Feedback Timing and Student Control in the Lisp Intelligent Tutoring System

Albert T. Corbett

John R. Anderson

Advanced Computer Tutoring Project

Carnegie-Mellon University

Abstract

The Lisp Intelligent Tutoring System provides assistance to students as they do Lisp coding exercises. It has been in use over the past four years and has proven to be an effective environment for doing exercises. Much of its effectiveness is achieved by maintaining strong control over the student's behavior. Students are constrained to type their code top-down left-to-right, are informed immediately of errors and are required to repair errors before moving on. In this paper we describe a new version of the tutor that is more flexible. In this version students control the order in which code is entered and control when revisions are made. The tutor continues to provide immediate feedback, but does not interrupt the student's activity. We describe the results of an initial study comparing the effectiveness of the new tutor and the standard version and examine students' use of the enhanced flexibility in the new tutor.

The Lisp Intelligent Tutoring System assists students with Lisp programming exercises (Anderson & Reiser, 1985). The program presents problem descriptions and as the students type their answers the program monitors their progress step-by-step, providing feedback on errors and providing correct steps if the student appears to be floundering. The tutor currently includes about 240 exercises, covering the first twelve chapters of an introductory text (Anderson, Corbett & Reiser, 1987) and has been used to teach an introductory Lisp course at Carnegie-Mellon University each term since the fall of 1984. Three studies have been completed evaluating the tutor's effectiveness; two were conducted after its initial development (Anderson, Boyle & Reiser 1985) and a third unpublished study was completed this past spring. These studies have shown that the tutor is an effective environment for doing Lisp exercises - chiefly because it saves time. Students complete the exercises more quickly using the tutor than working on their own. Generally, there is no difference in how well the material is learned, as measured by a paper-and-pencil posttest, although in one study students performed better on the posttest after using the tutor.

We have been using the tutor for the past two years to examine issues concerning the content and timing of feedback in tutoring. One of the tutor's distinctive characteristics is that it provides immediate feedback after the student types each unit of code (generally a single atom or "word"). If the student makes a mistake, the tutor immediately interrupts, provides feedback to the student and requires the student to correct the error before moving on. If at any point in the code, the student makes repeated mistakes that the tutor does not recognize, or if the student repeatedly makes the same type of mistake, the tutor intervenes and provides a correct code unit along with an explanation. The effect of this feedback procedure is to keep students on a correct solution path and to prevent floundering. Interestingly, the immediacy of the feedback appears more important in this regard than the content (Corbett & Anderson, in press). In two experiments we modified the tutor so that it did not provide comments. It continued to interrupt immediately to indicate an error had been made and required errors to be repaired immediately, but did not provide any diagnosis or explanation. It also provided correct code steps when the student appeared to be floundering, but again, did not provide any explanation. Shutting off comments in this fashion did have an effect; after making an error, students using the standard tutor (with comments) were more likely to correct the error on their very next attempt. However, there was no effect on total time to complete the exercises, apparently because students using the no-comment version did not spend time reading explanations.

The present paper describes the second in a series of studies that directly evaluate the timing and use of feedback in the tutor. While the immediate feedback principle appears central to the tutor's success, there are several reasons we are interested in varying the timing of feedback and relaxing the tutor's control over the tutorial interaction. First, some students who use the standard tutor say they would like it better if it did not give immediate feedback. Second, immediate feedback does not uniformly give rise to enhanced learning (Anderson, Boyle, Corbett & Lewis, in press; Kulik & Kulik, 1988). Third, human tutors do not always intervene immediately when a student makes a mistake (Fox, 1988; Lepper & Chabay, in press). A year ago we developed a "student-controlled" version of the tutor that gave students control over the delivery of feedback (Corbett & Anderson, in press). In this version of the tutor students were given a structured editor, allowing them to enter their code in any order and to make whatever revisions they wished. The tutor did not provide assistance automatically, but students could request help at any time. They could ask for help from the tutor on a specific step (a goal hint or an explanation of the correct action) or they could request the tutor to check over their code, in which case the tutor examined the code and provided feedback on the first error it discovered. This "student-controlled" version of the tutor did not prove as effective as the standard tutor. Students using the student-controlled version took about 60% longer to complete the exercises than students using the standard version, while the two groups did equally well on a posttest.

In this paper we describe an initial study with a new version of the tutor which combines properties of the standard version and student-controlled version. In this version, the tutor notifies students immediately when they make mistakes, by displaying the errors in boldface on the screen. However, the tutor does not provide feedback messages unless requested and does not demand any specific action from the student when an error occurs. We have informally termed this new version the "flag tutor" since it merely flags errors when they occur, leaving control of events in the hands of the students.

Interacting with the Tutors

Figure 1 provides two "snapshots" of the screen as a student is using the standard tutor and Figure 2 provides a snapshot of a student using the flag tutor In Figure 1a, the student has just begun defining a function called ***ends***. The problem description appears in the "tutor window" at the top of the screen and the student has finished typing the first unit of code, ***defun***, in the "code window" at the bottom of the screen. At that point, the tutor did some work for the student, expanding the template for a call to ***defun*** It provided a balancing right parenthesis and put three placeholder symbols on the screen These placeholders are not themselves lisp code, but represent goals the student needs to satisfy (code the student will need to provide). Students are constrained to type their code top-down and left-to-right, so after expanding the template, the tutor moves the cursor over the next placeholder *<name>*, and the student is allowed to continue. Figure 1b, depicts the screen after the student has made a mistake When the student typed ***car***, which is an error, the tutor presented a feedback message in the tutor window. In this case, the tutor has recognized that the student is anticipating a subsequent portion of the code and reminds the student about the operation that has been omitted

The flag tutor deviates from the standard tutor in several ways, each of which is intended enhance the student's control of the coding process. First, students are provided a true structured editor, enabling them to deviate from left-to-right and top-down coding in entering their code Second, the students have free access to a Lisp interpreter (labelled the "LISP Window" in Figure 2) while working on a coding exercise. This allows students to experiment with possible solution components by trying them in Lisp. Third, while the tutor monitors the student's performance continuously and immediately notifies the student of an error, it does not interrupt the student When the student types an incorrect code symbol, the tutor displays the error in boldface, but otherwise allows the student to continue Figure 2 depicts the flag tutor screen when a student makes the same mistake as in Figure 1b. The error ***car*** is displayed in boldface, but no feedback message is displayed. Finally, the criterion for intervening to provide the student a correct step is relaxed in the flag tutor. As described earlier, the standard tutor provides a correct step if, at any goal, the student repeatedly makes the same type of error. The flag tutor, on the other hand, does not monitor for floundering while the student is typing. It only assesses floundering at a goal when a student asks for help at that goal Thus, the flag tutor does not intervene with a correct step unless the student has repeatedly asked for help at a given goal and the student appears to be floundering based on the code that is present when those requests are made.

Students using either version of the tutor can ask for two types of help at any placeholder symbol. They can request a reminder about what goal they are trying to achieve and they can directly request an explanation of the correct step for achieving that goal. Students using the flag tutor can also request a third type of help; they can request feedback on an error that has been flagged by the tutor. When this request is made, the student gets the same error feedback message as students get automatically with the standard tutor.

Tutor Implementation

LISPITS provides assistance to students on the basis of a *model tracing* paradigm. The tutor contains a student model, derived from observations of students learning Lisp (Anderson, Farrell & Sauers, 1984; Pirolli & Anderson, 1985). The student model takes the form of a production system with approximately 1200 production rules for generating correct and incorrect LISP code. The tutor uses these rules to model the individual correct and incorrect steps that a student might take in solving a problem and uses that information in evaluating the student's responses.[1] In the original implementation of the tutor, the production system model ran step-by-step along with the student, but had trouble keeping up in real time. As a result, problem solutions are now pre-compiled That is, the production system runs ahead of time on each exercise It systematically expands every solution path it can find for the exercise and stores the information relevant to tutoring in a goal tree.[2] A diagram of the goal tree for the function *ends* is presented in Figure 3. Nodes in this structure represent goals and links represent production firings A branch point,

[1]See Sleeman & Brown, 1982, part III, or Wenger, 1987, for a description of other tutors that take a similar approach.

[2]A similar process is described in Sleeman, 1983.

e.g., at the *code-process* node, indicates that a goal can be satisfied by more than one production, leading to alternative solutions to the exercise. Information can be accessed in this data structure more quickly than it can be generated by the production system, thus speeding execution of the standard tutor. The existence of such data structures also makes it feasible to implement versions of the tutor, such as the flag tutor, that deviate from the standard tutor constraints. (See Corbett, Anderson & Patterson, 1988, for additional details on problem compilation and tutoring flexibility).

Since the standard tutor constrains the student to type code top-down left-to-right and requires that each coding step be completed correctly before the next is attempted, the student's solution progresses in an orderly step-by-step fashion down-through the goal tree; at each step a unique goal is active. The situation is more complicated in the case of the flag tutor, since the students may deviate from top-down left-to-right coding and may delete correct code. Since many exercises have more than one possible solution, each code symbol generated by the student may map to more than one goal. For example, consider the following code fragment:

```
(defun ends (lis)
      (<function> (car <list>) ...))
```

The symbol ***car*** maps to the three goals at the fifth level in the goal tree. Since the function call at the parent goal has not been filled in, any of the three corresponding paths is viable. The tutor must recognize that ***car*** is correct, since it satisfies goals on two of the paths. However, if the student subsequently types ***append*** in the ***<function>*** slot, which maps to the *code-process* goal, then ***car*** is no longer on a viable path and there is an error in the student's code. This error may be repaired either by deleting ***append*** or by deleting ***car***.

Given these considerations, code checking necessarily entails more computation in the case of the flag tutor. Whenever the student fills in a code unit that maps to more than one goal, the tutor begins by looking up through the goal tree to find the closest correct code to determine which paths are valid. If the code matches a goal on at least one valid path, then it is taken as correct; if no valid goals are matched, it is flagged as incorrect. When correct code is typed at or below a branch point, the tutor must also check correct code lower in the goal tree to ensure that it remains on a valid path. Finally, when correct code is deleted at or below a branch point it may relax constraints on downstream code. As a result, when correct code is deleted at or below a branch point, incorrect code lower in the goal tree is checked to see if it now matches a valid path.

Testing the Flag tutor

Twenty-five subjects took part in the initial study of the flag tutor. Thirteen students used the standard immediate feedback tutor while twelve used the new flag tutor. The twenty-five subjects completed the first two lessons in the curriculum and took a paper-and-pencil cumulative test.

Evaluation Measures

Two overall measures of the tutor's effectiveness were computed: performance on the posttest and time to complete the exercises. There was essentially no difference between the two groups on the final test. Subjects using the standard tutor scored 87% correct, while student using the flag tutor scored 84% correct. However, time to complete the exercises varied between the two groups. Subjects who used the standard version of the tutor spent an average of 2.46 minutes on each exercise, while students who used the flag tutor spent an average of 3.97 minutes [3]on each exercise, $t(23)= 2.76, p<.05$.

We thought the flag tutor might be an improvement on the earlier student-controlled tutor. since the flag tutor provides both flexibility and immediate information on progress toward a correct solution. However. the flag tutor did not proved more efficient. Students using the flag tutor took about 60% longer to complete the exercises than students using the standard tutor, which is about the same level of performance as was obtained with the student-

[3]Subjects using the flag tutor encountered a number of bugs in this initial study - an average of about 3.5 per student. Seventy percent involved a minor problem with the editor; under certain circumstances it generated a spurious symbol, which subjects simply deleted. Since each coding cycle is time stamped, the effects of bugs could be readily removed from the measure of coding time

controlled tutor.

Processing Measures

Detailed analysis of the log files generated by the flag tutor suggest that students took relatively little advantage of the flexibility provided by the tutor. As was the case with the student-controlled tutor, students almost never deviated from top-down left-to-right coding. There were three instances of right-to-left coding and two instances of bottom-up coding. Students also made minimal use of the Lisp window. Seven students did not make use of it all all and only two students used it in more than one exercise. The final, and perhaps most important question concerns the timing of error corrections. Students using the flag tutor revised 72% of their code mistakes immediately. They revised an additional 17% of their mistakes after typing just one additional code unit or repairing one other existing error, and revised 11% of their errors after longer delays.

There are several reasons to expect that the greater freedom provided by the flag tutor will be associated with a cost in total time spent on the exercises. First, when a mistake is made with the flag tutor, the student must decide whether to go back and fix the error and must take the time to execute the required editor commands. Second, the code-checking computation is more complex in the case of the flag tutor than in the case of the standard tutor. There is considerable evidence in the log files, however, that students take longer to complete exercises with the flag tutor because they are floundering to a larger extent. First, across the two lessons, students are simply making more errors with the flag tutor than with the standard tutor (31.7 errors/student vs. 24.8 errors/student). Second, after receiving feedback on an incorrect code unit, students using the flag tutor are more likely to eventually retype that same incorrect code than students using the standard tutor (6.7 times/subject vs. 3.8 times/subject). Third, students using the flag tutor also delete correct code units an average of 6.9 times across the experiment, something that cannot be done with the standard tutor. Finally, the standard tutor intervened an average of two times per subject to provide a correct answer because of floundering, while the flag tutor never intervened.

Two other interesting results should be noted. First, students had surprising difficulty in asking the flag tutor to comment on errors. When making this request, the student needs to move the cursor over the error in question. However, about 35% of the time students made this request, the cursor was over correct code or an unfilled placeholder.

Finally, students were asked to rate various aspects of the tutor and there was a suggestion that students were happier with the flag tutor. Specifically, when asked to rate how well they liked immediate feedback (on a seven point scale) the ratings were marginally more positive in the flag tutor condition than in the standard tutor condition, 5.92 vs. 4.92, t(23)=1.61, p< .11.

Conclusion

This initial flag tutor implementation did not live up to our expectations, but this study suggests that two modifications may lead to substantial improvement. The most important modification would be to limit floundering by having the flag tutor monitor all code the student types for evidence of floundering instead of just monitoring when help is requested. The second modification would be to introduce a general help key, which would allow the tutor to direct the student's attention to an appropriate error to correct.

While we think we can improve the efficiency of the flag tutor, it is becoming increasingly apparent to us that we will not be able to create a system that allows students more control and yet still gets them through the exercises as quickly. There is evidence, however, that students want more control, both from informal comments over the years and from the ratings in the present experiment. This creates a standard philosophical dilemma in education between freedom and educational efficiency. We imagine it will be resolved by putting the decision in the student's hands - informing them of the characteristics of the various tutors and letting them choose.

References

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.

Corbett, A.T., Anderson, J.R. and Patterson, E.G. (1988) Problem compilation and tutoring flexibility in the Lisp Tutor. *The Proceedings of the International Conference on Intelligent Tutoring Systems*, 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. Scheftic (eds.) *Computer assisted instruction and intelligent tutoring systems. Establishing communication and collaboration*. Hillsdale, NJ: Lawrence Erlbaum.

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

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

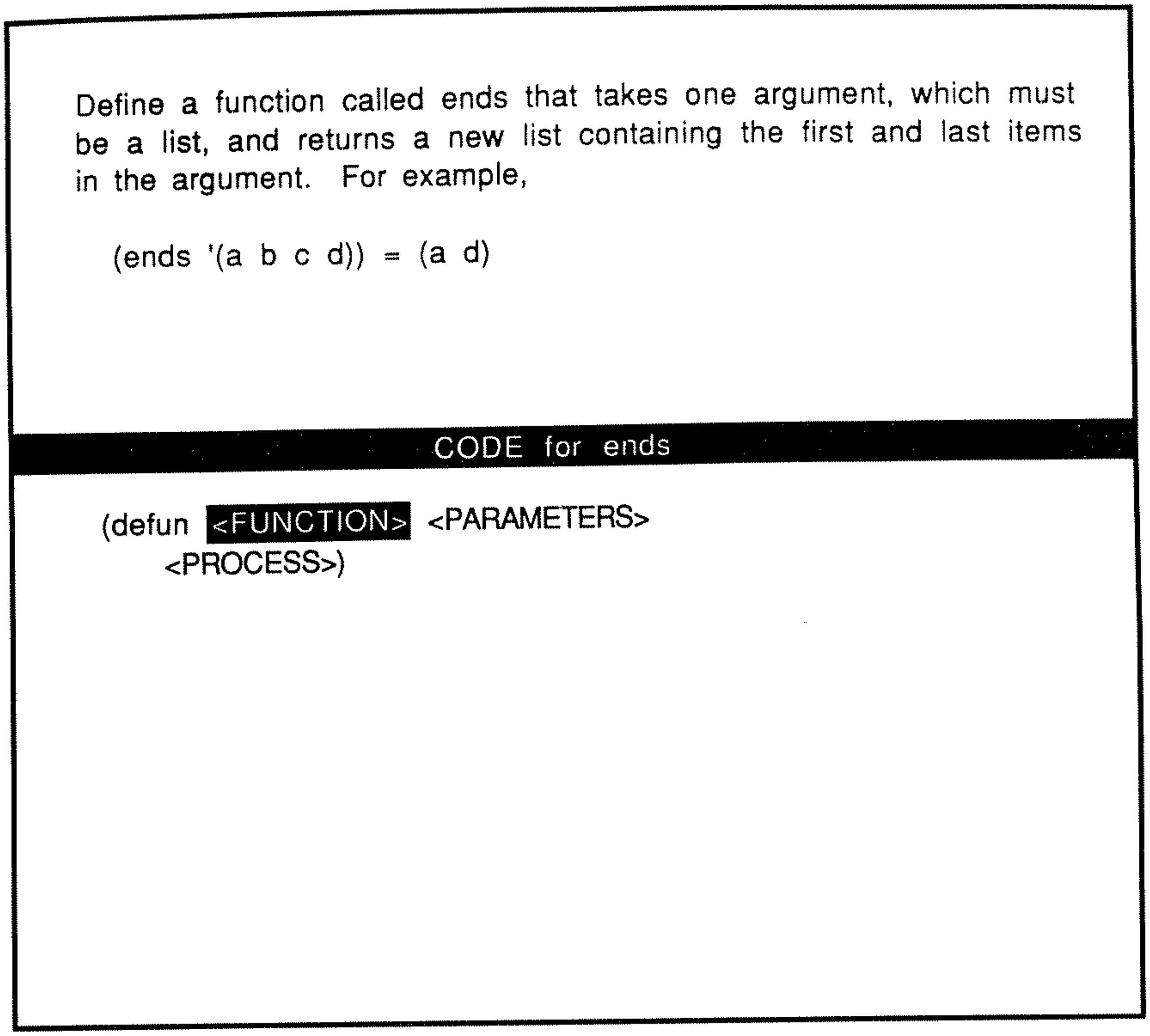
Lepper, M.R. and Chabay, R.W. (in press) Socializing the intelligent tutor: Bringing empathy to computer tutors. In H. Mandl and A. Lesgold (eds.) *Learning issues for intelligent tutoring systems*. New York: Springer.

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.

Wenger, E., (1987) *Artificial Intelligence in Tutoring Systems*. Morgan Kaufmann, Los Altos, CA.



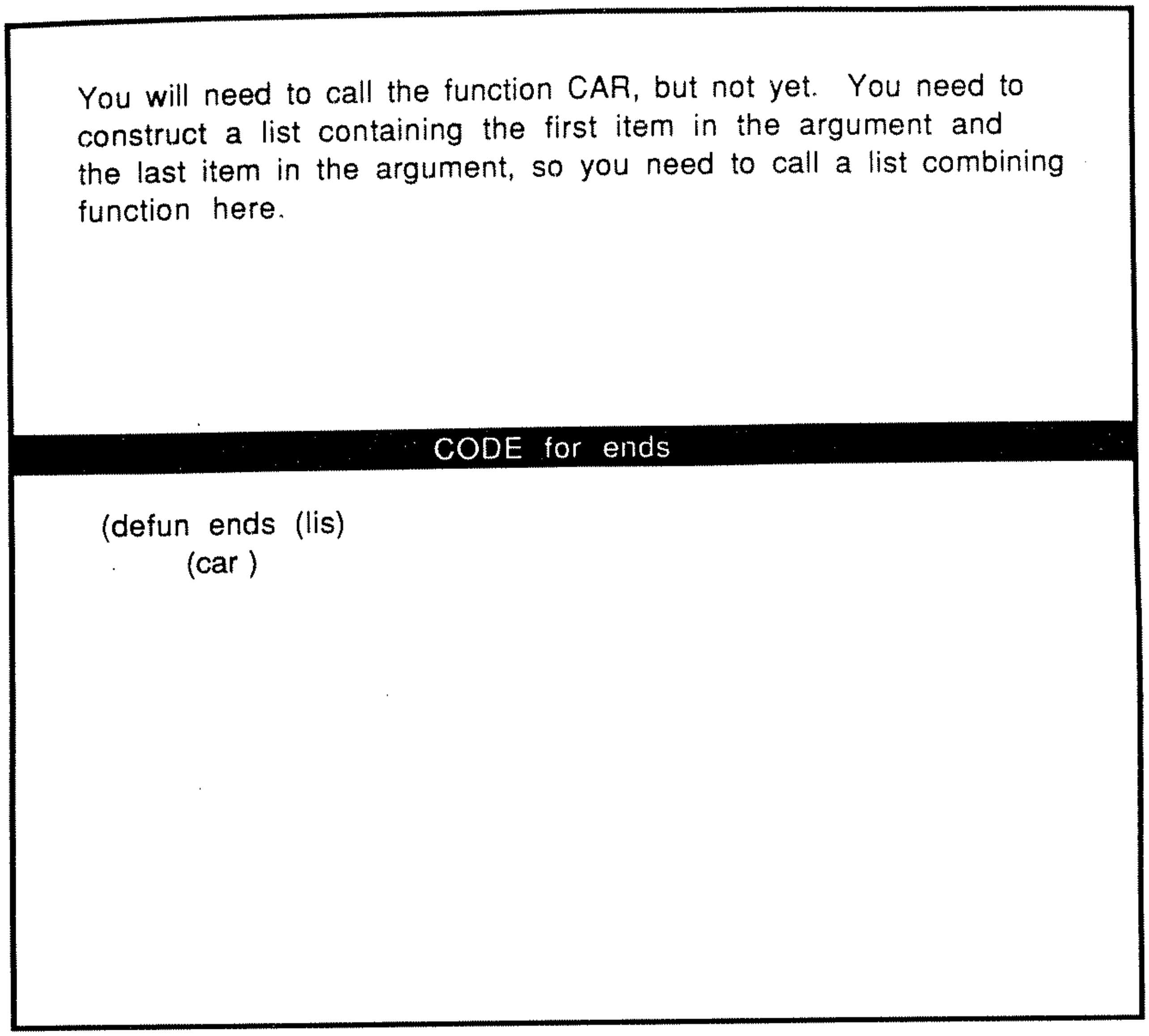
Define a function called ends that takes one argument, which must be a list, and returns a new list containing the first and last items in the argument. For example,

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

CODE for ends

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

FIGURE 1a

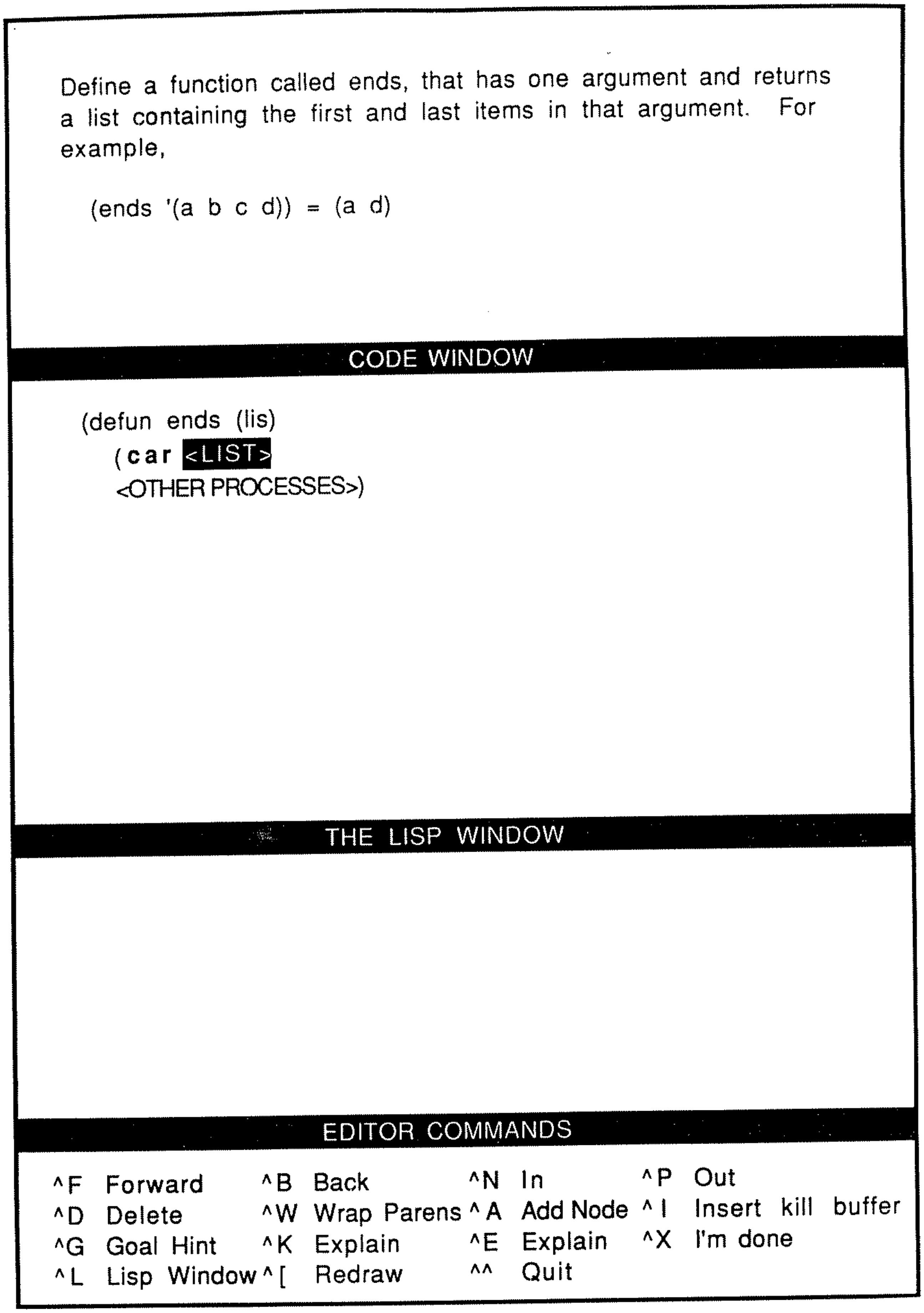


You will need to call the function CAR, but not yet. You need to construct a list containing the first item in the argument and the last item in the argument, so you need to call a list combining function here.

CODE for ends

```
(defun ends (lis)
       (car )
```

FIGURE 1b



Define a function called ends, that has one argument and returns a list containing the first and last items in that argument. For example,

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

```
(defun ends (lis)
   (car <LIST>
   <OTHER PROCESSES>)
```

^F	Forward	^B	Back	^N	In	^P	Out
^D	Delete	^W	Wrap Parens	^A	Add Node	^I	Insert kill buffer
^G	Goal Hint	^K	Explain	^E	Explain	^X	I'm done
^L	Lisp Window	^[	Redraw	^^	Quit		

Figure 2

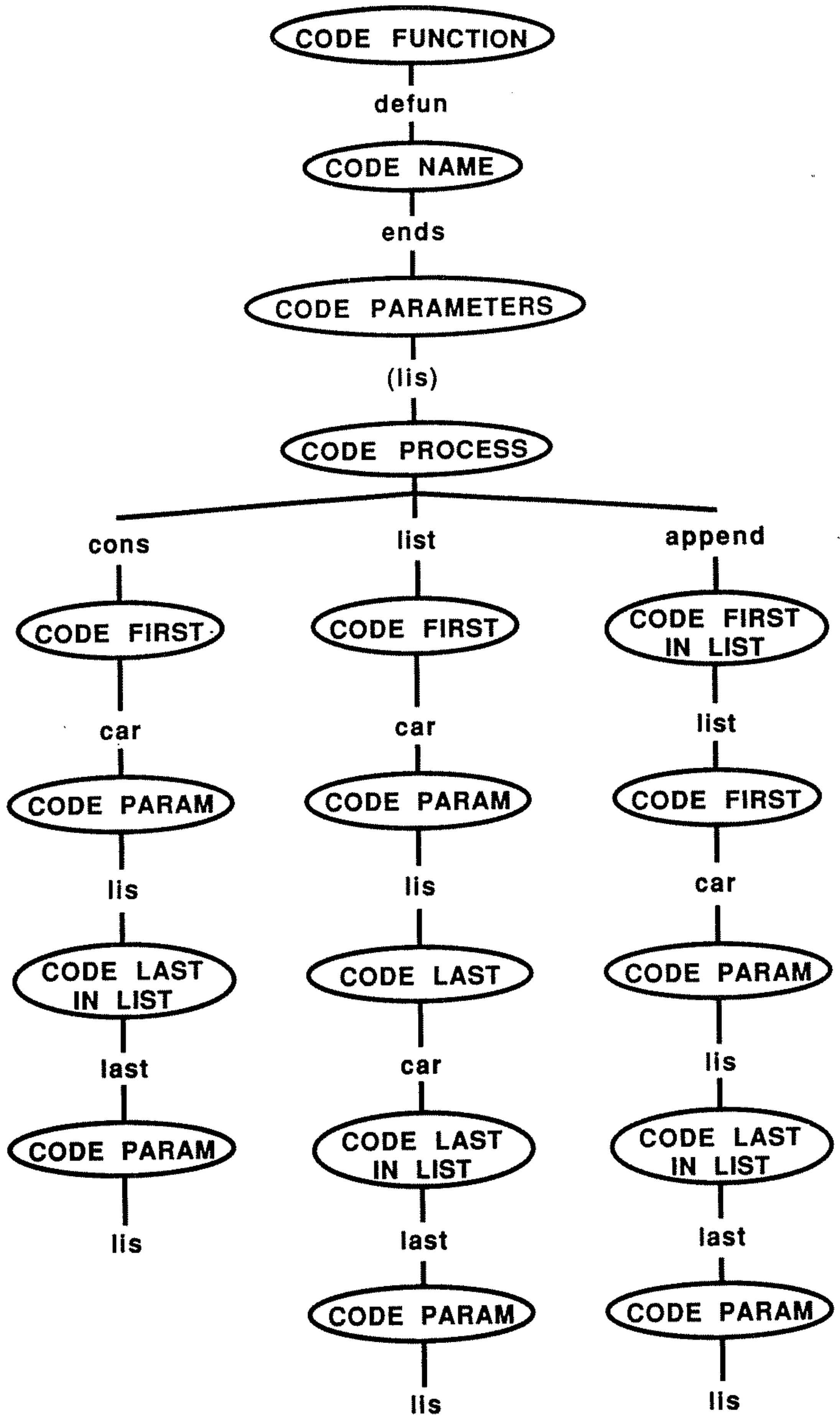

CODE FUNCTION
defun
CODE NAME
ends
CODE PARAMETERS
(lis)
CODE PROCESS
cons
list
append
CODE FIRST
car
CODE PARAM
lis
CODE LAST IN LIST
last
CODE PARAM
lis
CODE FIRST
car
CODE PARAM
lis
CODE LAST
car
CODE LAST IN LIST
last
CODE PARAM
lis
CODE FIRST IN LIST
list
CODE FIRST
car
CODE PARAM
lis
CODE LAST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

FIGURE 3