Student Modeling and Tutoring Flexibility in the Lisp Intelligent Tutoring System

Albert T. Corbett

John R. Anderson

Eric G. Patterson

Advanced Computer Tutoring Project

Carnegie-Mellon University

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 each step. 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 revised and extended over the years, and now consists of approximately 240 exercises covering the first twelve chapters of an introductory Lisp text (Anderson, Corbett & Reiser, 1987), 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 system.

The tutor was developed to serve as a "real-life" application of the ACT* model of skill acquisition (Anderson, 1983). The tutor's design and behavior are based in large part on the principles of this model, as is described in more detail below. One goal of the tutor, of course, was to teach LISP more effectively, but a second goal was to collect detailed data with the tutor on the course skill acquisition in a natural setting. The tutor 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 (Anderson, Boyle & Reiser, 1985; Anderson & Reiser, 1985). In one study, students using the tutor completed the exercises in a little over half the time required by the students working on their own and scored equally well on a posttest. In the other study, students completed the exercises 30% faster and scored 43% higher on a posttest. The tutor's effectiveness, as indicated by these global measures of time-on-task and posttest success, provides some general confirmation of the underlying theory. However, the data provided by the tutor on the course of acquisition bears more directly on the theory. 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. As is described below, these coding rules are the basic units of skill acquisition in ACT* and the theory makes some predictions about the time-course of acquisition. For example, there should be a large speed-up in application of a rule from the first to second use, followed by a subsequent more gradual speed-up. This prediction is confirmed by the results (Anderson, 1987a, in press; Conrad & Anderson, 1988).

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. First, the theory underlying the tutor has undergone revision since the tutor was first developed (Anderson & Thompson, in press). 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, some students complain about aspects of the tutor and believe they would be happier with 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. Specifically, we will focus on the impact of one component of the tutor, the student model, on the tutor's behavior and implications for modifying that behavior. We will begin with a brief description of the tutor's current behavior, followed by the principles that gave rise to the tutor. Then we will discuss the architecture of the tutor, the role of the student model in governing the tutor's 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 Tutorial Interaction

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 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, and is able to provide feedback accordingly.

Insert Figure 1 about here

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. One possible solution to this exercise looks like this:

```
(defun ends (lis)
    (cons (car lis) (last lis)))
```

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 a left parenthesis and the symbol ***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 angle-bracket symbols on the screen. These angle-bracket symbols represent subgoals that the student must satsify, more specifically, they represent arguments of ***defun*** 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.

ACT* and the Tutor's Design

The tutor's behavior as described in the prior section reflects both general assumptions of the ACT* theory and observations of students learning to program in LISP (Anderson, 1987b; Anderson, Farrell & Sauers, 1984; Pirolli & Anderson, 1985). ACT* is a general theory of cognition, but only a few of its assumptions are relevant to tutoring.[1] These assumptions were in turn distilled into a set of tutoring principles by Anderson, Boyle, Farrell and Reiser (1987).

One central assumption is that problem-solving behavior is hierarchically structured. A problem represents a goal that can be solved by decomposition into subgoals. For example, the template in Figure 1b indicates that the goal of defining a function can be satisfied by typing ***defun*** and setting three subgoals: (1) coding the function name, (2) coding the function parameters and (3) coding the process or body of the function. Some goals can be directly satisfied by the execution of a rule (e.g., the first two subgoals in our example), while other goals require additional decomposition (e.g., the third

[1]These assumptions are not unique to ACT*, but are shared by a variety of cognitive theories.

goal in the example). Figure 2 depicts the hierarchical goal tree that underlies the task of defining the function ***ends.***

Insert Figure 2 about here

The nodes in this figure represent goals and the branches are labelled with code symbols that are generated in satisfying the goals. As can be seen, when ***defun*** is coded, three subgoals are set (the arc at this juncture in the figure indicates that each of these subgoals must be satisfied). This goal structure represents three different solutions to the exercise, since the CODE-PROCESS goal can be satisfied by any of three code symbols, ***cons, list*** or ***append.*** Each of these three symbols gives rise to a unique subgoal structure that must be satisfied. One tutoring principle is derived from the hierarchical goal-structure assumption: a tutor should make the goal structure explicit.

An important cluster of assumptions, giving rise to several tutoring principles, concerns the representation and acquisition of knowledge. ACT* distinguishes between declarative and procedural knowledge. It assumes that knowledge underlying a skill is encoded in declarative form initially, on the basis of communication or observation. Declarative knowledge can be encoded readily, but does not lead directly to behavior. Instead, behavior requires procedural knowledge. Procedural knowledge is represented as a set of independent IF-THEN rules called ***productions.*** An English translation of some productions rules that students learn early in LISP programming would be:

```
(1) IF the goal is to define a function called name
        that accepts n arguments and performs the task process
    THEN code a call to defun
        and set subgoals to code
            (a) the function name name
            (b) a list of n parameters
            (c) the process process
```

```
(2) IF the goal is to form a list
        by inserting newitem at the beginning of
        an existing list oldlist
    THEN code a call to CONS
        and set subgoals to code
            (a) newitem
            (b) oldlist.
```

In the course of skill acquisition, the declarative knowledge that is encoded is applied by means of general problem-solving productions. This application of declarative knowledge is relatively slow, but results in the formulation of domain specific productions (e.g., the examples above) that can be applied more rapidly. With additional experience, productions become stronger and give rise to larger order productions. These assumptions concerning skill acquisition give rise to several tutoring principles: instruction should be presented in the context of problem solving, the student's knowledge should be represented as a set of productions and the grain size of the representation should be adjusted with learning.

Finally, ACT* assumes that skill acquisition and performance depends on a limited-capacity working memory. This assumption gives rise to the general principle that working memory load should be minimized. Along with some specific production-learning assumptions discussed below, it also gives rise to the principle that error feedback should be presented immediately.

Although the tutor as described here has proven generally effective, there are two aspects of its behavior that we would like to manipulate experimentally, the constraints on coding order and the immediate feedback policy.

Coding Order

Students are constrained by the tutor to expand goals left-to-right and depth-first. These constraints do not represent a general principle of problem-solving in ACT*, but represent a generalization based on observations of students learning Lisp (Anderson, Farrell & Sauers, 1984) The tutor itself deviates from the generalization under some circumstances. For example, when a student creates a helping function in an exercise, the tutor imposes a breadth-first expansion; that is, the student completes the top-level function before defining the helping function. In some exercises students refer to local variables before declaring them, which violates left-to-right goal expansion. The tutor also recognizes that students may deviate from strictly top-down goal expansion. For example, imagine an exercise in which students have to define a function called ***remove-last*** that removes the last element of a list, for example, ***(remove-last '(a b c d))*** returns ***(a b c).*** This function can be defined as follows:

```
(defun remove-last (lis)
    (reverse (cdr (reverse lis))))
```

The body of this function is non-obvious to novices, even when students understand what the functions ***cdr*** and ***reverse*** do. The function reverses the list, deletes the resulting first element and then flips the list back again. The tutor contains coding productions that generate this code top-down, but if a student flounders at this point, the tutor branches to mean-ends analysis planning productions that derive the solution in a different order. The first means-ends production that applies recognizes that ***cdr*** is the only known function that deletes a list element. The second production recognizes that the list should be reversed before ***cdr*** is called, since ***cdr*** only deletes an element from the front of a list. A final means-ends production then recognizes that the resulting list should be reversed again. Conceivably, if students implicitly work through this plan, they would deviate from top-down code generation.

Our chief motivation for relaxing these constraints is that some students complain about the restrictions on coding order. Relaxing the restrictions and observing when students deviate from the standard order may provide additional evidence on the development of coding productions.

Immediate Feedback

There are several reasons for manipulating the timing and control of feedback. Anderson, Boyle, Farrell & Reiser (1987) proposed that feedback be provided immediately in skill acquisition for practical as well as theoretical reasons. Two practical reasons for immediate feedback are that it saves time and frustration on the part of the student by reducing floundering and that it simplifies the model tracing task. On the other hand, there are several practical reasons for varying from the principle. Some students complain about immediate feedback, in part because they don't like being interrupted and in part because they would prefer to find and fix mistakes themselves. In addition, human tutors do not necessarily intervene immediately when mistakes are made (Fox 1988, Lepper & Chabay, in press) and the effectiveness of immediate feedback varies (Kulik & Kulik, 1988).

An additional reason for varying feedback, however, is that the learning assumptions of ACT* have changed. Previously in ACT* production formation was based on a working memory trace of the problem solving episode. That is, production formation was based on all that transpired between the initial setting of a goal and the action that satisfied it. Optimal production formation required both the presence of relevant information in working memory and the ability to filter out irrelevant information. Since immediate feedback reduces floundering, it reduces the the load on working memory (by reducing the size of goal-satisfaction episodes) and fosters the formation of appropriate productions. Since the Lisp tutor was developed, however, assumptions have been elaborated and revised concerning the initial proceduralization of domain-specific productions (Anderson & Thompson, in press). In the revised formulation, production formation is based on working memory data structures representing the initial goal and ultimate solution and not on a trace of the entire goal-satisfaction episode. As a result, production formation is no longer directly related to the size of the episode (amount of floundering), and immediate feedback is no longer viewed as crucial in production formation.

Model Tracing and the Tutor's Architecture

The tutor's architecture can be analyzed into three basic components: domain knowledge (the student model), tutoring rules, and the interface (cf. Sleeman & Brown, 1982; Wenger, 1987). One might guess that the characteristics we wish to manipulate, coding order and immediate feedback chiefly involve the interface and pedagogical component. However, because of details of the tutor's architecture, these aspects of the tutorial interaction are closely tied to properties of the student model.

The Interface

Currently the tutor's interface is fairly simple. It is responsible for accepting the student's code and displaying the code on the screen. It is not responsible for maintaining an internal representation of the student's code. The interface accepts expressions from the student roughly a symbol at a time and does some syntactic checking. For example, it ensures that students do not embed illegal syntactic characters within atoms and ensures that students type a left parenthesis at the beginning of function calls. (Note that the interface must have information on correct solutions to perform the latter task). If the input satisfies these syntactic constraints the interface passes it through for checking. Communication in the other direction, from the tutor to the interface is accomplished by means of a symbol table, which is described below. At the end of each cycle, the interface converts the symbol table to Lisp code for display on the screen, sets the cursor on the appropriate goal symbol and awaits a new input.

The Student Model

As specified by the tutoring principles, the student model is implemented in the form of a production system. 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. In 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 the top-level goal of coding the function is added to a goal stack. In each cycle a goal is pulled from the stack and the set of production rules is compared to the goal and the problem description. This comparison process results in a set of productions, called the *conflict set*, that match the problem state and therefore could be applied. This set always contains one or more correct rules and generally contains one or more buggy rules. The student's input is then compared to this set. If the input matches the coding action of a correct production, that production "fires." It adds an expression to the symbol table which represents the student's solution and may add goals to the goal stack.[2] No tutorial action is taken, and control is returned to the interface. If the student's input does not match a correct production then the tutorial component responds.

The Tutorial Component

The tutorial component consists of a set of simple rules that apply when the student makes a mistake. If the student's input matches the coding action of a buggy production, then a feedback message associated with the production is presented. If the input does not match any production, the tutor cannot diagnose the error and indicates this to the student. If the student makes two errors at a goal that cannot be diagnosed or triggers the same buggy production three times, the tutor assumes the student is floundering and provides a correct code symbol along with an explanation. In each case, after the student hits the ***return*** key, control again passes to the interface.

In addition to providing error feedback, the tutorial cmponent performs a second function we call *knowledge tracing*. As the student solves exercises, the tutorial component maintains an "overlay" model of the student's knowledge of Lisp coding. For each correct production in the student model, the tutor maintains a probability estimate that the student has the production correctly encoded. The student's first response at each goal, correct or incorrect, is used to modify the estimate of a correct production in the conflict set. These probability estimates do not influence the tutor's response to error, but are used to select exercises and to decide when a student is ready to move on to a new topic. (See

[2] This is an oversimplification since not all productions generate code. Some only modify the goal structure or the problem description and as described earlier, some productions model planning processes. However, these productions are not directly relevant to the topic of modifying the tutorial interaction.

Corbett & Anderson, in press, for additional implementation details and an evaluation of knowledge tracing).

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. A second difficulty with model-tracing concerns the disambiguation of students' responses, since under some 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 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*,[3] 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 CODE-PROCESS node in Figure 2). 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, ***cons***, ***list*** or ***append***, can be employed to construct the required list of two elements. Not surprisingly, the subsequent 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, whichever function is chosen at the top-level goal here, ***cons***, ***list*** or ***append***, determines the correct code at

3 A similar process is described in Sleeman, 1983.

the subgoals (the correct arguments). 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 problem, 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.

Insert Figure 3 about here

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 the corresponding and-or goal tree itself.[4]

[4] Alternative solutions to the space demands of representing trace structures are being pursued in other tutoring projects. Skwarecki (1988) describes a tutoring project that employs a more compact goal tree. Anderson and Ray Pelletier are developing a more efficient production system interpreter that should be fast enough to run on-line in tutoring, eliminating the need for compiled solutions.

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.

Modifying the Tutorial Interaction

As is suggested in the preceding section, much of the tutor's complexity resides in the student model. The interface and tutorial components are fairly simple, as is the communication between components. Given this architecture, the tutorial characteristics we wish to manipulate, coding order and immediate feedback, depend heavily on characteristics of the student model. In particular, input order is not an optional attribute of the interface, but is governed by properties of the student model, because the student model rather than the interface determines what the student can do next. Since the student model can only expand goals top-down, depth-first and left-to-right, the student is contrained to type code in that order.

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 not strictly a consequence of immediate feedback, but a consequence of applying immediate feedback to a student model of minimum grain size.

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.

To relax the tutor's input and feedback constraints, we need to dissociate the tutor's interface component from the student-modelling component and to revise the tutorial component. We are following a multi-stage plan in accomplishing this task. The first step is to isolate the interface functions from the modelling functions. To do this, we have introduced a true structured editor into the the tutor. It provides editing commands that allow the student to enter code in any order and to delete code they have entered. The editor assumes the responsibility of maintaining a symbol table representing the student's code and ensures that the code is syntactically legal, but has no capability for checking the code is functionally correct. The structured editor provides code templates with angle-bracket symbols that are similar to, though not uniformly identical to the tutor's templates.[5] As in the case of the tutor, the student can only enter code by expanding angle-bracket symbols on the screen, but the student is able to select the next angle-bracket symbol to be expanded, to generate arbitrary angle-bracket symbols (as long as they are structurally legal) and to embed existing code or angle-bracket symbols in a template for a higher-level function call, and so has control over the order in which the code is generated.

[5] 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 finally deletes the last empty argument node.

Student-controlled Feedback

Having introduced the editor, the next step is to integrate with the student-modelling and tutorial components to enable feedback. In our initial research with the revised architecture, we have implemented a student-controlled tutor, in which students not only control the order in which they input code, but also when feedback is given. In this tutor students can type code, make whatever changes they want, and ask for feedback from the tutor at any point. The transition from tutor-controlled feedback to student-controlled feedback is a fairly small one. Instead of feeding each unit of code to the student-model and tutorial component, the code is buffered in the editor's symbol table and matched to the student-model only when the student requests help. 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 template 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.

In the student-controlled 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 nodes in the goal 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 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). We have begun collecting data with this version of the tutor, and preliminary results are described below.

Further Developments: Restoring Tutor Control

As data is collected with the student-controlled tutor, we have also begun work on the next phase in our project: restoring the ability to evaluate code on a symbol-by-symbol basis, while allowing students to deviate from the standard input order. Restoring symbol-by-symbol evaluation will allow us substantial flexibility in implementing tutor-controlled feedback rules. In addition, it will enable greater flexibility in knowledge tracing. In the standard tutor, knowledge tracing is based on the student's first coding attempt at each goal. Under the code-buffering implementation of the student-controlled tutor, however, the model-tracing and knowledge-tracing mechanisms only have access to the state of the code when the student requests help, with no record of the order in which the code is entered and no record of deletions and revisions the student made. Symbol-by-symbol evaluation will restore maximum information to the knowledge-tracing mechanism.

Since the fully-expanded goal tree is available for each exercise as a result of problem-compilation, we can implement symbol-by-symbol evaluation by means of a purely structural mapping of editor symbols onto goals. As described earlier, the editor generates code templates much like those created by the productions in the student-model. For example, when the student types a call to ***defun*** the editor generates the template ***(defun <name> <parameters> <process>)***, which is identical to the template generated by the production that codes ***defun***. While the angle-bracket symbols created by the editor do not strictly speaking represent goals in a correct solution, they do correspond closely to those goals. Thus, the solution to integrating the editor with model-tracing is to map each node in the editor symbol table, as it is generated ,to goals in the compiled solution tree. This mapping allows us to continue evaluating code on a symbol-by-symbol basis, even if the student diverges from depth-first left-to-right expansion and even if the student makes mistakes.

If there is a single solution to an exercise, each editor node will map to at most one goal in the solution tree. If an exercise has more than one solution, then editor nodes below a choice point will map to multiple goals on various branches. Issues arise concerning such mappings to multiple branches, however the issues can be readily resolved. First, a given input may satisfy goals on mulitple paths. For example, if the student begins the body of ***ends*** by typing ***(cons (car...)***,the symbol ***car*** matches a goal on two branches. In this, case, we can identify the appropriate branch, because only one branch involves ***cons***. However, the same disambiguation is not possible if the student begins the body as follows: ***(<function> (car <list>) <other-expressions>)***. Since the entire goal tree has been expanded through compilation, though, there is no particular reason to perform the disambiguation. The appropriate branch will emerge as the student continues typing.

A second issue in multiple mappings also arises. Suppose the student types the following code: ***(cons (car lis) (list (last lis)))***. The first three symbols match one branch in the tree and the latter three symbols match a branch of a different solution. In light of the theoretical assumption of top-down goal expansion, when such conflicts arise, we will disambiguate them in favor of the higher level code. Given this resolution, each time code is added to the editor table at or below a branch point, it is necessary to check downstream nodes to see if previously consistent code is now on a mismatching branch. On the other hand, each time code is deleted from the symbol table, it is necessary to check downstream to see if previously inconsistent code is now on a consistant branch.

Thus, while computational complexity is introduced, we can return to something approximating the input cycle of the original tutor. On each cycle, new code is matched to a production conflict set iat some goal (or more than one set if the code maps to more than one goal), any new editor nodes are mapped to goals in the tree, and finally downstream code is checked if the modified node falls below a branch point.

A final issue concerning this structural node mapping concerns bottom up coding. Although the editor expands code templates in a top-down fashion, much like the tutor, it is possible, by means of an editor command to generate code bottom up. (This command takes an existing Lisp expression or angle-bracket symbol and embeds it as the first argument of a new function call template). To the extent that students deviate from top-down expansion, a purely structural mapping will lead to code mismatches. For example, if a student coded the body of ***ends*** bottom up, he or she would begin by coding ***lis.*** This symbol would be structually mapped to the CODE-PROCESS goal, at which it does not match any production. Eventually, as the student completes a bottom-up expansion, the code would be recognized, but at intermediate states the tutor would fail to recognize it as a possible solution If we find this happening frequently it would warrent moving to a more complex goal-mapping scheme which makes reference to the content of the goals. As the student types symbols under this system, the tutor would search the goal tree for goals that match the input and are topologically consistant with the structure of the editor table. Thus, if the student started by typing ***lis*** the tutor would tentatively map the input to the six CODE-PARAM goals in the goal tree. If the student then embedded this symbol in a call to ***car,*** i.e., ***(car lis),*** only four topologically consistent mappings would remain from ***lis*** to CODE-PARAM goals (i.e., mappings that would not require the subsequent deletion of code). Such a system would enable more immediately accurate evaluation of bottom up coding, but yet higher computational expense.

The immediate code evaluation process described in this section enables us to implement a variety of tutor-controlled feedback rules. One option, of course, is to restore the standard tutor's policy: inform the student of errors immediately and require that each step be accomplished correctly. However, symbol-by-symbol evaluation does not necessitate immediate feedback. One structural alternative that might prove less disruptive is to provide feedback on the basis of larger order units, e.g., complete function calls.

Another option is to make feedback timing contingent on the type of error made. For example, given the demonstrable effect of working memory limitations on coding errors (Anderson & Jeffries, 1985) it would be ideal to present feedback messages immediately only when the benefit of the message outweighs the cost of disrupting working memory. One rule that would approach this goal and would be fairly easy to implement given the knowledge tracing mechanism would be to delay feedback on "slips," (errors in which a student fails to fire a well-learned production) that do not seriously disrupt the goal structure of the exercise. An alternate scheme to minimize working memory disruptions would be to provide immediate notification when an error is made, but not to insist on immediate correction. Such a tutor would signal that an error has been made by displaying the error in a distinctive font, but leave it up to the student when a correction is made. Under this system it is up to the student to evaluate the relationship of the error to his or her current goal structure and to choose the optimal time to suspend goal expansion and correct the error.

Multiple Errors

A final issue deserves comment before we consider the preliminary data from the student-controlled tutor. If we deviate from immediate feedback and error correction, then students will be generating code with more than one error. When the student requests help, a decision will be required concerning the error on which to comment.

The student-controlled buffering approach described above answers this question implicitly. Given the standard student-modeling mechanism, when the student asks the tutor to check over the code, the tutor checks it top-down, depth-first, left-to-right and provides feedback on the first error encountered. Indeed, if the student asks for goal-specific feedback (a goal hint or explanation of correct code), the tutor checks over the code from the beginning. If an error is encountered before reaching the target goal, the tutor will instead provide feedback on the earlier error. In part this is because it could be difficult to continue tracing down to the target goal once an error is made. However, another important reason concerns the content of the feedback itself. Given the operation of the standard tutor, the feedback messages, including the goal reminders, bug diagnoses and descriptions of correct code, assume that the upstream code is correct and that there is no code downstream. These messages may mislead students if these assumptions are violated. Currently, we are keeping the standard feedback messages in the new implementations for two reasons: experimental control and efficiency. We want to keep the content of feedback constant as we vary its control and timing, to avoid confounding the factors. However, another important consideration is that there are more than a thousand feedback messages to be modified. We would like to determine that an alternative to immediate feedback and error correction is effective, before tuning this body of messages accordingly.

Testing Student-controlled Feedback

As described earlier, our initial research in varying the nature of the tutorial interaction employs 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. We have collected data with this tutor for the first two lessons of the tutor's curriculum. The first lesson introduces basic arithmetic and list functions, the structure of function calls, and variables. The second lesson covers function definitions.

Thirty-four subjects took part in this 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 the 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, depth-first, 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 necessarily 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 coding order. In no case did any of these partial solutions show evidence of right-to-left, depth-first 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. Subjects never deviated from depth-first coding in these exercises, although the structure of the exercises provided relatively few opportunities to distinguish depth-first vs. breadth-first coding - a total of three per subject. Across the 16 subjects using the student-controlled tutor and the seven exercises in lesson two, however, 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, depth-first 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. In that lesson students detected and revised a total of 86 errors, while the tutor detected 165 errors. Of these 86 errors students revised, 49 (57%) were corrected, while the remaining 37 (43%) are changed to a different error.[6] Thus, there may be some ben efit in deviating from symbol-by-symbol assistance in tutoring. It should be noted though, that in addition to correcting errors (and "miscorrecting" errors) students also changed correct code symbols 21 times (changing them to

[6] 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.

alternative correct symbols 9 times and "discorrecting" them 12 times). Thus, of all the spontaneous changes students made, 20% were changes to correct code.

A final issue we examined concerns the relative position of the coding revisions the subjects made. Gray and Anderson (in press) investigated the code revisions students made when writing fairly difficult iterative search functions in Lisp. They found that subjects are most likely to change code at the goal they are currently working on or have just completed and are next most likely to go back and change code at a goal superordinate to the current goal. They are relatively unlikely to change code at other goals. For example, suppose a student is typing code in the standard order andhas reached this point in coding a solution to ***ends***:

```
(defun ends (lis)
    (list (car lis) (car (last <list3>))))
```

Gray and Anderson found that the student would be most likely to change the symbol ***last***, next most likely to change the second instance of ***car***, or the occurrences of ***list*** and ***defun***, all of which satisfy superordinate goals of the current goal, and least likely to change any of the other symbols in the code. The detailed analyses of lesson 2 revealed the same pattern for the simpler functions in the current experiment. Sixty-one of the students changes (57%) were at the current goal, 31 (29%) were at superordinate goals and 15 (14%) were at other goals. In addition, the probability that a revision actually corrects an error varied with the position of the correction relative to the current goal. Fifty-one percent of the changes at the current goal corrected an error, while only 42% of the changes at superordinate goals and 33% of changes at other goals corrected errors.

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 may not save much time relative to the student-controlled tutor; 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, relative to the standard tutor, 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. (1983). *The architecture of cognition.* Harvard University Press, Cambridge, MA.

Anderson, J.R. (1987a). 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. (1987b). Skill acquisition: Compilation of weak-method problem solutions. *Psychological Review, 94*, 192-210.

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., Farrell, R. & Reiser, B.J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.) *Modelling cognition.* Wiley, New York.

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 Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction, 22*, 403-423.

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

Anderson, J.R. and Thompson, R. (in press). Use of analogy in a production system architecture. In S. Vosniadou and A. Ortony (Eds.) *Similarity and analogical reasoning.* Cambridge University Press, New York.

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.

Conrad, F.C. and Anderson, J.R. (1988). The process of learning Lisp. *The Proceedings of the Tenth Annual Conference of the Cognitive Science Society*, Montreal.

Corbett, A.T. and Anderson, J.R. (in press). The Lisp Intelligent Tutoring System: Research in skill acquisition. In J. Larkin, R. Chabay and C. Sheftic (Eds.), *Computer assisted instruction and intelligent tutoring systems: Establishing communications and collaboration.* Erlbaum, Hillsdale, NJ.

Fox, B.A. (1988). Cognitive and interactional aspects of correction in tutoring. *Technical Report #88-2*, Institute of Cognitive Science, University of Colorado.

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.

Kulik, J.A. and Kulik, C.C. (1988). Timing of feedback and verbal learning. *Review of Educational Research, 58*, 79-97.

Lepper, M.R. and Chabay, R.W. (in press). Socializing the intelligent tutor: Bringing empathy to computer tutors. In H. Mandl and A. Lesgold (Eds.) *Learning issues for intelligent tutoring systems.* Springer, New York.

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.

Skwarecki, E. J. (1988). Improving the engineering of model-tracing diagnosis. *The Proceedings of the International Conference on Intelligent Tutoring Systems*, Montreal.

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.

Wenger, E. (1987). *Artificial intelligence and tutoring systems.* Morgan Kaufmann, Los Altos, CA.

Figure Captions

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

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

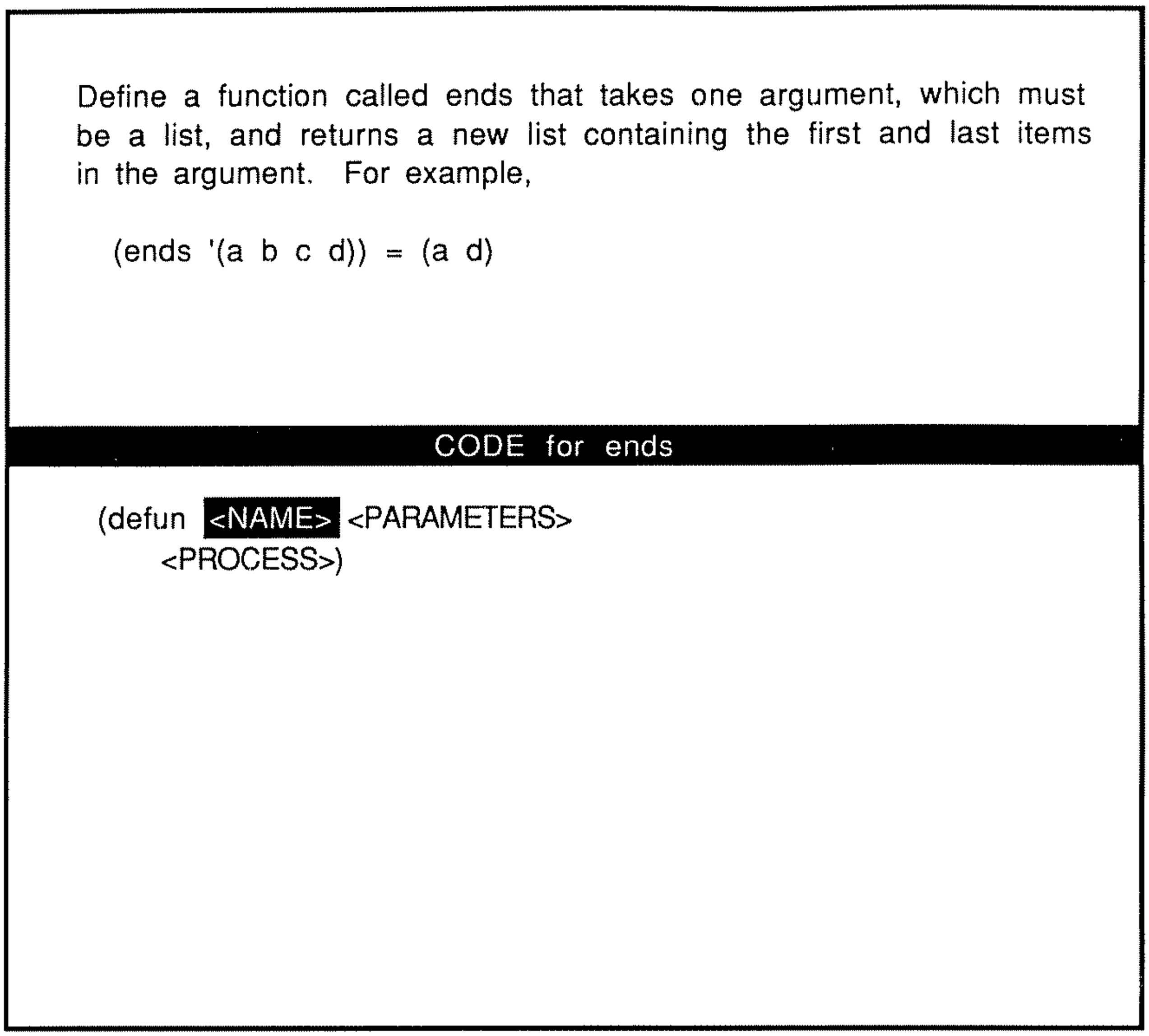
Figure 3. A depth-first transformation of the goal structure for the function *ends*.

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
```



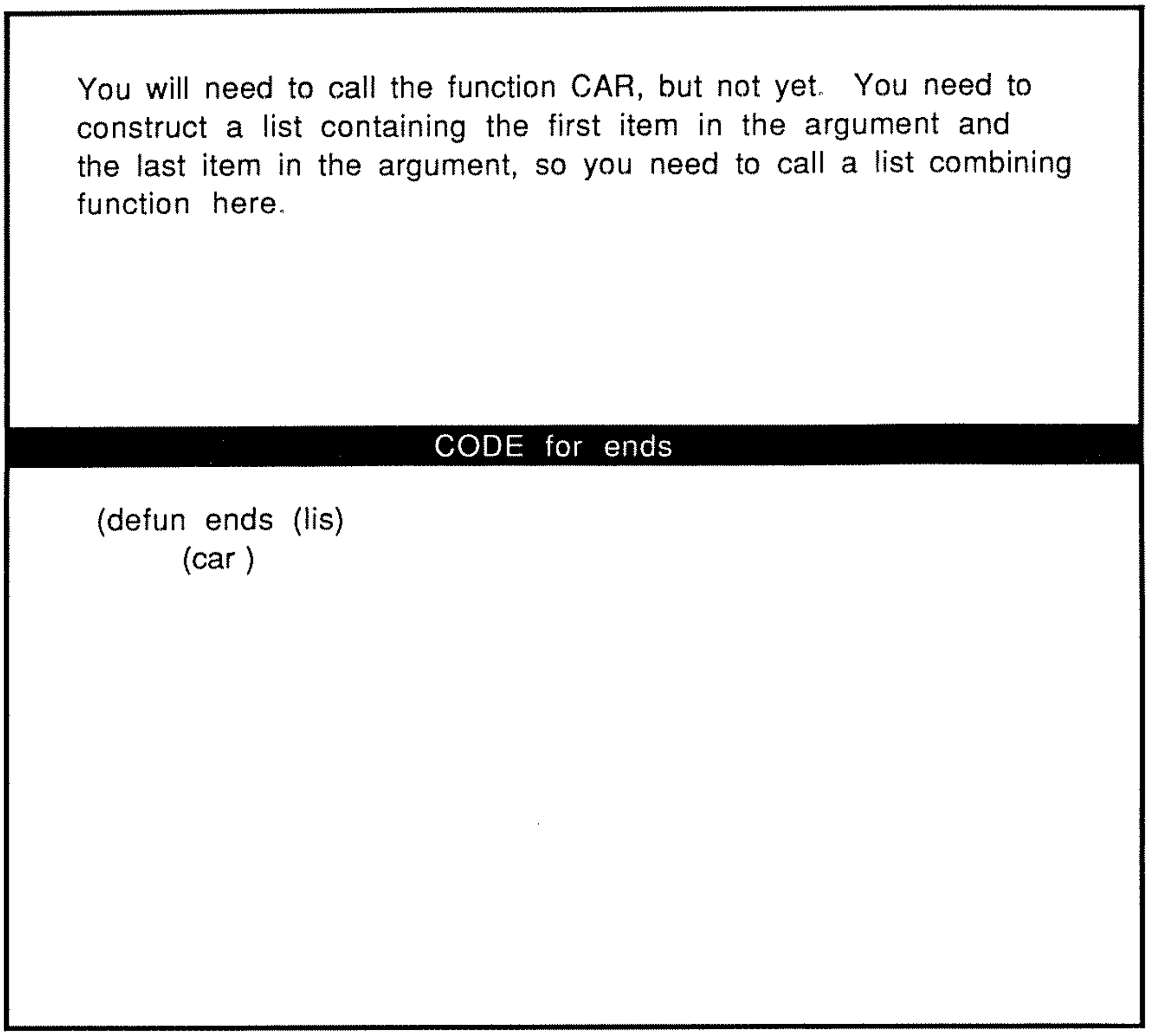
Define a function called ends that takes one argument, which must be a list, and returns a new list containing the first and last items in the argument. For example,

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

CODE for ends

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

FIGURE 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 )
```

FIGURE 12

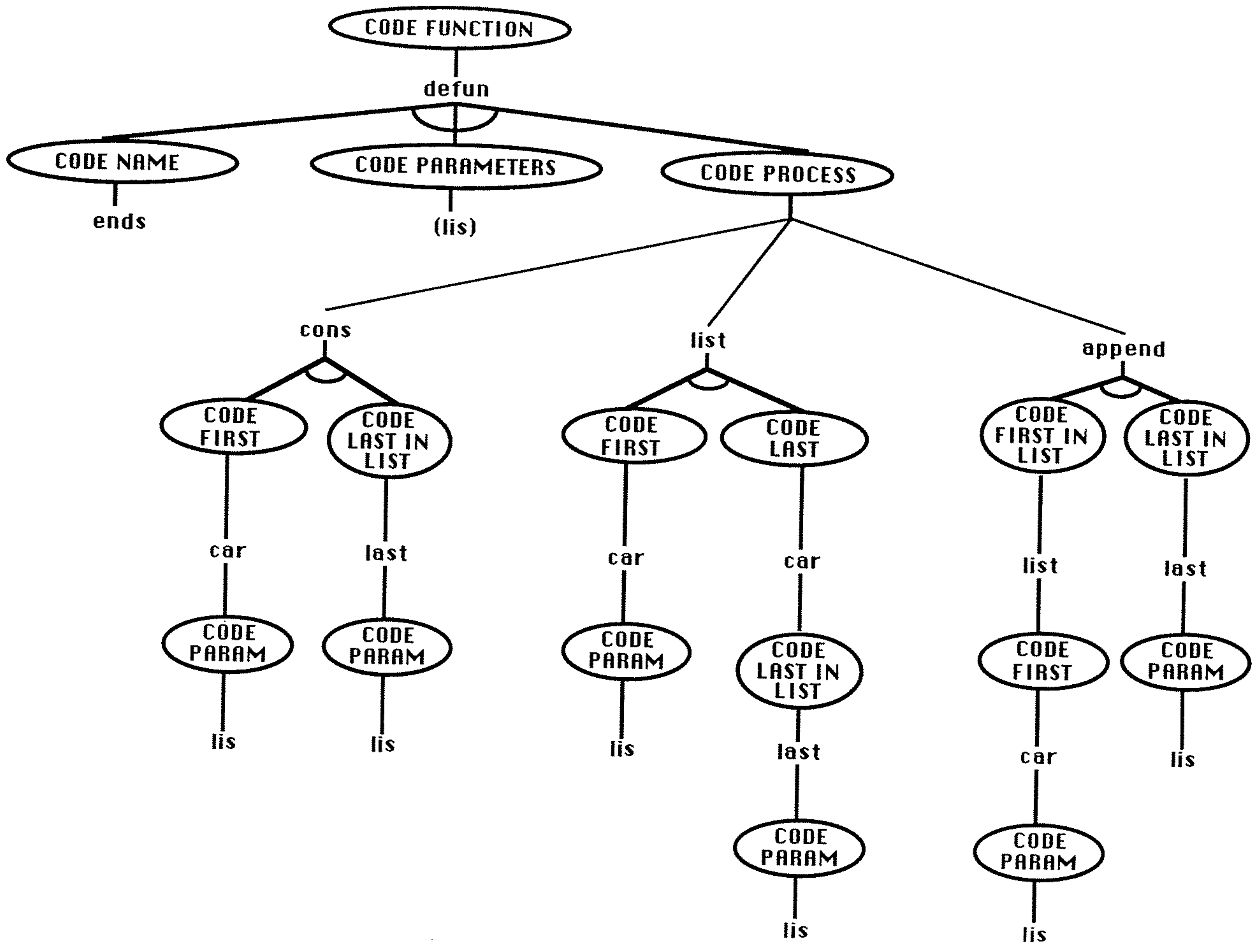

CODE FUNCTION
defun
CODE NAME
ends
CODE PARAMETERS
(lis)
CODE PROCESS
cons
CODE FIRST
car
CODE PARAM
lis
CODE LAST IN LIST
last
CODE PARAM
lis
list
CODE FIRST
car
CODE PARAM
lis
CODE LAST
car
CODE LAST IN LIST
last
CODE PARAM
lis
append
CODE FIRST IN LIST
list
CODE FIRST
car
CODE PARAM
lis
CODE LAST IN LIST
last
CODE PARAM
lis

FIGURE 2

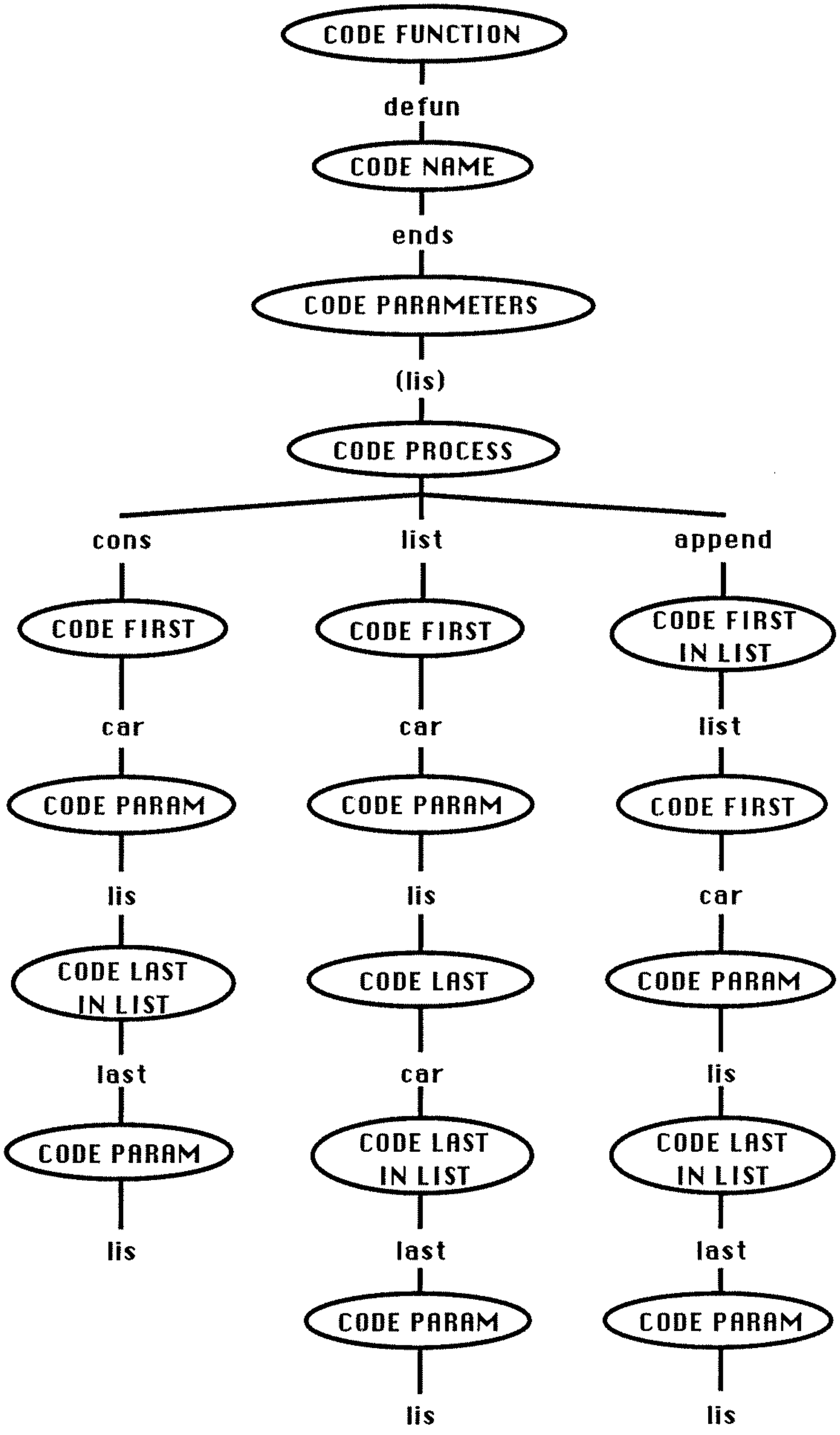

CODE FUNCTION
defun
CODE NAME
ends
CODE PARAMETERS
(lis)
CODE PROCESS
cons
list
append
CODE FIRST
CODE FIRST
CODE FIRST IN LIST
car
car
list
CODE PARAM
CODE PARAM
CODE FIRST
lis
lis
car
CODE LAST IN LIST
CODE LAST
CODE PARAM
last
car
lis
CODE PARAM
CODE LAST IN LIST
CODE LAST IN LIST
lis
last
last
CODE PARAM
CODE PARAM
lis
lis

FIGURE 3