's behavior require a recoding of the student model. This is an important realization because the student model, which currently consists of approximately 1200 rules for generating correct and incorrect code, represents about 75% of the code involved in the tutor. Fortunately, we can implement model tracing in a way that makes modification of the student model a less imposing task than this statistic suggests.

Implementation of Model Tracing

One difficulty with model tracing within the framework described above is that production systems have high computational costs (due to pattern matching demands) and require an unrealistically high level of computational resources to keep up with students in real time.[4] A second difficulty with model-tracing concerns the disambiguation of students' responses, since under a limited set of circumstances, a student's response may match more than one production instantiation. For example, consider the function call

(+ (car lis1) (car lis2))

Since the ordering of arguments to the function + is unimportant, the tutor will allow the student to code the two arguments in either order. Thus, when the goal is set to code the first argument, there are two viable production instantiations, each of which codes ***car***. When the student types ***car***, it is not possible to determine which argument the student is coding. This ambiguity could be resolved in the next cycle when a variable is typed. However, to postpone resolution for a cycle, it would be necessary for the production system to follow both possible branches. That entails matching the student's next response to the subgoal of each production, which increases the amount of matching required.

Problem Compilation

Both of these difficulties can be resolved when it is recognized that model tracing does not require on-line execution of the production system while the tutor is running. The tutor's ability to recognize correct solutions and standard bugs depends on the student model, since the tutor is only presented a problem description and not a solution. However, it is possible to run the production system model ahead of time as long as a trace of the that run is stored that retains whatever information is relevant to tutoring. In this way the cost of pattern matching in identifying relevant rules can be borne ahead of time. Once those rules have been identified and stored in a data structure, is easy to match the student's response to the rules and have the tutor respond accordingly. This process, referred to as *problem compilation*[5], not only enhances the efficiency of modelling, but as an added benefit, enables functional modifications in the student model at relatively low cost.

Implementing Problem Compilation

There is one substantial difference between running the production system on-line and running it ahead of time. Most of the exercises in the tutor can be solved in more than one way and some have literally hundreds of acceptable solutions. Thus a goal tree representation of the student model's potential behavior contains or-nodes (e.g., the node "CODE PROCESS" in Figure 2, which represents a goal that can be satisfied by any of three productions). When the student model is being run on-line and an or-node is encountered, it is only necessary to follow the branch selected by the student. When the model is run ahead of time, however, it is necessary to follow each branch at an or-node so that subsequently the tutor can follow the student down any branch that is selected. Thus, problem compilation requires the exhaustive representation of alternative expansions of the goal tree and the resulting data structures can become quite large.

The need to expand the goal tree exhaustively poses an additional complication in representing mutual constraints among productions. Whenever there is more than one production that satisfies a goal, the production selected will almost certainly constrain the way at least one other goal in the problem is satisfied. This can be seen in the body of the function ***ends***. Three different functions can be employed to construct the required list of two elements. Not surprisingly, the code for the two elements is different in each case:

(cons (car lis) (last lis))

(list (car lis) (car (last lis)))

(append (list (car lis)) (last lis))

In this example the components of the code that co-vary are hierarchically organized. That is, the production which satisfies the top-level goal in these expressions determines the correct response at the subgoals. It is easy to represent such hierarchically organized constraints in a goal tree. (Each branch at a choice point only represents legitimate actions at the subordinate goals). However, it is sometimes the case that mutually dependent code is not hierarchically organized. In iteration, for example, variable initializations and loop actions are not hierarchically ordered (at least in the tutor's student model) but the initial values assigned to variables interact with the order in which loop actions are performed and the nature of the variable updates. When a goal tree containing such constraints is exhaustively expanded by the student model it is essential that some convention be adopted for marking the constraints.

One solution to this problems, adopted by Anderson and Ross Thompson, in compiling problems for the tutor, is to represent the student model traces not as a goal tree, but as a depth-first expansion of the goal tree. An example of such a representation is presented in Figure 3. The effect of this transformation is that temporal relations are represented hierarchically. If goal b follows goal a temporally, then goal b is structurally a descendent of goal a. The advantage of this is that coordinate goals in the basic goal tree become hierarchically arranged and any mutual constraints among goals can be easily represented. The disadvantage of this solution is that identical substructures are represented redundantly on various branches in the tree. Even when branches are allowed to converge whenever possible, an exhaustive depth-first expansion of an and-or goal tree is larger than

[4]An independent attempt by Anderson and Joe Parker is underway to reduce computational demands by applying a discrimination-net approach to rule-matching, that should substantially reduce the computational demands of modelling programming skills.

[5]A similar process is described in Sleeman, 1983

the corresponding and-or goal tree itself.[6]

Problem Compilation and Tutoring Flexibility

There are at least two advantageous side effects of problem compilation. First, it becomes relatively inexpensive to search multiple steps down alternative branches in the goal tree when necessary to process ambiguous responses. Second, it becomes relatively inexpensive to modify the behavior of the tutor by effectively modifying the student model. The production system is no longer running on-line as the tutor is at work. Instead a relatively simple interpreter exists which accepts the trace structure as input and simulates the running model. As a result, it is possible to simulate a new model without changing the production system by writing an interpreter that accepts the same data structure, but behaves differently.

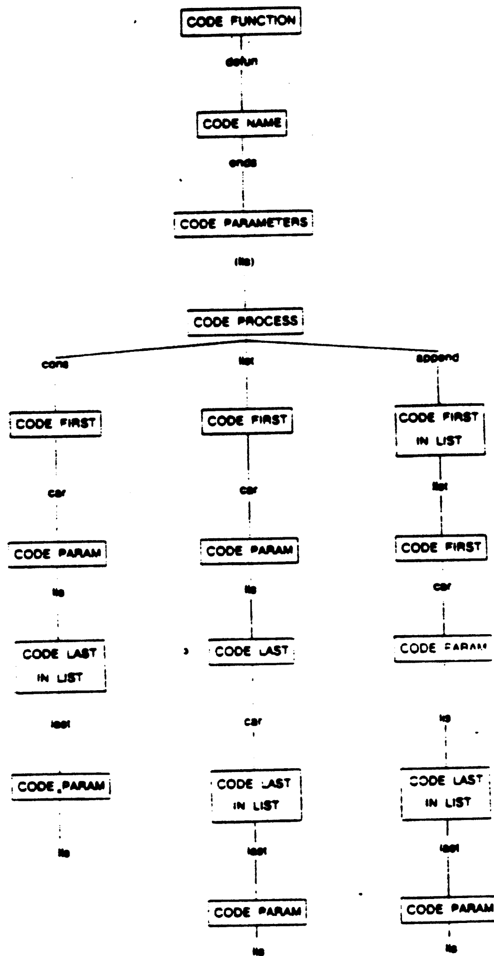

Figure 3. A depth-first transformation of the goal structure for the function *ends*.

Modifying the Tutorial Interaction: Student-controlled Feedback

In our initial research in varying the nature of the tutorial interaction we have employed problem compilation to implement a tutor that gives the student more control over the coding process in two ways. First, we have relaxed the constraint on input-order, so that the students can generate code in any order they wish. Second, in the new tutor, students have control over when feedback is presented.

The transition from tutor-controlled to student-controlled feedback is a fairly small one. Instead of feeding each unit of code to the tutorial engine as it is generated, the code can be buffered and submitted to the tutor at the student's request. Problem compilation is important in converting to student-controlled feedback largely for the purpose of resolving ambiguity. As described earlier, there are some situations in which a student's response may match more than one correct step that the tutor is prepared to take. Further ambiguity can arise in the student-controlled tutor since students can deviate from left-to-right input order and as a result, may have unexpanded goal symbols in the middle of their code when they ask the tutor for help. Problem compilation makes it convenient to resolve both types of ambiguity by looking ahead through the rest of the student's code.

We have begun implementing this "student-controlled" tutor and we have collected data with it for the first two lessons of the tutor's curriculum. In this tutor the student is provided with a true structured editor. As in the case of the standard tutor interface, the structured editor provides templates for the student[7] and ensures that the code is syntactically correct. The student can only expand goal symbols that exist on the screen, but the student is able to generate arbitrary goal symbols (as long as they are structurally legal) and so ultimately has complete control over the order in which the code is generated. In addition, unlike the standard interface the student may go back and modify code that has been completed.

In this tutor, the student can request three types of help. As in the standard tutor, the student can ask for a hint at any unexpanded goal, and the student can ask for an explanation at any unexpanded goal. A third option is provided the student in this version of the tutor, however. At any time the student can ask the tutor to check over all the code that has been written so far. At that point the tutor checks over the code in the same top-down, left-to-right sequence that it ordinarily would. The tutor ignores any unexpanded template symbols it encounters by skipping over the corresponding goals in the node tree. If no errors are found, it tells the student that everything is fine so far. If no errors are found and the code is

[6]Alternative solutions to the space demands of representing trace structures are being pursued in other tutoring projects. In the case of a geometry tutor (Anderson, Boyle & Reiser, 1985) the trace structures and tutor programs have been ported from Lisp to C by Anderson and Ray Pelletier. Every attempt was made to develop an efficient representation and when the tutor is running, the code for the program and exercises requires less than .5 megabytes. At the same time Ed Skwarecki is tackling the problem of representing trace structures as and-or trees and dealing with complications that arise.

[7]The templates are very similar but not uniformly identical to the tutor's templates for several reasons. For example, many functions such as +, ***list*** and ***equal*** can take a variable number of arguments and the editor has no information on what the student intends. Thus, while the tutor generates templates with the correct number of argument nodes for the solution, the editor simply generates one argument node when the function is first called and generates a new argument node each time an earlier one is expanded until the student fianlly deletes the last empty argument node.

complete, the tutor advances the student to the Lisp window just as in the standard configuration. If an error is detected, however, the tutor gives the same feedback as it would in the standard condition and removes the erroneous code from the screen. The tutor does not check any farther and any code that is down or to the right is popped out of the solution and into a separate buffer (since leaving it in place might suggest to the student that it is correct and in the proper position).[8]

Testing Student-controlled Feedback

Thirty-four subjects took part in the first study of student-controlled tutoring. Half the students used the standard immediate-feedback tutor, while half used the new student-controlled-feedback tutor. Students in both conditions completed the first two lessons in the tutor curriculum and then took a cumulative quiz. One student dropped out in the immediate-feedback condition and one student in the student-controlled condition failed to complete the two lessons in the allotted time, leaving sixteen subjects in each group.

Evaluation Measures

Two measures of tutor effectiveness are of interest: performance on the final quiz and time to complete the lessons. There was no difference between the two groups on the quiz: the mean score for both groups was 83% correct. However, there was a reliable difference in time to complete the exercises: Subjects in the immediate-feedback condition required an average of 5.7 minutes to complete each exercise, while subjects in the student-controlled condition required 8.6 minutes, $t(30)=3.9$, $p < 0.001$. Part of this time difference may reflect the fact that the subjects in the student-controlled condition were working with a true structured editor which is necessarily more complicated than the constrained interface in the standard version of the tutor. However, as described in the next section, students are doing additional processing in the student-controlled condition (in catching their own errors) and part of the time difference may reflect that extra processing.

Processing Measures

The log files in the student-controlled condition can be used to address three issues concerning interface design in programming tutors: (1) when do students request feedback, (2) to what extent do students deviate from top-down, left-to-right coding, and (3) to what extent do students catch their own errors when immediate feedback is suspended?

In answer to the first question, subjects in the student-controlled condition showed an overwhelming inclination to complete their code before requesting feedback. Students asked the tutor to "check over the code" a total of 661 times across the exercises in both lessons and in 646 of these cases their code was complete, though not necessarily correct. Students also requested a goal-hint 33 times and a goal-explanation 39 times and these requests also require the tutor to give feedback on partial code. Even when these goal-specific requests are included, however, the proportion of tutoring requests that involved partial code is still relatively small (12%). This suggests that students could be happy with a tutor that does not provide feedback on partial solutions but only on complete code, as for example in the case of Proust (Johnson & Soloway, 1985).

Examination of those instances in which students request tutoring on partial code also provides indirect evidence on the issue of top-down, left-to-right coding. In no case did any of these partial solutions show evidence of right-to-left or bottom-up coding. To obtain direct evidence on this issue, however, it is not sufficient to simply examine the state of the code when a student asks the tutor for assistance. Rather, it is necessary to trace through the students' complete interaction with the editor, which we did for the second lesson. Across the 16 subjects using the student-controlled tutor and the seven exercises in lesson two, there were about 400 goals which required subgoals and hence could be satisfied in a bottom-up rather than top-down fashion. In addition, there were about 450 opportunities for the students to complete goals in a right-to-left rather than left-to-right fashion. Detailed inspection of the editor interactions revealed only five cases in which a goal was completed in a bottom-up fashion and just one case in which goals were completed in right-to-left order.

It should be noted that these results concerning tutoring requests and coding order may hinge on the relative simplicity of the exercises under study here. As functions become more complex, students may show more inclination to have the tutor confirm parts of the code before proceeding with the rest. Similarly, as functions become more complex, there may be some payoff for jumping around and filling in the parts the student is sure of before tackling the more difficult parts of the solution. Moreover, this pattern of results may be specific to the functional quality of Lisp and may not generalize to more procedural languages (or to more procedural operations, such as iteration, encountered in later Lisp lessons). However, at least for the early lessons, the top-down left-to-right interface of the standard tutor seems entirely adequate.

In answer to the final question, analyses of the log files suggest that subjects are catching and correcting their own errors in the student-controlled condition. Across both lessons, the tutor caught reliably more bugs per exercise in the immediate feedback condition, 1.15, than in the student-controlled condition, 0.83, ($t(30)=2.48$, $p < .05$). This suggests that subjects in the student-controlled condition are catching their own errors, though again, this is only indirect evidence since it is conceivable that students are being more cautious and making fewer errors in the student-controlled condition. Detailed inspection of the editor interactions in lesson two confirmed the conclusion, however; subjects in the student-controlled condition in fact corrected 23% of their own errors in that lesson.[9] Thus, there may be some benefit in deviating from symbol-by-symbol assistance in tutoring. It should be noted though, that in addition to correcting errors, students also "miscorrected" errors (changed an error to a different error) and changed correct code symbols (sometimes "discorrecting" them). Specifically, of the code changes students made in lesson two, 46% were

[8] When a student requests a hint or an explanation at a specific goal, the tutor actually checks over all the code "upstream" (up and to the left) of the goal. If an error is found upstream, the tutor provides the usual feedback on that error rather than trying to provide help on the downstream goal, which may not be part of a correct solution.

[9] The data in this section exclude errors that would not register as such in the immediate feedback tutor, e.g., errors that were corrected by deleting characters before typing a delimiter and certain syntactic errors that are caught by the interface rather than the student model.

error corrections, 33% were error miscorrections and 21% were changes to correct symbols. The latter two operations, which account for slightly more than half the changes students made, clearly add to the time it takes students to complete the exercises without yielding any obvious benefits. This result exemplifies one of the chief issues we are facing: how to balance the temporal efficiency which can be obtained with the strong tutor control built into the immediate-feedback condition, with the possible cognitive and affective benefits that can be obtained by relaxing that control.

One possible control structure is suggested by some results reported by Gray and Anderson (1987) and replicated in this study. Gray and Anderson investigated the code revisions that students made when writing fairly difficult iterative search functions in Lisp. They found that subjects are likely to go back and change code only at the currently active goal or at a direct ancestor of the currently active goal. The detailed analyses of lesson 2 revealed the same pattern for these simpler functions: 86% of the changes students made were at the currently active goal or at a superordinate goal. These results suggest that a more optimal tutor might track students' responses all the way down to leaf nodes in the goal tree but only provide feedback as the students pop back up through the tree. Such a tutor would (1) allow students editing freedom while working on incomplete subgoals, (2) check each subgoal after it is complete, providing feedback and ultimately answers where necessary, and (3) move the student forward after each subgoal is complete. A tutor with this control structure would not be expected to save much time; since students make most changes on the way down through the tree, they would still be making almost as many productive and unproductive changes as if the tutor never intervened. However, such a tutor would have the advantage of allowing students to catch whatever errors they are likely to catch, while providing feedback as soon as possible on errors that the student is not likely to correct.

References

Anderson, J.R. (1987). Production systems, learning and tutoring. In D. Klahr, P. Langley and R. Neches (Eds.) *Production System Models of Learning and Development*. MIT Press, Cambridge, MA.

Anderson, J.R. (in press). Analysis of student performance with the LISP tutor. In N. Fredericksen, R. Glaser, A. Lesgold and M. Shafto (Eds.), *Diagnostic Monitoring of Skill and Knowledge Acquisition*. Erlbaum, Hillsdale, NJ.

Anderson, J.R., Boyle, C.F., Corbett, A.T. and Lewis, M.W. (in press). Cognitive modelling and intelligent tutoring. *Artificial Intelligence*.

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

Anderson, J.R., Corbett, A.T. and Reiser, B.J. (1987). *Essential Lisp*. Addison-Wesley, Reading, MA.

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

Anderson, J.R. and Reiser, B.J. (1985). The Lisp Tutor. *Byte, 10*, 4 (Apr.), 159-175.

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.

Gray, W. and Anderson, J.R. (in press). Change episodes in coding: When and how do programmers change their code? In G. Olson, S. Sheppard and E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop*. Ablex, Norwood, NJ.

Johnson, M.L. and Soloway, E. (1985). PROUST: An automatic debugger for Pascal programs. *Byte, 10*, 4 (Apr.) 179-190.

Pirolli, P.L. and Anderson, J.R. (1985). The role of learning from examples in the acquisition of recursive programming skill. *Canadian Journal of Psychology, 39*, 240-272.

Sleeman, D.H. (1983). Inferring student models for intelligent tutor-aided instruction. In R. Michalski, J. Carbonell and T. Mitchell (Eds.) *Machine Learning*. Tioga, Palo Alto, CA.

Sleeman, D.H and Brown, J.S. (1982). *Intelligent Tutoring Systems*. Academic Press, New York, NY.