Designing and Using Human-Computer Interfaces and Knowledge Based Systems
edited by G. Salvendy and M.J. Smith
Elsevier Science Publishers B.V., Amsterdam, 1989–Printed in the Netherlands

REASONING AND EXPLANATION IN AN INTELLIGENT TUTOR FOR PROGRAMMING

Michael Ranney and Brian J. Reiser
Princeton University, Cognitive Science Laboratory, 221 Nassau Street, Princeton, NJ 08542

ABSTRACT

This paper describes GIL, the graphical tutor for LISP programming, according to criteria that are emerging from the field of intelligent tutoring. We begin with a brief overview of GIL's explanatory and visual characteristics, then discuss ways in which the system reduces and obviates many difficulties in learning to program. Finally, GIL's present and future performance is considered with respect to several problematic issues in the design of intelligent tutors.

INTRODUCTION

Over the past few years, various research groups have been developing intelligent tutoring systems, including several designed to teach computer programming (e.g., Anderson & Reiser, 1985; Bonar & Cunningham, 1988). Even more recently, attention has been focused on just what makes such tutoring technology successful (Collins, in press), particularly given the considerable difficulties inherent in learning to program (du Boulay, 1988). In this paper, we apply some of these criteria to GIL, our intelligent tutor for generating LISP programs.

GIL: GRAPHICAL INSTRUCTION IN LISP

Since the system is described more extensively elsewhere (Reiser, Kimberg, Lovett, & Ranney, in press), this section represents only a very quick overview of GIL's essential features. Using a problem solver, an explainer, a response manager, and a graphical interface, GIL tutors students in writing simple LISP programs. In doing so, the system (1) explains its own reasoning via a set of plans and problem solving rules and (2) employs a visual representation that facilitates students' program generation.

Problem solving rules that yield generative explanations

GIL constructs explanatory feedback and hints directly from the content of the problem solving rules -- rules that trace the student's behavior as the solution progresses. Such explanations are hence generative, in contrast with other rule-based programming tutors that use "canned" English text that is handcrafted for each new situation (e.g., Anderson, Boyle, & Reiser, 1985). This is possible because GIL's problem solver makes casual knowledge about useful programming operations explicit.

To guide the student along novel chains of reasoning, GIL's problem solver includes rules and plans that understand how each LISP step transforms the data; the system not only encodes the results of sequences of operations, it can also communicate the desirable aspects of such sequences. This knowledge is represented as properties of each step's input and output, allowing the problem solver to explain why a proposed step is effective in a given situation.

GIL's explainer uses these same plans and rules to construct explanations that respond to either poor strategies, legal errors, or requests for hints. When a student solicits a hint, GIL finds a rule that best continues the student's progress. The tutor then proposes the problem solving rule's associated step, explaining its choice in terms of the properties of the step's input and output. These properties are also used when GIL explains discrepancies between a near-miss error and a comparable rule from the problem solver. Thus, GIL does not need a catalog of buggy rules, as explanations for errors are dynamically constructed via such comparisons. Two basic sorts of errors are explained: legal errors, in which a step's input and function do not result in the indicated output, and strategic errors, in which a legal step is not useful. Figure 1 illustrates part of GIL's explanation of a legal error.

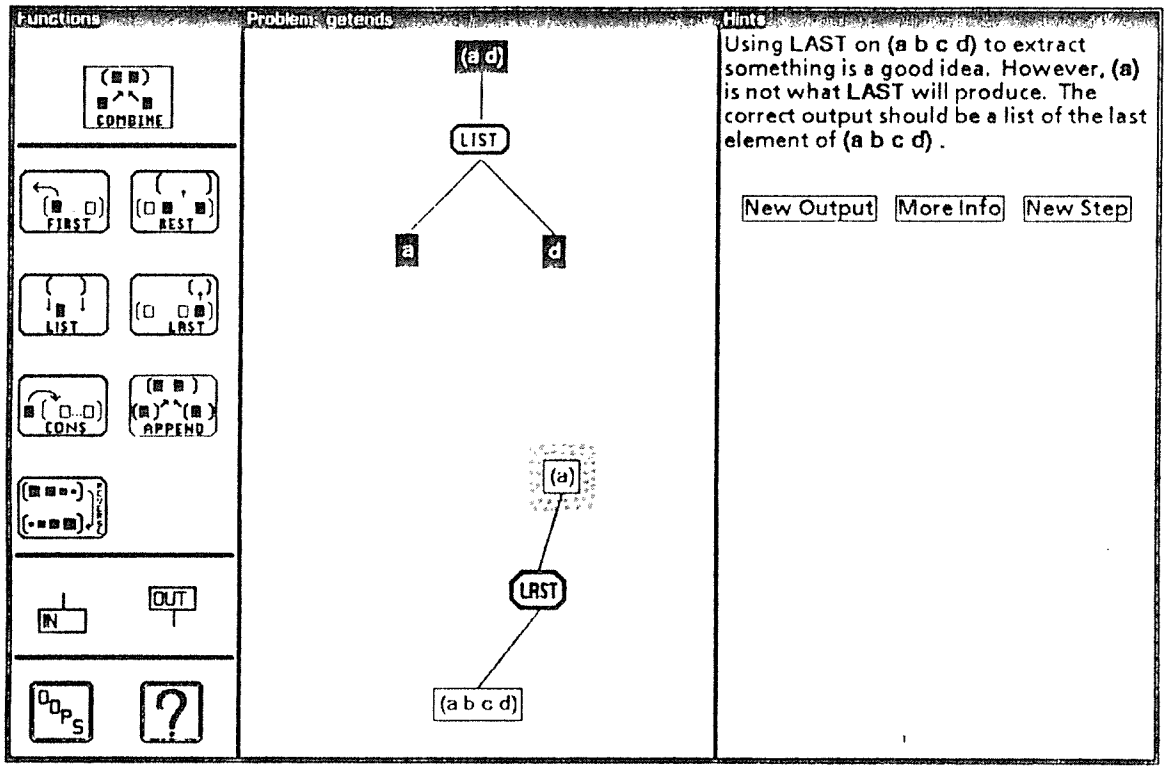

Fig. 1. A forward-working legal (output) error, with one level of explanation and three choices.

A Graphical Representation of Programming

With its graphical interface, students construct a program in GIL by connecting objects that represent programming constructs. The resulting graph stands in sharp contrast with standard, nested, text-based, LISP functions. Each step involves selecting a LISP function from the menu, then specifying its input and output by typing or clicking. The graph makes each of the program's intermediate products explicit, as the student specifies how the output for one function becomes the input for another. Thus, students indicate chains of transformations, rather than a mere sequence of operations. Intermediate products allow for GIL's stepwise character, and tutoring is provided based on this unit of analysis. Figure 1 shows two steps for the problem *getends* (in which the goal is to make a list from the first and last elements of an input list). The correct *list* step is a backward step that works from the goal, *(a d)*, in black, toward the input, *(a b c d)*, in white. The incorrect *last* step is a forward step that creates the intermediate product *d*. A

complete program would eventually link the subgoals *a* and *d* to the original input, *(a b c d)*.

The graph uses a structure that mirrors the planning of a program, with reasoning chains represented as branches that link the original input and the ultimate goal. GIL's explicit intermediate products help students understand the developing algorithm. Most importantly, perhaps, the visual interface supports multidirectional planning and problem solving. Students can work both forward and backward, and can alternate laterally from one branch to another (i.e., rightward or leftward). GIL's problem solver contains rules relevant to all of these options, providing a very powerful basis for the explanations needed for hints and error feedback.

HOW DOES GIL REDUCE PROGRAMMING DIFFICULTIES?

In describing the difficulties involved in learning computer programming, du Boulay (1988) provides a partial, interrelated, list of a programmer's activities. In designing GIL, we have addressed about five of these eight activities. As du Boulay casts an appropriately wide net, let us first make it clear what activities GIL does *not* tutor. Since GIL provides students with well-specified problems, it does not facilitate the establishment of a problem, i.e., problem finding and the initial stages of problem representation. Furthermore, since GIL's present curriculum is geared toward naive programmers and simple problems, professional activities like documenting, maintaining, and extending existing code are not addressed. Now, let us consider how the tutor reduces some of the more salient difficulties of the other programming activities.

The program as process

Both Collins (in press) and du Boulay (1988) note that it is difficult to generate solutions that represent the product, rather than the process, of problem solving. In this respect, standard text-based programs are much like two-column proofs in geometry (Anderson et al., 1985); they hardly provide a sense of the reasoning that generates them. Students who similarly believe that complete programs are usually generated line by line are mistaken, and this misconception was the primary motive behind GIL's graphical nature: to change the programming environment of LISP such that the product and the process are more congruent (cf. Collins & Brown, 1988). Such changes have several desirable outcomes. The graphical structure makes flow of control much more explicit, the intermediate products reduce the mental load involving the results of data transformations, and the use of concrete input eliminates occasional textual confusions between variables and functions. Thus, GIL makes subjects' tacit reasoning more visible.

To some extent, GIL's direct-manipulation character makes it a different language than LISP. However, like Bonar and Cunningham's Bridge programming tutor (1988), GIL can be viewed as an intermediate representation between natural human planning and a text-based programming language (du Boulay, 1988). The spatial reification of such planning processes aids a student in reflecting about a problem's solution, which is easier to construct when the communication medium is congruent with the individual's reasoning. Thus, like Anderson et al.'s (1985) geometry tutor, GIL abstracts the problem solving of programming into a cogent problem space. When combined with GIL's capacity for self-explanation and multidirectional problem solving, the tutor serves as an environment in which the apprentice programmer can develop some of the expert's skills (cf. Collins, in press).

Problem decomposition

A theme present in several of du Boulay's programming activities (e.g., problem comprehension, method specification, coding and compiling, etc.) is that of decomposition.

Since breaking a problem into its parts is central to productive programming, difficulties with this skill are quite serious. GIL aids students' problem decomposition in several ways: First, the interface's explicit visual constructs make decomposition easier, as the students can always see the current problem state; the salient function icons remind one of what operations are available, while the concrete data elements help suggest which transformations will be necessary to achieve the goal. Next, GIL's explainer provides hints that are based on plans, matching problem solving rules, and a knowledge base that explicitly represents compositional considerations in its goal structure. In fact, GIL often recommends decompositional steps when students ask for hints. Figure 2 shows such a situation at the beginning of problem *getends*, in which two levels of information have been requested. Note that the first level offers a rather general specification of the decompositional method; the second level describes an appropriate coding, concretely translating the first level into a specific backward step, making explicit a useful function and its input. (The suggested step is actually the backward step illustrated in Figure 1.)

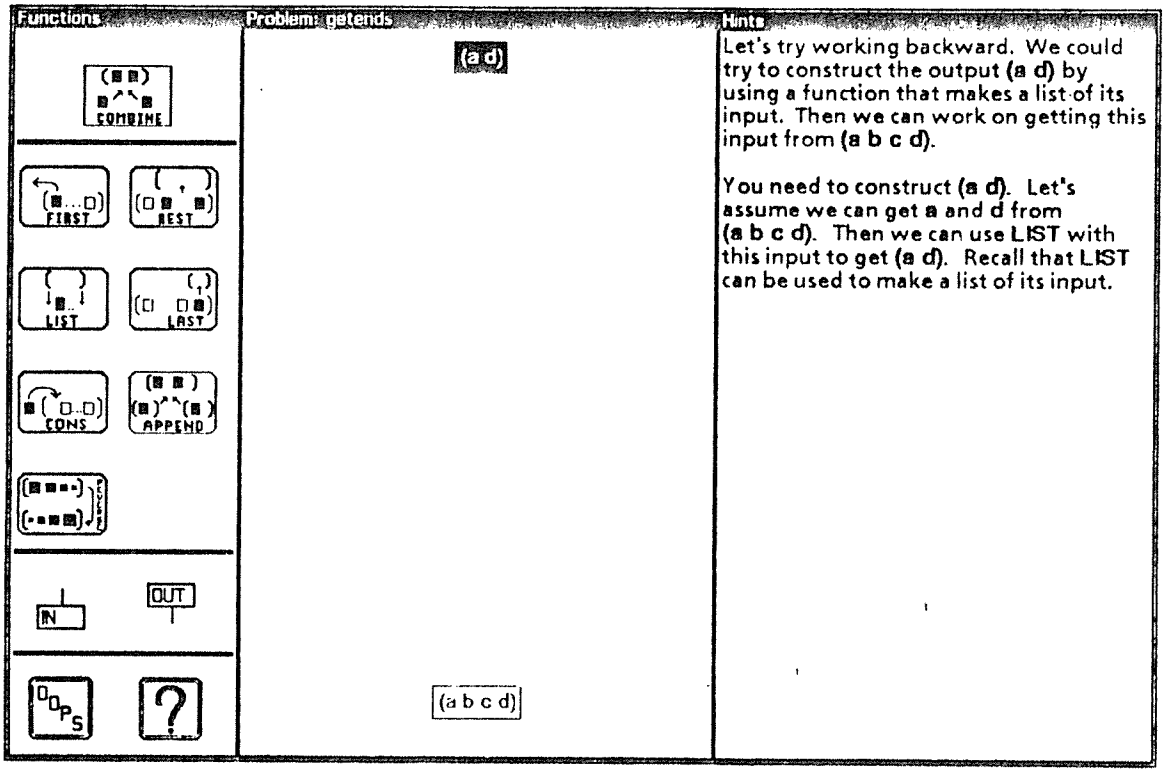

Fig. 2. Two decompositional hints provided at the start of problem *getends*.

Interpreting output and debugging

It is generally considered rather difficult to comprehend and debug a complete program (du Boulay, 1988). GIL's environment obviates this difficulty by imposing its stepwise program development, which provides for a kind of ad hoc debugging (in contrast to the post hoc analyses of Johnson & Soloway, 1987). However, the system's explanations can also provide a global perspective, including multi-step plans and feedback that suggest alternative approaches. In general, the stepwise constraint means that GIL's output is quite interpretable. The two levels of error feedback are usually easily understood in the local context, and problematic constructs are visually highlighted by placing gray boxes around them (as in Figure 1 above). Thus, GIL

severely limits the potential for student floundering due to uninterpretable program behavior. There are no hard-to-parse "compiler error messages," due to the tutor's model-tracing and explainer capabilities. There are also few "run-time surprises" (du Boulay, 1988) since each "run" involves a single function, corresponding to a single problem solving step. Both legal and strategic errors are quickly caught, so thrashing, in which students move about a problem space without reducing the distance to the goal, is minimized. Although not as authoritarian as the CMU LISP Tutor (Reiser, Anderson, & Farrell, 1985), GIL's suppression of such floundering is based on a similar instructional philosophy (and one that human tutors generally share) -- that help should be provided promptly at critical moments.

Different perspectives and tutor flexibility

Both Collins (in press) and du Boulay (1988) point out that another difficult component of learning involves the need to consider alternative paths to a solution. In programming, both student and tutor often neglect this facet of reflection, focusing only on the development of a single working program. GIL's reasoning allows alternative solution paths in a variety of ways, and the system can be configured to provide even more support for students' reflective processes.

First, as a byproduct of its generative explanatory character, GIL can offer as many perspectives as there are matching problem solving rules (not to mention perspectives that result when matched rules are steps in multi-step plans). Since the tutor is multidirectional, even simple problems may provide a handful of loci for the next step, with several possible operations at each locus. Furthermore, as GIL is capable of generating all subsequent solution paths from any legal problem state, it is a simple extension of the current system to include an intervention that explicitly contrasts the student's solution with more elegant or efficient variants. This facility would complement current efforts to get students to reflect on alternative solutions.

Because of GIL's multidirectional character, some legal steps are ambiguous with respect to why they were taken. The system must then maintain multiple perspectives regarding the active goals that such a step invokes, pending disambiguating steps. This makes subsequent error feedback more tricky, as sometimes it is unclear as to which goal a subject was trying to achieve by taking the flawed step -- a difficulty common to some other formal domains, like geometry.

GIL also has enough flexibility to provide some backtracking help when students discover a new plan. In particular, when students try to implement the new plan before eliminating vestiges of the old plan, the tutor lays out a set of appropriate options. Because geometry proofs are not (temporally) interpreted in the same way that computer programs are, unlike the Geometry Tutor, GIL does not permit dead ends in its graph (cf. Anderson et al., 1985). Figure 3 illustrates the options provided: The student can either continue along the previously committed path or delete that path in order to pursue the new one. (Note that students often change strategies, delete partial solutions, and implement the new code without ever seeing an error message.)

Collins (in press) suggests that computer coaches can offer "new eyeglasses" for students, providing new terms and concepts about problem solving. GIL offers new perspectives in several ways, with many of them driven by the interaction between the system's interface and explainer, such that new distinctions are both graphically illustrated and textually described to students. Among these distinctions are backward/forward and leftward/rightward problem solving, as well as the differences between data and functions. The result is a tutor that can offer interventions based upon a number of levels and types of knowledge (cf. du Boulay, 1988); GIL

can provide several different types of feedback that deal with the interaction of a program's topography and its conceptual structure.

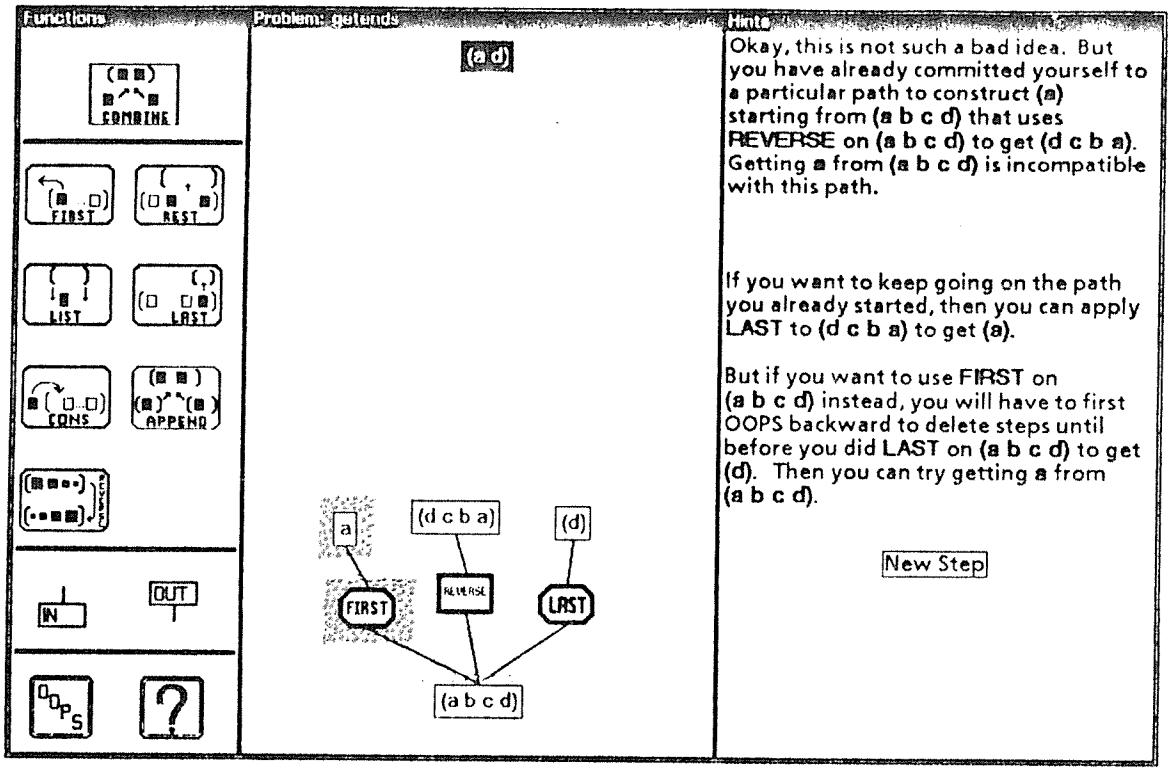

Fig. 3. GIL diagnoses a redundant step, providing two levels of feedback.

Program reliability

A final difficulty in learning to program relates to testing a solution to determine whether the boundary conditions of its functionality are appropriate. Such testing is presently unnecessary, since GIL's problem solver helps students construct solutions that conform to the constraints laid out in each unambiguous, well-structured, problem statement. In essence, the system does not accept steps which follow an overly specific plan, even though there is no library, as proposed by du Boulay (1988), that explicitly recognizes such "buggy plans." (For instance, the library would recognize that the LISP code *(cons (first some-list) (rest (rest (rest (some-list)))))* is only functional for *getends*' problem statement when "some-list" has four elements.) This sort of catalog would allow GIL more sophistication in explaining why a particular step leads to a less general solution. We are concerned, however, that if such buggy plans begin to approximate the buggy rules that are characteristic of the CMU LISP Tutor, a proliferation of such plans may jeopardize GIL's overall, rule-driven, generative nature.

We are now developing a variant of GIL that would have its authoritarian character dramatically reduced by silencing its explainer at times -- and allowing the student to continue on after an error has been made. In this way, GIL would become an environment for program exploration, in which the student could choose to invoke the problem solver and explainer whenever they were needed to facilitate the individual's hypothesis testing (Collins, in press). In fact, while GIL presently allows only the creation of general solutions, it can readily be modified to permit more program testing: At present, the tutor focuses on concrete examples, rather than

programs that employ input variables; we plan to change GIL's interface to allow students to vary a program's input, hence testing its reliability over a range of concrete examples (and demonstrating that the input can be variable).

POTENTIAL PITFALLS IN TUTORING PROGRAMMING

Intelligent tutoring, and computer-based instruction in general, often suffers by contrast with human instructors. For instance, du Boulay (1988) notes that expert teachers have a much greater breadth of knowledge than do intelligent tutoring systems. This will probably be true for some time. However, the contrast is more striking for programming tutors that employ buggy rules with canned English and text-based formalisms than for a system like GIL, with its generative explanations and graphical interface. On one hand, generative explanations rely on much the same information that human experts use when describing a situation. On the other hand, the tutor's multidirectional environment offers so much freedom that an expert human tutor who watched a student interacting with the graphical interface would probably have as much difficulty discerning the student's intentions as GIL does. (This would be especially true prior to the completion of a step, e.g., before the links between a function and its data are specified.)

Preliminary results (e.g., Reiser, Ranney, Lovett, & Kimberg, in press), both from students working with GIL and with human tutors (in a standard, text-based environment), support this hypothesis of comparable diagnosis. For instance, students working with GIL take about as much time as human-tutored LISP students do when covering comparable material. Furthermore, analyses of error rates and verbal protocols indicate that both GIL and human tutors intervene often.

Another contrast by which intelligent tutors suffer relates to the pruning of explanations. Human tutors can cut right to the heart of an issue, tailoring their feedback to focus on a critical aspect (du Boulay, 1988). The result is often a few cogent syllables, e.g., "That's not a list." Although some of this advantage is due to the broader communication bandwidth enjoyed by human tutors, much of it is driven by a strong understanding of redundancy. GIL incorporates some of this knowledge by representing entailment relations among clauses, such that its explainer can reason about them and avoid redundancies when comparing a flawed step to a desirable step. For example, this feature eliminates the redundant second clause of the following error feedback: "This output is not a list, and it does not have the proper first element." In the future, we plan to further focus such explanations by using a subject's individualized student model to constrain GIL's feedback and hints.

A final pitfall for intelligent tutoring systems concerns the start-up costs of their usage. As du Boulay (1988) justly warns, tutors can easily become unwieldy by providing too much flexibility and too many features. There is always the danger of diminishing returns, in which it could take as much time to learn the uses for a rather arbitrary set of windows and icons as the tutor saves in more expedient learning. Happily, this is not (at least not *yet*) the case with GIL, as students are briefed on its interface with a ten-minute demonstration that yields a much more dramatic saving in learning time. Whether this will continue to be the case as the GIL curriculum grows remains to be seen. It it our hope, however, to reduce potential unwieldiness by activating new features and functions only as they are introduced into that curriculum.

CONCLUSIONS

GIL is a hybrid system when considered in relation to the classes du Boulay (1988) lays out. Although primarily a tutor, it has local debugging capabilities and contains a graphical programming environment that dramatically supports program generation. The product, when compared to standard LISP, can almost be considered a new programming language, given that the fundamental representation (i.e., graphical vs. textual) and the unit of analysis (i.e., a chained step vs. nested function-calls) are so different. Either way, GIL offers much promise as a programming coach. It has specifically been designed to reduce some of the more difficult activities of programming, while avoiding some of the pitfalls to which intelligent tutoring systems often fall prey.

Acknowledgments

We are grateful to John Connelly, Jody Gevins, Patricia Friedmann, Eric Ho, Daniel Kimberg, Marsha Lovett, and Antonio Romero for assistance in programming GIL. The research reported here was supported in part by contract MDA903-87-K-0652 from the Army Research Institute and by a research grant from the James S. McDonnell Foundation to Princeton University. The research was performed using equipment donated to Princeton University by the Xerox Corporation University Grant Program. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Army or the McDonnell Foundation.

REFERENCES

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

Anderson, J.R., and Reiser, B.J., 1985. The LISP tutor. Byte, 10: 159-175.

Bonar, J. and Cunningham, R., 1988. Bridge: Tutoring the Programming Process. In: J. Psotka, L.D. Massey, and S.A. Mutter (Eds.), Intelligent Tutoring Systems: Lessons Learned. Erlbaum, Hillsdale, NJ.

Collins, A., in press. Cognitive Apprenticeship and Instructional Technology. To appear in: B.F. Jones and L. Idol (Eds.), Dimensions of Thinking and Cognitive Instruction. Erlbaum, Hillsdale, NJ.

Collins, A., and Brown, J.S., 1988. The Computer as a Tool for Learning Through Reflection. In: H. Mandl and A. Lesgold (Eds.), Learning issues for intelligent tutoring systems. Springer-Verlag, New York.

du Boulay, B., 1988, November. Towards more versatile tutors for programming: Some speculative comments. NATO Workshop on Educational Technology, Milton Keynes, UK.

Johnson, W.L., and Soloway, E., 1987. PROUST: An Automatic Debugger for Pascal Programs. In: G. Kearsley (Ed.), Artificial Intelligence and Instruction: Applications and Methods. Addison-Wesley, Reading, MA.

Reiser, B.J., Anderson, J.R., and Farrell, R.G., 1985. Dynamic Student Modeling in an Intelligent Tutor for LISP Programming. Proceedings of the Ninth International Joint Conference on Artificial Intelligence. Los Angeles, CA.

Reiser, B.J., Kimberg, D.Y., Lovett, M.C., and Ranney, M., in press. Knowledge Representation and Explanation in GIL, an Intelligent Tutor for Programming. To appear in: J. Larkin, R. Chabay, and C. Scheftic (Eds.), Computer-Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration. Hillsdale, NJ: Erlbaum.

Reiser, B.J., Ranney, M., Lovett, M.C., and Kimberg, D.Y., in press. Facilitating Students' Reasoning with Causal Explanations and Visual Representations. To appear in: Proceedings of the Fourth International Conference on Artificial Intelligence and Education.