An Interactive Computer-based Tutor for LISP[1]

Robert G. Farrell
John R. Anderson
Brian J. Reiser
Advanced Computer Tutoring Project
Department of Psychology, CMU
Pittsburgh, PA 15213 USA

[1]This research was supported by grants N00014-81-C-0335 and N00014-84-K-0064 from the Office of Naval Research

Abstract

This paper describes an intelligent computer-based tutor for LISP that incorporates some of the ingredients of good private tutoring. The tutor consists of a problem-solver that generates steps toward a solution and an advisor that compares the problem-solver's steps to the student's steps. Our system can interact with students in a number of different problem spaces for algorithm design and coding. The tutor reduces memory demands by displaying relevant contextual information and directs problem-solving by immediately intervening when a student generates an unacceptable partial answer. Initial experiments indicate that our tutor is approximately twice as effective as classroom instruction.

Introduction

Students have extreme difficulty learning their first programming language. This difficulty is magnified by the learning environment - a cold terminal, an unforgiving textbook, and an inaccessible teacher. The student may be entirely lost until an experienced student or teaching assistant volunteers their expertise. We estimate that private instruction is between two and four times as effective as classroom instruction. Students taught by private tutors learn both more quickly and more deeply than students in classrooms (McKendree, Reiser & Anderson, 1984). Our goal is to capture private tutors' expertise by constructing intelligent computer-based tutors that can interactively help students solve problems. We also want to test our theory of how people learn complex skills (Anderson, 1983) and more specifically how people learn to program (Anderson, Farrell, & Sauers, 1984).

In this paper, we describe an initial version of a computer-based LISP tutor that incorporates some of the ingredients of good private tutoring (Anderson, Boyle, Farrell, and Reiser, 1984). Students learn LISP with our tutor by first reading some short instructional material and then working through a series of problems. We plan to use our tutor to teach a 30 hour course in LISP which covers the basic structures and functions of LISP, function definition, conditionals and predicates, helping functions,

recursion, and iteration. We have currently implemented 18 hours of this instruction.

Our tutor works through the problems with the student interactively. It consists of a problem-solver and an advisor. We first describe how the problem-solver helps to interactively model students as they learn to program. We then describe the advisor and its tutoring strategy. Finally, we discuss three features of the tutor which we feel contribute to its effectiveness:

1. Use of different problem spaces to cover a broad range of programming behavior

2. Use of the graphic reminders to reduce the amount of information that a student must remember while programming

3. Use of immediate feedback to direct problem solving and reduce learning time.

Interactive Student Modelling

In previous work (Anderson, Farrell, & Sauers, 1984; Anderson, Pirolli, & Farrell; in press) we outlined a detailed theory of how students learn to program in LISP. We used GRAPES (Sauers & Farrell, 1982), a Goal-Restricted Production System, to model students at many different levels of performance. Our current work incorporates these models into a tutoring system that can interactively assess a student's knowledge during problem-solving.

A good human tutor can follow a student's problem solution, giving suggestions when the student makes an incorrect or non-optimal step or when the student is lost. Human tutors can give this type of tutorial assistance because they infer a model of the student's knowledge. We follow students' problem-solving through a similar process, called *interactive student modelling*. Our system continually monitors the student's progress and tries to assess the knowledge that the student must have in order to produce the given behavior. This knowledge is represented in the form of GRAPES production rules and goals. In addition, the tutor has a set of common "buggy" rules (Brown & Burton, 1978) and "buggy" goals that it can recognize. Interactive student modelling is achieved by inferring which rules and goals in the tutor's catalog could possibly produce the observed student behavior. Because of our detailed model, our system can convey the heuristic knowledge needed to solve a wide range of beginning programming problems.

The Tutoring System Control Structure

The LISP tutoring system consists of two major components: the problem-solver and the advisor. The problem solver consists of the GRAPES interpreter, novice model rules, and buggy or "mal" rules (Sleeman, 1982). The advisor is a production system interpreter much like GRAPES; it also provides a tutorial strategy and many facilities for creating tutoring sessions including graphics, text generation, and parsing. The tutor interpreter executes tutoring rules (*t-rules*) (Clancey, 1982) which contain patterns for creating explanations and menu entries.

A problem input to the tutor consists of a small data base of facts and an initial goal. The problem-solver tries to decompose the initial goal into easier subgoals, using the novice model rules and the facts in the data base. The student also tries to decompose the goal using a goal description generated by the tutor and the facts that appeared in the instruction booklet given before each problem session.

The advisor matches the problem-solver's next step against the student's next step and categorizes the student's response. Since there are many ways to solve any interesting programming problem, the problem-solver must generate a list of possible correct and incorrect actions and the advisor matches all of them against the student's answer. If the student generates a correct step toward the solution, the advisor directs the problem-solver to execute the rule corresponding to the student's step. If the student displays a bug, the system generates text explaining why the answer was incorrect. If the student fails to produce a correct answer after a number of trials, the system provides the best answer and generates an explanation of why the answer was the best choice.

Interacting with the System

The tutor's top window displays explanations, hints, and querys, the "code" window provides a structured editing environment for entering LISP code, and the bottom window displays the problem statement and any planning information or reminders. The tutor brings the student's attention to new information by flashing the appropriate window.

The tutor interprets each keystroke typed by the student and gives immediate feedback about correct and incorrect steps. At any time, the student can press a *clarify* key to get additional help about the problem or an *info* key to access a tree-structured help facility.

When a student types a function name, place holders

appear for the function arguments. The structured editor allows the student to code these arguments in any order. A spelling corrector and parentheses checker help the student enter code in a graceful manner.

Using Different Problem Spaces

Producing a program in any language consists of a medley of algorithm design, coding, and debugging (Brooks, 1977). A good human tutor can converse with the student in a variety of problem spaces. In this section we describe how our tutor communicates in the problem-spaces involved in algorithm design and coding. We are not concerned with debugging since the tutor never allows the student to produce a final solution that is incorrect. Our tutor currently utilizes three problem spaces for coding and algorithm design: a coding space, a means-ends analysis space, and a problem decomposition space.

The LISP Coding Problem Space

The LISP coding problem space is used in normal problem-solving. The student enters LISP code in a syntax-based editor. The hierarchical structure of the solution is represented by symbols to be expanded. The student's plan for the solution is represented by the structure and name of the symbols. For example:

```
(defun subset (list1 list2)
    (cond <TERMINATING-CASE>
          <RECURSIVE-CASE>
    )
)
```

illustrates that the student is using CDR recursion to solve the *subset* problem. The student can choose to code either the terminating case or recursive case first.

CDR recursion is a programming plan(Soloway, 1980; Rich & Shrobe, 1978) well known to expert LISP programmers but difficult for novices to induce on their own. Part of the utility of a programming tutor is to introduce powerful programming techniques like CDR recursion during problem-solving (Anderson, Boyle, Farrell, and Reiser, 1984).

The Means-Ends Analysis Problem Space

The means-ends analysis (Newell & Simon, 1972) space is used when the student is having trouble producing code for a problem that can be characterized by a set of successive operations on an example. In this problem space, the student can develop a solution by supplying LISP operators that reduce differences between the current state and the goal state in the

example.

Figure 1 illustrates a sample interaction with the tutor during means-ends analysis. The student is trying to produce some code to get all but the last element of a list. Menus list both correct and incorrect ways of performing an operation. The menu entries are generated from patterns associated with both good and buggy rules in the novice model. Once the student picks a correct entry, he or she must provide a function that will perform the operation described. The tutor separately assesses the student's knowledge of what operations must be performed and their ability to implement those operations in LISP.

Problem Decomposition

The problem decomposition space is used when the student is having trouble producing code for a problem that can be easily decomposed into pieces. The conceptual pieces of the problem may not correspond exactly to the form of the code. The system displays a menu of possible decompositions of the problem and the student must pick the correct answer. The tutor makes sure that the student actually implements their algorithm when finally producing the LISP code. Again, the tutor separately assesses the students' ability to derive the algorithm from their ability to implement the algorithm.

Reducing Memory Demands

Solving programming problems requires holding a great deal of requisite information in a mental working memory. This requisite information consists of unsolved goals, partial products of calculations, and descriptions of LISP functions. We estimate that half of students' time spent solving programming problems is spent recovering from working memory failures (Anderson, Farrell, & Sauers, 1984). Anderson and Jeffries (1984) demonstrated that working memory load in one part of a task causes students to err on other parts of the task, even if those parts are logically unrelated. Therefore, it is extremely important that tutors keep working memory load to a minimum.

One way that the tutor keeps working memory load low is by displaying descriptions of the student's goals on the terminal screen. The student's goals are represented in GRAPES and the tutoring system uses this representation to generate english descriptions. The tutor displays the overall goal and the current goal as well as the goals along the shortest path between these two goals. For example, if the student is solving for the second argument to **lessp** in the following code:

```
(defun lessoreqp (x y)
    (or (equal x y)
        (lessp x _)))
```

then the system would display the following goal context:

```
Write a function called lessoreqp.
Test if x is less than or equal to y.
Test if x is less than y.
Write code for the second argument to lessp.
```

Students solving LISP problems also have trouble remembering partial results. In our LISP tutor, any calculations that the student performs on examples are displayed in a window for later reference. In addition, the partially-correct code is always displayed on the screen.

Immediate Feedback

Novices spend a large amount of time exploring incorrect solutions that result in little learning. A good human tutor directs the student toward correct answers, while still letting the student learn from mistakes Lewis and Anderson (1984) have shown that students learn more slowly when they are given delayed feedback about their erroneous applications of operators. In our studies of LISP learning (Anderson, Farrell, & Sauers, 1982), our subjects spent more than half of their time exploring wrong paths or recovering from erroneous steps.

Our tutor monitors the student with every keystroke, giving immediate feedback when it detects an error. Since the student never strays more than one step off of a correct solution path, our tutor can model the student in great detail. When the student makes an error, an explanation is generated from a pattern stored with the buggy rule and a query is generated from the student's current goal, directing the student toward a correct answer.

Our tutor cannot generate immediate feedback when the student's behavior does not disambiguate which goal he or she is pursuing. The tutor is silent until it can disambiguate the goal. If the student is generating an especially ambiguous piece of code, the tutor may display a menu of goals and ask the student to decide among them. Once the student's goal is known, the tutor can then intervene with tutorial assistance.

Conclusion

Our computer-based tutor for LISP incorporates some abilities of good human tutors. Our system can interact with students in a number of different problem spaces for algorithm design and coding. The tutor reduces memory demands by displaying relevant contextual information and directs problem-

solving by immediately intervening when a student generates an unacceptable partial answer. Our system interactively models the student by updating a set of production rules. These production rules also serve as a novice model that follows the student as he or she solves the problem. We performed an evaluation study on our tutor (McKendree, Reiser, & Anderson, 1984) which confirms our belief that it is about twice as effective as classroom instruction. We plan to further test the tutor's pedagogical effectiveness by automating a 30 hour LISP course taught in the fall of 1984.

References

Anderson, J.R. *The Architecture of Cognition*. Cambridge, MA: Harvard University Press 1983.

Anderson, J.R., Farrell, R., and Sauers, R. Learning to program in LISP. *Cognitive Science*, 1984, , . in press.

Anderson, J.R., Pirolli, P. and Farrell, R. Learning recursive programming. In forthcoming book edited by Chi, Farr, & Glaser.

Anderson, J.R., Boyle, C. F., Farrell, R.G., and Reiser, B.J. Cognitive Principles in the Design of Computer Tutors. Paper submitted to the CACM.

Brooks, R.E. Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 1977, 9, 737-751.

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

Clancey, W. J. Tutoring rules for guiding a case method dialogue. In D. Sleeman and J.S. Brown (Eds.), *Intelligent Tutoring Systems*, New York: Academic Press, 1982.

Lewis, M. and Anderson, J.R. The role of feedback in discriminating problem-solving operators. Submitted.

McKendree, J., Reiser, B.J., and Anderson, J.R. Tutorial goals and strategies in the instruction of programming skills. Paper submitted to the 1984 conference of the Cognitive Science Society.

Newell, A. and Simon, H. *Human Problem Solving*. Englewood Cliffs, N.J.: Prentice-Hall 1972.

Rich, C. and Shrobe, H. Initial report of a LISP programmers' apprentice. *IEEE Trans. Soft. Eng.*, 1978, , 456-466.

Sauers, R., and Farrell, R. *GRAPES user's manual*. ONR Technical

Report ONR-82-3, Carnegie-Mellon University, 1982.

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

Sleeman, D. & Brown, J.S. (Eds.). *Intelligent Tutoring Systems.* New York: Academic Press 1982.

Soloway, E.M. From problems to programs via plans: The context and structure of knowledge for introductory LISP programming. COINS Technical Report 80-19, University of Massachusetts at Amherst, 1980.

Figure 1

What can we do to change (a b c d) into (a b c)?

PRESS:	IF YOU WANT TO:
1	remove d from (a b c d)
2	get each element and gather them into a list
3	have the tutor choose
4	get information about LISP

Menu choice: **1**

Very Good.

What function will remove just 1 element from a list and return the result?

Function name: **I don't know**

The function CDR will remove an element from a list. You will have to apply CDR to (a b c d) to remove d. Remember that in this case, d is the last element of the argument list. So, in order to apply CDR, you have to get d in the first position of the list.

The LISP Tutor Teaching with Means-Ends Analysis