Interactive Student Modelling in a Computer-based Lisp Tutor

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

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

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 a teaching assistant or more experienced student volunteers his or her expertise. We estimate that private instruction is somewhere between two and four times as effective as classroom instruction. Our research program is aimed at finding those aspects of private tutoring that can be implemented as a computer program, so that we can provide automated private tutoring to a large number of students.

In this paper we describe an initial version of a computer-based tutor for LISP that incorporates some of the ingredients of good human tutoring. We will first describe how we have applied our previous cognitive modelling work to the domain of intelligent tutoring. We then describe the structure of our tutoring system for LISP. We show how this tutor makes learning easier by conveying the problem-space structure, reducing working-memory demands, and directing problem-solving. Finally, we report the results of some initial evaluation studies with our tutor and give some future directions for our research.

Interactive Student Modelling

To interactively model a student, a tutoring system must continually recognize students' goals and their procedures to achieve those goals. We developed a Goal-Restricted Production System (GRAPES) to model how novices write LISP functions (Sauers and Farrell, 1982) to achieve programming goals. We have used GRAPES to construct an ideal model that incorporates all of the correct procedures that students might use in a particular tutoring session. During students' problem-solving, the tutoring system must discover which procedures or deviations from procedures a student is actually using. We also developed a **knowledge compilation** mechanism for GRAPES which accounts for the difference in performance between novices and advanced students (Anderson, 1983). Using knowledge compilation, we can adjust instruction according to the student's

learning rate.

Tutorial Interaction

Our LISP tutor consists of a domain-independent interpreter that incorporates tutoring strategies, a set of LISP programming rules for modelling the student, a set of tutorial rules that analyze student code and provide feedback, and various problems characterized by an initial goal and a problem statement.

Our tutoring system interacts with the student by first explaining a LISP problem-solving goal; the student reads the goal description and enters an answer that should achieve that goal. If the student's choice is acceptable, the tutor pursues the chosen path and generates more problem-solving goals. If the student's choice is unacceptable, the tutor explains why the choice was incorrect and permits the student to try again. If the student cannot generate a good answer, the system will explain the best possible response.

We plan to use our tutor to teach a short course in LISP, including basic structures and functions, function definition, conditionals and predicates, helping functions, recursion, and iteration.

Conveying Problem Space Structure

Producing a program in any language consists of a medley of algorithm design, coding, and debugging (Brooks). 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 we never allow the student to produce an final solution that is incorrect.

Our tutor currently utilizes four problem spaces for coding and algorithm design:

* 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 problem is represented by symbols to be expanded. For instance, (cons <1> <2>) tells the student that he or she can choose to produce code for either <1> or <2>.

* The means-ends analysis 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. The student produces a solution by supplying LISP operators that reduce differences between the current state and the goal state in the example.

* 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 system displays a menu of possible decompositions of the problem and

the student must pick a correct decomposition.

* The case analysis problem space is used when the student is having trouble producing code for a problem that has a decomposable input-output behavior. The student specifies an action for each input-output case and then produces code that achieves all of the actions.

Reducing Working Memory Demands

In previous work (Anderson, Farrell, and Sauers, 1983) we estimated that half of students' time spent solving programming problems is spent recovering from working memory failures. A good human tutor constantly reminds the student of the information necessary to solve the problem that the student is attempting. Our tutor reduces working memory demands in the following ways:

* The tutor always displays the problem statement in a separate window.

* The tutor displays the entire student answer and that portion of the answer that is correct so far.

* The tutor provides a tree-structured help facility that describes all of the LISP operators that the student has learned so far.

Directing Problem Solving

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 feedback about their erroneous applications of operators only after a delay.

Our tutor directs problem-solving by first focusing the student on a single problem-solving goal. The tutor formulates a query that directs the student to supply a particular piece of LISP code to do a specific task. The tutor can supply examples to illustrate a sample input or output to the code it is requesting.

The tutor keeps the student from generating incorrect solutions by providing immediate feedback on errors. If the student cannot solve a problem subtask after a small number of tries, the tutor explains the best answer. Both explanations and queries are generated by instantiating natural language patterns associated with each rule or goal. The resulting english is modified by a set of transformational rewrite rules to enhance readibility.

Conclusion

We have constructed a computer-based tutor for LISP based upon some abilities of good human tutors. The tutor can interact with the student in a number of different problem spaces, corresponding to different student solution strategies. Our tutor reduces working memory demands by use of pop-up windows and directed dialogue. It also directs problem-solving by immediately intervening when a student generates an unacceptable answer. Our system interactively models the student by updating a set of production rules. We have performed an evaluation study on our tutor which confirms our belief that our tutor is about twice as effective as classroom instruction, but is only half as effective as a good private tutor. We plan to further test the pedagogical effectiveness of our tutor by automating a short LISP course taught in the fall of 1984.