

Knowledge Base Compilation and the Language Design Game*

Warren Sack
131 Peyton Street, Santa Cruz CA 95060
email: wsack@mcimail.com

Abstract: The ProgramCritic is a system designed to analyze and critique students' computer programs. After analyzing a program, the ProgramCritic provides the student with a list of English-language comments detailing the strengths and weaknesses of the student's program. The foundation of the ProgramCritic's analytic abilities is a set of "knowledge bases" which describe a range of programming problems and the ways in which parts of the problem can be solved. Several other systems have been built by other researchers with a functionality similar to the ProgramCritic's; notable among them is PROUST [Johnson 1986]. Differences between the ProgramCritic and PROUST are described through a detailed explanation of how one might build a compiler for PROUST's knowledge base language. Shortcomings with PROUST's knowledge base language are pointed out one-by-one and for each shortcoming a fix is proposed. Integrating the proposed fixes together serves to explain the knowledge base language used in the ProgramCritic.

Introduction: Automatic Program Debuggers and Language Games

Recently there has been an explosion in new paradigms for research in computers and education. Among others, these have been advanced as "new" frameworks: "situated learning", "constructionism", "discovery learning", and "interactive learning environments." Perhaps however "paradigm" is too strong a term to describe the banners under which these small subgroups of the computer and education community function. In fact, the manner in which the recent "paradigms" have been introduced has more closely resembled a political process whereby manifestos are drafted and dissenters are excommunicated. These sorts of rhetoric games are visible in most disciplines -- even most sciences (see [Latour 1987] for some interesting examples) -- and it is not with distaste that I point them out now. Quite the contrary. I rather enjoy the lively debate the manifestos engender and will soon co-author one on constructivist technologies for education [Sack, Soloway, Weingrad and Guzdial (in preparation)].

The historian of science, Thomas Kuhn, might say that the field of computers and education is in a "pre-paradigm period:"

Throughout the pre-paradigm period when there is a multiplicity of competing schools, evidence of progress, except within schools, is very hard to find. This is the period described ... as one during which individuals practice science, but in which the results of their enterprise do not add up to science as we know it. And again, during periods of revolution when the fundamental tenets of a field are once more at issue, doubts are repeatedly expressed about the very possibility of continued progress if one or another of the opposed paradigms is adopted. [Kuhn 1962: 163]

In the 1930's the philosopher Ludwig Wittgenstein coined a term which might be of use to us in sorting out the multiple "pre-paradigms" of the computers and education field; a *language game* was what Wittgenstein eventually called the endeavors of philosophers (of whom he was one). Thus if I refer to the multiple "pre-paradigms" of computers and education as language

* Appears in *Intelligent Tutoring Systems, Second International Conference (Lecture Notes in Computer Science)* Claude Frasson, Gilles Gauthier, and Gordon McCalla (editors), (Berlin: Springer-Verlag, 1992).

games I do so not flippantly, but rather with the utmost respect and with a very solid and serious Wittgensteinian philosophical tradition behind me.

This paper will focus on some "progress" that I have made in one of the language games of the field of computers and education. To more exactly explain this "progress" I will compare my design for a recently implemented automatic program debugger for novices, the ProgramCritic [Sack and Bennett (patent pending)], with a system of similar functionality that is considered a milestone in the field of intelligent tutoring systems, PROUST [Johnson 1986]. Automatic program debuggers are, essentially, sophisticated parsers which can analyze a student's computer program, identify strengths and weaknesses in the student's work, and then present to the student a critique of their work in the form of a list of comments (written in a natural language, like English, for example). Good program debuggers can comment, not only on errors of syntax, but, more interestingly, on errors in a student's solution which are specific to the problem that the student is trying to solve. Addressing a student's non-syntactic errors requires that an automatic program debugger be provided with a description of the problem that a student is working on and, also, with a library of ways in which students often solve (or fail to solve) specific parts of the problem. These kinds of machine-readable, problem descriptions that are employed by "intelligent" program debuggers are often referred to as "knowledge bases."

In staking out PROUST's importance for the field of computers and education it has been claimed that

- *PROUST analyzes a program by constructing a model of the student's intentions and their realization.* [Johnson 1986: 15]; and,
- *... PROUST's explicit reconstruction of the programmer's intentions constitutes an important advance whose ramifications are not limited to computer programming.* [Wenger 1986: 251]

In showing why my system, the ProgramCritic, constitutes "progress" over PROUST I steer around the language game played by PROUST's designers which would claim some sort of meta-hermeneutical status for PROUST; i.e., I am not going to discuss the "interpretive" powers, or the (re)construction of "intentions", by either PROUST or the ProgramCritic. It is problematic to claim that the ProgramCritic is "better than" PROUST if one will not -- as I will not -- play the language game of the designers of the precedent system. After all, how shall I claim any "progress" if I do not claim that the ProgramCritic can "reconstruct intentions" better than PROUST can?

Difference *does* exist between technological artifacts (like automatic program debuggers), but any notion of "progress" is *purely fictitious*. However, fictions of "progress" are often framed within a recurrent narrative common to a field. I will explain the ProgramCritic's "superiority" over PROUST by referring to an "Occam's Razor" sort of story that is often heard in the field of artificial intelligence. This sort of story often runs like this: X writes a 10,000 line program for a dissertation to illustrate a theory about Q; Q is "proven to be true" by a variety of practitioners (often from related fields like psychology, or linguistics); Y shows how a program of 100 lines (and of equivalent functionality to X's original program) can be written by throwing out the original theory Q and replacing it with something much more mundane.

Let me list a couple of examples of "Occam Razor" like stories that have occurred in the literature of artificial intelligence. SAM [Schank and Riesbeck 1981] was a very complicated program constructed to show how SCRIPTs (i.e., a particular flavor of schemas which have appeared throughout the history of psychology ever since Kant invented them) explained human understanding of stories. SAM was constructed to "understand" stories using SCRIPTs. Now, in several introductory programming books, one can find a "rewrite" of SAM, with functionality equal to the original dissertation work, that is a trivial program one page in length (e.g., [Sterling and Shapiro 1986: 234]).

"Explanation-Based Learning"[Minton et al. 1989] has suffered a similar, yet still debated, fate. In their 1988 paper, van Harmelen and Bundy show how the supposedly complex, artificial intelligence technique of explanation-based generalization can be re-explained as plain-old partial evaluation (a technique used by many computer language designers). In van Harmelen and Bundy's re-description "explanations" become "proofs" and partial evaluation is illustrated by a tiny one-page program.

It is instructive to note that the people who are the best at telling "Occam's Razor" sorts of stories in the artificial intelligence literature are often computer programming language designers. By looking at, so-called, "knowledge structures" (like SCRIPTs and "explanations") with the critical eye of the language designer one often finds that "knowledge structures" look more like ordinary computer programs than some sort of mental entities which constitute "psychological reality." For the rest of this paper I will call this sort of critical examination of knowledge representation languages the *language design game*, because it is language designers who do it so well.

In what follows I play the language design game with entities used in PROUST called "goals, plans, and bugs." According to the designer of PROUST, PROUST uses a "knowledge" of "goals, plans, and bugs" to diagnose a student's "intentions." I will show why one can view "goals, plans, and bugs" as plain-old procedures. By rephrasing them in such a manner, I explain why of the 15,000 lines of Lisp code that it took to construct PROUST most was devoted to, what I would call, an unsuccessful attempt to implement a functionality equivalent to a Prolog interpreter. Using this insight I was able to duplicate PROUST's functionality in my system, the ProgramCritic, with about 100 lines of code.

Dispensing with Bug Rules

PROUST's knowledge base contains goals, plans and bug rules. Goals and plans were written by [Johnson 1986] to describe correct methods for solving particular parts of a certain programming problem: goals and plans describe idiomatic methods for solving a given programming problem. PROUST's bug rules were designed to describe ways in which the goals and plans might be transformed, or permuted, into incorrect methods.

The architecture of PROUST's goals, plans and bug rules appears to have been influenced by the production rule literature (and most especially by the work of Brown, Burton and VanLehn [Brown and Burton 1978; Brown and VanLehn 1980]). Two sets of rules are normally used in production system models of students: correct rules, and incorrect, mal-, or bug-rules. From the perspectives of the production rule game, the construct of bug rules is theoretically important.

However, from the perspective of the language design game PROUST's bug rules are a weakness for two reasons:

(1) Execution suspension: PROUST suspends the matching of goals and plans if a plan cannot be matched. Bug rules are applied to "explain" the mismatches encountered during the matching of a plan. If the mismatches can be accounted for by some bug rule, then goal and plan matching is resumed. Suspending and resuming execution of PROUST's matching interpreter requires that PROUST do a lot of "bookkeeping." I.e., while it is analyzing a student's Pascal program, PROUST builds an "interpretation tree" (a sort of fancy parse tree) in order to record the state of the matching process. The "interpretation tree" contains the information necessary to resume a suspended matching process. Keeping a trace of the matching process (like the "interpretation tree") does not necessarily demand a lot of computer space (i.e., memory). For example, most interpreters of the Prolog programming language keep a very space-efficient execution trace to allow backtracking. However, the way in which "interpretation trees" are stored by PROUST is very space demanding.

(2) Unclear semantics: Johnson was unable to devise a declarative form for PROUST's bug rules: ... *a fully declarative plan-difference [i.e., bug] rule test representation cannot be achieved.* [Johnson 1985: 173] As a consequence, bug rules in PROUST are arbitrary Lisp functions. They are arbitrary in the sense that bug rules can be (and are) written to

- (a) modify the "interpretation tree", i.e., the state of the goal and plan matching process;
- (b) modify the structure of the student program being analyzed; and,
- (c) modify the variable bindings of the plans and goals being matched.[Johnson 1986: 180]

Consequently, not even the goals and plans in PROUST have a semantics that is independent of the state of the matching interpreter because bug rules in PROUST modify variable bindings and other vital pieces of information necessary to describe the state of the matching process.

From a language designer's point of view the alternative is necessary: eliminate bug rules from the architecture and replace them with buggy plan and goal variants written in the same notation as the correct goals and plans. Without bug rules, execution of the matcher does not need to be suspended and one gains a declarative semantics for bugs.

MicroPROUST's Goals and Plans

After Johnson wrote PROUST he designed a "micro" version of it which he call MicroPROUST [Johnson and Soloway 1985a; Johnson and Soloway 1985b]. The main difference between the goal and plan knowledge structures employed in PROUST and those used in MicroPROUST is that one cannot specify subgoals within a MicroPROUST plan. I will center the following discussion around MicroPROUST plans because they are a bit easier to understand than PROUST's plans. In a later section I will reintroduce subgoals into plans, thus bridging the difference between the plans of PROUST and MicroPROUST.

The MicroPROUST plan shown below is designed to match the piece of Pascal code which follows it. (Those readers who have written plans for MicroPROUST will recognize that the plan syntax shown is not exactly the one used in the MicroPROUST described by [Johnson and Soloway 1985b]. However, the syntax shown is "isomorphic" to the original syntax in that one can mechanically translate from one syntax into the other. I have written a short piece of Lisp code to do this translation.)

MicroPROUST plan:

```
(DEFPLAN SENTINEL-PROCESS-READ-WHILE (^NEW ^STOP)

  "~%read the first value of ^NEW at line (LINE-OF INIT) ~
  ~&read each of the following values at line (LINE-OF NEXT)"

  MAINLOOP (WHILE (<> ^NEW ^STOP) ^?)  NIL
  PROCESS  (BEGIN ^*)                  ((AT MAINLOOP))          T
  NEXT     (READ ^NEW)                  ((AT PROCESS) (BOTTOM)) T
  INIT     (READ ^NEW)                  ((ABOVE MAINLOOP)))
```

Pascal code to which the plan could be matched:

```
...
Read(X);
...
While X <> Y Do Begin
...
    Read(X)
End;
...
```

The BNF specification of a MicroPROUST plan looks like this:

```
<mp plan> ::= (DEFPLAN <plan name> ({<variable>})  
              <message>  
              {<slot name> <pattern> <constraints> <marker>})
```

where ...

<plan names> and <slot names> are symbols (i.e., Lisp tokens);

<variables> are symbols prefixed with a caret (i.e., a ^);

<messages> are strings which can contain references to <slot names> and the lines of Pascal code to which the <patterns> mentioned in the plan slots are matched;

<patterns> are lists of symbols and <variables> that are matched against portions of an ast, abstract syntax tree (Before being analyzed by MicroPROUST, Pascal programs are lexed and then parsed into an internal format called an ast);

<constraints> list either one or two constraints which specify where, in the ast, a <pattern> can be matched (Thus, in the plan SENTINEL-PROCESS-READ-WHILE the INITIALization of the <variable> ^NEW must get matched ABOVE the MAINLOOP); finally, a

<marker> can be T or NIL or absent and is used by the MicroPROUST matching interpreter to mark which parts of the ast can be matched by more than one <pattern>.

MicroPROUST goals are simply disjuncts of plans.

```
(DEFGOAL SENTINEL-CONTROLLED-LOOP
  (OR SENTINEL-READ-PROCESS-WHILE
      SENTINEL-PROCESS-READ-WHILE
      SENTINEL-READ-PROCESS-REPEAT))
```

In order for a goal to be matched, only one of the plans listed in the goal needs to be matched.

Compiling a MicroPROUST Plan into C

I wrote a two step compiler to transform MicroPROUST plans into C code. The first step turns a plan into a set of Lisp functions. The second step takes the generated Lisp functions and rewrites them as C functions. Almost immediately after I built the compiler for MicroPROUST plans, I threw it and the plan language out in order to start from scratch and build the knowledge base language for the ProgramCritic. Nevertheless it is instructive to examine the output of the plan compiler to see what features of MicroPROUST's plan language needed to be "cleaned up."

The following is the main Lisp function output by the first phase (plan-->Lisp) of the plan compiler for the plan shown above (the SENTINEL-PROCESS-READ-WHILE plan).

```
(defun sentinel_process_read_while (new stop mainloop next process init
                                   messages)
  (let ((lmainloop (pointer-value mainloop))
        (lnext (pointer-value next))
        (lprocess (pointer-value process))
        (linit (pointer-value init))
        (lmessages (pointer-value messages))
        (lnew (pointer-value new))
        (lstop (pointer-value stop)))
    (find_mainloop_of_sentinel_process_read_while mainloop
      (create_context *nil* *nil*) nil new stop (make-pointer :value t))
    (find_process_of_sentinel_process_read_while process
      (create_context (create_constraint 'at mainloop) *nil*) nil
      mainloop)
    (find_next_of_sentinel_process_read_while next
      (create_context (create_constraint 'at process)
                     (create_constraint 'bottom *nil*))
      nil new process)
    (find_next_of_sentinel_process_read_while init
      (create_context (create_constraint 'above mainloop) *nil*) nil new
      next)
    (cond ((pointer-value init)
           (setf (pointer-value messages)
                 (format *nil*
                        "Input from the user is initially read by the ~
                        ~A statement on ~
                        ~&line ~D . Subsequently, input is repeatedly ~
                        read by the ~A ~
                        ~&statement on line ~D . No more values are ~
                        read for ~A after ~A ~
                        ~&is read by the program . |"
                        (name-of (pointer-value init))
                        (line-of (pointer-value init))
                        (name-of (pointer-value next))
                        (line-of (pointer-value next))
                        (pointer-value new)
                        (pointer-value stop))))))
```

```

t)
(t
  (setf (pointer-value mainloop) lmainloop)
  (setf (pointer-value next) lnext)
  (setf (pointer-value process) lprocess)
  (setf (pointer-value init) linit)
  (setf (pointer-value messages) lmessages)
  (setf (pointer-value new) lnew)
  (setf (pointer-value stop) lstop)
  nil)))

```

The Lisp function created for the plan is called `sentinel_process_read_while`. It returns T (i.e., "true") if it is successfully matched. It returns NIL (i.e., "false") if it does not match successfully. Its parameters are all typed data structures whose slots are filled in in the course of matching; the structures are called pointers and their most important slot is called value.

By comparing the Lisp function to the plan it was compiled from one can notice that two important things have been "cleaned up" in the plan language.

(1) *Static binding of variables*: The <slot names> and <variables> of the MicroPROUST plans have both been replaced by Common Lisp variables with *static binding*. MicroPROUST's matching interpreter implements *dynamic binding* of variables. *In a language with dynamic binding, free variables in a procedure [or plan] get their values from the environment from which the environment is called rather from the environment in which the procedure is defined ... dynamic binding violates the principle that a procedure should be regarded as a "black box," such that changing the name of a parameter throughout a procedure's definition will not change the procedure's behavior.*[Abelson and Sussman 1985: 321]. MicroPROUST's plans are not "black boxes" in this sense because the matching interpreter implements dynamic rather than static binding. In a language with static binding a free variable in a procedure (or plan) gets its value from the environment in which it is defined. Static binding has two advantages: (a) it allows one to rename variables in one plan without without having to worry about how the renaming will affect other plans; and, (b) since most contemporary computer languages use static rather than dynamic binding (APL is a notable exception), plans with static binding are easier to compile because they employ the same binding mechanisms as the target language (in this case C).

(2) *Plan slots as function calls*: The slots of MicroPROUST plans (denoted in the BNF definition given above by the sequence of four elements <slot name> <pattern> <constraints> <marker>) have been made into simple function calls in their compiled Lisp form. For example, the first slot of the plan SENTINEL-PROCESS-READ-WHILE MAINLOOP

```
(WHILE (<> ^NEW ^STOP) ^?) NIL
```

has been transformed by the plan compiler into the function call

```
(find_mainloop_of_sentinel_process_read_while mainloop
  (create_context *nil* *nil*) nil new stop (make-pointer :value
t))
```

(The greater part of the definition of the Lisp function `find_mainloop_of_sentinel_process_read_while` created by the plan compiler consists of references to various parts of the data structures which get instantiated if the pattern (WHILE (<> ^NEW ^STOP) ^?) is successfully matched against a student's Pascal program. The definition is lengthy (about as long as the main Lisp function generated for SENTINEL-PROCESS-READ-WHILE) but simple in form.)

Consequently it is now easy to see how subgoals can be introduced into plans. Subgoals can be rendered as function calls to other Lisp functions which represent plans or goals.

Goals and Plans as Equivalent and the Problem of Backtracking

In fact, this insight destroys the distinction (necessary in MicroPROUST and PROUST) between plans and goals. Since plans can call goals, or other plans, simple plans can be written to function like the disjunctive statements that MicroPROUST and PROUST goals are. A further embodiment of the plan compiler supported an extension to the plan syntax to take advantage of this insight, but I am not going to describe in detail that extension here.

I will, however, point out the key problem encountered if one allows plans to call other plans; the problem concerns backtracking. In order to recover from a plan match error the variable bindings have to be undone and the last successful match has to be backtracked to in order to try another plan. Neither PROUST nor MicroPROUST have more than rudimentary backtracking facilities.

After I constructed the last version of the plan compiler I realized that some of its features were quite reminiscent of a Warren abstract machine (the standard method used to implement logic programming languages, like Prolog). Consequently, I realized that the problems discussed above (concerning variable bindings, function calls, and backtracking) could *not* be handled in PROUST, but were adequately resolved in all implementations of the programming language Prolog.

So, the critique that I have sketched out in this paper is probably the most unflattering portrait one could paint of Johnson's work: from the vantage point of the language design game, one can see that most of the code of PROUST constitutes an unsuccessful attempt to implement a Prolog interpreter.

The ProgramCritic

The ProgramCritic is implemented in Prolog and has a knowledge base language that is "more powerful" (from a language designer's point of view) than PROUST's in the following ways:

(1) The ProgramCritic's knowledge base patterns (i.e., its "plans", "goals" and "bug-rules") have a declarative semantics that is not necessarily dependent upon the state of the matching interpreter;

(2) Variables in the knowledge base patterns are lexically scoped (i.e., get their values from a static binding mechanism, Prolog's variable instantiation mechanism);

(3) "Slots" in the knowledge base patterns are simply "function calls" to other knowledge base patterns; and,

(4) Backtracking is supported (via use of Prolog's built-in chronological backtracking mechanism).

It takes about 3 pages of Prolog code to define the ProgramCritic's knowledge base language (as compared to about 150 pages of Lisp necessary to define PROUST's matching mechanisms). The "core" of this implementation is a match/1 predicate which allows one to write patterns that can be matched against the internal format of a student's Pascal program (i.e., against the abstract syntax tree (ast)); and, a set of relation predicates which allow the relative positions between two nodes in the ast to be compared. For example, ancestor(X,Y) checks to see if X is Y's ancestor in the abstract syntax tree.

Using the ProgramCritic's knowledge base language I can define the SENTINEL-PROCESS-READ-WHILE "plan" like this:

```
sentinel_process_read_while(Mainloop,Process,Next,Init) :-  
    match([while,Mainloop,
```

```

[[<>,_,[variable,_,[New,_,[]],[empty,_,[]]],
  [variable,_,[Stop,_,[]],[empty,_,[]]]]_]],
match([begin,Process,_]),
match([read,Next,[variable,_,[New,_,[]],[empty,_,[]]]]),
match([read,Init,[variable,_,[New,_,[]],[empty,_,[]]]]),
proper_ancestor(Mainloop,Process),
proper_ancestor(Process,Next),
not proper_right_sibling(_,Process),
proper_left_sibling(Init,Mainloop).

```

Using a commercially available Prolog compiler the ProgramCritic's knowledge base patterns are compiled into fast, executable programs. Knowledge bases for the ProgramCritic are fairly compact (on the order of twenty pages of Prolog code) and they can be run on machines widely available in American high schools (IBM ATs with 640 kilobytes of memory and a hard drive).

The ProgramCritic, equipped with several knowledge bases, is the "smarts" of a programming environment for novices referred to at the Educational Testing Service (ETS) as the Advanced Placement Computer Science (APCS) Practice and Feedback System. The APCS System contains the ProgramCritic with knowledge bases, a set of programming problems that the student can select from, a text editor, a means to access a Pascal compiler, and a front end to the ProgramCritic which outputs the ProgramCritic's analysis to the student by highlighting errors in the student's program in reverse video and displaying advice in accompanying windows. During the spring and fall terms of 1991 the APCS System was field tested in several high schools in New Jersey and Pennsylvania. I have also implemented a device (called the ExampleCompiler) which partly automates the task of knowledge base construction. A description of the full APCS System and the ExampleCompiler is forthcoming [Sack, Soloway and Bennett (in preparation)].

Conclusions

In this paper, my comparison of PROUST and the ProgramCritic was done using the lexicon and rules of the *language design game*. It was only possible to do such a comparison with a vocabulary and set of aesthetics peculiar to the language design game. My claims were phrased in terms like these: declarative semantics, matching interpreter, dynamic and static scoping, function calls, and Prolog's variable instantiation and backtracking mechanism

In contrast, Johnson did not design PROUST with the language design game in mind. Rather, Johnson was working under the constraints of a different "pre-paradigm", a different language game, that was current in the environment in which he designed PROUST (the Yale Artificial Intelligence Lab from 1981 to 1985). One might call this Yale game, the *schema game* because the point of the game was to identify "real", mental structures (likes goals and plans) and then code them up in computer programs meant to exhibit some sort of "intelligence."

What I have tried to make clear by example in this paper is the idea that it is important to identify the language game which characterizes one's research goals before one tries to evaluate whether or not one has designed a "better" knowledge base language. After all, how can one know if one is winning unless one knows what the game is?

By refusing to play the schema game and by taking the perspective of a language designer I have been able to point out the weaknesses of the knowledge representation languages used in PROUST [Johnson 1986] and MicroPROUST [Johnson and Soloway 1985]. I have built a compiler for a modified version of MicroPROUST's knowledge base language and, by so doing, I was able to show how one might characterize the bulk of PROUST as simply an unsuccessful attempt to implement a Prolog interpreter.

Following through on this insight I also described how a far more powerful and compact knowledge base interpreter can be written. I have implemented the ProgramCritic in Prolog. It is currently being used as the "smarts" for a programming environment for novices (known at ETS as the APCS Practice and Feedback System). The APCS System has been field tested in several high schools during this past year.

Acknowledgements

Most of the work described in this paper was done while I was a consultant to the Educational Testing Service in Princeton, NJ. Randy Bennett has served as my project advisor at ETS. Elliot Soloway was a co-consultant and my supervisor during this work. I would like to thank both of them for the encouragement they provided for the work described here as well as for a very fruitful two years in which the three of us collaboratively designed the APCS Practice and Feedback System in association with Henry Braun, Randy Kaplan, Marc Hoover, Irv Katz, Tara Kalro (who was our knowledge engineer), Doug Forer (who built the interface and editor for the APCS System), Gail Chapman (who wrote the programming problems included with the APCS System), Mary Edwards (who wrote the high school teacher's manual) and the ETS lawyers (of whom there seem to be many) who are copyrighting and patenting (!) the APCS System [Sack and Bennett (patent pending)].

References

- Anderson, John R. (1983) *The Architecture of Cognition*, Harvard University Press, Cambridge, MA.
- Brown, J.S. and Burton, R. (1978) Diagnostic models for procedural bugs in basic mathematics skills, *Cognitive Science*, 2, 155-192.
- Brown, John Seely and VanLehn, Kurt (1980) Repair theory: a generative theory of bugs in procedural skills, *Cognitive Science*, 2.
- Chapman, David et al. (1987) How to be a graduate student at the MIT AI Lab, MIT AI Lab Memo, Cambridge, MA.
- Charniak, Eugene, Riesbeck, Christopher, McDermott, Drew and Meehan, James (1986) *Artificial Intelligence Programming*, 2nd Edition, Lawrence Erlbaum associates.
- Johnson, W. Lewis (1985) Intention-Based Diagnosis of Errors in Novice Programs, Yale University Computer Science Department Research Report #395, New Haven, CT.
- Johnson, W. Lewis (1986) *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, Los Altos, CA.
- Johnson, W. Lewis and Soloway, Elliot (1985a) Micro-PROUST, Yale University Computer Science Department Research Report #402, New Haven, CT.
- Johnson, W. Lewis and Soloway, Elliot (1985b) PROUST: An automatic debugger for Pascal programs, *BYTE*, April 1985, 179-190.
- Kuhn, Thomas S. (1962) *The Structure of Scientific Revolutions*, 2nd Edition, U of Chicago Press, Chicago.
- Latour, Bruno (1987) *Science in Action: How to follow scientists and engineers through society*, Harvard U. Press, Cambridge, MA.
- Minton, S., Carbonell, J.G., Knoblock, C.A., Kuokka, D.R., Etzioni, O., and Gil, Y. (1989) Explanation-based Learning: A Problem Solving Perspective, *Artificial Intelligence* 40, 63-118.
- Sack, Warren (1990) *Technical Description of the Analytic Component: A Pattern Matcher for Analyzing Computer Programs*. (Unpublished Working Paper) Princeton, NJ: Educational Testing Service.
- Sack, Warren (1991) *Knowledge Engineer's Manual: How to Build KBs for the ProgramCritic*. (Unpublished Working Paper) Princeton, NJ: Educational Testing Service.
- Sack, Warren, Soloway, Elliot and Bennett, Randy (in preparation) The Advanced Placement Computer Science Practice and Feedback System.
- Sack, Warren and Bennett, Randy (patent pending) United States Patent Application: Method and System for Interactive Computer Science Testing, Analysis and Feedback.

- Schank, Roger and Abelson, Robert (1977) *Scripts Plans Goals and Understanding: An Inquiry into Human Knowledge Structures*, Lawrence Erlbaum Associates, Hillsdale, NJ.
- Soloway, Elliot and Ehrlich, Kathrine (1984) Empirical Investigations of Programming Knowledge, *IEEE Transactions on Software Engineering* SE-10(5).
- Steele, Guy Lewis and Sussman, Gerald Jay (1975) Scheme: An interpreter for the extended lambda calculus, MIT Artificial Intelligence Laboratory Memo 349, Cambridge, MA.
- Sterling, Leon and Shapiro, Ehud (1986) *The Art of Prolog*, MIT Press, Cambridge, MA.
- Sussman, Gerald Jay and Abelson, Harald (1985) *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA.
- van Harmelen, F, and Bundy, A. (1988) Explanation-based generalization = Partial evaluation, *Artificial Intelligence* 36401-412.
- Warren, D.H.D. (1977) Implementing Prolog: Compiling Predicate Logic Programs, DAI Report Nos. 39 & 40, U. of Edinburgh.
- Wenger, Etienne (1986) *Artificial Intelligence and Tutoring Systems*, Morgan Kaufmann Publishers, Los Altos, CA.
- Wittgenstein, Ludwig (1960) *The Blue and Brown Books: Preliminary Studies for the "Philosophical Investigations"*, Harper & Row, New York.