WORKING PAPER 84

THE EVOLUTION OF PROCEDURAL KNOWLEDGE

A Master's Thesis Proposal And Progress Report

Mark L. Miller

Massachusetts Institute of Technology

Artificial Intelligence Laboratory

January 16, 1975

Abstract

A focus on planning and debugging procedures underlies the enhanced proficiency of recent programs which solve problems and acquire new skills. By describing complex procedures as constituents of evolutionary sequences of families of simpler procedures, we can augment our understanding of how they were written and how they accomplish their goals, as well as improving our ability to debug them. To the extent that properties of such descriptions are task independent, we ought to be able to create a computational analogue for genetic epistemology, a theory of *procedural ontogeny*. Since such a theory ought to be relevant to the teaching of procedures and modelling of the learner, it is proposed that an educational application system be implemented, to help to clarify these ideas. The system would provide assistance to students solving geometry construction problems.

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defence and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0005.

Working Papers are informal papers intended for internal use.

THE EVOLUTION OF PROCEDURAL KNOWLEDGE

A Master's Thesis Proposal and Progress Report

Table of Contents:

Introduction:	3
A Look at Related Research:	12
Turtle Geometry -- A Scenario of Planning and Debugging:	18
Sorting Algorithms -- Another Scenario:	28
Toward a Theory:	37
A Consultant For Geometry Construction Problems:	47
Conclusion:	59
Bibliography:	60

Introduction:

> [The behaviorist] says: "There is no problem of consciousness, none of imagery, none of meaning". But what he really intends to say is: "There is no problem that cannot just as well be formulated without using these particular terms". ...there may be very good reasons for saying that meaning *arises* partially as a result of the organisation of reactions. But this is a very different view from the one which states roundly: "Meaning is action".
>
> <Bartlett, 64>.

Hampered by inadequate formalisms, or frustrated by the replacement of the study of phenomena by the study of formalism itself, psychologists, computer scientists, linguists, philosophers, educators, logicians, and mathematicians have begun to forge a substantial new discipline: *cognitive science*. The central problem of cognitive science is to understand and represent the meaning content of human knowledge. The artificial intelligence [AI] approach to the problem of meaning is to strive for precision by expressing our intuitions in the language of computer programs. Perhaps the single most significant feature of the AI method is the attempt to describe the notion of *process*.

There is a fruitful interplay between this theoretical approach and *cognitive engineering*, the attempt to build intelligent machines. The design of applications-oriented programs can be a powerful methodology for clarifying theoretical issues. This section discusses certain controversies which are the overriding concern of the current proposal for research. Other sections scrutinize the application of some of the ideas to the domains of turtle geometry and sorting algorithms. A later section attempts to formulate the beginnings of a theory. The final section proposes the implementation of an educational computer system (which would provide assistance to students

solving geometry construction problems), as a technique for pursuing the investigation of these issues.

Shifting emphasis away from earlier axiomatic and search paradigms, recent AI research has focused on the understanding of program planning and debugging, and the explicit representation of knowledge in a domain. This theme underlies the enhanced proficiency of systems which solve problems and acquire new skills. The embedding of such knowledge in procedures, as opposed to more static data structures, has been termed *procedural epistemology*. Examination of the genesis of problem solving procedures for several domains leads to the observation that procedural knowledge exhibits a tendency to evolve toward greater flexibility and generality. By describing complex procedures as constituents of evolutionary sequences of families of simpler procedures, we can augment our understanding of how they were written and how they accomplish their goals, as well as improving our ability to debug them. To the extent that properties of such descriptions are task independent, we ought to be able to create a computational analogue for genetic epistemology, a theory of *procedural ontogeny*.

> [The phrase "procedural ontogeny" can be criticized as both loose and pompous. These objections are not without substance, and it might have been preferable to avoid this terminology entirely. The overriding heuristic value of providing names for concepts is the justification for its retention in this document. Nonetheless, this *particular* nomenclature is still questionable. The choice of the word "ontogeny", instead of some less specific term, reflects an emphasis on similarities among the developmental histories of individual procedures, rather than on differences between collections of procedures (i.e., speciation), as might be suggested by some more inclusive phrase. Even then, one might legitimately inquire as to the sense in which the procedures being discussed exist. Are they embodied, for example, in the adaptive mechanisms of biological organisms? The intention is that the procedures represent abstractions from reality; this is a special case of the philosophical distinction between epistemology and ontology, the implications of which cannot be pursued in a paper of this nature.]

Such a theory (i.e., of procedural ontogeny) would be relevant to the teaching of procedures, and, more importantly, the modelling of the learner. By addressing fundamental questions of

problem solving and learning, the project should contribute to progress in program understanding research, and help to strengthen the developmental foundations of the study of intelligence. Knowledge of the evolutionary history of programs can be utilized in several different ways, including explanation, generation, and debugging.

Several alternative techniques for organizing the debugging process have already emerged:

1. *Model Driven Debugging* operates by fixing a program so as to eliminate violations of a model of the intended result. The MYCROFT system written by Ira Goldstein for the LOGO domain is organized in this manner, where the model language provides primitives for describing pictures in declarative form, and the debugging monitor attempts to insure the consistency of declarative and procedural descriptions.

2. *Process Driven Debugging* is directed by the abstract state of the process at the time of the error interrupt. Gerry Sussman's HACKER program emphasizes this understanding of the chronology and purpose of the computation in writing and debugging BLOCK'S WORLD programs.

3. *Algorithm Driven Debugging* is guided by interpretive descriptions of standard algorithms for a given task. Gregory Ruth has built an analyzer for sorting programs written by beginners which attempts to recognize the underlying algorithm.

4. *Validation Driven Debugging* bases its strategy on the structure of the procedure definition during attempted verification. Carl Hewitt's group has proposed an environment for knowledge-based programming in which procedural *contracts* are reconciled with corresponding ACTOR code by *meta-evaluation*.

Deriving guidance from the editing history of the program, *Evolution Driven Debugging* could provide further insight. By keeping a record of the successive versions of a program, the monitor system can become sensitive to bugs arising from recent patches of various semantic types. Evolution driven debugging analyzes the conceptual origins of bugs in more complex programs (such as those with inputs), tracing them to planning heuristics which produced the code by generalizing from earlier families of simpler programs. There is no quarrel with the utility or validity of other techniques; rather, it is contended that a powerful debugging monitor would need to exhibit many such sources of direction.

It is important to recognize, in program generation, that the complete specifications very often are not known in advance. Nor does one always know the correct way to proceed. At each stage, some bugs will be introduced; others will be eliminated, reflecting an evolving understanding, and adapting to changing specifications. The planning and generation of procedures are inextricably bound up with debugging. What is being dealt with here is the relationship of bugs to longer range plans for building a complex program by starting with a simplified, less efficient, or less general version. These concepts are unified by a provision for *program readable annotation* of causal relationships. There is a need for careful evaluation modes, history lists, and means for representing the dynamics of a changing world. There have to be mechanisms for discovering whether and why a program is indeed malfunctioning, and this may depend on an important role for typical and pathological examples.

Research on procedural evolution is necessarily concerned with the nature of expertise: the representation of domain dependent knowledge so as to be accessible by situational goals; the problem of recognition; the tradeoff with structural problem solving techniques; and the importance of geometrical, physical, or mechanical analogies in thought about other domains. Most important in studying children's programming is the role of learning. The emphasis is on learning *how* versus learning *that*, and the importance of a carefully organized sequence of problems, solutions, and near misses to solutions.

A crucial issue for this work concerns the alternative paradigms for organizing a problem solving system. One school of thought favors Fregean representations, making a sharp distinction between facts and inference rules. There have been certain problems with this approach, such as the difficulty of updating facts which can change over time. Another school of thought, embodied in GPS <Newell & Simon, 72>, organizes the problem solver as a set of productions which search a

problem space, using plausible move generators and static evaluation functions. This contrasts with a new *knowledge* paradigm based on planning and debugging, and providing for explicit (often procedural) domain dependent knowledge. Forgetting is allowed, since knowledge is seen as limited and changeable. An important sub-issue is the extent to which debugging or problem solving in general is dependent on the particular domain. Does each mini-world have its own characteristic bugs, and/or appropriate top-level debugging strategy? To what extent can existing programs be extended or generalized? Can there be a complete catalogue of bug types, or does skill in trouble-shooting merely reflect the operation of a myriad assortment of *ad hoc* techniques? What is the role of paradigmatic examples as analogies or powerful ideas for other domains?

The distinction between procedural and declarative knowledge is a difficult issue. This controversy of meaning-as-action versus meaning-as-context parallels the earlier psychological issue of Behaviorism versus Gestaltism. However, the current debate reflects a more advanced understanding, being essentially mentalistic and having reference to internal states. The questions are more refined: intrinsic versus extrinsic computation, intensional versus extensional contexts, the modularity/efficiency tradeoff, heuristic power versus logical consistency. What is most important is that the programmer be able to put knowledge anywhere in the system that it can be used, and that facts be linked to advice as to their use. Moreover, it is critical to be able to specify all and *only* what is intended. For example, it may be that an iterative factorial program serving to model the intentions of a recursive factorial program requires too much -- it prescribes the order in which the multiplications are to be done -- complicating the task of showing the equivalence of alternative versions. This might be particularly true in the presence of bugs, where equivalent contracts could produce different manifestations. Likewise, when we wish to express (IMPLIES A B), we may not know whether it should be used in antecedent or consequent form. If forced to choose, we have

decided too much too soon. It may be desirous to leave the decision to a neutral (perhaps as yet unspecified) interpreter. On the other hand, much of the elegance and power of LISP lies in its blurring of such distinctions. This closure property has been put to powerful use in programs which create, modify, and execute code. It is not so much the *form* of the language but the amount and location of *imperative content*. The objections in the case of factorial can be met, for example, by the introduction of new primitives such as *bags*, which leave the order unspecified. Then it becomes a matter of raising the level of the interpreter. Some of these problems may be related to dissatisfaction with the invocation of database methods on the basis of the pattern of a single assertion. Minsky's FRAME theory may be seen as a possible step toward synthesis.

More perplexing than even the declarative/procedural controversy is the general proliferation of proposals for representing knowledge. What criteria does one use in determining to employ Semantic Nets instead of Conniver Assertions, Actors instead of Frames, or Conceptual Dependency Diagrams instead of Deep Structures? Can there be a scientific basis for such choices, or is AI bound to wallow in the "Turing Machine Tarpit"? Is there a useful sense in which these are not merely notational variants? Can the best aspects of each proposal be unified into a single, superior representation? For example, Bobrow and Norman <74> have attempted to merge features of Actors and Frames into a single mechanism called "Schemata". Can such a device help to draw parallels between Piaget's vague but intuitive explanations, and the precise operations of a computer program?

A growing movement to find ways of constructing more reliable software raises another issue: validation versus debugging. It is not inconceivable that within five or ten years there might exist a "complete" catalogue of programming bugs. Would this eliminate the need for debugging? Could one compile a *critic* for each, to preprocess all programs, and then send them

off for automatic verification? Likewise, is it possible that the Structured Programming movement will supply us with such a sound design methodology that it becomes almost perverse to make mistakes? Can *top-down-stepwise-refinement* replace *simplify-and-debug*? Is debugging obsolete? It seems doubtful. Every programming construct with any power is dangerous. Even critic compilation, which surely distinguishes the expert from the novice in programming, cannot eliminate the efficiency of not explicitly providing for all exceptional conditions. For the fifth-grader, this might mean not checking for accidental equality when variabilizing. For the LISP interpreter, it might mean that the MAP series of functions must terminate on NULL checks (or, for that matter, tolerate attempts to RPLAC NIL), because the expense of NLISTP checks would be prohibitive.

A good example of a powerful but dangerous construct is the free variable. The scope of identifiers has been a difficulty for every programming language to date. Various improvements have been and will continue to be considered. Undoubtedly some of these will eliminate certain bugs. But consider the following evolutionary sequence. Program ALPHA has been written to call subroutine GAMMA. GAMMA uses the variable FOO freely, as it is declared by ALPHA. It is irrelevant whether the scope rules of the language are ALGOL-like or LISP-like, or whether they make fluid access the default or the exception. For efficiency reasons (which will not be argued further here), GAMMA intends explicitly to use FOO freely. Now a few years later, when the listing for the system has grown very large, it is discovered that for certain configurations, users would like ALPHA to call an intermediate routine, BETA, which will take over responsibility for calling GAMMA in these cases. BETA needs a temporary storage cell. Because of homonymy, coincidence, or prevalence of the favorite name FOO, BETA declares FOO as a local variable -- thereby clobbering GAMMA. Could this be avoided? The answer to this, and a thousand similar

questions, is a matter of computational efficiency. If a sophisticated on-line indexing facility were available (and this bug was in the catalogue), a check for such a situation could be made (the listings are sufficiently long to make habitual hand-checking unreasonable). But it seems probable that there will always be a number of such pitfalls which, individually, occur sufficiently infrequently, and cost enough to routinely test for, that it is best to let subsequent errors direct the processing. The cost-effectiveness of such a decision will of course always have to be weighed against possible results of future errors. When building a real-time hospital-patient monitor, or an air-traffic controller, it might be necessary to modify the criterion. But there seems to be little chance that modular methodology can totally compensate for the eventuality that the specifications for a program -- whether because of new user needs at the top or replacement of the hardware below -- may change drastically.

Similarly, it seems preposterous to attempt to prove that a program is correct when it has never been tried on a single example. Like the examination of *sample variations* in chess, the use of well-chosen test examples is intended not as a substitute for, but as a complement to, validation. Within this framework one can ask related subquestions, such as comparing the relative efficiencies of various approaches for particular bugs, or for the problem of inputs. When (how much?) does a declarative model help? Are bugs caught by run-time interrupts (unsatisfied prerequisite to primitive) more amenable to Process Driven Debugging? Would compile-time type-checking eliminate or reduce manifestations such as wrong number or order of arguments? What would be the role of evolutionary information in a comprehensive programming-laboratory style monitor? Should completeness and even strict logical consistency be traded for increased practical problem solving power (and resultant need to debug)? Local versus global: what is the role of major program reorganization as opposed to patching? Analog versus digital: what is the role of

simulation models in reasoning? Are such techniques as the use of a diagram filter in geometry helpful or necessary? Is program writing more like debugging a blank sheet of paper or implementing a proof? Must we distinguish between cognitive and emotive bugs? At least some spelling and syntax errors can be easily fixed, if not understood.

A central concern is to understand the structure of natural language dialogue when it occurs in a goal-directed context, such as that between tutor and student. What is the best way to communicate procedural problem solving knowledge to the novice? In particular, what is the nature of a "hint"? The focus is on the connections between a general theory of procedural evolution, and its application to a model of the progress of a given student. To what extent are hints based on structural problem solving techniques (e.g., "look at the degenerate case"), as opposed to, for example, knowledge of the approach used in recently solved problems, and prediction of bugs which can arise in attempts to extend it? How can tutor-like systems carry on a more natural dialogue with a human user? What is the effect of expressing procedures in natural language -- a loss in precision or a gain in programmer power? Most generally, what are the postulates of *purposeful* human conversation?

In many cases, there are no formal differences between the positions posed here. Criteria such as naturalness, efficacy, and perspicuity come into play. If AI is to be an experimental science, such issues can not be left to speculation, personal style, or religious preference. They must be resolved by analysis of successful and unsuccessful attempts to build systems such as the one proposed here. Evolutionary considerations may provide some insight.

A Look at Related Research:

> ... genetic epistemology deals with both the formulation and the meaning of knowledge. We can formulate our problem in the following terms: by what means does the human mind go from a state of less sufficient knowledge to a state of higher knowledge? The decision of what is lower or less adequate knowledge, and what is higher knowledge, has of course formal and normative aspects. It is not up to psychologists to determine whether or not a certain state of knowledge is superior to another state. That decision is one for logicians or for specialists within a given realm of science. For instance, in the area of physics, it is up to physicists to decide whether or not a given theory shows some progress over another theory. Our problem, from the point of view of psychology and from the point of view of genetic epistemology, is to explain how the transition is made from a lower level of knowledge to a level that is judged to be higher. The nature of these transitions is a factual question. The transitions are historical or psychological or sometimes even biological ...
>
> <Piaget, 71>.

A brief review of related literature will help to put this proposal into perspective. The ideas behind the project have grown out of recent work directly concerned with planning and debugging procedures. More generally, these efforts are related to: AI research into the representation of knowledge and the learning of descriptions; efforts to build intelligent, reactive, computer-based learning environments; cognitive and educational psychology; and computer science work on programming language semantics, computational linguistics, automatic programming, and program understanding.

The research described herein began as an attempt to extend MYCROFT, the debugging system described by Ira Goldstein <74a>. This thesis emphasizes powerful ideas like linearity, ordering the attack on multiple bugs, and the importance of finding the plan. The model driven debugging system generates patches which eliminate violations of a declarative model of an intended picture, described in terms of predicates such as "connected", "left-of", and "inside-of".

The model specifies *what* is to be accomplished; the program tells *how*. Understanding the program involves finding its *plan*, commentary indicating which pieces of code accomplish which model parts. Additional information is obtained from *performance annotation*, including a Cartesian representation of the resulting picture. The debugging monitor was designed to take advantage of features unique to its domain of fixed instruction turtle programs (such as the Rigid Body Theorem), although extensions to more complex constructions are suggested. Studying the extension to programs with inputs suggested the desirability of (or need for) additional sources of direction. In a quest for generality, there also arose an obligation to examine other domains. Nonetheless, MYCROFT is quite effective at isolating and correcting a large catalogue of bugs. It (he?) does not attempt to explain *why* they are there.

Gerry Sussman's HACKER <73> combines the ability to debug (BLOCK'S WORLD) programs with a library of techniques for writing them as well. It models the way in which skills are acquired by adapting a plan based upon a related problem, and then patching it when it fails. It debugs programs which fail by requesting illegal computations, such as attempting to pick up a block whose top has not yet been cleared off. By keeping a thorough record of the history of the computation (the *CHRONTEXT*), and providing for causal commentary such as protection of the scopes of goals, bugs involving the interaction of conjoined subgoals and similar errors can be repaired. Some of these bugs are linked to the way in which the programs have been constructed -- for example, HACKER is subject to overvariabilization -- but knowledge of why the bugs are there is not used. This suggests that more than one approach can be effective in handling a given type of bug: HACKER favors a style of process driven debugging analogous to the stack tracing commonly employed with on-line systems.

Gregory Ruth <74> has built a Computer Aided Instruction [CAI] type system for use in an

undergraduate programming course. Debugging sorting programs written by beginners, it takes the pragmatic philosophy that most people do not invent brand new algorithms for standard programming tasks. Knowledge of basic sorting algorithms is built into the system. It expects the plan of the user program to be one of the two or three common ones, such as "bubble sort". This algorithm driven debugging system incorporates a grammar for recognizing similar or equivalent implementations. Based on possible sequences of design decisions which still support the basic algorithm, this grammar includes several common error types as variations. The bugs are repaired by expert modules designed to incorporate knowledge of loops, conditionals, and other programming constructs.

Carl Hewitt's ongoing PLANNER project <Hewitt *et.al.*, 73> approaches program understanding from the perspective of validation. In the framework of developing a comprehensive model of computation, all of the knowledge in a system is embedded in *ACTORS*, procedures with generalized control structure, based upon a unidirectional message-passing scheme. This includes knowledge of the *intentions* and *contracts* of code. *Meta-evaluation* is a facility whereby the system attempts to verify that the code in fact satisfies its advertised contract with the outside world. It is based upon the inductive assumption that all ACTORS with which it communicates do likewise. This is a procedural analogue of the "generalized snapshots of performance" method <Naur, 66> for handling the problem that testing on a few inputs cannot demonstrate the absence of bugs <Dijkstra, 70>. Hewitt's formalism is entirely procedural, bringing the declarative/procedural controversy to the fore. For example, the contract for a recursive FACTORIAL program is an iterative version. This confronts the problem of meaning with an interpretive (as contrasted with denotative) semantics, replacing questions of the consistency of descriptions with questions of the equivalence of computations. Here, the concern is with the

proposal to build a *programmer's assistant* (similar to Teitleman's <70> concept of a *programming laboratory*) which should be able to repair programs by examining the point at which attempted meta-evaluation breaks down. Some examples of how this validation driven debugging might be done are provided in <Smith *et.al.*, 73>. The ability to discover discrepancies very close to the underlying source of the error naturally depends on the amount of intentional commentary provided. The first point at which an intention fails to be provable causes entry into the error repair system. It seems likely that the incorporation of extrinsic purpose commentary into the formalism will be essential. Closely related to efforts to understand the evolution of programs is the work on teaching procedures by telling (in a high level goal oriented language), deducing the bodies of canned loops, and (especially) procedural abstraction from protocols of examples <Hewitt, 70b>, <Hewitt, 71>. It is clear that a synthesis of the contributions of proceduralists, declarativists and those who employ multiple representations is badly needed.

There are a number of projects not directly concerned with planning and debugging procedures which nonetheless have a similar spirit. Many of these have adopted knowledge representations similar to Minsky's <73> *Frame* theory, which has influenced the approach being taken here. This trend reflects a need for larger, more structured representations, where knowledge is inseparably bound to procedures for its use. Issues of recognition of classes of problem situations (as opposed to invocation by the pattern of a single assertion), shared data (e.g., *Clusters* <Liskov & Zilles, 74>), and schemas versus productions are pursued elsewhere (e.g., <Miller, 74a>). Some projects in this vein are the *Electronics Repairman* <Brown, A., 74>, <McDermott, 74> and work on medical diagnosis (e.g., <Silverman, 74>). Similar ideas have been employed in work on vision <Fahlman, 73b>, <Kuipers 74>, <Dunlavey, 74a,b> and robotics <Fahlman, 73a>. Description comparison, near misses, and discrimination networks are features of Winston's <70> learning

program, and recent attempts <Freiling, 73> to extend it.

Work on functional reasoning in geography, geometry and electronics <Brown & Collins, 74> suggests how people create a "pseudo-proof" of a theorem by intelligently creating and manipulating examples. Relevant systems include SOPHIE <Brown *et.al.*, 74>, a CAI program for teaching the qualitative knowledge needed for trouble-shooting electronic circuits, SCHOLAR <Collins *et.al.*, 74>, a tutoring program for geography, and a few other projects <Goldberg, 73>, <Kimball, 73>. There is more concern with efforts to teach mathematical problem solving skills than those which primarily present factual information. The applications system described in a later section is inspired by a proposal to build a more complete *Geometry Laboratory* <Burton, 73>, <Brown, J., 75>. The hope is that an understanding of the evolution of procedures will help to alleviate some of the problems which these authors see as confronting the next generation of *Intelligent* CAI [ICAI] systems. The problem solving ability of the ICAI system proposed here, which would teach students to apply powerful ideas *about problem solving* to geometry construction problems, relies partly on the procedural approach to constructions described by Funt <73>.

Psychological and pedagogical literature on learning and problem solving provide the background for this sort of research. One of the domains considered in this paper is based on the LOGO project, an attempt to provide computational tools and concepts to elementary school children <Papert, 71a,b,72,73>, <Papert & Solomon, 71>. Much thought has gone into an attempt to find computational analogies to the theories of Piaget (e.g., <Piaget, 71>), and Polya (e.g., <Polya 71>). Studies of human information processing, especially attempts to represent the semantic content of natural language utterances (<Norman, 73>, <Schank, 72>, <Newell & Simon, 72>), suggest how process models of long term memory, such as *active semantic networks*, can be used to model the learning of procedural knowledge (e.g., cooking recipes and programming languages).

It is important to try to relate this work to general computer science, as well. The natural interface is automatic programming. There have been numerous efforts to build systems that write programs <Fikes, 70>, <Manna & Waldinger, 71>, <Waldinger & Lee, 69> or learn to perform simple tasks such as arithmetic <Friedberg, 58>, <Friedberg *et.al.*, 59>. These have encountered various difficulties (e.g., the "mesa" phenomenon) discussed elsewhere <Miller, 74b>, <Sussman, 73>, <Balzer, 72>. Structured Programming (e.g., <Dijkstra, 70>, <Wirth, 71>, <Hoare, 69>) provides *rational form criteria,* where *design caveats* (e.g., "avoid free variables") suggest *demons* looking for erroneous code, (e.g., "incorrect scoping of identifiers"). Comparative Schematology <Paterson, 70>, <Paterson & Hewitt, 70> suggests looking for situations where the power of the plan-type or schema is inappropriate for the model to be achieved; for example, a flowchart schema to implement an inherently recursive model. The semantics of programming languages <Floyd, 67>, <McCarthy, 59> provides various alternative representations for the meanings of programs (such as Pratt's <74> idea to use modal logic). Finally, numerous attempts to automate proofs of correctness (e.g., <Laventhal, 74>) provide yardsticks against which to measure the progress of the less formal (heuristic) planning/debugging approach.

Turtle Geometry -- A Scenario of Planning and Debugging:

> The fundamental hypothesis of genetic epistemology is that there is a parallelism between the progress made in the logical and rational organization of knowledge and the corresponding formative psychological processes. Well, now, if that is our hypothesis, what will be our field of study? Of course the most fruitful, most obvious field of study would be reconstituting human history -- the history of human thinking in prehistoric man. Unfortunately, we are not very well informed about the psychology of Neanderthal man or about the psychology of *Homo sinensis* of Teilhard de Chardin. Since this field of biogenesis is not available to us, we shall do as biologists do and turn to ontogenesis. Nothing could be more accessible to study than the ontogenesis of these notions. There are children all around us. It is with children that we have the best chance of studying the development of logical knowledge, mathematical knowledge, physical knowledge, and so forth.
>
> <Piaget, 71>.

A major aspect of the research leading up to this proposal has been an investigation of the planning and debugging of picture programs from the mini-world of *LOGO turtle geometry*. In the LOGO laboratory, children are exposed to computers and computational concepts, as a way of understanding and improving their own efforts to learn and solve problems. The *turtle* is a *virtual graphics cursor*, with various physical embodiments. The turtle has a *state*, which consists of its location, its heading, and whether its pen is up or down. Primitives in the LOGO programming language (FORWARD, RIGHT, PENUP) alter these attributes of the state vector independently.

The system built by Ira Goldstein debugs *fixed instruction* turtle programs. A fixed instruction program is a sequence of non-recursive calls to primitives and user subroutines. Calls to primitives can take only constant inputs; in user procedures, inputs, variables, counters, conditionals, iteration, recursion, and exiting commands are not allowed. This research grew out of an attempt to extend Goldstein's work to include some of these previously disallowed constructs.

This section develops an evolutionary scenario as an illustration of the understanding of more complex programs -- and their associated bugs -- through knowledge of the editing sequences by which they were written.

The LOGO programs of concern draw pictures by manipulating the state of a *display turtle* <Abelson, *et.al.*, 73>. Standard control primitives such as looping, conditionals, and recursion are available, but not interrupts, co-routines, or more general control mechanisms. Actually, prior to processing, the programs are translated to equivalent MACLISP <Moon, 74> code, as described in <Goldstein, *et.al.*, 74>. For convenience, we employ the LOGO syntax, as in the following fixed instruction TRIANGLE routine.

```
TO TRIANGLE
10 FORWARD 100
20 RIGHT 120
30 FORWARD 100
40 RIGHT 120
50 FORWARD 100
60 RIGHT 120
END
```

The debugging leading up to such a routine would be easily handled by Goldstein's system. Furthermore, his system is sensitive to such considerations as evolutionary *insert plans*, where *rational form criteria* (such as a *caveat* against consecutive calls to the same primitive) can be temporarily suspended. An example of such a situation is the following triangle program, which will be a building block for a later TREE.

```
TO TREETOPTRI
10 FORWARD 50
30 FORWARD 50
40 RIGHT 120
50 FORWARD 100
60 RIGHT 120
70 FORWARD 100
END
```

The TRUNK would be added later, as line 20.

It would be natural for a fifth grader who had just learned the concept of "inputs" to generalize the first triangle routine to take variable length sides. The correct routine might look something like the following.

```
TO TRIANGLE2 :X
10 FORWARD :X
20 RIGHT 120
30 FORWARD :X
40 RIGHT 120
50 FORWARD :X
60 RIGHT 120
END
```

But consider the unfortunate case of a child who happens to use sides of length 120 in the original program. Forgetting the semantics of each constant, the *accidental equality* leads to *overvariabilizing* the routine:

```
TO TRIANGLE3 :X
10 FORWARD :X
20 RIGHT :X
30 FORWARD :X
40 RIGHT :X
50 FORWARD :X
60 RIGHT :X
END
```

This also emphasizes the fact that, when dealing with inputs, bugs may be intermittent. This version would work correctly if :X happened to be 120, but would produce rather odd violations for other values. Since the violations vary considerably with the input (for example, from "unconnected" to "X-connected"), trying to isolate the error on the basis of model violations alone might be difficult. *Knowledge of the recent edit, however, helps to pinpoint the problem.* This is facilitated by domain dependent knowledge of primitive semantics. FORWARD takes a length as input, whereas RIGHT takes an angle. It is seen as odd to pass the same parameter to both, except

to achieve unusual special effects.

Note that encouraging the child to use mnemonic names might help her/him to detect this. "RIGHT :SIDE" would seem "weird", whereas "RIGHT :X" does not. This suggests that such bugs could be avoided, since the programmer could keep track of the meaning of each constant, variabilizing accordingly. Indeed, it would be surprising to find a competent programmer falling into this kind of trap. But to the novice, such a refinement seems unnecessary, since such situations have not arisen frequently enough to warrant the additional mental effort. Likewise, for expert programmers there undoubtedly exist more sophisticated bugs, sufficiently improbable to cause them to be overlooked on the first pass.

The plan of the simpler program constrains the relative likelihoods of the kinds of bugs which can occur. Consider the following program, which draws an equilateral triangle beginning in the center of one side.

```
TO TRIANGLE4
10 RIGHT 90
20 FORWARD 50
30 RIGHT 120
40 FORWARD 100
50 RIGHT 120
60 FORWARD 100
70 RIGHT 120
80 FORWARD 50
90 RIGHT 90
END
```

Here, the fact that one side is accomplished in two pieces might be considered an accidental inequality. It makes the bug of *incomplete variabilization* much more likely.

```
TO TRIANGLE5 :SIDE
10 RIGHT 90
20 FORWARD 50
30 RIGHT 120
40 FORWARD :SIDE
```

```
50 RIGHT 120
60 FORWARD :SIDE
70 RIGHT 120
80 FORWARD 50
90 RIGHT 90
END
```

Of course this incomplete editing bug can and does occur with the other plan, but it would be considered less probable. Likewise, overvariabilizing could occur here. But the bug of forgetting to scale the first side by a factor of two can only occur with this type of algorithm.

If the child happens to notice the repeated pattern of instructions in these programs, perhaps by comparing similar ones for fixed size squares or pentagons, it may occur to her/him to subroutinize the common code. Even if she/he does not, it can be quite natural to introduce iteration or recursion at this point. Note that without conditionals, such programs do not terminate.

```
TO TRIANGLE6
10 FORWARD 100
20 RIGHT 120
30 GO 10
END
```

```
TO TRIANGLE7
10 FORWARD 100
20 RIGHT 120
30 TRIANGLE7
END
```

Bugs, however, are in the eye of the beholder. What is a feature to the implementors of a system may turn out to be a bug for the users. In this case, failure to halt becomes a bug when the programs are used as subroutines.

The fix for the iterative version is a local counter and a stop rule. The recursive version may pass the counter as a parameter, somewhat of an inconvenience at the top level.

```
TO TRIANGLE8
10 LOCAL :X
20 MAKE "X" 3
```

```
TO TRIANGLE9 :X
10 FORWARD 100
20 RIGHT 120
```

```
30 FORWARD 100
40 RIGHT 120
50 IF :X=1 THEN STOP
60 MAKE "X" :X-1
70 GO 30
END
```

```
30 IF :X = 1 THEN STOP
40 TRIANGLE9 :X-1
END
```

While these versions do not help to *abbreviate* the code, they can be a useful intermediate step for the student, suggesting areas for further generalization. Although such *conceptually intermediate forms* might not be explicitly manifested in the evolutionary sequence because of their lack of brevity, a debugging monitor should nevertheless be cognizant of them and their associated bugs. These versions are subject, for example, to counting errors in the loops, such as *fencepost* and *slipthrough*. [An earlier draft of this paper in fact contained several fencepost (counting the "holes" instead of the "posts") bugs which were not discovered until actual on-line testing!] These could be caught by an approach based upon the editing history, in concert with a loop specialist such as is utilized in <Ruth, 74>.

TRIANGLE9, like most programs, can be interpreted differently. It might be a bugged polygon, where the side length is fixed, but the number of sides is an input. However, it has the bug that the rotation was not edited to correspond to the number of sides. The result is that it draws triangles, repeating some of the sides.

Very odd effects result from overvariabilizing a program which specifies a polygon by its rotation angle.

```
TO POLYGON1 :A
10 FORWARD :A
20 RIGHT :A
30 POLYGON1 :A
END
```

For familiar angles (divisors of 360 with small integers), the program draws perfectly satisfactory,

connected polygons *which would satisfy a qualitative model.* The only manifestation is that the size fluctuates seemingly unpredictably. This is an example of the principle that very mysterious behavior often has quite simple origins. Other possible bugs in the constant side, variable angle polygon routine would include invocation errors, such as *input-not-a-divisor-of-360*, or *round-off-error* -- resulting in unexpected violations of connectivity when used by higher level routines.

Usage errors are not usually considered bugs in the routines being used. But when dealing with the full capabilities of co-recursion possible in recent languages, such distinctions are relative. The assignment of guilt for a manifested violation can depend on what seems most natural or convenient. A very simple example of this can be seen in the *wrong-order-of-inputs* syndrome in the following recursive polygon procedure.

```
TO POLYGON2 :ANGLE :SIDE
10 FORWARD :SIDE
20 RIGHT :ANGLE
30 POLYGON2 :SIDE :ANGLE
END
```

The picture drawn by (POLYGON2 90 200), for example, is aesthetically pleasing, if somewhat hard to describe. Should we blame line 30, or the procedure heading? It might seem more natural to reorder the heading, but if this were done in the context of other user routines which used POLYGON2, *perturbation analysis* <Hewitt *et.al.*, 73> would be necessary. This bug might be expected to arise more frequently in a sequence which had variabilized the angle before the side, than one in which the converse were the case.

When two evolutionary paths merge, as is the case for a fully general polygon routine with a stop rule, the patterns of edits are reversed. Consequently, the linear assumption is that the corresponding bugs are also reversed. This is precisely the sense in which knowing the editing history constrains the bugs which should be considered -- if the user just edited a correct, iterative

triangle routine to take a variable side input, there is no reason to expect errors associated with the loop structure -- *even though there is a loop* in the resulting program. Likewise, paths below a correct node in the sequence are not subject to the bugs which are prior to that node (although they may of course be subject to other bugs of the same type). As a consequence, the catalogue of possible bugs in a long sequence can be expected to grow linearly, rather than exponentially, provided that no steps are skipped. This may be a partial explanation for the value of carefully graded training sequences. When the polygon is attempted on the first pass, every conceivable bug is a possibility, and other sources of debugging direction are needed. It is plausible that the heuristic, "consider bugs near the end of the sequence first" might be valuable, if the user had proceeded along these lines without the aid of the system.

However, in some unfortunate situations, bugs may also arise from the *interaction* of two edits. For example, a correct straight-line polygon routine which is clever enough to calculate the rotation angle from the number of sides, might be edited to operate recursively. This merges with a correct recursive version which is edited to calculate the rotation angle. The result might be something like the following.

```
TO POLYGON3 :SIDE :SIDES
10 LOCAL :ANGLE
20 MAKE "ANGLE" 360/:SIDES
30 FORWARD :SIDE
40 RIGHT :ANGLE
50 IF :SIDES = 1 THEN STOP
60 POLYGON3 :SIDE :SIDES-1
END
```

Note that :ANGLE will be 180 degrees on the second-to-last invocation; consequently the most serious manifestation is that a model part (one side) seems to be missing entirely; the retracing done by the innermost call could easily be mistaken for a preparation step. An approach based

solely upon ordering the manifestations according to severity and then trying to localize one at a time might begin by hypothesizing missing code, or a fencepost-type counting error in the recursive stop rule. But this *clobbered-local-variable* bug is actually a common pitfall of *recursivizing*. Without explicitly testing the level of the call, the patch requires creation of a *setup* super-procedure. Although a related bug is possible in a corresponding iterative version, the patch is *local* to a single statement within the body of the procedure. Thus, while iteration may be a special case of recursion in the formal sense, debugging systems must distinguish them, to reflect the differences in the bugs -- and patches -- to which they are subject.

```
TO POLYGON4 :SIDE :SIDES
10 LOCAL :ANGLE
20 MAKE "ANGLE" 360/:SIDES
30 FORWARD :SIDE
40 RIGHT :ANGLE
50 IF :SIDES = 1 THEN STOP
60 MAKE "SIDES" :SIDES-1
70 GO 30
END
```

Whether examining the programming of a VILLAGE from a HOUSE, a FOREST from a TREE, or (in the case of the LOGO music box) a SONG from a single THEME, one finds editing sequences which introduce countless examples of a relatively small number of such evolution-related bug types: inconsistent/incomplete scale factors, the confusion of types and tokens (and the associated problem that not all changes are beneficial to all users), the clobbering of preparation steps, the incorrect scoping of identifiers. These are the sorts of errors commonly encountered by fifth graders <Goldstein, G., 73>. Were the horizons of the domain to be broadened -- for example to include touch and light sensitive turtles, or, perhaps, to include distinctions between alternative argument passing conventions (call-by-variable-substitution, call-by-name-substitution, call-by-variable-reference-substitution) -- other bugs would begin to be included:

unexpected side effects, forgetting previous conventions, addition of an input parameter which unintentionally denotes a component of an existing parameter. At least some of these evolutionary bugs may be common for more experienced programmers as well.

Sorting Algorithms -- Another Scenario:

> In real situations the complete specification of a problem is unknown, and what we really see happening is an evolutionary process. The sloppily formulated problem is given to the programmers, who produce a concrete realization. The users then complain about those properties of the realization which do not reflect their needs. This process iterates until the users are satisfied. As the users debug their ideas of what they need, the programmers debug their programs. At no point in this process do either the users or the programmers believe that they fully understand the problem. The iteration usually doesn't terminate because the users continue to evolve new ideas and requirements; so the programs must continually undergo revision due to "bugs" resulting from a misunderstanding or changing of intent. This remains true even in the case where the users are the programmers.
>
> <Sussman, 73>.

The last section attempted to illustrate how certain bugs can arise from the sequence of edits which extend the utility of a procedure. This section will emphasize a slightly different aspect of procedural evolution: how a complex program can be written by a succession of edits designed to eliminate problems in earlier versions. In order to avoid being misled by the peculiarities of a single domain, the next set of examples are taken from a different sphere: they are based upon a class of algorithms which sort an array of numbers into ascending order.

According to Knuth <73>, computer manufacturers estimate that over a quarter of the running time on their computers is spent on sorting. This relevance to practical programming should make a study of the evolution of algorithms for the task the more compelling, hopefully lending some generality to the ideas. Moreover, the ordering structure (i.e., a lattice) implicit in the operations of sorting reflects a fundamental aspect of human reasoning. For example, it is one of the three "mother structures" of the Bourbaki mathematicians. Furthermore, it corresponds to a logic of relationships present in the practical intelligence of even very young children. Prior to the

stage of concrete operations, children can place two or three sticks at a time into a series of ascending lengths (although they cannot coordinate all of the sticks of a larger group into a single sequence). By the age of seven, children have a remarkably advanced sorting algorithm. First, they find the shortest stick, and, having set that to one side, they proceed recursively to find the shortest of those remaining, and so on <Piaget, 71>. These kinds of intuitions can become the basis for quite sophisticated procedures.

The program constructed by Gregory Ruth analyzes and debugs implementations of sorting algorithms written by beginners. Its analysis is based on a program generation model [PGM] for each of several standard algorithms, knowledge of which is programmed into the system. Expert modules incorporate knowledge of common coding errors, and how to correct them. In the following, sorting is considered from a slightly different perspective, with a view to understanding the evolution of the algorithms themselves.

The simplest form of the sorting problem is to arrange an array of numbers into non-decreasing order. More generally, the task is to create a *file* of N *records*, each of which has a *key*, such that the records in the file are a permutation of the records in an input file, and the keys in the output file satisfy some ordering relation. This discussion is restricted to the former; while the latter formulation greatly complicates the specifications (e.g., by requiring *functional arguments*), it is not hard to edit simpler programs so as to encompass these options. Of course, if the harder problem were attacked by trying to modify a solution to the easier one, certain types of bugs might be expected to appear! For example, the final procedure might be required to be *stable* (i.e., records with equal keys retain their original relative order), but the purely numerical version might not have provided for this property.

Since the cardinality of the set of sorting algorithms is of the order of the number of

programmers in the universe squared, attention is further restricted to a particularly simple family of such methods, known as *sorting by insertion* <Knuth, 73>. The basic form of insertion sorting is an obvious technique which is certain to be re-invented by anyone who gives the problem a few minutes thought. Although in a strict sense, the current discussion does not concern *programs*, which are *implementations* of *algorithms*, for consistency a LOGO-like syntax will continue to be employed.

```
TO SORT1 :A :N                              !A is input array of nums!
10  ARRAY A,B[1:N]                          !B is output array; size N!
20  LOCAL I,J                               !A[:J] is the item we are doing!
25                                          !I is the search ptr for B!
30  MAKE B[1] A[1]                          !insert first num!
40  MAKE "J" 2                              !start processing nums 2-N!
50  MAKE "I" :J-1                           !searching for insrt loc!
60  IF B[:I]≤A[:J] THEN GO 100
70  MAKE B[:I+1] B[:I]                      !move right!
80  MAKE "I" :I-1                           !keep looking!
90  IF :I>0 THEN GO 60
100 MAKE B[:I+1] A[:J]
110 MAKE "J" :J+1                           !increment J; if done, return!
120 IF :J>:N THEN RETURN B
130 GO 50                                   !not done -- insert A[j]!
END
```

Of course, in most situations, there is no particular need to maintain the original array intact. Consequently, it becomes natural to consider whether the second array can be dispensed with altogether. If so, the storage requirements of the algorithm would be halved. A cursory analysis reveals that the role of array B is, in fact, minor, and it can be eliminated. For clarity, an additional local variable is introduced.

```
TO SORT2 :A :N                              !array A sorted in place!
10  ARRAY A[1:N]                            !N is the size of A.!
20  LOCAL I,J,K                             !J is where we are so far.!
30  MAKE "J" 2                              !The first num is ok wrt self!
40  MAKE "I" :J-1                           !I indexes compares for each J!
50  MAKE K A[:J]                            !If A[i]≤K, insrt loc found!
```

```
60  IF A[:I]≤:K THEN GO 100
70  MAKE A[:I+1] A[:I]                  !move right!
80  MAKE "I" :I-1                       !keep looking!
90  IF :I>0 THEN GO 60
100 MAKE A[:I+1] :K
110 MAKE "J" :J+1                       !increment J; if done, return!
120 IF :J>:N THEN RETURN A
130 GO 40                               !not done -- insert A[j]!
END
```

Unquestionably, this could be made more readable in a language which provided structured looping facilities, but the idea is clear: the left hand part of the array is in the correct relative order; when the next element from the right hand part is inserted into the left part, it is inserted into the correct place, and larger elements are bumped to the right; by induction, when the right part is null, the array will be completely sorted.

The annoying truth is that the above algorithm does in fact perform the specified task. In the usual sort of model of computation, specifications may describe *what* a program is to do, but not *how*, and certainly not *how well*. After formalizing the above demonstration of correctness, there would simply be nothing more to be said. There are many problems (e.g., playing chess) for which a straightforward but impossibly slow solution exists; these miss the point that the essence of procedural knowledge lies in the *efficient* use of available resources.

It would be desirable to be able to formalize various common sense notions about what it means to perform a task well; of course, this is an extremely difficult problem in general. In the present task, for example, there are certain requirements which one might wish to place on the algorithm. In some situations, the ordering function might be very costly to execute; for these, the goal would be to find solutions in which the number of comparisons made was minimal. If, as is the case with numerical keys, comparison is relatively inexpensive, the amount of reshuffling performed on the array might be of greater concern. For now, suppose that the constraints have

evolved so that the number of comparisons made is the relevant factor.

There is something very bad about the above algorithm. It exhibits a certain stupidity in its actions which deserves to be called a "bug". Let us suppose that we were given the task of sorting the numbers manually -- in the LOGO environment, this is known as "playing turtle". Perhaps we have already taken care of the first ten or so numbers, and we are now working on the eleventh. We need to find the proper location of this item relative to the other ten. Since "sorting turtles" have tunnel vision, we can only examine one of these ten numbers at a time. Does it seem reasonable that we would first look at the tenth, and then the ninth, and so on, until we found the correct spot? Maybe we would with ten, but if there were a hundred we would quickly tire of this, and start skipping around until we got into the right vicinity. This bit of common sense is the intuition behind *binary insertion*.

```
TO SORT3 :A :N                               !binary insertion sort!
10  ARRAY A[0:N]                             !A[0] a dummy, in case even!
20  LOCAL I,J,K,L,M                          !I,J,K approx as before!
21                                           !L delimits interval!
22                                           !M a dummy for interchanges!
30  MAKE A[0] -999999                        !some very small number!
40  MAKE "J" 2
50  MAKE "K" A[:J]
60  MAKE "L" :J DIV 2                        !truncation division!
70  MAKE "I" :L+(:J REM 2)
80  IF :K<A[:I] THEN GO 160
90  IF :K>A[:I] THEN GO 200
100 MAKE "I" :I+1                            !reshuffle and insert!
110 IF :I>:J THEN GO 240
120 MAKE "M" A[:I]                           !interchange!
130 MAKE A[:I] :K
140 MAKE "K" :M
150 GO 100
160 IF :L=0 THEN GO 110                      !K < A[:I] here!
170 MAKE "I" :I-((:L DIV 2)+(:L REM 2))
180 MAKE "L" :L DIV 2
190 GO 80                                    !continue searching!
200 IF :L=0 THEN GO 100                      !K > A[:I] here!
```

```
210 MAKE "I" :I+(:L DIV 2)+(:L REM 2)
220 MAKE "L" :L DIV 2
230 GO 80                                   !continue searching!
240 MAKE "J" :J+1                           !outer loop!
250 IF :J>:N THEN RETURN A
260 GO 50
END
```

Note that the plan of this version is analogous to that of the previous one. This might be made more apparent if the searching and reshuffling code were separately subroutinized. The basic idea is to start by comparing the current item to the middle of the left part of the array; this tells which interval to check next; then, this is repeated on the appropriate half of the left part, and so on. But it is still not quite what we originally had in mind. Since the range of values for the part of the array being examined is known, this information should be taken into account. Suppose that the next item were relatively large. Again, pretend that we are forced to perform the operations manually. Rather than starting in the middle of the already sorted part, we would normally try somewhere toward the right. This line of reasoning would lead us to re-invent the *interpolation sort.*

```
TO SORT4 :A :N                              !interpolation search sort!
10  ARRAY A[0:N]                            !sorted in place!
20  LOCAL I,J,K,L,M,U                       !L, U lower and upper bnds!
30  MAKE A[0] -999999
40  MAKE "J" 2
50  MAKE "K" A[:J]
60  MAKE "L" 1
70  MAKE "U" :J-1
80  MAKE "I" :L                             !I is next loc to check!
90  IF A[:L] = A[:U] THEN GO 130
100 IF :K < A[:L] THEN GO 160
110 IF :K > A[:U] THEN GO 270
120 MAKE "I" :I +                           !choose new I in [L,U]!
125              (:K - A[:L])*(:U-:L) DIV (A[:U]-A[:L])
130 IF :K<A[:I] THEN GO 210
140 IF :K>A[:I] THEN GO 240
150 MAKE "I" :I+1                           !insert between I and I+1!
```

```
160 IF :I>:J THEN GO 290
170 MAKE "M" A[:I]                    !M is dummy for interchange!
180 MAKE A[:I] :K
190 MAKE "K" :M                       !shove right!
200 GO 150
210 MAKE "U" :I-1                     !improve upper bound!
220 IF :U>:L THEN GO 80
230 GO 160                            !bounds narrowed to zero!
240 MAKE "L" :I+1                     !improve lower bound!
250 IF :U>:L THEN GO 80
260 GO 150                            !bounds narrowed to zero!
270 MAKE "I" :U                       !K>A[U], so insert above!
280 GO 150
290 MAKE "J" :J+1                     !outer loop!
300 IF :J>:N THEN RETURN A
310 GO 50
END
```

The latest version employs a fairly sophisticated technique for locating the spot to insert the next item. But the routine is thrashing for a different reason: it spends all of its time "reshuffling" -- bumping elements to the right in order to make room for the newly inserted item. If we were performing the operations on paper, we would run out of erasers!

There are a number of tacks which can be taken to surmount this problem. One approach is known as the *Shell sort*, but it is fairly complex and not central to this argument. A less sophisticated alternative will serve the purpose. The idea, again, is quite simple -- given the progress which has been made so far. Suppose we could write the numbers on a second sheet of paper. Would we start out writing them at the top? After a little consideration, it would occur to us to start somewhere in the middle; this would be a *two-way insertion* sort. But without even writing the program, we can apply what has been learned in a previous step: we would not want to insert the numbers right in the *center* of the available space. Rather, we would take into account the relative proportions, leading us to what might be called an *interpolation-search interpolation-insertion* sort. Most of us have probably used such a method, say, to alphabetize a list. The details

of this sort of algorithm are cumbersome, because of the danger that all of the items may "bunch up", causing us to get out the eraser again. When we weary of that, we start drawing arrows instead, and we have discovered *list insertion* sorting.

```
TO SORT5 :A :N                                      !list insertion sort!
10  ARRAY A[0:N]
20  LOCAL I,J,K,L                                   !L a linked list of nums!
30  MAKE A[0] -999999
40  MAKE "J" 2
50  MAKE "L" *LIST A[0] A[1]
60  MAKE "I" :L                                     !I is a tail of L!
70  MAKE "K" A[:J]
80  IF (BUTFIRST :I)=NIL THEN GO 120
90  IF :K≤FIRST(BUTFIRST :I) THEN GO 120
100 MAKE "I" (BUTFIRST :I)
110 GO 80                                           !next parsed as "RPLACD"!
120 REPLACE (BUTFIRST :I) BY *CONS :K (BUTFIRST :I)
130 MAKE "J" :J+1                                   !outer loop!
140 IF :J>:N THEN RETURN (FILLARRAY "A" :L)
150 GO 60
END
```

This latest version has decreased the amount of reshuffling considerably. However, a larger number of comparisons are now being made. It seems as though the sequence has split into two branches, each appropriate to pursue for different applications. It turns out, however, that there is a way to merge these into a single algorithm which improves upon both. The result is known as a *tree insertion* sort, and is the culmination of this effort. Detailed analysis of more sophisticated approaches to the sorting problem -- and some discussion of their evolution -- can be found in <Knuth 73>.

```
TO SORT6 :A :N                                      !tree insertion sort!
10  ARRAY A[0:N]                                    !L is the binary tree!
20  LOCAL I,J,K,L                                   !each entry a triple -- !
30  MAKE "J" 2                                      !value, lo link, hi link!
40  MAKE "L" *LIST A[0] NIL (*LIST A[1] NIL NIL)
50  MAKE "K" A[:J]
60  MAKE "I" :L                                     !I ptr into L for "RPLACA"!
```

```
70  IF :K > (FIRST :I) THEN GO 130
80  IF FIRST(BUTFIRST :I)=NIL THEN GO 110
90  MAKE "I" FIRST(BUTFIRST :I)
100 GO 70                                      !follow lo link!
110 REPLACE FIRST(BUTFIRST :I) BY *LIST :K NIL NIL
120 GO 170                                     !grow new lo link!
130 IF FIRST(BUTFIRST(BUTFIRST :I))=NIL THEN GO 160
140 MAKE "I" FIRST(BUTFIRST(BUTFIRST :I))
150 GO 70                                      !follow hi link!
160 REPLACE FIRST(BUTFIRST(BUTFIRST :I)) BY *LIST :K NIL NIL
170 MAKE "J" :J+1                              !above grows new hi link!
180 IF :J>:N THEN RETURN FILLARRAY "A" (TRESYM :L)
190 GO 50                                      !TRESYM -> symmetric order!
END
```

Pursuing this scenario has been instructive for a number of reasons. It seems fairly clear that it would not have been easy to arrive at the final version of the algorithm by "top-down-progressive-refinement" from the original specifications. Furthermore, at each step, the improvements which were incorporated were based on common sense notions of efficiency, which were reasonably straightforward to implement. Whereas the previous section examined how bugs can be introduced by successive edits which extend the utility of a procedure, the current section emphasized how a complex program can be written by a sequence of edits to a cruder, simpler version, each of which eliminates or reduces the severity of various deficiencies.

Toward a Theory:

> Any large program will exist during its life-time in a multitude of different versions, so that in composing a large program we are not so much concerned with a single program, but with a whole family of related programs, containing alternative programs for the same job and/or similar programs for similar jobs. A program therefore should be conceived and understood as a member of a family; it should be so structured out of components that various members of this family, sharing components, do not only share the correctness demonstration of the shared components but also of the shared substructure.
>
> <Dijkstra, 70>.

Procedures, like all symbolic descriptions, can be compared. The resulting *difference description* is itself a program, operating over a domain of legal edits to symbolic structures. When the programs being compared are related by causal evolutionary chains, this difference can be interpreted and classified in terms of purposeful plans in a meaningful way. The *Thesis of Evolution Driven Debugging* is that these longer range planning heuristics can be fruitfully associated with particular common types of bugs. The evolutionary approach to *program-writing* is to begin with a crude or existing program for a simpler related task, or a simpler but inefficient program for the same task, and successively extend and improve it, by analysis of its shortcomings according to straightforward but powerful common sense rules. The first phase of an evolution driven debugging monitor involves the semantic categorization of evolutionary edit types. The output of this phase would be an editing summary such as "variabilization of a rotation angle", "subroutinization of generic round", or "recursivization of repetitive open code". The first phase of an evolutionary program generator involves the recognition that the problem statement is either amenable to a coarse solution by a known general-purpose plan, or similar to the problem statement

for a related special-purpose plan. The output would be a generator function for instantiated candidate solutions, along with information about the kinds of difficulties which should be anticipated.

A unifying concept of the evolutionary approach is that procedures bear family resemblances to one another. This idea is a familiar concept in the literature on software portability and adaptability. However, the intention is to suggest a deeper relationship, one based upon the more global planning structures involved. This corresponds to Goldstein's suggestion that a new type of equivalence -- with respect to the plan -- be considered.

One kind of family is the collection of programs satisfying a given model. This is somewhat ambiguous, as it could be a reference to the fact that many programs will satisfy any given specification, however complete. In this discussion, there is more concern with the leeway provided by an *underdetermined* model. For example, most standard definitions of an equilateral triangle will not mention size or orientation. Consequently, any program drawing any size equilateral triangle in any orientation is correct with respect to such a model. That is, it satisfies all of the required properties. Natural extensions of these programs will identify these degrees of freedom and variabilize them. The resulting programs, which take, say, size and orientation as input parameters, might be said to summarize the simpler family. Only very rarely are the specifications for a programming task so tight that only a single class of equivalent computations can satisfy them. This is as it should be. Some constraints which might have been imposed because of a desire to cleanly characterize the task may not be essential to the users' needs, and could prove to be costly or difficult to implement. This is analogous to the role of possible worlds in model theoretic semantics. The model should specify all and *only* what is intended.

There is yet a deeper, more subtle family relationship. This is provided by the connections

between various models. For example, one kind of generalization is structural generalization of the model <Winston, 70>. When the model is being inferred by induction from a series of examples -- and refined by near non-examples -- a tree of such models is created, where previously explored branches correspond to rejected hypotheses about the important attributes of the description. When the model is made more general, the class of programs which are correct is enlarged; when it is refined, certain previous members are ruled out. Debugging of the model will have second order effects on debugging the program. A difficult but important area for investigation here is the relationship of model structure to program structure. It may be difficult to find the plan for, and therefore debug, programs with control structures not reflecting that of their models. Confusing or obscure code may be related to attempts to cut across the boundaries of model families.

As the specifications and code for a program evolve, a network of related programs is being constructed. Looking at this network from the perspective of a given program to be debugged, it can be viewed as a tree, with fixed instruction programs at the roots. It seems useful to assert that underlying every complex program is a family of simpler (ultimately fixed instruction) programs for which it can be considered the general case. One of the arguments advanced in favor of capturing such generalizations in code is that they abbreviate the total text involved, as in the usual explanations which novices receive about the virtues of subroutines. Of course attempting to recognize these abstractions is conceptually worthwhile for a variety of other reasons -- including clarity, debuggability, and modularity -- even if the routine is only to be used once. Conversely, ill-considered attempts to find the ultimate abstraction can be dangerous, or at best vacuous, as the following example illustrates.

```
TO MISCELLANEOUS :INPUT
5  LOCAL :ANGLE
10 MAKE "ANGLE" 72                          !default: draw pentagon!
20 IF :INPUT="SQUARE" THEN GO 70
30 IF :INPUT="TRIANGLE" THEN GO 110
40 FORWARD 100
50 RIGHT :ANGLE
60 GO 80
70 MAKE "ANGLE" 90
80 FORWARD 100
90 RIGHT :ANGLE
100 GO 120
110 MAKE "ANGLE" 120
120 FORWARD 100
130 RIGHT :ANGLE
140 FORWARD 100
150 RIGHT :ANGLE
160 FORWARD 100
170 RIGHT :ANGLE
END
```

The utility of describing individual procedures as members of such a network is that it suggests a possible basis for the theory of procedural ontogeny, consisting of syntactic and semantic components. The grammar would prescribe the reasonable ways in which programs -- and the programmer's insight underlying them -- develop. Presumably "reasonable" would be interpreted to exclude such oddities as the following.

```
TO FOO1
10 FORWARD 100
20 IF :X=1 THEN STOP
30 FOO1
END
```

```
TO FOO2
10 LOCAL :X
20 MAKE "X" 1
30 IF :X=1 THEN STOP
40 FOO2
END
```

It is difficult to imagine any purposeful routine for which either of these could be a bugged version.

In the more usual case, the intention would be to include common errors among the correct versions, so that they would appear in the structure along with their brethren. This could be

imagined to specify the linkages of an augmented transition network [ATN] <Woods *et.al.*, 72> or some comparable representation. The importance of the "transformational component", which might consist of arbitrary computations which test arcs or manipulate registers, (here as in the case of natural language), would then be that it provides a mechanism whereby one may describe regularities about families of programs which, derived from a common fixed instruction base, differ only by the application of a transformation such as "identify indistinguishable nodes and variabilize". The semantic component would provide interpretations of the resulting structures in terms of global planning strategies and their associated bugs. The conditions on arcs which lead to incorrect programs would specify particular bugs. Since the possible bugs would be described, in such a framework, as a function of the (semantic editing) transformation, rather than merely the particular program involved, there should be no constraint to produce a new network for each programming *task*. The extent to which one might need to supply such information for each new *domain* would thereby become a precise technical question.

To the extent that such evolutionary bugs turn out to depend on the domain, one might next be led to investigate high level ways to modify the network in a modular fashion. If new links could be "grown" when bugs which slipped through undetected were later pointed out, one could have a domain-sensitive *critic compiler*, analogous to that in HACKER, but based on the pattern of edits. When the earlier versions of a program were known (or could be ascertained on other grounds), a set of *demons* could be generated which are appropriate for the arcs associated with the transformational pattern. (A demon is a rule of the form "(pattern --> action)", which is triggered by events such as the addition of information to the knowledge base.) The demons would attempt to interpret future suspicious occurrences in light of the known evolutionary bug types when the system was run in *careful* mode. Where extremely high reliability became so

desirable that considerations of computational efficiency were secondary, the demons could be fired up at once. The advantage of the trial-and-error philosophy is that only those demons which are seen to be relevant later need actually be run. *This method is more powerful and exact than the crude evolution based heuristic "look at recent patches first"*. It is much more like the *analysis-by-synthesis* heuristic, "ask what you could do to a correct module to make it behave in this obnoxious way", which often leads straight to the source of difficulty.

The development of a program would usually follow a single path through the network. Although intermediate states might be skipped, (probably every state would need to have a default "JUMP" arc), the related bugs could be carried along until a correct node or leaf was reached. Paths below a correct node are no longer subject to bugs above them in the hierarchy. This is probably why programs are often built in this gradual evolutionary way -- at each stage, only a few bugs can be introduced. These are (hopefully) repaired before the more difficult next stage is attempted. If the full-blown program were attempted all at once (as the "top-down" school might advocate), every conceivable bug would have to be considered, and more difficult strategies for obtaining reliability would have to be adopted -- such as providing detailed simulations of lower level routines in order to verify the top level <Zurcher & Randell, 68>.

An important question for this research is the mechanism for determining that a bug in fact exists. Different approaches will result in different manifestations for the same underlying cause. What are the possible manipulations one may perform in trying to understand a procedure? The most obvious is to examine its code as data, looking for rational form violations, for example. Another is to run it, observing its behavior. This strategy of performance annotation can be quite valuable, but a problem arises for more complex programs: there are infinitely many possible inputs and runtime environments to test. One solution is to replace this *process annotation* by an

abstract *schematic annotation*, in which identifiers are bound to anonymous objects, and the code is simulated symbolically. This results (for the case of geometry) in a set of analytic equations describing the class of pictures, whose solution for some predicates may prove difficult or intractable. An interesting area for research is to attempt to provide an interface for such a scheme with programs such as MATHLAB <Martin, 67>. An alternative method, which seems favorable at the present time, is to test the procedure on examples, as human programmers currently do. This has the difficulty that no amount of testing can conclusively demonstrate that a program is correct, a serious concern in the drive toward greater software reliability. However, the art of testing reflects an important human ability to reason by carefully chosen examples. Some test cases are chosen because they are typical of the expected use. Others are specifically chosen to be extreme, testing the pathological possibilities. This is based on the powerful idea of the *continuity* of the physical world. With sufficient formal trappings, such arguments can form the basis for a more rigorous demonstration. Understanding the criteria for selection of such examples is an exciting researchable question. The role of models and/or runtime checking for such things as datatypes or prerequisite conditions on invocations of primitives is somewhat unclear at present. For example, a program which draws only squares must satisfy a rectangle model as a special case. But this fails to capture the intuition that a *typical* rectangle has adjacent sides unequal.

Such considerations argue for a knowledge representation more like Frame-Systems. Whereas the ATN-like approach described so far (with procedures located at the nodes of a network), might be said to emphasize what Piaget calls the "figurative" aspect of thought, its *dual* (with procedures as the links, and models as the nodes) emphasizes the "operative" aspect. A representation is sought in which these elements are unified, and yet the active, operative aspect remains primary.

The currently envisioned problem solving subsystem (and its model of the student's problem solving subsystem) is assumed to have two parts. The first is a base performance system which constitutes the current state of knowledge about the domain. This is organized as a linked network of frames. Each frame corresponds to an abstract situation, such as the current state of progress in developing a new procedure. Schematized states of the world are grouped together on the basis of certain patterns of features, for the purpose of common action specific to those features.

Effort at classification naturally breaks up into the frequently observed (e.g., <Groot, 65>) phases of thought. The earliest phase, *orientation*, consists of testing for key features in order to generate a working hypothesis. In an *exploratory* phase, instantiation of major terminal nodes takes place. During these first classifying phases, many of the methods being developed by work on hypothesis-driven recognition systems can be utilized. One of the desiderata leading to the effort to combine work on natural language with other aspects of intelligence is the quest for a unified formalism which can account for many such specific phenomena. This is evolutionarily sound (in the biological sense), since such mechanisms could have been adapted, for example, from existing visual routines. Conversely, it is difficult to imagine how "searching a problem space" or "resolution theorem proving" could have suddenly evolved in higher organisms.

Within each frame are procedures for further *investigation* and *verification*, and for handling unexpected problems. There will be a model of the situations to which the frame applies, which might be expressed as a procedure for recognizing them. Identifiers in the frame can be bound on invocation or receive typical default values. Their meanings can be linked to those of other frames by analogy or explicit generalization hierarchies. There will generally be commentary linking the model to one or more associated methods, and pointers to typical uses of the methods in higher level frames. The methods will be programs which transform the situation into new ones,

linking pairs into "action schemes" or "before-after transformations". For example, in the case of language, there might be a scheme for Schank's "PTRANS", another for "MTRANS", and a higher level, generalized "TRANS" scheme. Such schemes (linked pairs of frames) can be used in forward reasoning from the current state, or backward (teleological) reasoning from a desired goal state. A chain of schemes at a given level may represent a single unitary action -- and its associated commentary -- at the next higher level. Thus, whether a given frame represents a procedure (linking two other frames into a scheme), a plan, or a model of a situation, depends on the context. In particular, the evolution of a procedure is an instance of a chain of such schemes, where the nodes represent successive versions of the routine, and the links represent the editing history, some of which may include the introduction of bugs. Invocation of database methods is *frame-directed*, rather than being directed by the pattern of a single assertion. When faced with a problem, if there is no perfectly matching scheme in the database, the most similar scheme is used as a plan to be debugged. A huge network of such schemes are connected on the basis of similarity (common features) -- a powerful generalization of the simplistic but suggestive "difference operator table" <Newell & Simon, 63>.

The second major part of the problem solving subsystem (and the system's model of the student's problem solving subsystem) is responsible for monitoring progress, generalizing successful results, and assimilating new knowledge. It relies on the highly structured modular base being able to make its assumptions about the rest of the system explicit. Learning proceeds by discrete stages of development, or plateaus of problem solving ability. The mechanism for proceeding from one stage to another is the incorporation of new action schemes. These are created by copying, extending, and debugging existing ones. They are *not* like new, independent axioms, because they may interact in complex ways, and even contradict previous knowledge. Although superficially

similar to the addition of new productions, this process differs in the following ways: (a) the frames are larger and more structured than the condition part of productions; (b) the frames are linked (not necessarily one-to-one) into schemes which include both initial and *final* state specifications; (c) the control structure is explicit, as opposed to the implicit linear sequence of testing conditions; (d) the use of associative connections (hash coding techniques, similarity/discrimination networks) helps to find relevant schemes quickly.

The theory of procedural evolution to be used by a tutoring program or debugging monitor, then, is only superficially similar to an ATN; rather, it is embodied in the relationships of a network of schemes. The student's progress on a particular problem is seen as a subscheme in a larger chain. The system attempts to ensure that the chain is extended in certain ways, which are specified by its own, more abstract chain (one might draw an analogy to protein synthesis). This is to be accomplished by providing hints. A hint might help the student to avoid a blind alley or recognize a useful subscheme, or provide a prototypical example of a desired generalization. The most powerful hint, of course, is implicit in the order in which problems are posed.

The current section has suggested only the barest outlines of a theory of the evolutionary planning and debugging of procedures. An attempt has been made to marry the most valuable features of a wide range of candidate formalisms for representing such knowledge. What is significant is that the resulting synthesis seems to be applicable to other AI tasks, such as natural language understanding, as well. The next section will look at the problem of building a tutor program for geometry constructions using straight-edge and compass. The design of this system will provide a coherent methodology for fleshing-out and debugging these sketchy ideas.

A Consultant For Geometry Construction Problems:

> Anyone who has followed the argument this far will nevertheless feel the need to ask why the evolutionary process should work. What must nature, including man, be like in order that science be possible at all? Why should scientific communities be able to reach a firm consensus unattainable in other fields? Why should consensus endure across one paradigm change after another? And why should paradigm change invariably produce an instrument more perfect in any sense than those known before? From one point of view those questions, excepting the first, have already been answered. But from another they are as open as they were when this essay began. It is not only the scientific community that must be special. The world of which that community is a part must also possess quite special characteristics, and we are no closer than we were at the start to knowing what these must be. That problem -- What must the world be like in order that man may know it? -- was not, however, created by this essay. On the contrary, it is as old as science itself, and it remains unanswered. But it need not be answered in this place. Any conception of nature compatible with the growth of science by proof is compatible with the evolutionary view of science developed here. Since this view is also compatible with close observation of scientific life, there are strong arguments for employing it in attempts to solve the host of problems that still remain.
>
> <Kuhn, 62>.

In order to further pursue the investigation of issues surrounding the evolution of procedural knowledge, this section proposes the design and implementation of an educational application system. The proposed system would act as a consultant to students solving geometry construction problems. The task of writing such a program would help to clarify the ideas, ensure their generality by the extension to a new domain, and test their practical applicability. The program could be valuable as one facet of a larger reactive educational environment, such as Seymour Papert's *Mathland* project or the ICAI system envisioned in a recent proposal by John Seely Brown <75>. Examples of subsystems one might wish to include in such an environment would be a symbolic integration laboratory, an assistant to students learning to prove trigonometric

identities, or a simulator for simple physics problems.

In a typical situation, the geometry consultant is tutoring a high school student who is learning to construct geometric figures satisfying certain constraints, while employing only a straight-edge and compass. The program acts as bookkeeper, graphics display, critic, and advisor. When the student "gets stuck", it makes suggestions. It attempts to provide guidance in planning construction algorithms, and counterexamples to incorrect ones. By maintaining a detailed model of each students' progress, it is able to present a carefully organized sequence of related problems, illustrating variations or extensions of known construction techniques.

It is not taken for granted that computerizing the traditional high school curriculum is worthwhile. It may well be the case that very little of that which is currently taught in the schools ought to be taught at all. In particular, it is not the objective of this research project to automate the mindless memorization of dogma. What the schools ought to teach are skills in *thinking*.

The domain of straight-edge and compass constructions has much to recommend it as a vehicle for teaching problem solving skills. Euclidean geometry as a whole provides a paradigm for formal reasoning. Construction problems in particular require insight and ingenuity not unlike that of the engineer or programmer. In fact, the solutions to such problems are procedures quite analogous to those used in BLOCK'S WORLD construction or turtle geometry. The formal structure of the task is isomorphic to that of the mechanical engineer: e.g., build a bridge able to support a given load, withstand various stresses, using certain available techniques.

Such a program would be theoretically interesting in a number of respects. It could be regarded as a warm-up exercise for more difficult domains, such as the Mechanics of Solids <Miller, 74c> or the assembly of components on a circuit board subject to various constraints (e.g., heat dissipation, length of leads). The program would necessarily address those controversies to

which this paper has addressed itself in earlier sections. It would be forced to deal with the interfacing of multiple representations: Euclidean, analytic, computational. It would have to verify the correctness of procedures and soundness of plans; generate typical examples and pathological counterexamples; provide for plausible reasoning and qualitative explanation; understand logical entailment and debugging. The search for helpful similarities and analogies will require an instantiation of the *frame/scheme* concept, unclouded by the vagueness of less well-understood domains. There must be an analysis of tutorial dialogue, an examination of the postulates of purposeful conversation, an inquiry into the interplay of natural-language syntax, semantics, and pragmatics -- in a mini-world which is neither mysterious nor trivial. Most importantly, there is the chance to examine the effectiveness of the planning/debugging approach to learning and problem solving in a context to which many high school students are exposed.

A small, well-traveled domain underlies the technical feasibility of the project. The description problem has been addressed by a variety of theorem provers (e.g., <Goldstein, I., 73a>, <Gelernter *et.al.*, 63>, <Nevins, 74>). The problem solving aspects of constructions have been dealt with by a recent system <Funt, 73>. A number of self-help style texts (e.g., <Rich, 63>) catalogue the standard construction techniques, in a manner strongly suggesting frame-systems. Considerable success has been achieved, as noted earlier, by programs which debug analogous kinds of algorithms. A similar consultant program for the domain of electronics trouble-shooting has already been alluded to. Traditionally, insights into the problem solving process have been couched in examples from geometry constructions. Progress in very high level languages provides valuable primitives in which to express a theory; for example, a methodology for the intelligent handling of fault interrupts in a generalized control regime is emerging <Fahlman, 73a>. Even the secondary goal of conversing in a comfortable subset of English seems realistic in view of the

available parsing technology and strong expectations provided by the domain. In short, such a project provides a balanced proportion of the do-able and the worthwhile.

Nonetheless, there are serious problems with which previous ICAI-type systems have been unable to deal satisfactorily. The most important of these concerns the development of a good model of the student. The difficulties stem from inadequate description of the development of knowledge structures, and failure to recognize that the problem solver is operating at a number of different levels. The planning/debugging paradigm has begun to provide the tools necessary to surmount these barriers. With the development of a theory of the ontogenesis of procedural knowledge, it becomes conceivable to model the student as an entirely *procedural* problem solving system. Still, a great deal of care must be exercised in such an analysis. In particular, it is essential to distinguish between debugging the student's overt responses, which might be called the *object code*, and debugging the student's internal *procedure generation heuristics*, (which may also be operating at more than one level). The object code can be patched directly, but an inference is required to go from an error at this level to the bug it reflects in the internal process. The relationship between these *second order manifestations* and the conceptual underlying causes may be obscure. The monitor system must rely on the constraints provided by its *own* solution (and a model of its *own* internal procedures), the student's solutions to previous problems, knowledge of recently acquired theorems and construction algorithms, and so forth. At the same time, it must be prepared to accept partial plans for the solution which differ from those provided by its own construction algorithms, and determine whether they can be completed successfully. Where they can, it must rely heavily on a sound understanding of the student's approach in determining which hints will be helpful, and which would be "giving away too much", failing to demand a fair share of active participation from the student. Few systems to date have seriously attempted to deal with

these kinds of issues.

There are two primitive operations in the domain of straight-edge and compass constructions. One takes as input the names of two points and draws the straight line containing them. Either can be defaulted to a random point in the plane. The other expects to receive as input the name of a point, a length, and an internal angle, drawing the arc with center at the point, and the corresponding radius and extent. These are embedded in frame-like data structures, which include procedures for recognizing when a Cartesian representation of a figure is indeed a line or an arc, intrinsic commentary for justifying the legality of the procedures in terms of Euclidean theorems ("two points determine a line"), and links to extrinsic commentary in frames which employ them in typical higher-level purposes. Student remarks are to be interpreted as: factual questions intended to clarify the goal of a problem, partial plans for the solution, intensional and purpose commentary, and partial trace output from either the problem solving process or an interrupting monitor process.

The following sample interaction is presented in order to provide a clearer view of the intended system. It is based partly on experience gained by tutoring high school students on numerous similar examples. Commentary about the basis for system response appears in {curly braces}.

S: I'm confused. I don't understand how to do this problem.

T: Which problem are you working on? {presumably the system has already read in its file for this student and is updating its model.}

S: To inscribe a circle in a given triangle.

T: Ok, here is the triangle. Call it $\triangle ABC$. {importance of drawing diagram, naming. Discussion refers to figure 1.}

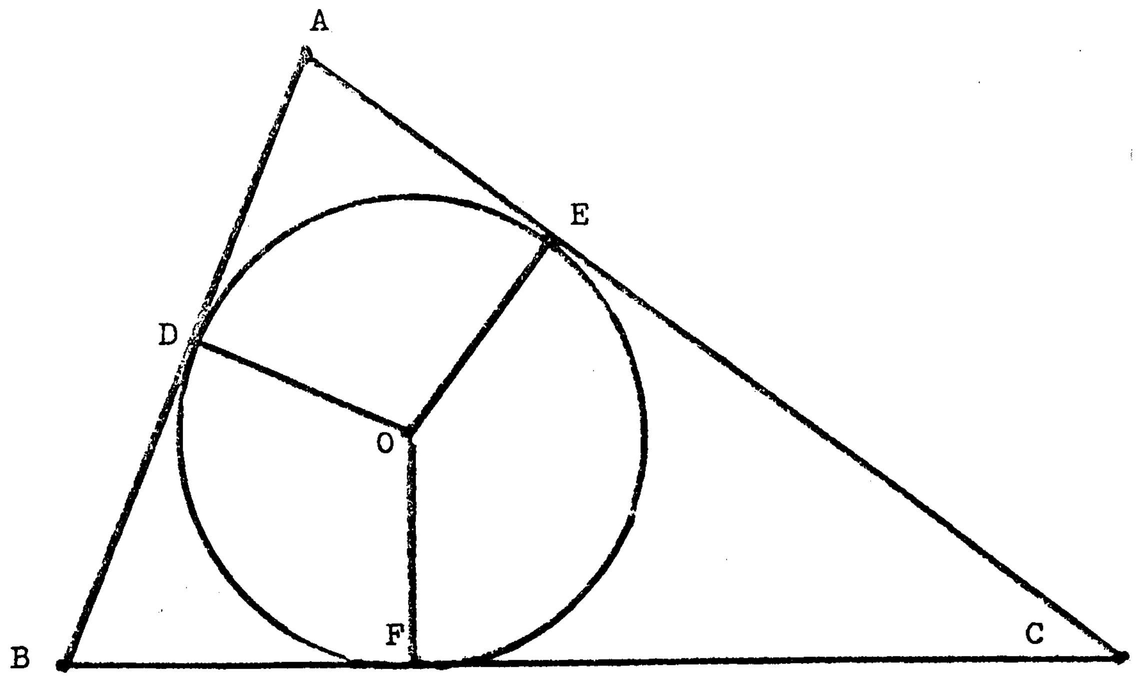

FIGURE 1

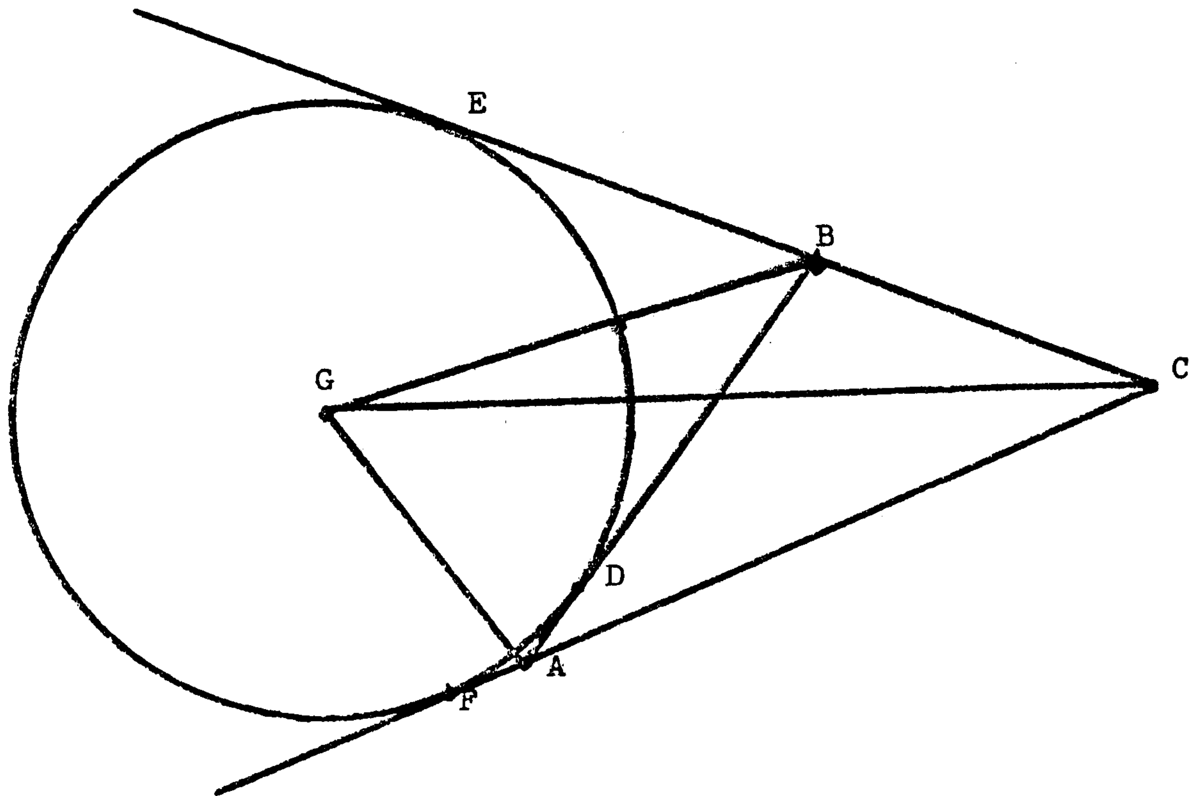

FIGURE 2

S: Now what do I do?

T: You are asked to draw a certain circle. What must you know in order to construct a circle? {Emphasizing the unknowns. Note that one reason for the difficulty that students experience with such problems is that they fail to recognize, at first, what the relevant unknowns *are*.}

S: Well, the radius, for one thing.

T: That's true. So at some point we will need to determine the radius, a line segment. What else must you know, which is even more important? {A more experienced student would immediately have seen that the essence of this problem is to find the location of the center.}

S: The center.

T: Right, the location of the center, which is a point. Let's start with that. {Breaking the problem into subproblems. Note that choosing to search for a point suggests the use of one of the primary schemes for construction problems -- Polya's "pattern of two loci". It is not coincidental that a mere handful of such schemes are adequate for solving a wide range of problems in our domain. Nevertheless, even the system's method has a "bug", in that it can fail to determine whether a solution exists; in such cases, the system should be able to resort to more powerful techniques outside of the purely Euclidean ones.}

S: Well, it has to be somewhere inside the triangle.

T: Ok. {The claim is accepted at face value, although this student might find the proof quite hard; e.g., closely related problems, for which the statement happens to be false, come to mind at once. It would hardly be appropriate to challenge the student's correct intuition at this point.} Very well, call the center "O". Does the circle touch side AB? {Expanding definition of "inscribed".}

S: Yes, in one point.

T: So segment AB must be *tangent* to the circle? {paraphrasing, trying to draw connections to relevant frames.}

S: Right, the radius is perpendicular to AB.

T: What about the other two sides? {To elicit the obvious generalization.}

S: It's the same.

T: Ok, look at the points where the circle touches the triangle. Call them D, E,

and F. {Updates the display.} What can you say about △DAO and △EAO? {Here, using intensional commentary from the system's own solution to prompt a corresponding step in the student's plan. This seems to be one fairly powerful standard strategy that many tutors employ. Of course, there are several others as well.}

S: They are congruent.

T: And how about △CEO and △CFO?

S: Congruent, also.

T: So what does that tell you about angles DAO and EAO, for example? {Still pursuing the strategy of working backward through its own plan, prompting with the intrinsic commentary.}

S: They're equal.

T: Ok. Now if you were given angle EAD, could you construct the ray AO so that <EAO = <DAO? {Pointing out a useful subroutine.}

S: That's just bisecting a given angle.

T: What about the ray CO?

S: Oh, ok, so we've got O.

T: What else do you need? {Restoring the previous context. During this strategy of simplifying the problem by relaxing constraints, the student may have forgotten that the original problem involves more than one step.}

S: The radius. But that's just segment OE.

T: How do you construct OE? {The next main step.}

S: It's already there!

T: No, we just pretended we knew where it was, so we could talk about it. {The nominal fallacy bug.} Given O, and AC, how do you construct E?

S: ???

T: What is the relation between segments OE and AC? {Using the intensional commentary strategy on a subproblem.}

S: Perpendicular.

T: Can you construct a perpendicular to a given line (AC) from a given point (O)? {Describing the fully instantiated invocation pattern for the appropriate frame.}

S: Oh yes, that's easy.

T: {Note that the above required T to have a model of the student which included the appropriate frame.} Well, then what is the procedure to inscribe the circle? {Popping back up to the main problem for which all the subroutines have been written.}

S: First, bisect <BAC. The center is on the bisecting ray. Drop a perpendicular from the center to AC.

T: Wait a minute. {Unsatisfied prerequisite bug.} You left something out. From what you said, I could draw O on the bisector ray of <BAC *outside* the triangle. {Generation and display of counterexamples.}

S: Oh, ok you have to bisect the other two angles, too. The intersection is O.

T: But what if they don't meet at a point? {It is not enough for a procedure to be correct, it must also *appear* to be correct!}

S: Um, I don't know.

T: Can you think of an example where they don't meet at a point?

S: What about a long skinny triangle?

T: Like this? {The system moves point C slowly to the right.}

S: Well, I guess they always have to meet at the same point.

T: That's right, but do you see how you could prove it?

S: Not exactly, maybe if we undid what we said before?

T: That's ok for now. {This was an interesting point to mention, but the proof is beyond the level of this student. People can understand procedures more readily than their proofs of correctness.} Anyhow, do you see any place that you did more work than you had to? {Rational Form violation -- the purpose of a step is already satisfied.}

S: No.

T: Why did you need to construct the third bisector? {Just localizing the error to a particular step.}

S: I guess you don't.

T: Ok. What's left?

S: Drop the perpendicular. That is the radius. Draw the circle.

T: Ok. Here is the final figure. Do you remember the problem we began working on the other day? {While willing to work on problems in the order preferred by the student, the system takes the initiative in pursuing the most logical (from an evolutionary point of view) line of development.}

S: Yes: "Given a triangle, to construct a circle which is tangent to one side and to the extensions of the other two sides."

T: Good. Here is the figure we were using. ABC was the given triangle. {Discussion refers to figure 2. Note that the type of the unknown (a circle) and the type of the given (a triangle) are the same as those in the previous example. This suggests extending that solution to a more general form; moreover, it suggests that the *method* of finding that solution is a special case of a more powerful planning scheme.} What is the most essential unknown?

S: The location of the center, point G.

T: Good. And what conditions can be used to constrain it? {In order for the "scheme of two point sets" (our rendition of Polya's terminology) to be applicable, G must lie at the intersection of two loci whose components are rectilinear or curvilinear.}

S: Well, I have one idea. Since segments GE and GF are radii, they must have equal length. Since they are also perpendicular to the two sides, triangles FGC and EGC must be congruent. So, by corresponding parts of congruent triangles, G must lie on the bisector of angle C.

T: All right. {This was not the procedure that the system intended to use, but it can be completed successfully, so the system accepts it for now. It can still be used to illustrate the basic reasoning pattern. The system uses this more abstract scheme to generate a particular hint.} So you have restricted G to lie on a certain line. If you could find another line containing G, then the center would be located at their point of intersection.

S: That's as far as I could get, though. The only other lines are GE and GF, but to draw these I would need to know E or F.

T: No, there is also GD. Do you see that D does not necessarily lie on the bisector of angle C? {In the original display, triangle ABC was draw so as to cause some confusion when segment GC was added. This isolates a sort of "accidental equality"

bug. Note, however, that in this gradual sequence of related problems, there are fewer such bugs; if the system had posed a problem which was further afield, there would tend to be more. Furthermore, it would be harder for the monitor to discover their underlying conceptual causes.}

S: Oh. Well, I suppose if side AB were more *slanted*, ...

T: {An extremely acute version is displayed, as in the figure. At this point, the system should provide a hint which will help the student to draw on previously learned schemes. The idea of the hint is to mention one or two key features of the frame's precondition.} Can you do anything with quadrilateral GDBE?

S: Um, ...

T: What is a generally good thing to do with quadrilaterals?

S: Oh! Divide them into triangles! Let's see, please show chord ED.

T: Well, that's one possibility. {An unfortunate choice, but it is shown.} Or, you could try to get some line containing G.

S: Ok, show GB instead.

T: Go ahead. {ED is erased to avoid cluttering up the display.}

S: I think the two triangles are similar ... but I only know one angle.

T: If you don't know the angles, what about the sides?

S: Of course! The triangles are congruent, by, um, ... A.S.S. ...?

T: What!?

S: Er, by Hypotenuse-Leg.

T: All right. So what can you say about <EBG and <DBG?

S: Equal. So GB is the angle bisector. We're done.

T: Good. Now, I want to illustrate a different solution for the same problem. Is that alright? {It would probably be well for the system to review the current solution at this point, perhaps even printing out the entire procedure in a LOGO-like syntax. In any case, it uses the now-familiar problem to explore alternative approaches, extending the planning method to a more general form.}

S: I would have thought *one* would be enough!

T: The method we have been using so far has been to simplify the problem to locating a point, and then to try to describe the point as the intersection of two point sets which could be constructed separately. {Teaching procedures by telling is always legitimate!} In your solution, one of the point sets was segment GB, the bisector of <EBD. The other was GC, the bisector of <BCA. Can you think of any others you could have used?

S: Not offhand.

T: Is there anything special about vertex B? {"Nothing Special" is a standard clue to look for symmetry.}

S: No. I suppose we could bisect angle FAD instead of angle EBD. It's the same.

T: Is there a way to solve it without constructing segment GC? {This first step, bisecting angle C, is still central to the student's approach. Challenging it may lead the student to additional insights.}

S: I guess you could just use segments GB and GA as the two point sets... {etc.}

This hypothetical scenario should help to illustrate the goal of a geometry constructions consultant program, and by analogy, the goal of other tutor-like systems. It is doubtful that such an intelligent and flexible system could be achieved in six months or a year. Nevertheless, the planning/debugging paradigm and ideas stemming from it provide a new collection of tools with which a good start can be made.

Conclusion:

A focus on planning and debugging procedures underlies the enhanced proficiency of recent programs which solve problems and acquire new skills. By describing complex procedures as constituents of evolutionary sequences of families of simpler procedures, we can augment our understanding of how they were written and how they accomplish their goals, as well as improving our ability to debug them. To the extent that properties of such descriptions are task independent, we ought to be able to create a computational analogue for genetic epistemology, a theory of *procedural ontogeny*. Since such a theory ought to be relevant to the teaching of procedures and modelling of the learner, it is proposed that an educational application system be implemented, to help to clarify these ideas. The system would provide assistance to students solving geometry construction problems.

Bibliography:

<Abelson *et.al.*, 73>
Abelson, Hal, Nat Goodman, and Lee Rudolph, *LOGO Manual*, M.I.T. A.I. Lab, LOGO Memo #7, August 10, 1973.

<Balzer, 72>
Balzer,"Automatic Programming", Item 1, Institute Technical Memorandum, University of Southern California, Information Sciences Institute, September, 1972.

<Bartlett, 64>
Bartlett, Sir Frederic C., *Remembering: A Study in Experimental and Social Psychology*, Cambridge University Press, 1964.

<Bobrow & Norman, 74>
Bobrow, Daniel G., and Donald A. Norman, "Towards Provocative Dialogue", (paper presented at Carbonell Symposium, to appear in Bobrow & Collins, 1975.), 1974.

<Brown, A., 74>
Brown, Allen L., "Qualitative Knowledge, Causal Reasoning, and the Localization of Failures", M.I.T. A.I. Lab, Working Paper #61, March 1974.

<Brown, J., 75>
Brown, John Seely, "Uses of Advanced Computer Technology and Artificial Intelligence for Teaching Mathematical Problem Solving Skills", B.B.N. Proposal #P75-CSC-4A, January 1, 1975.

<Brown & Collins, 74>
Brown, John Seely, and Allan Collins, *Proposal for: Computer Models of Functional Reasoning*, B.B.N. Proposal #P74-BSC-17, January, 1974.

<Brown *et.al.*, 74>
Brown, John Seely, Richard R. Burton, Alan G. Bell, *SOPHIE: A Sophisticated Instructional Environment for Teaching Electronic Troubleshooting (An Example of AI in CAI)*, Final Report, B.B.N. Report #2790, A.I. Report #12, March, 1974.

<Burton, 73>
Burton, Richard S., "A Thesis Proposal -- A Computer Based Geometry Laboratory", University of California at Irvine, (unpublished), 1973.

<Collins *et.al.*, 74>
Collins, Allan, Eleanor H. Warnock, Nelleke Aiello, Mark L. Miller, "Reasoning From Incomplete Knowledge", (paper presented at Carbonell Symposium, to appear in Bobrow & Collins, 1975.), 1974.

<Dijkstra, 70>
Dijkstra, E.W., "Structured Programming", in *Software Engineering Techniques*, Report on a Conference sponsored by the N.A.T.O. science committee, Rome, Italy, (Oct., 1969), April, 1970.

<Dunlavey, 74a>
Dunlavey, Michael R., "Knowledge About Interfacing Descriptions", Thesis Proposal, Information and Computer Sciences Dept., Georgia Institute of Technology, Feb., 1974.

<Dunlavey, 74b>
Dunlavey, Michael R., "Knowledge About Interfacing Descriptions", M.I.T. A.I. Lab, Working Paper #62, March, 1974.

<Fahlman, 73a>
Fahlman, Scott E., *A Planning System for Robot Construction Tasks*, M.I.T. A.I. Lab, TR-283, Cambridge, 1973.

<Fahlman, 73b>
Fahlman, Scott E., "A Hypothesis-Frame System for Recognition Problems", M.I.T. A.I. Lab, Working Paper #57, December, 1973.

<Fikes, 70>
Fikes, R., "REF:ARF: A system for solving problems stated as procedures", *Artificial Intelligence Journal*, 1: No. 1, 1970.

<Floyd, 67>
Floyd, Robert W., "Assigning Meanings to Programs", *Proc. Symp. App. Math.*, AMS, vol XIX, 1967.

<Freiling, 73>
Freiling, Michael J., "Functions and Frames in the Learning of Structures", M.I.T. A.I. Lab, Working Paper #58, December, 1973.

<Friedberg, 58>
Friedberg, R.M., "A Learning Machine, part I", *IBM Journal of Research and Development*, 2: 2-13, Jan, 1958.

<Friedberg *et.al.*, 59>
Friedberg, R.M., B. Dunham, and J.H. North, "A Learning Machine, part II", *IBM Journal of Research and Development*, 3:282-287, June, 1959.

<Funt, 73>
Funt, Brian V., "A Procedural Approach to Constructions in Euclidean Geometry", Master's Thesis, University of British Columbia, 1973.

<Gelernter, 63>
Gelernter, H., "Realization of a Geometry-Theorem Proving Machine", in Feigenbaum & Feldman (eds.), *Computers and Thought*, pp. 134-152, New York, McGraw-Hill, 1963.

<Gelernter *et.al.*, 63>
Gelernter, H., J. R. Hansen, and D. W. Loveland, "Empirical Explorations of the Geometry-Theorem Proving Machine", in Feigenbaum & Feldman (eds.), *Computers and Thought*, pp. 153-163, New York, McGraw-Hill, 1963.

<Goldberg, 73>
Goldberg, A., "Computer-Assisted Instruction: The Application of Theorem-Proving to Adaptive Response Analysis", Institute for Mathematical Studies in the Social Sciences, Stanford University, Technical Report #203, 1973.

<Goldstein, G., 73>
Goldstein, Gerrianne, "LOGO Classes Commentary", M.I.T. A.I. Lab, LOGO Memo #10 (LOGO Working Paper #5), February 6, 1973.

<Goldstein, I., 73a>
Goldstein, Ira P., *Elementary Geometry Theorem Proving*, M.I.T. A.I. Memo #280, April, 1973.

<Goldstein, I., 74a>
Goldstein, Ira P., *Understanding Fixed Instruction Turtle Programs*, M.I.T. A.I. Lab, Ph.D. Thesis, AI-TR-294, 1974.

<Goldstein, I., 74b>
Goldstein, Ira P., "Summary of Mycroft: A System for Understanding Simple Picture Programs", M.I.T. A.I. Lab, Memo #305, LOGO Memo #10, May 1974.

<Goldstein *et.al.*, 74>
Goldstein, Ira. P., Henry Lieberman, Harry Bochner, Mark L. Miller, *LLOGO, An Implementation of LOGO in LISP*, M.I.T. A.I. Lab, A.I. Memo #307, LOGO Memo 11, 1974.

<Groot 65>
Groot, Adriaan D. de, *Thought and Choice in Chess*, translated from the Dutch (1946) version by G.W. Baylor, New York, Mouton, 1965.

<Hewitt, 70a>
Hewitt, Carl, "More Comparative Schematology", M.I.T. A.I. Lab, Memo #207, 1970.

<Hewitt, 70b>
Hewitt, Carl, "Teaching Procedures in Humans and Robots", M.I.T. A.I. Lab, Memo #208, 1970.

<Hewitt, 71>
Hewitt, Carl, "Procedural Semantics: Models of Procedures and the Teaching of Procedures", Courant Computer Science Symposium 8, (Dec. 20-21), *Natural Language Processing*, ed. Randall Rustin, New York, Algorithmic Press, Inc., 1971.

<Hewitt, *et.al.*, 73>
Hewitt, Carl, Peter Bishop, Richard Steiger, Irene Greif, Brian Smith, Todd Matson, Roger Hale, *Automating Knowledge Based Programming Using ACTORS*, Revised, Dec., 1973.

<Hoare, 69>
Hoare, C.A.R., "An Axiomatic Basis For Computer Programming", *Communications of the ACM*, 12, 10, pp. 567-583, October, 1969.

<Kimball, 73>
Kimball, R.B., "Self-Optimizing Computer Assisted Tutoring: Theory and Practice", Institute for Mathematical Studies in the Social Sciences, Stanford University, Technical Report #206, 1973.

<Knuth, 73>
Knuth, Donald E., *The Art of Computer Programming*, vol. 3, "Sorting and Searching", Reading, Mass., Addison-Wesley Publishing Co., 1973.

<Kuhn, 62>
Kuhn, Thomas S., *The Structure of Scientific Revolutions*, Chicago, The University of Chicago Press, 1962.

<Kuipers, 74>
Kuipers, Benjamin, "An Hypothesis-Driven Recognition System for the Block's World", M.I.T. A.I. Lab, Working Paper #63, 1974.

<Laventhal, 74>
Laventhal, Mark Steven, "Verification of Programs Operating on Structured Data", M.I.T. Project MAC TR-124, 1974.

<Liskov & Zilles, 74>
Liskov, Barbara, and Stephen Zilles, "Programming with Abstract Data Types", M.I.T. Project MAC (unpublished), 1974.

<Manna & Waldinger, 71>
Manna, Zohar, and Richard J. Waldinger, "Toward Automatic Program Synthesis", *Communications of the ACM*, vol. 14, no. 3, pp. 151-165, March, 1971.

<Martin 67>
Martin, William A., "Symbolic Mathematical Laboratory", M.I.T. Project MAC TR-36, 1967.

<McCarthy, 59>
McCarthy, J., "Programs With Common Sense", in: Blake, E.D., and A.M. Uttley, (eds.), *Proceedings of the Symposium on Mechanization of Thought Processes*, National Physical Laboratory, (Teddington, England), London, HM Stationary Office, 1959.

<McDermott, 74>
McDermott, Drew V., "Advice on the Fast-Paced World of Electronics", M.I.T. A.I. Lab, Working Paper #71, May, 1974.

<Miller, 74a>
Miller, Mark L., "Thoughts on Chess, Learning, and A.I.", M.I.T. A.I. Lab, (unpublished), 1974.

<Miller, 74b>
Miller, Mark L., "Some Ideas on Program Writing", M.I.T. A.I. Lab, (unpublished), 1974.

<Miller, 74c>
Miller, Mark L., "Learning to Solve Problems in the Mechanics of Solids", M.I.T. A.I. Lab, (unpublished), 1974.

<Minsky, 73>
Minsky, Marvin L., "Frame-Systems: A Framework for Representation of Knowledge", M.I.T. A.I. Lab, (unpublished), 1973.

<Moise & Downs, 64>
Moise, Edwin E., and Floyd L. Downs, Jr., *Geometry*, Teachers' Edition, Reading, Mass., Addison-Wesley Series in Science and Mathematics Education, 1964.

<Moon, 74>
Moon, David A., *MACLISP Reference Manual*, Revision 0, Cambridge, Mass., M.I.T. Project MAC, 1974.

<Naur, 66>
Naur, P., "Proof of Algorithms by General Snapshots", *BIT 6*, 1966.

<Newell & Simon, 63>
Newell, Allen, J.C. Shaw, and H.A. Simon, "Chess-playing Programs and the Problem of Complexity", in Feigenbaum & Feldman (eds.), *Computers and Thought*, New York, McGraw-Hill, 1963.

<Newell & Simon, 72>
Newell, Allen, and Herbert A. Simon, *Human Problem Solving*, Englewood Cliffs, N.J., Prentice Hall Inc., 1972.

<Nevins, 74>
Nevins, Arthur J., "Plane Geometry Theorem Proving Using Forward Chaining", M.I.T. A.I. Lab, Memo *303, 1974.

<Norman, 73>
Norman, Donald, "Learning and Remembering: A Tutorial Preview", reprinted from *Attention and Performance IV*, New York, Academic Press, Inc., 1973.

<Okumura, 73>
Okumura, K., "LOGO Classes Commentary", (Oct. 16, 1972 - June 8, 1973), M.I.T. A.I. Lab, Working Paper *6, LOGO Memo *11 1973.

<Papert, 71a>
Papert, Seymour A., "Teaching Children to be Mathematicians Versus Teaching About Mathematics", M.I.T. A.I. Lab, Memo *249, 1971.

<Papert, 71b>
Papert, Seymour A., "Teaching Children Thinking", M.I.T. A.I. Lab, Memo *247, LOGO Memo *2, 1971.

<Papert, 72>
Papert, Seymour A., "Theory of Knowledge and Complexity", Amsterdam Lectures (anonymous notes), 1972.

<Papert, 73>
Papert, Seymour A., "Uses of Technology to Enhance Education", M.I.T. A.I. Lab, Memo *298, LOGO Memo *8, June, 1973.

<Papert & Solomon, 71>
Papert, Seymour A., and Cynthia Solomon, "Twenty Things to do With a Computer", M.I.T. A.I. Lab, Memo *248, LOGO Memo *3, 1971.

<Paterson, 70>
Paterson, Michael S., *Equivalence Problems in a Model of Computation*, M.I.T. A.I. Lab, Memo #211, 1970.

<Paterson & Hewitt, 70>
Paterson, Michael, and Carl Hewitt, "Comparative Schematology", M.I.T. A.I. Lab, Memo #201, 1970.

<Piaget, 71>
Piaget, Jean, *Genetic Epistemology*, (translated by Eleanor Duckworth), The Norton Library, New York, W. W. Norton & Co., Inc., 1971.

<Polya, 71>
Polya, G., *How To Solve It*, Princeton, N.J., Princeton University Press, second edition, 1971.

<Pratt, 74>
Pratt, Vaughn R., "A Modal Logic for Correctness and Termination", M.I.T. Seminar, *Semantics of Programming Languages*, 6.891, (unpublished class notes), April 23, 1974.

<Rich 63>
Rich, Barnett, *Plane Geometry with Coordinate Geometry*, Schaum's Outline Series, New York, McGraw-Hill Co., 1963.

<Rubinstein, 74>
Rubinstein, Richard, *Computers and a Liberal Education: Using LOGO and Computer Art*, Ph.D. Thesis, U.C. Irvine, 1974.

<Ruth, 74>
Ruth, Gregory Robert, *Analysis of Algorithm Implementations*, M.I.T. Project MAC TR-130, 1974.

<Schank, 72>
Schank, Roger, "Conceptual Dependency: A Theory of Natural Language Understanding", in *Cognitive Psychology*, vol. 3, pp. 552-631, 1972.

<Silverman, 74>
Silverman, Howie, "Understanding Digitalis Therapy", M.I.T. A.I. Lab, (unpublished), 1974.

<Smith *et.al.*, 73>
Smith, Brian, Dick Waters, Henry Lieberman, "Comments on Comments", or, "The Purpose of Intentions and the Intention of Purposes", M.I.T. A.I. Lab, (unpublished), 1973.

<Solomon, 73>
Solomon, Cynthia, "LOGO Classes Commentary", (April 23, 1973 - June 8, 1973), M.I.T. A.I. Lab, 1973.

<Sussman, 73>
Sussman, Gerald Jay, *A Computational Model of Skill Acquisition*, M.I.T. A.I. Lab, TR-297, 1973.

<Teitleman, 70>
Teitleman, W., "Toward a Programming Laboratory", in *Software Engineering Techniques*, Report on a Conference sponsored by the N.A.T.O. science committee, Rome, Italy, (Oct., 1969), April, 1970.

<Waldinger & Lee, 69>
Waldinger, Richard J., and Richard C.T. Lee, "PROW: A Step Toward Automatic Program Writing", *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, D.C., (eds. Donald E. Walker and Lewis M. Norton), May 7-9, pp. 241-252, 1969.

<Winston, 70>
Winston, Patrick, *Learning Structural Descriptions From Examples*, M.I.T. A.I. Lab, TR-231, 1970.

<Wirth, 71>
Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM*, *14*, 4 pp. 221-227, April, 1971.

<Woods *et.al.*, 72>
Woods, W.A., R.M. Kaplan, and Bonnie Nash-Webber, *The Lunar Sciences Natural Language Information System: Final Report*, B.B.N. Report #2378, 1972.

<Zurcher & Randell, 68>
Zurcher, F.W., and B. Randell, "Iterative Multi-level Modelling -- A Methodology for Computer System Design", Thomas J. Watson Research Center, Yorktown Heights, New York, I.F.I.P. Congress, Edinburgh, August 5-10, 1968.