DOCUMENT RESUME

ED 336 062 IR 015 059

AUTHOR McAllister, Alan
TITLE An Analysis of Problem Solving Strategies in LOGO Programming Using Partially Automated Techniques.
PUB DATE Mar 91
NOTE 72p.; Paper presented at the Annual Meeting of the American Educational Research Association (Chicago, IL, April 3-7, 1991).
PUB TYPE Information Analyses (070) -- Reports - Research/Technical (143) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS Academically Gifted; Behavioral Science Research; *Cognitive Processes; *Computer Assisted Instruction; Computer Software; Educational Strategies; Elementary Secondary Education; *Interaction Process Analysis; Microcomputers; *Problem Solving; *Programing
IDENTIFIERS LOGO Programing Language

ABSTRACT

Computer assisted instructional environments offer unique opportunities for the analysis of learning and problem solving processes. In this paper, partially automated techniques for process analysis are applied in a study of the problem solving strategies of a group of gifted students learning the Logo programming language. The objectives of the paper are to develop strategy specifications, construct a model of the students' programming process, and illustrate the application of partially automated interaction process analysis. The computer software programs included tools for collecting, displaying, encoding, and dynamically representing the students' problem solving in the Logo environment. Two tasks were administered: a graphics task midway through the program, and a word-and-list task at the end. Problem solving strategies were identified in each task, and it was found that they differed across tasks and across subjects. Results indicate that students' strategies are a function of specific task demands and the students' representations of the problem, their knowledge of program design, and their knowledge-base of specific methods for implementing their designs in code. Recommendations for future behavioral science research in the area of problem solving include routine use of microcomputers to monitor students' performances and provide appropriate individualized assistance instantaneously. (66 references) (Author/DB)

An analysis of problem solving strategies in LOGO programming using partially automated techniques

To be presented at the 1991 annual meeting of
the American Educational Research Association,
Chicago, Illinois,
April 3, 1991

Alan McAllister
Department of Psychology
York University
March 1, 1991

Abstract

By facilitating the collection and analysis of records of behavior, computer-based learning environments offer unique opportunities for the analysis of learning and problem solving processes. In this paper, partially automated techniques for process analysis are applied in a study of the problem solving strategies of a group of students learning LOGO.

The objectives of the paper are to develop strategy specifications, to construct a model of the students' programming process, and to illustrate the application of partially automated process analysis. The software tools used in this study have as their theoretical foundation the concept of problem solving as a goal-directed search in problem spaces associated with a task. The tools consist of programs for collecting, displaying, encoding, and dynamically representing the students' problem solving in the LOGO environment. A group of nine gifted students, ten to fourteen years of age, were trained over a fifteen week period. Two tasks were administered, a graphics task midway through the program, and a word-and-list task towards the end of the program. The tools are used to identify the students' problem solving strategies in each of the tasks, and these strategies are framed within the context of an agenda model of search control. The strategies identified differed across tasks and across subjects. According to this model, the students' strategies are a function of specific task demands and the students' representations of the problem, their knowledge of program design, and their knowledge-base of specific methods for implementing their designs in code.

Key words and phrases: problem solving strategies, programming, LOGO, and automated analysis.

Table of contents

1. Introduction 1
2. Paradigms applied to the study of programming 1
   - 2.1. The application of the effects paradigm 2
   - 2.2. The application of the expert-novice paradigm 3
   - 2.3. The application of the process analytic paradigm 5
     - 2.3.1. The paradigm itself 5
     - 2.3.2. Themes in research on programming 7
     - 2.3.3. Limitations of the existing paradigm 10
3. The approach in this study 11
4. Subjects, training, and tasks 13
5. The tools and their theoretical foundation 14
6. A preliminary formulation of the model 17
7. Results 19
   - 7.1. The first task: House-Playhouse 20
     - 7.1.1. The programs 20
     - 7.1.2. The strategies 21
   - 7.2. The second task: High-Low 28
     - 7.2.1. The programs 29
     - 7.2.2. The strategies 30
8. Discussion 35
9. Conclusions and implications 37

List of Tables

Table 1: Components of the House-Playhouse figure programmed 39
Table 2: Components of the High-Low game programmed 40

List of Figures

Figure 1:	Traditional protocol analysis	41
Figure 2:	The two problem spaces	42
Figure 3:	The record generator	43
Figure 4:	The encoder	44
Figure 5:	The problem behavior graph generator	45
Figure 6:	The agenda model	46
Figure 7:	The output of the House-Playhouse programs	47
Figure 8:	The structure of the House-Playhouse programs	48
Figure 9:	Example 1 (episode 1, task 1, S2)	49
Figure 10:	Example 2 (episode 1, task 1, S3)	50
Figure 11:	Example 3 (episode 1, task 1, S1)	51
Figure 12:	Example 4 (episode 1, task 1, S4)	52
Figure 13:	The structure of the High-Low programs	53
Figure 14:	Example 5 (episode 1, task 2, S1)	54
Figure 15:	Example 6 (episode 1, task 2. S2)	55
Figure 16:	Example 7 (episode 1, task 2, S4)	56
Figure 17:	Example 8 (episode 1, task 2, S3)	57

1. Introduction

In the field of education few would doubt that it is more illuminating to look at the process a student goes through to achieve a result than merely to look at the product of that process. Yet process analysis is seen by many teachers and researchers as too time-consuming, subjective, and unreliable to provide a basis for evaluation and instruction and for academic research. However, computer-based learning environments offer unique opportunities for overcoming the problems associated with process analysis. The interactions of a student with a computer can be automatically recorded by the computer, and programs can be constructed to automate the coding and analysis of these records.

The study described in this paper takes advantage of the opportunities for process analysis within computer-based environments by focusing on LOGO programming and analyzing the problem solving strategies employed by a group of young gifted students on two tasks involving graphics and word-and-list processing. While young students' LOGO programming has been extensively studied over the past decade and more, the process analytic approach has not been employed to any great extent in this research. The particular objective of this paper is to illustrate how automated techniques can be used to model the programming process; the more general objective is to illustrate in part the development and application of a process analytic paradigm of research in a field in which it is particularly apt but largely ignored. As a preliminary to the discussion of the study and its results, some of the paradigms that have been applied within the literature on programming are outlined and the results they have yielded examined.

2. Paradigms applied to the study of programming

A variety of paradigms has been applied to the study of programming (Curtis, 1988); three of these are particularly relevant to this discussion. The effects paradigm examines the issue of the

transfer of skills from the domain of programming; this has been the predominate paradigm in the study of LOGO programming within the educational research community. The expert-novice paradigm has been widely applied in studies of adult (university-age and professional) programmers to reveal the nature of programming knowledge by focusing on differences that the acquisition of knowledge makes. The third paradigm, the process analytic paradigm, employs process records, primarily recordings of verbalizations, to analyze the strategies employed by programmers of various levels of skill. In this section, the various paradigms and their application to the study of programming are examined.

2.1. The application of the effects paradigm

In much of the research on LOGO programming, young students are pre-tested and post-tested with measures thought to have some relationship to the kinds of skills exhibited in programming (e.g., geometric skills, problem-solving, or recipe reading), and the LOGO trained groups are matched with controls (e.g., students receiving a regular academic program, training on CAI software, or computer literacy instruction).

Transfer of other than specific skills is notoriously difficult to demonstrate (Perkins & Saloman, 1989; Saloman & Perkins, 1989), and attempts to demonstrate the transfer of skills acquired through LOGO programming to other domains have been true to form. The use of this paradigm has produced, at best, mixed and sometimes contradictory results, and negative judgements have increasingly been made on the justifiability of teaching programming to young students (Khayrallah & Meiraker, 1987; Krasner & Mitterer, 1984; Michayluk, 1986). However, rather than abandoning the teaching of programming in schools, it may be more appropriate to look at how the paradigm has been applied in much of this research.

What the decade-long investigation of this issue has accomplished

is the recognition that to achieve transfer some appreciable level of programming skills must first be developed (Kinzer, Littlefield, Delclos, & Bransford, 1985; Klahr & Carver, 1988) and that how programming is taught (e.g., discovery, guided discovery, mediated learning, structured learning) makes a substantial difference to both learning and transfer (Delclos & Kulewic, 1985; Delclos, Littlefield, & Bransford, 1985; Emihovich & Miller, 1986; Kull, 1986; Lehrer, 1986; Lehrer & Smith, 1986; Leron, 1985: Pea and Kurland, 1984). When training is based on a solid analysis of the activity of programming and sensitive measures of learning and transfer are used, transfer of skills acquired through programming can be achieved (Carver and Klahr, 1986; Klahr & Carver, 1988).

2.2. The application of the expert-novice paradigm

One of the major deficiencies of the LOGO debate within the educational community is the relative lack of awareness of the literature that has been generated within the larger research community on the nature of programming and the programming process. Much of this research has been conducted using the expert-novice paradigm in which the performance of groups of novices and experts is compared on programming and related tasks.

A number of studies (Adelson 1981, 1984b; Bateson, Alexander, & Murphy, 1987; McKeithen, Reitman, Rueter, & Hirtle, 1981; Shneiderman, 1976; Shneiderman, 1977; Shneiderman & Mayer, 1979) have looked at differences between novices and experts' recall for programs and, consistent with findings in other domains, have found that experts are no better than novices in recalling programming material that lacks inherent meaning (e.g., randomly chosen program fragments), but are better at recalling meaningful material (e.g., fragments from a functional program). Experts' superior recall is attributed to their ability to chunk information and organize it in relation to the programming knowledge that they have available to them. With

experience, this information is hierarchically organized into larger chunks which are indexed to facilitate efficient access (Bateson et al, 1987).

This organization of knowledge allows experts to form higher level representations than novices. Whereas novices form concrete representations based on surface features of the language ("syntax") and the specifics of how a programme works (Adelson, 1981, 1984b) and tend to formulate solutions for specific instances of a problem based on its literal features (Hoc, 1977; Weiser & Shertz, 1983), experts form representations which have to do with the meaning of statements within a programming language ("semantics"), with the function of these statements as parts of programs, and with the generalities of what a program does (Adelson, 1981, 1984b).

The nature of the functional units of domain specific knowledge in programming has been investigated in a number of studies (Davies, 1990a, 1990b; Gilmore & Green, 1988; Rist, 1986; Soloway & Ehrlich, 1984; Soloway, Ehrlich, Bonar, & Greenspan, 1982). Sheil (1981), relying on the concept of schemata, has characterized programming knowledge as

> a collection of units ... each of which is organized as a program fragment, abstracted to some degree, together with a set of propositions about its behavior and rules for combining it with others, and indexed in terms of the problem classes for which it is appropriate. (p. 118)

The elements of these schemata that have been focused upon are: (1) the programming plan structures or abstract program fragments which represent stereotypical action sequences in programming (Soloway & Ehlich, 1984; Davies, 1990b); (2) rules of programming discourse which specify conventions in programming and govern the composition of plan structures into programs (Soloway & Ehrlick, 1984); and (3) the selection rules which govern the implementation of programming plan structures in appropriate situations (Davies, 1990b). This research (Davies, 1990b) suggests that only novice programmers lack plan structures, but that what develops from intermediate to expert levels is the use and deployment of appropriate selection rules and the correct

use of discourse rules.

2.3. The application of the process analytic paradigm

These concepts of knowledge organization, level of representation, and functional units of domain-specific knowledge appear as well in studies of programming using the process analytic paradigm. Since this is probably the more unfamiliar of the three paradigms and so much of the later discussion in the paper depends on a familiarity with it, the discussion needs to be somewhat extended.

2.3.1. The paradigm itself

In this paradigm, the objective is to construct process models which explain the behavior studied. Ideally, these are computational models which are "sufficient" in the sense that they generate the behavior which is to be explained, subject, of course, to known constraints of human psychology. Strategies take a central role in models of problem solving processes; they are the "programs" which generate a path through a problem to its solution and determine the sequences of processes which occur.

The empirical basis for the construction and testing of these models consists of records of subjects carrying out the process which is modeled. These records can be recordings of such things as step-by-step moves, response times, eye movements, and verbalizations, although verbalizations are generally employed. Protocol analysis, which involves recording and analyzing subjects' verbalizations during problem solving, is the most widely applied method for collecting and analyzing process data (Ericsson & Simon, 1984; Newell & Simon, 1972). In this method, subjects are asked to think aloud during problem solving and

their verbalizations are recorded[1], transcribed, encoded, and then subjected to various forms of analysis. Figure 1 depicts the method in its traditional form.

One way of developing a coding vocabulary is through a task analysis which identifies the "problem spaces" for a task (Newell and Simon, 1972). This approach to task analysis assumes that the structure of the task environment determines the ways in which subjects represent a problem and that those representations are in the form of problem spaces (Simon, 1978).

A problem space is defined as a set of symbolic structures, the states of the space, and a set of operators for transforming one state into another; sequences of state-operator pairs define paths for moving through the space. A problem within a problem space is defined by an initial state, a goal state and a set of path constraints; some problems involve more than one problem space (Kant & Newell, 1984; Simon & Lea, 1976). Within this framework, the problem solving process is a goal-directed search which begins from an initial state constituted by a given configuration of a problem space and, through the application of operators to states, transforms one state into another and proceeds towards a goal state along a path subject to specific constraints and under the guidance of search control knowledge (Newell, 1980).

Once a vocabulary of "objects and relations" (Ericsson & Simon, 1984) has been developed from such a task analysis and the records are encoded, the behavior can be represented in an abstract form. One form of behavioral representation is a problem behavior graph which plots the subject's path through a problem space as a sequence of state-operator pairs in which operators are applied to states to produce new states.[2]

---

[1]If the problem solving involves observable actions, these actions are usually recorded as well and represented in the transcript as referents for the verbalizations.

[2]The graph usually takes the form of a kind of flow diagram consisting of boxes, representing states of the problem space, and arrows representing the application of operators to those states to

A problem behavior graph can be used in two ways with respect to a process model. In constructing a model, rules can be abstracted from the problem behavior graph's representation of the conjunctions of states and operators in a subject's solution path to account for the subject's successive steps in solving the problem. These rules can in turn be used to describe the subjects' strategy and to construct a computational model capable of reproducing the subject's behavior (Ohlsson & Langley, 1985). If, on the other hand, the process model has been developed independently, problem behavior graphs can be used to test the model by matching the graphs to the "trace" generated by the process models (Newell & Simon, 1972).

Protocol analysis as a paradigm of research is not without its limitations; however, before examining them, some results from studies of the programming process which have employed variants of this approach need to be discussed.

2.3.2. Themes in process analytic research on programming

Clear themes have emerged from the application of this paradigm to the study of the programming process. Some of these themes have to do with contrasts in how strategies direct the process, with ways in which knowledge is developed in relationship to two problem spaces, and with the nature of programming knowledge.

The programming process can be characterized in terms of a design phase and an implementation phase, although these phases are seldom distinct in actual practice. In the design phase an overall representation of the problem is developed from the task specifications. This may amount to a sketchy solution or a well-articulated algorithm. In the implementation phase, the overall representation is translated into code, tested, and debugged.

Consistent with expectations based on research on the different

---

produce new states.

types of knowledge available to experts and novices, research on the programming process (Adelson & Soloway, 1985; Jeffries, Turner, Polson, and Atwood, 1981) suggests that, because experts are more able to represent problems in the abstract than are novices, they can develop an overall solution to a problem by decomposing it into a hierarchy of subproblems; because they have a rich repertoire of previously developed representations, they can move from one level of the design solution to the next lower level by drawing on available knowledge. Even when appropriate knowledge is not available, experts persist in this mode of operating at one level of abstraction at a time (Adelson & Soloway, 1985). Novices, on the other hand, are not as able to develop an overall representation of the problem, and, while they may attempt to decompose the problem into subproblems, they tend to work on one aspect of the decomposed problem at a time and expand it into a detailed local solution (Jeffries et al, 1981).

According to this research, the contrast between experts and novices is in terms of the experts' use of strategies which drive the solution process in a top-down and breadth-first direction on a decompositional search tree as opposed to the novices' use of strategies which drive the solution in a bottom-up and depth-first direction. However, other studies (Fisher, 1986, 1988; Rist, 1986, 1989) have qualified this view. These studies suggest that novices and experts proceed in a more flexible manner depending upon the availability of knowledge and the requirements of the problem. Novices successively refine solutions to a problem when they can easily retrieve appropriate programming knowledge, but when they cannot, they tend to focus on a local solution to a problem and expand it (Rist, 1986, 1989). Flexibility is shown by advanced programmers as well, and they may adopt strategies which drive the process in a variety of different directions at different points in the solution process depending upon the requirements of the problem (Fisher, 1986, 1988).

Protocol analysis studies suggest that programming knowledge can be extended and elaborated through simulation and experimentation

(Adelson & Soloway, 1985) and in relationship to more than one problem space (Dunbar & Klahr, 1989; Kant & Newell, 1984; Klahr & Dunbar, 1988). In skilled design, mental simulation and search in an hypothesis space, a space of partially articulated solution models, are done as a check on the evolution of the design as it is refined. Feedback through trials in an experimental space, such as running a program on a computer system, may also be used; trials of this nature tend to be used in situations in which mental simulation alone is not sufficient and more specific information is necessary to develop a solution. Strategic approaches can be differentiated on the basis of the space which is searched with some subjects preferring search in one space rather than another. Mental simulation and search in an hypothesis space tend to be linked with higher levels of skill and the availability of knowledge, and hence with retrieval and expansion of previously developed knowledge, whereas test-case execution and search in an experimental space tend to be linked with less skilled approaches and lack of knowledge, and hence with the development and expansion of new representations.

Research focused on the implementation process suggests that it has many characteristics in common with the design process. Similar strategic approaches are found in both processes; the predominant view in the literature on implementation, as in design, is that skilled implementation is a largely top-down, breadth-first process. Just as skilled planning requires keeping the overall representation of the problem in mind as the design evolves, skilled implementation involves being able to keep the program structure in mind while dealing with the detail of the program (Vessey, 1985, 1986). As in the design process, the overall representation of the program provides a context for the development and deployment of functional units of knowledge. These units of knowledge allow the programmer to map the evolved plan into the code of a specific language (Brooks, 1977), to evaluate and change code before it is executed (Gray & Anderson, 1987), and to seek clues to bugs in written code and to evaluate hypotheses when they are tested (Gould,

1975).

2.3.3. Limitations of the existing paradigm

Most of the models which have emerged in research on the programming process are either in the form of step-by-step descriptions of the programming process (Adelson & Soloway, 1985; Kant & Newell, 1984; Gray & Anderson, 1987) or flow-charts or "process hierarchy" models (Carver, 1987; Dunbar & Klahr, 1989; Fisher, 1986, 1988; Gould, 1977; Klahr & Carver, 1988; Klahr & Dunbar, 1988; Vessey, 1986, 1988). Only one effective computational model has been developed (Brooks, 1977), and that model is based on one subject writing 46 lines of code.

There are a number of reasons for the lack of effective computational models in the field, most having to do with the difficulties of following through with the paradigm. Some of these difficulties are related to the tremendous costs in terms of time and resources associated with collecting, transcribing, and coding verbalizations. As a result of these costs, the number of subjects in these studies is severely restricted; in the process analytic research surveyed above, none involved more than 20 subjects, and most had fewer than 10. The use of single subjects and small groups is typical of research within this paradigm, and there are plausible theoretical justifications for this (see Newell & Simon, 1972); however, results from the application of this approach are likely to be viewed within the larger research community as restricted in their generalizability. Furthermore, because of the intensity of the data collection method, even one record can compose a huge data set; not only are the costs high in collecting and analyzing such records, but there are great risks to reliability in dealing with that amount of data and making inferences from it. Each inference that is made from the record of transcribed utterances introduces an uncontrolled element of intelligence and care (Waterman & Newell, 1972). Finding consistent patterns in the data can be extremely difficult (Fisher, 1988), and often several passes through

the record will be necessary to ensure that the coding has been done consistently.

Reliability and generalizability are the Achilles' heels of protocol analysis. One way of addressing these weaknesses is to automate as much as possible of the data collection and analysis tasks, the idea being that, with some degree of automatization of the encoding and analysis, protocol analysis will be easier, faster, and more reliable, and more subjects can be used. The prospect is held out by automatization is that the reliability and generalizability of the results of the analysis will be greater and the construction and testing of computational models will be facilitated.

Two approaches to automating protocol analysis as it relates to the programming process can be distinguished. One approach has been to develop systems to provide automated assistance for the coding of verbal protocols and for data analysis of the coded protocols (Fisher, 1988; Sanderson, James, and Seider, 1989; James, Sanderson, & Seider, 1990). Another approach is to develop more fully automated systems which focus on non-verbal trace records which can be collected as the task is performed and analyzed using pre-determined encoding schemes (Redmond & Gasen, 1988, 1989).

There are clearly trade-offs in using one approach rather than another. On the one hand, the more automated a system is the more reliable it is likely to be and the easier it is to apply to larger numbers of subjects; on these grounds, automated recordings of interactions on a computer are preferable to verbalizations as the focus of analysis. On the other hand, verbalizations are undoubtedly a richer source of information about cognitive processes. Furthermore, while building intelligence into a system can eliminate possible sources of unreliability, this can itself be a costly process and may restrict the sensitivity of the analysis.

While approaches such as these hold out promise for strengthening the paradigm, they are limited in their present state of realization. For instance, the systems developed so far to analyze the programming

process do not generate full-bodied behavioral representations in the form of problem behavior graphs, nor do they do cognitive modeling. Automated process analysis systems with these potentials have been developed for domains such as mathematics (Brown & Burton, 1978; Kowalski & VanLehn, 1988; Ohlsson & Langley, 1985; VanLehn & Garlick, 1987) but not for the domain of programming.

3. The approach in this study

The study described in this paper involves the use of automated tools for the collection, coding, and analysis of records of the programming process. The approach used is a form of partially automated process analysis. It is distinct from traditional protocol analysis in that it focuses on non-verbal trace records rather than verbalization records[3]. It relies on a predetermined coding scheme and transforms the coded records into goal-based problem behavior graphs from which strategy specifications can be derived. Two basic assumptions are made for this analysis:

1) that a correspondence exists between the thought processes of programmers and the code they generate, and

2) that the appearance of new code and changes to code from one point in time to the next are significant and measurable points in the programmer's problem solving (Redmond & Gasen, 1988, 1989).

The "system" for collecting and analyzing the records consists of a set of tools in the form of programs written in LOGO by the author. These tools collect, display, and code the records and assist in the development of behavioral representations in the form of problem behavior graphs. No attempt has been made here to construct an effective computational model; rather than giving detailed sets of rules

---

[3]Verbalizations during performance of the tasks in this study were recorded and transcribed; however, these verbalizations are merely used for clarification of the analysis.

to specify strategies (a "production system"), the model which is developed specifies types of strategies and the search control regime within which they operate. This means that the sense of sufficiency that is claimed for the model is relatively weak (although no more so than the majority of studies in the area). Furthermore, the records used to develop the model are the ones for which it is designed to account, so that the model cannot be claimed to be generalizable beyond the records analyzed[4]. This paper, then, is a report of the partial realization of the goal of automating process analysis and a step on the way to developing a method for studying processes which is both reliable and generalizable.

In the sections which follow, the specifics of the study are discussed, a theoretical basis for the software tools and a preliminary formulation of the model are given, and the results from the application of the tools to the subjects' performance on the two tasks are analyzed.

4. Subjects, training, and tasks

Nine students, ten to fourteen, who had been identified as gifted according to the criteria of their school board and who had had previous programming experience, were given instruction on LOGO programming on a two hours per week basis for fifteen weeks. These students were chosen because it was felt that they would be highly motivated and would be able to acquire the relatively large amount of knowledge required in the time that was available. The instructional approach varied according to the need for mastery of content and the individual needs of the students. The approach can best be described as eclectic and included discovery learning, mediated learning (see McAllister, 1985), and structured lessons.

---

[4]The model as it is presented here is to be evaluated against an independent set of records in an extension of this study which is in progress.

Two tasks were chosen as the focus of the study; the first was administered midway through the program and covered graphics programming, the second was administered at the end of the program and covered word-and-list and interactive game programming. (The actual tasks and the circumstances of their administration are detailed below.)

5. The tools and their theoretical foundation

Three sets of software tools are involved in the analysis of the subject's performance on the two programming tasks: (1) a "record generator" which consists of programs for recording and displaying the records, (2) an "encoder" which consists of a program which displays the records segmented into units of analysis and identifies changes from one version of a procedure to the next, and (3) a "problem behavior graph generator" which consists of programs which assist in the development of a problem behavior graph from a coded record. The theoretical foundation for these tools is derived from an analysis of certain invariants of the LOGO programming environment (specifically, the IBM LOGO used in this study) in terms of two problem spaces.

LOGO is an interpreted language and highly interactive. Programming in LOGO involves writing procedures; within this environment, procedures are developed in either a define mode or an edit mode, and instructions are carried out in an immediate mode. Programming in this environment can be conceptualized as taking place as a search in two basic problem spaces, a primary problem space (the "program space"), which is linked with the define and edit modes in which procedures are developed, and a secondary space (the "trial space"), which is linked with the immediate mode in which experimentation, testing, and debugging takes place. Search in these two spaces involves going back and forth between them-- writing procedures, trying them out, perhaps seeing what some specific instructions will do, and using the feedback that the computer gives to make changes to the procedures. (Figure 2 is a broad characterization

of the two problem spaces in the LOGO environment.)

The record generator consists of a recording program and a displaying program (see Figure 3). The recording program constructs a record which consists of "snapshots" of procedures in the editor when it is exited, the definition of a procedure when it is completed in define mode, and the set of instructions entered in immediate mode when they are invoked (with a carriage return)[5]. The record that results from the use of this program during a programming session is a time-stamped series of these snapshots in the form of lists. The displaying program can display a record either in print or on the screen or it can replay the record so that what was occurring on the system at the points at which the snapshots were taken can be observed.

<Insert Figure 3 about here>

There are clear referents for the states and operators of the problem spaces in the record[6]. The snapshots of the define and edit modes can be taken to represent states of the program space and the snapshots of the immediate mode to represent states of the trial space. Operators are evident in overt actions within the programming environment. In the LOGO implementation used for this study, IBM LOGO (IBM, 1980), the variety of types of these actions is limited: within the define mode, procedures can only be defined one at a time and instructions can only be appended; within the edit mode, it is possible to work on several procedures, but the only editing actions are appending, inserting, and deleting instructions; and within the immediate mode, it is possible only to invoke instructions and, in the case of an interactive program, make inputs to the keyboard.

---

[5]Although errors always occur within the immediate mode, the record generator records the initiating action and the error message as a separate category.

[6]States and operators in a problem space are "mental constructs" (Newell, 1980), but when a program is developed on a computer, the symbolic states may designate states of the device and the operators may lead to actions on the device (Newell, 1980; Newell & Simon, 1976).

Accordingly, the actions of defining, appending, inserting, and deleting evident in the record within the define and edit modes represent types of operators within the program space, and the actions of invoking instructions and making inputs to the keyboard within the immediate mode represent types of operators within the trial space.

The encoder (see Figure 4) is used to identify changes to the states of the program space and the types of operators used to affect these changes. For its analysis it takes as its basic unit an instruction, which is defined as a LOGO primitive or a procedure and whatever inputs it may have. The program compares the present formulation of a procedure with its last formulation on a unit by unit basis, and identifies the changes made and the types of operators used. The output of this program, combined with a classification of the types of operators used in the trial space, supplies a coded record of the states visited and operators used in the subject's search through the two problem spaces associated with a task.

<Insert Figure 4 about here>

The path of a subject through a task consists of this sequence of states and operators evident in the coded record. A plot of the path of the subject through the problem spaces can be made using a problem behavior graph. The particular type of problem behaviour graph adopted for this study is goal-based and groups operators according to goals and plots the subject's path in terms of its sequential goal structure.

The problem behavior graph generator (see Figure 5; see Appendix A for a key to interpreting the graph) automates the plotting of the problem behavior graph. At its present stage of implementation, there are two steps. An analysis program generates a file in the form of a series of schematic lists. These lists contain information about what occurred in each line of the record, and there are open slots for the operators and the goals. Working from a coded record, the researcher uses domain knowledge concerning the effects of the operators and

knowledge of the intentions of the subject[7] to insert the appropriate goals and operators in the open slots. This file is used as input to a program which displays the operators in terms of their goal structure, and this is the problem behavior graph.

<Insert Figure 5 about here>

This type of problem behavior graph simplifies and reduces the data, while, at the same time, it provides potentially informative hypotheses about the cognitive processes which occurred. Specifically, it represents the intentions of the subject and the means that were employed to realize those intentions. To extract the implications of these graphs, a preliminary model of the programming process must be formulated.

6. A preliminary formulation of the model

The model of the programming process formulated here is based on the idea that there is a fixed cycle within the search process and that, within this cycle, there is an underlying control mechanism for scheduling goals. That control mechanism can be thought of according to the metaphor of an agenda (VanLehn & Garlick, 1987).

The idea is that the programmer is given the specifications for a program embodied in the task instructions; this is a kind of imposed agenda, but it is on the basis of it that the programmer draws upon a knowledge base to form an overall representation of the problem. In representing the problem, the programmer seeks to decompose it into more easily manageable parts; these are the "items on the agenda" and become subgoals of the task. However, this representation will change

---

[7]Knowing the intentions of the subject is relatively simple since the subjects were working from a precise list of specifications for the task. It is also possible to work backward in the record from what was actually achieved to hypothesize what the programmer's intentions were, and, since there is a verbal record, it is possible to consult it to see what the subject had to say at the time.

throughout the programming process, and with these changes, the agenda will have to be updated. In this way, the agenda becomes an ordering of the active goals of the problem solver.

Human-computer interactions are frequently characterized in terms of a control cycle[8] consisting of

1) selecting a goal and deciding to act on it,
2) selecting an operator and executing the operator,
3) perceiving and interpreting the system state that results,
4) comparing the system state with the set of goals for the task,
5) and completing the cycle by selecting a new goal

(Newell, 1980; Norman, 1986).

This is essentially what occurs within the programming process, as well (see Figure 6). The scheduling of goals is a strategic selection from the agenda which is based on a comparison of the current state of the agenda and the current state of the problem. A goal is selected and the decision is made to act on it, and operators are selected and executed by overt actions on the system. These actions change the system, which in turn changes the state of a problem space. A comparison of the agenda with the state of the problem changes the programmer's representation of the problem and the agenda is updated. At this point, a new cycle may begin or the cycle may be terminated if the agenda has been completed or the problem abandoned.

<Insert Figure 6 about here>

Methods, which describe procedures for accomplishing goals, link goals and operators (Card, Moran, & Newell, 1983) and can be cast in terms of conditional sequences of states, goals, and operators (e.g., "if the state of the problem is x and the state to be achieved is y, do

---

[8]Frequently, the idea of a goal-stack is used to describe the ordering and scheduling of goals. However, there is a lot of rigidity to a last-in, first-out goal-stack, and it is not a particularly helpful idea in describing semantically rich tasks like programming.

z")[9]. A programmer may have available as stored knowledge a set of possible methods for accomplishing a goal; in this situation, the programmer must merely select which method is preferable and apply it. On the other hand, methods for accomplishing a given goal may not be available, and the programmer must develop a method for accomplishing that goal; even though methods may not exist for accomplishing the original goal, the programmer may still have methods (e.g., search procedures) available for finding an appropriate method to accomplish it.

The problem solving strategies which control the programming process can be described in terms of the particular way in which goals are scheduled and selected and in terms of the specific methods selected and employed to accomplish these goals. In the development of a problem solving strategy, the programmer's knowledge is drawn upon to determine the overall representation and decomposition of the problem as embodied in the agenda and to provide methods for carrying out goals or for finding the appropriate methods for carrying out goals.

This sketch of a model can be used as a framework for understanding the results obtained from the application of the tools to the two programming tasks. Through an examination of these results, the nature of the strategies can be elucidated and the model filled-out to provide an interpretation of the students' programming.

7. Results

In this section, the results of the application of these tools to the two programming tasks are examined. The two tasks are discussed separately; the format for this discussion involves a review of the training leading up to the task and the circumstances of its

---

[9]The concept of methods employed here borrows much from the concept of programming plans or schemata as discussed in the literature (see above), but it is a more generic concept.

administration and an examination of the programs that the students developed and the strategies that they used to develop them.

7.1. The first task: House-Playhouse

Prior to the administration of the first task, the students had spent eight weeks working on graphic programming and were generally proficient with navigating the screen and drawing simple objects. In the weeks immediately preceding the test, the students had been taught how to make basic shapes (e.g., square, triangle, and any sided polygon) using constants and variables, and a test was administered in class on the material that had been covered. The results from the test were reviewed with the group as a whole, and individual tuition on this material was given to any of the students according to their needs as identified in the test.

The House-Playhouse task required the students to draw a house and a playhouse one quarter the size of the house. The students were given a detailed set of specifications for the figures (see Appendix B) and 90 minutes to complete the task.

The task was designed to be done by the students with little or no involvement by the researcher. The majority of the students required little or no help to continue working on the task productively, so the records of their sessions are reliable indications of their independent levels of functioning at the time of the administration of the task.

7.1.1. The programs

Figure 7 shows the output of the programs that the students produced; the drawings are ordered according to the number of components of the task instructions that were implemented. Table 1 shows the students ranked from left to right according to the number of components they successfully implemented. The components are ranked from top to bottom according to the number of programs in which they were

successfully implemented. In general, two students, S1 and S3, programmed substantially more of the components than the others, while two others, S5 and S9, programmed substantially fewer. The frequency with which a given component was programmed probably reflects such factors as the order in which it was mentioned in the task instructions, the importance that the students attached to accomplishing it, and the relative ease with which it could be programmed.

<Insert Figure 7 about here>

<Insert Table 1 about here>

Tree diagrams can be used to exhibit much of the structure of the final programs. The program PROCTREE, which is included on the IBM LOGO disk, was used on each of the subjects' final programs, and the resulting procedural structures were abstracted to derive the diagrams shown in Figure 8. The dashes represent procedure names and the indentations show the layers and subprocedural relationships. (The number of lines of dashes do not necessarily represent the actual number of procedural calls in any specific program.)

<Insert Figure 8 about here>

Four different procedural structures are illustrated: simple decompositional structures with two layers of procedures, single procedures, chained procedures which consist of one procedure having as its last line a call to a second procedure, and two layer decompositional structures with chained procedures. A structural feature not shown in these diagrams is the use of REPEAT structures. Most of the students (S2 was the exception) used REPEAT at some point in their programs, usually to draw such shapes as squares, windows, rectangles, stairs, pickets and so forth.

7.1.2. The strategies

An analysis of the protocols of the subjects indicates that they used a similar approach in dealing with the task as a whole, but that different strategies were used to program the main components of the

task. Essentially what they all did in carrying out the task was to group components of the problem together, establish a subgoal based on the group and then implement it before going on to program the next group of components. In this manner they slowly expanded their programs so that all the components were coded. However, in programming these components they did use a variety of different strategies which are the focus of this analysis.

To make the strategy analysis concrete, the discussion refers to specific examples. The examples show the three basic component-based strategies that the students used and some of the strategies which are subsidiary to them[10]. Each of the strategies is described in terms of essential characteristics. The main strategies are characterised in terms of how the goal of programming the component is decomposed and how the code is developed in relationship to the two problem spaces. The subsidiary strategies, which occur within the context of the component-based strategies, are characterized as ways in which the trial space is used to find methods to implement goals. The strategies are compared in terms of the way in which the problem is represented and knowledge is deployed and developed, in terms of their relative cognitive advantages and disadvantages, and in terms of their relationship to the program structures that were evolved in the course of the sessions.

The first of the component-based strategies, the "incremental" strategy, is a highly linear, experimental strategy in which the programmer approaches achieving the goal in small bits and relies heavily upon feedback from the trial space. This strategy has the following characteristics:

1) Components of the problem are coded in the program space one at a time, a bit at a time;

2) once a bit of the component is coded, it is tested in

---

[10]This is not meant to be an exhaustive catalogue of all possible strategies but it does represent the strategies that this group of students employed most frequently.

the trial space;

3) the feedback from the trial space is used to refine and debug the bit of code; and

4) method-finding is based primarily in the trial space.

<Insert Figure 9 about here>

The student in this example used a procedure to carry out a search in the trial space. While he may have been able to retrieve a method for fulfilling a goal, it is more likely that he simply coded provisionally and used the trial space for feedback on whether he was correct or not. This is an example of the subsidiary strategy of "reactive debugging" which is used as a means of finding an appropriate method for fulfilling a goal. Its essential characteristics are:

1) A bit of a component of a task is coded provisionally in the program space and then tested in the trial space; and

2) the feedback from the trial space is used to refine and debug the bit of code.

In another form of the incremental strategy (see Figure 10), the programmer uses the trial space in an anticipatory fashion to gather information about the effects of instructions and subsequently the successful instructions are incorporated into a procedure which can be tested. This is an example of the subsidiary strategy of "anticipatory search." Its essential characteristics are:

1) Information is gathered about successful instructions by experimentation in the trial space; and

2) these instructions are then reproduced in the program space.

<Insert Figure 10 about here>

The third example (Figure 11) illustrates another strategy, the "refinement strategy." In this strategy a component-based goal is established and an attempt is made to implement it completely and fully within the program space before testing the procedure. The refinement strategy involves the following characteristics:

1) A component is selected for coding and it is completely coded in the program space before any testing takes place;

2) the code is tested in the trial space and the feedback is used to make alterations in the existing code;

3) the new code is tested and refined until the component is successfully implemented; and

4) method-finding may occur in the trial space prior to initial coding or entirely within the program space, while refinement can take place in both the trial space and the program space.

<Insert Figure 11 about here>

Another type of search strategy is illustrated in this example. At two points the (lines 7 to 14 and lines 17 to 21) the subject moved the turtle on the screen to measure distances and used the information gathered to make alterations to the procedure under construction. This illustrates a subsidiary, method-finding strategy of "inferential search," the essential characteristics of which are:

1) Information is gathered by experimentation in the trial space; and

2) this information is used to make inferences which are used to code in the program space.

The fourth example (see Figure 12) illustrates a "modular strategy." It involves a decomposition of goals and uses procedures or other structures as modules or chunks that can be separately tested and refined, and combined into wholes. This modular strategy has the following characteristics:

1) A component is analyzed into two or more subcomponents;

2) each of these subcomponents is coded in the program space;

3) these subcomponents are integrated into a whole in the form of a procedure;

4) the subcomponents may be tested singly or tested as parts of the whole;

5) the feedback from the tests are used to refine and debug the subcomponents and the whole; and

6) method-finding takes place both in the program space and

in the trial space.

<Insert Figure 12 about here>

The subsidary strategies of reactive debugging, anticipatory search, and inferential search use the trial space to develop knowledge in three different ways:

1) to test hypotheses in the form of previously coded procedures,

2) to test instructions for possible use in procedures, and

3) to make inferences from the results of experiments.

Reactive debugging is effective primarily as a way of verifying hypotheses but becomes cumbersome as a way of developing new knowledge. Anticipatory search is a relatively risk-free way of experimenting within the trial space, but it poses difficulties in terms of transferring the knowledge gained into the program space; the information must be stored in memory and retrieved or it must be transcribed to paper and then into the editor. Although it can be used to sift-out faulty code, debugging is often needed once the code is tested. Inferential search depends upon the programmer knowing what kind of information to gather and on knowing what to do with the information once it is obtained.

The incremental strategy can be pursued with little thought of what is to happen next. It is economical in that it only requires dealing with circumscribed goals and small chunks of code and the knowledge can be acquired largely on an experimental basis within the programming process. However, it is very ad hoc and the code is highly dependent upon the experimental context in which it is written. Once written, the code is embedded in the context of what proceeds and follows it in the body of the procedure, which leads to non-recyclable code, unwieldy procedures with little discernible structure, and programs which are difficult to comprehend or revise once they are written.

The difficulties of this strategy become most apparent when it is used in conjunction with a reactive debugging strategy. The program of the first student consisted of a single procedure with no REPEAT

commands. As the session progressed, the problem became more complex and the unstructured procedure became longer and more unwieldy. As a result, it became extremely difficult for him to locate buggy code. Perhaps in order to avoid the problems associated with debugging, he increasingly used an anticipatory search strategy to select instructions for inclusion in the procedure.

The refinement strategy in the third example would appear to be the polar opposite of the incremental strategy. Extensive planning is required, and the programmer must be resourceful in finding appropriate methods to carry out these plans. The knowledge required can be simply retrieved, anticipatory search may be carried out, or the programmer may rely on such processes as symbolic execution. The demands of this strategy are relatively high, and because of this, the strategy is probably the least economical of all the strategies.

This strategy becomes most problematic when it is used by students who do not have the required knowledge base or resourcefulness in method-finding. Even when the student is highly skilled (as the student in the example was), refinement and debugging are almost always required and sometimes protracted. As well, subsidiary strategies such as anticipatory search can become problematic in the context of a strategy which strives for completeness. For instance, before defining the procedure to draw the deck, the student in the example tried out a number of instructions in the trial space. He copied the successful ones down on paper, and then transcribed them into the editor. However, he made mistakes in the transcription and had difficulty locating the errors.

The planning involved in the modular strategy is quite economical in its use of cognitive resources. In this strategy, there is a decomposition of the problem so that the segmented parts can be treated as independent units. This permits the programmer to look for solutions to circumscribed problems without concern with the effect of one command on the whole program, and these units can be developed flexibly using any of the subsidiary strategies. Because these independent chunks are

accessible as units, in combing and integrating them into a whole, the programmer is free to focus on them as they function within the whole without concern for the information they organize. However, this strategy requires the programmer to maintain an overall representation of the structure of the problem and the function of the elements in the program; in the example shown (Figure 12), the student appears to have had problems locating and correcting the source of the difficulties he encountered.

Each of the component-based strategies represents a different way of organizing chunks of code into wholes: the incremental strategy, sequentially by bits; the refinement strategy, as parts making up a whole; and the modular strategy, as independent elements which can manipulated as units within a whole. These different orientations to the organization of code were reflected to some extent in the structures of the programs that the students developed. As the session evolved, those who consistently used an incremental strategy (S2, S6, and S7) tended to develop unstructured programs consisting of a single procedure or simple chained procedural structures, while those who regularly used refinement and modular strategies (S1, S4, S5, and S8) for the separate components developed two layer decompositional structures.

In summary, the students approached this graphics task in a similar piece-meal fashion, but employed a variety of different strategies in programming the component elements of the problem. Each of these strategies involves a specific way of breaking the problem down into manageable proportions, of marshalling the appropriate knowledge of methods and language structures, and of searching the problem spaces of the task for a solution. In the following section, some strategies which evolved in relationship to a non-graphics task are investigated.

7.2. The second task: High-Low

After completing the section on graphic programming, the students had six weeks of training to prepare for the second programming task.

The training consisted of short lessons interspersed over the six weeks. The training focused on groups of word-and-list commands and programming techniques, and computer-based exercises were used to enable the students to practice the material taught. After the introduction of some basic concepts in the first session, the students started working on a graphics project which involved programming the game "Tic-Tac-Toe." Students worked individually and in groups, and the instructor assisted them in all phases of the programming of the game. Periodically, there would be group discussions of issues that arose in programming the game. All the students had to be assisted at various points to program the game, and some required extensive assistance; only one student (S1) was able to program the game with only minimal assistance.

High-Low is a simple number game which involves guessing a number between 1 and 100 and feedback is given as to whether the number guessed is too high or too low. The students were given a set of specifications (see Appendix C) for a game in which the computer generated a random number which the player had to guess, and they were allowed 90 minutes to complete the task.

Over the course of the training, wide disparities in skills developed within the groups. It was anticipated that the task would be relatively simple for some students and extremely difficult for others with most students falling somewhere in between. In order to avoid having the less proficient students become unduly frustrated by the task, it was decided that students would be assisted to the degree necessary for them to have a working game by the end of the session.

The interventions varied from relatively minor ones to major ones involving instruction. The content of the interventions included help with such things as syntax (punctuation, format, etc.), semantics (primitives and their significance), and logic (e.g., flow of control).

7.2.1. The programs

Because most of the students required some form of intervention

relatively early on in the programming session, the line between independent programming and assisted programming can be drawn immediately prior to the point at which major instructional interventions were required. Table 2 represents a scoring of the programs according to the components of the task instructions that had been constructed up to that point; the times varied from 5 to 45 minutes (mean 24, median 24). As the table indicates, one student (S1) was able to implement all but one of the components of the game without assistance, three were unable to implement any components (S3, S8, and S9), and the rest implemented an average of six components each (S2, S4, S5, S6, and S7). Five of the nine students (S1, S2, S4, S6, and S7) were able to program the major functions of the game so that it was possible to play it.

<Insert Table 2 about here>

Tree diagrams for the programs constructed prior to a major intervention (see Figure 13) show significant structural differences. The five most successful students (S1, S2, S4, S6, and S7) used procedural structures involving at least two layers: a superprocedure, a set of instructions for getting the player's name and generating the random number, and a recursive, looping structure which got the player's guesses and processed them until the number was guessed. The programs of the other students did not contain the essential looping structure; some only involved two layered structures (S3, S5, and S8) and one only had some elements of the superprocedure (S9).

<Insert Figure 13 about here>

Misunderstandings were evident even in the completed components of the five more successful students. In one program (S2), READCHARACTER rather than READWORD was used in the guess-entering component so that only single digit numbers could be entered. The component which generated the random number was fully implemented in only one program (S1); the others generated numbers other in the range of 0 to 101. Some of the main game loops were faulty. Two (S2 and S6) had incorrect sequences with the procedure giving feedback before the error-checking

component, and they had recursive calls to the main loop embedded in subprocedures so that incomplete loops could reappear once the number was guessed. Another (S7) had calls to an abandoned procedure.

More serious misunderstandings were evident in the programs of those who were unable to complete the basic components of the task. Only one of these students (S5) was able to complete even one component. They all sketched out the superprocedure for the program, but in doing so they exhibited basic confusions around the logic of the program and its flow of control. None of these students' superprocedures had the essential looping structure of the game, and in most the calls to subprocedures merely represented the steps in the game as identified in the instructions; in one (S9), the steps were not even in the correct sequence.

7.2.2. The strategies

Although each of the students required at least one major intervention at some point in the session, these interventions were directed at supplying needed information and not at altering the basic strategy that the students were using. However, to protect the integrity of the analysis here, the examples chosen to illustrate the strategies are drawn from intervals in which there were no interventions.

In LOGO there are significant differences between graphics programming and programming interactive games using words and lists. For one thing, while it is possible to draw complex shapes using sequences of instructions in a single procedure, an interactive game such as High-Low requires more complex structures. These differences in the tasks should be reflected, not only in the kinds of structures the subjects used, but in the strategies they used to develop them.

These differences are most evident in their initial approach to the second task. Whereas in the first task, all the students programmed in a component-by-component, depth-first fashion, in the second task,

they began the session by sketching out all or part of a superprocedure for the game with calls to procedures to carry out specific functions. The use of this top-down approach and simple decompositional structures was no doubt prompted by the way in which the Tic-Tac-Toe game had been developed in the weeks prior to the task.

Despite the dissimilarities in approaches to the two tasks, there were commonalities as well. The different strategies they used in the first task were in evidence in the second, although in somewhat different guises. In so far as they chunked instructions for specific functions and integrated these into wholes, they used a modular strategy. Incremental and refinement strategies and the various subsidiary strategies were employed as well in programming the separate components of the task. However, in keeping with the initial top-down approach used by the subjects in programming this task, the distinctive characteristics of the strategies employed in this task have to do with the students' overall approaches to program design.

Although they all had similar initial approaches to programming the game, once they had completed the definition of the superprocedure, their strategies diverged. The divergences can be described in terms of the contrast between a breadth-first approach and a depth-first approach and in relationship to the layers of the programs that the students developed.

Figure 14 shows an example of a top-down, breadth-first approach. The subject (S1) was highly successful in both tasks and he had employed a refinement strategy in programming most of the components of the first task. What is exhibited in this example is what looks like a combination of the refinement and modular strategies. The essential characteristics of this "breadth-first, refinement strategy" are:

1) The problem is decomposed completely or nearly so and the superprocedure and its component functions are coded at all levels;

2) method-finding and coding takes place wholly within the problem space prior to any tests within the trial space; and

3) the program is debugged largely by refinement of what has been previously coded.

<Insert Figure 14 about here>

The sixth example (Figure 15) is of an heterogeneous strategy. This student pursued a top-down, breadth-first approach up to the point where he had to use specific primitives rather than just calls to procedures. Once a greater level of detail was required, the subject switched to a depth-first, incremental strategy. The essential characteristics of this "self-limited, breadth-first strategy" are:

1) The problem is decomposed completely or nearly so and the superprocedure is coded,

2) the component functions of the game are coded at the abstract level of procedure calls, but once the level of specific instructions (involving the use of primitives) is reached, a switch is made to a depth-first strategy,

3) the component functions are coded using incremental or refinement strategies, and

4) method-finding takes place in both the program space and the trial space.

<Insert Figure 15 about here>

Another example (Figure 16) of a heterogenous strategy illustrates how the structural levels of the program affect its development. In this example, once the student had defined and coded the superprocedure, he proceeded to define and code a level at a time; when he reached a level where a procedure was thought to be completely coded, he tested and debugged it. This "stratification strategy" has the following characteristics:

1) The problem is decomposed completely or nearly so and the superprocedure is coded,

2) the component functions of the game are coded a level at a time; when a function is completely coded, it is tested and debugged,

3) the component functions are coded using refinement

strategies, and

4) method-finding takes place in both the program space and the trial space.

<Insert Figure 16 about here>

A subsidiary strategy exhibited in this example (line 9) is that of "verification search," which is a way of determining whether the purpose of a component function has been fulfilled. The essential characteristics of this strategy are:

1) A component function which generates a value for a variable is tested in the trial space;

2) the value of the variable is printed in the trial space; and

3) the feedback is used to determine if the component function works as intended.

The next example (Figure 17) illustrates a heterogeneous strategy which is also affected by program levels, but it is more depth-first than breadth-first. The difference is in the development process. The student failed to employ a looping function; as a result all the component functions of the program are at one level. The component functions were then developed and refined in a sequential fashion according to the order of the procedure calls in the superprocedure. Once one component function was debugged, an error message would appear indicating that the program had reached an undefined procedure, and then the student would define and develop that procedure. This cycle continued until all the component functions were programmed; the student then discovered that the game not did work as intended, and he had to introduce a game loop and work out the flow of control. This sequential design strategy has the following characteristics:

1) The problem is decomposed completely or nearly so and the superprocedure is coded,

2) the component functions are defined and coded sequentially according to the order in which they appear in the superprocedure,

3) the component functions are defined using incremental or refinement strategies,

4) method-finding takes place in both the program space and the trial space.

<Insert Figure 17 about here>

These strategies can be contrasted in terms of the programmer's overall representation and grasp of program design and how the programmer's knowledge is deployed and developed in the programming process.

The kind of breadth-first refinement strategy exhibited in the first example is linked in the literature with higher levels of expertise. This student was able to program the most components in the least time without assistance and finished the task well before the time was up; he demonstrated a mastery of the design process and could access functional units of programming knowledge which allowed him to program the game without need of extensive testing.

The use of the self-limited breadth-first strategy suggests a grasp of the fundamentals of game design but uncertainty about details; in other words, the student using the strategy could represent the problem as a whole but had difficulty accessing the functional units of knowledge necessary to carry it out to a solution. Consequently, he had to resort to developing components of the program through an incremental strategy and extensive tests in the trial space.

The stratification strategy demonstrates knowledge of game design and some sophistication about program structure. It is opportunistic in the sense that it takes advantage of the possibilities for testing out component functions as the stage of program development allows, but the programmer can remain in control of what elements of the program will be tested and refined.

The sequential strategy is the least developed of the four in terms of the grasp of game design involved and its control over the programming process. The subject in the example does not appear to have had a clear overall representation of the problem and exercises little

control over the course of program development; he simply repeatedly tested the program as it had been developed and used the feedback he received to determine what he should do next. In this respect, the strategy is quite similar to the incremental strategy used in the first task, with many of the same difficulties.

Perhaps the greatest common difficulty that the students had in programming this task was a result of the restrictions on search in the trial space that they imposed on themselves. With few exceptions, the students used tests of the superprocedure to get feedback from the trial space, and in doing so, they failed to take advantage of the decompositional structure of the programs they had constructed[11]. This was not particularly problematic in the early stages in program development when they were dealing with only a few components, but it caused significant difficulties in the later stages. As the session progressed, most of the students became embroiled in trying to sort out, at the same time, relatively minor problems which were internal to the components and major difficulties having to do with the relationships between the components (e.g., problems with the logic and flow of control of the programs).

8. Discussion

Problem solving strategies have been identified in the context of a graphics programming task and an interactive game task. This analysis of strategies, framed within the context of the agenda model sketched above, provides an interpretation of what the students did in these two programming tasks.

In terms of the agenda model, problem solving strategies are ways in which goals are scheduled and methods are used to achieve goals. The

---

[11]The verification search strategy could be used in a variety of ways to test individual components of the program. For instance, an error-checking procedure could be tested independently using a dummy input variable.

programmer's knowledge guides search in the two problem spaces at two levels of the model. The first level is that of the programmer's representation of the problem. According to the model, each of the strategies represents a different way of decomposing goals and thus a different kind of agenda for dealing with the task. The agenda is determined by the programmer's ability to grasp the problem as a whole and to use the necessary language structures to support a solution. The second level of the model is that of the specific methods used to implement goals. These methods can be specific sequences of operators which directly bring about changes in the program or they can be ways of finding these methods. To follow an agenda, the programmer must draw on a knowledge base of these methods. Accordingly, the availability or lack of availability of appropriate methods shapes the agenda and the ordering of goals.

The strategies exhibited in the first task do not involve a representation of the problem as a whole; the component-based goals appear to have been taken from the external agenda represented by the task instructions. In programming the specific components of the problem, the refinement and modular strategies involve an overall representation of the component, and the modular strategy involves a decomposition of the component as well as an awareness of the structures available within the language to express that decomposition; only the incremental strategy appears to lack an overall representation. Whatever limitations these strategies may have had, most of the programmers had a variety of available methods for achieving the goals they set; they appear to have had experiences with instructions that they could draw upon and search strategies for finding methods (e.g., symbolic execution, anticipatory search) within the two spaces of the task.

The strategies employed in the second task generally involve a representation of the whole problem and a decomposition of the goals of the task; however, there were clear limitations in the students' grasp of program design and in their ability to use appropriate language

structures. Some were able to follow the agenda of the goal decomposition and program the components completely from top to bottom, others could only sketch them out by levels or sequentially as it became necessary. Clearly the students had considerably greater difficulty with this task than they had with the earlier graphics task. However, for all but the most successful student, the difficulty was not so much at the level of their overall representation of the problem, but at the level of having the appropriate methods for implementing the goals that they initially set. The difficulty was that they had not developed the range of subsidiary strategies which had served them so well in finding appropriate methods in the graphics task.

9. Conclusion and implications

As computers become more common within the educational system, techniques for analyzing and assisting learning will need to be developed which take advantage of the opportunities that the technology presents. This paper is an illustration of an approach to the investigation of cognitive processes which has particular application to the investigation of problem solving in computer-based learning environments. The key features of this approach are the automated collection and analysis of trace records and the construction and application of models for explaining the cognitive processes represented in these records. At its present state of implementation, this approach is suitable only for research on circumscribed tasks. However, as the approach develops, it will reach the point at which it can be used routinely by teachers to understand what their students are doing in the computer-based environments so that they can provide the students with advice and direction. Eventually this approach will be developed to the point at which automated tutors will be able to develop models of the user and provide appropriate individualized assistance instantaneously. But the impact of this approach may be more widely felt than just in the area of computer-based learning. Educators who are dissatisfied with

basing evaluations on group comparisons and static, norm-referenced tests may find in the process-analytic paradigm principles which can be applied to underpin the observational techniques used within classrooms.

Table 1: components of the House-Playhouse figure programmed

	S1	S3	S6	S2	S4	S7	S8	S9	S5	Tot
1A	1	1	1	1	1	1	1	1	1	9
2A	1	1	1	1	1	1	1	1	1	9
5A	1	1	1	1	1	1	1	1	1	9
4A	1	1	1	1	1	0	1	.8*	.5	7.3
3A	1	1	1	1	1	0	1	1	0	7
6A	1	1	0	1	1	1	1	0	0	6
9A	1	1	0	0	1	1	1	0	0	5
7A	1	0	0	1	1	1	0	0	0	4
1B	1	1	1	0	0	0	0	0	0	3
3B	1	1	1	0	0	0	0	0	0	3
2B	1	1	0	1	0	0	0	0	0	3
8A	1	1	0	0	0	1	0	0	0	3
5B	1	1	0	0	0	0	0	0	0	2
6B	1	1	0	0	0	0	0	0	0	2
4B	0	1	1	0	0	0	0	0	0	2
9B	0	1	0	0	0	0	0	0	0	1
7B	0	0	0	0	0	0	0	0	0	0
8B	0	0	0	0	0	0	0	0	0	0
Tot	14	14	9	8	8	7	7	4.8	3.5	
min	93	88	90	85	94	94	102	99	86	

main house
1A body of house
2A door
3A first floor windows
4A second floor windows
5A roof
6A chimney
7A deck
8A picket fence
9A stairs

playhouse
1B body of playhouse
2B door
3B first floor windows
4B second floor windows
5B roof
6B chimney
7B deck
8B picket fence
9B stairs

* a fraction indicates that the component was implemented only in part

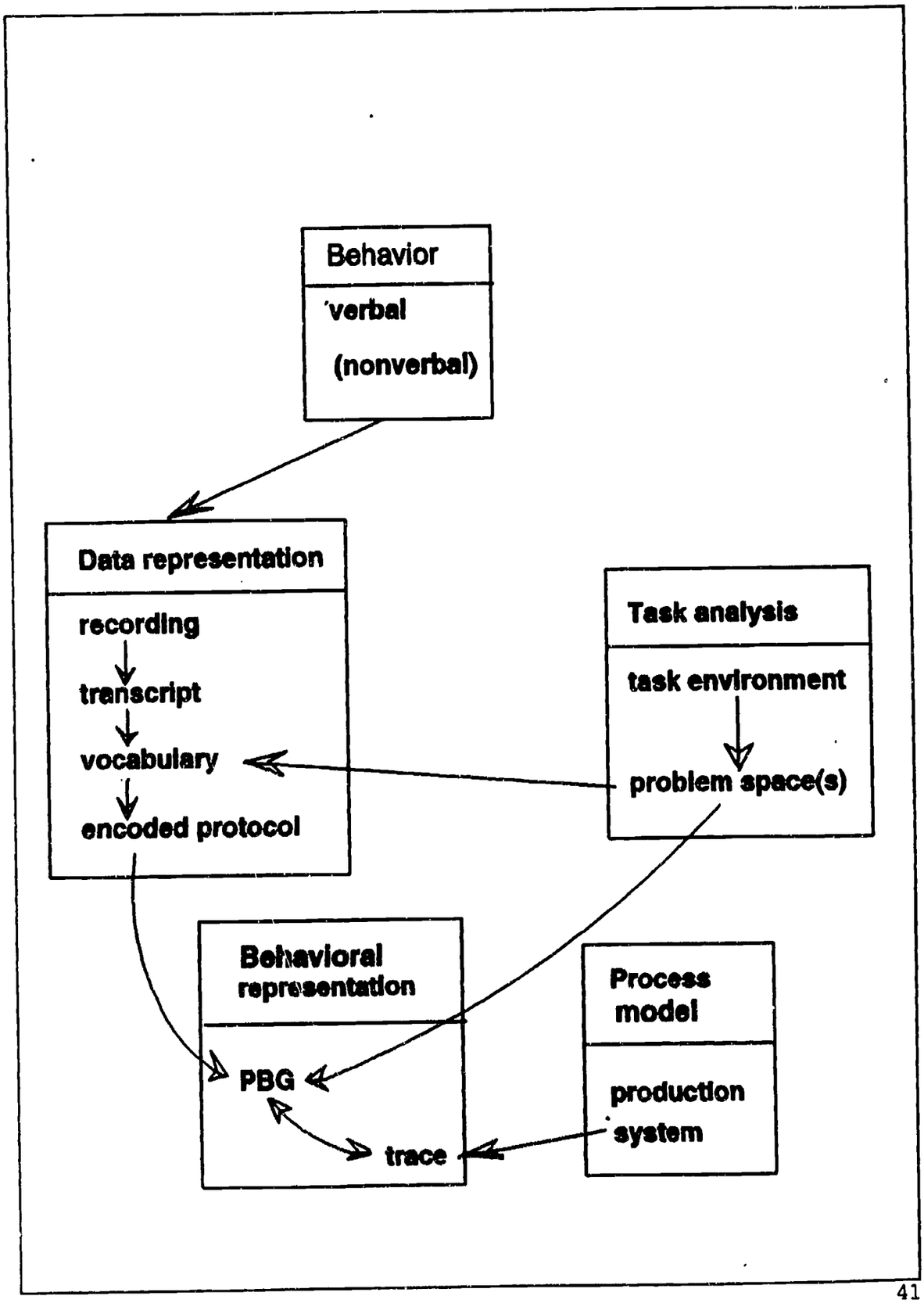
Table 2: components of the High-Low game programmed

	S1	S4	S6	S7	S2	S5	S3	S8	S9	Tot
1	1	1	1	1	1	1	0	0	0	6
3	1	1	1	1	1	0	0	0	0	5
6	1	1	1	1	1	0	0	0	0	5
5	1	1	1	.5*	.5	0	0	0	0	4
7	1	1	1	1	0	0	0	0	0	4
2	1	.5	.5	.5	.5	0	0	0	0	3
8	1	1	0	0	0	0	0	0	0	2
9	1	1	0	0	0	0	0	0	0	2
4	.5	0	0	0	0	0	0	0	0	.5
tot	1	1	1	0	0	0	0	0	0	3
min	24	32	24	42	45	11	16	19	5	3

1 get name function
2 function to generate random number
3 function for entering number
4 function for checking for incorrect input
5 loop
6 inform if high, low, or won
7 inform by name
8 print out number of guesses
9 print out guesses

* a fraction indicates that the function was implemented only in part.

Figure 1: Traditional protocol analysis



41

Figure 2: The two problem spaces

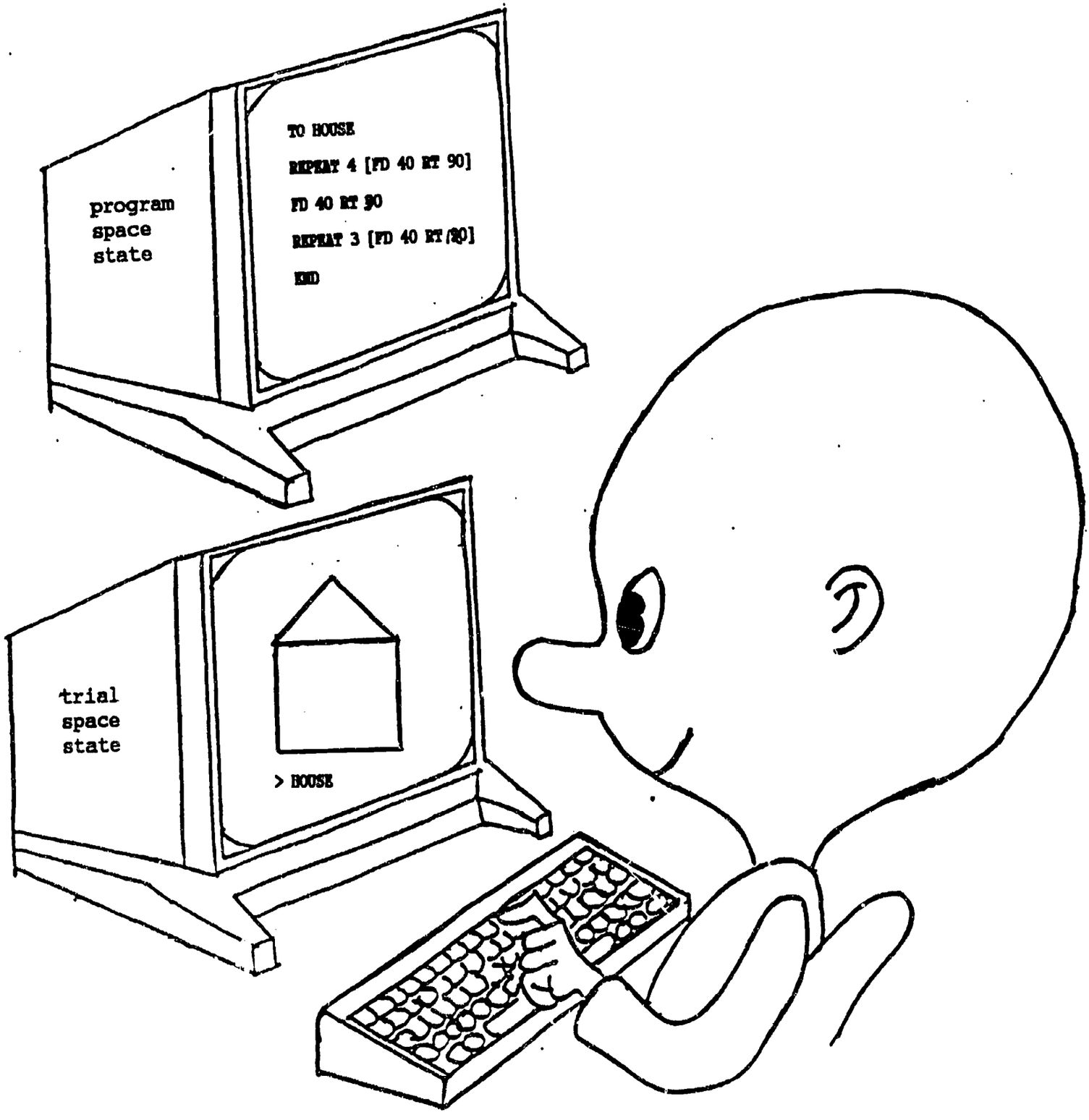


Figure 3: The record generator

Recording:

recording program records:
mode (immediate, define, edit, error, etc.)
initiating action and contents of mode
time stamps (in tenths of second, time from previous action, time to completion of action)

```
DE: [TO DOOR] [[RT 90] [FD :SIDE1 / 2] [FD :SIDE1 / 3 RT 90] [FD :SIDE1 / 4 RT 90]] [[466] [2161]]

ED: [ED "DOOR] [[[TO DOOR] [[SIDE1] [RT 90] [FD :SIDE1 / 2 LT 90]]]
      [[TO RECTANGLE] [[SIDE1] [REPEAT 2 [FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4]]]]] [[256] [2852]]

IM: [HOUSE1 50] [] [[241] [31]]

ER: [HOUSE1 50] [35 [I DON'T KNOW HOW TO SIDE1] ROOF [FD :SIDE1 RT 90 FD SIDE1 RT 180] RIGHT SIDE1]
[[85] [18]]
```

Displaying (printing or replaying):

put record lists into readable form:

```
{5}  (46.6 / 216.1 / 262.7 / 0:07:01)
DE:  TO DOOR
TO DOOR
RT 90
FD :SIDE1 / 2
FD :SIDE1 / 3 RT 90 FD :SIDE1 / 4 RT 90
END

{7}  (25.6 / 285.2 / 310.8 / 0:12:18)
ED:  TO DOOR
RT 90
FD :SIDE1 / 2 LT 90
END

{11}  (60 / 108.6 / 168.6 / 0:17:16)
ED:  TO HOUSE1 :SIDE1
HOUSE :SIDE1
DOOR :SIDE1
RECTANGLE :SIDE1
END

{12}  (24.1 / 3.1 / 27.2 / 0:17:43)
IM:  HOUSE1 50
```

displays trial state (when replaying):

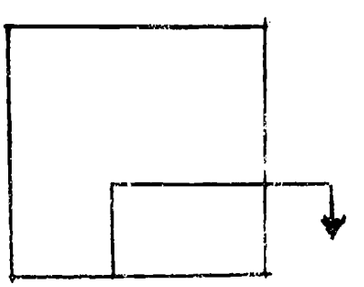


Figure 4: The encoder

Identifying changes and segmenting:

(focused on program states) indicates changes, prints lines of procedure, and prints the old and new version of the procedure segmented into units of instructions.

```
TO DOOR
RT 90
FD :SIDE1 / 2 LT 90
END

new list:  [TO DOOR] [RT 90] [FD :SIDE1 / 2] [LT 90]
old list:  [TO DOOR] [RT 90] [FD :SIDE1 / 2] [FD :SIDE1 / 3] [RT 90] [FD :SIDE1 / 4] [RT 90]
```

Coding:

```
delete 4

old list:  [TO DOOR] [RT 90] [FD :SIDE1 / 2]
                - [FD :SIDE1 / 3]
                - [RT 90]
                - [FD :SIDE1 / 4]
                - [RT 90]

append 1

new list:  [TO DOOR] [RT 90] [FD :SIDE1/2]
                + [LT 90]
```

Figure 5: The problem behavior graph generator

Output from analysis program:

```
[define HOUSE] [] [def] [1] [HOUSE] []
[] [] [app] [1] [HOUSE] [] [41.1]
[] [] [] [3] [HOUSE] [][98.2]
[define DOOR] [] [def] [5] [DOOR] []
[] [] [app] [5] [DOOR] [] [262.7]
[] [] [error] [6] [DOOR] [ED: DOOR] [DOOR HAS NO VALUE] [6.4]
[] [] [] [7] [DOOR[] [define RECTANGLE] [] [def] [7] [RECTANGLE] []
[] [] [app] [7] [RECTANGLE] [] [310.8]
```

Input to Problem Behaviour Graph Generator:

```
[square] [] [] [] [] []
[define HOUSE] [square] [def] [1] [HOUSE] [] [41.1]
[use variables] [square] [ins w/in] [3] [HOUSE] []
[make square] [square] [app] [3] [HOUSE] [] [98.2]
[square and door] [] [] [] [] []
[door] [square and door] [] [] [] []
[define DOOR] [door] [def] [5] [DOOR] []
[position, draw door] [door] [app] [5] [DOOR] [] [262.7]
[edit DOOR] [door] [error] [6] [ED: DOOR] [DOOR HAS NO VALUE] [6.4]
[take out door] [position, draw door] [del] [7] [DOOR] []
[turn in to draw door] [position, draw door] [app] [7] [DOOR] []
[rectangle] [door] [] [] [] []
[define RECTANGLE] [rectangle] [def] [7] [RECTANGLE] []
[do rectangle] [rectangle] [app] [7] [RECTANGLE] [] [310.8]
```

Output of Problem Behaviour Graph Generator:

```
G: square
     G: define HOUSE ( def ) < 1 > { HOUSE } < 41.1 >
     G: use variables ( ins w / in ) < 3 > { HOUSE }
     G: make square ( app ) < 3 > { HOUSE } <98.2>
G: square and door
     G: door
          G: define DOOR ( def ) < 5 > { DOOR }
          G: position, draw door ( app ) < 5 > { DOOR }
          G: edit DOOR ( error ) < 6 > { ED :DOOR } < 6.4 >
                  ( DOOR HAS NO VALUE )
          S: position, draw door
               G: take out door ( del ) < 7 > { DOOR }
               G: turn in to draw door ( app ) < 7 > { DOOR }
          G: rectangle
               G: define RECTANGLE ( def ) < 7 > { RECTANGLE }
               G: do rectangle ( app ) < 7 > { RECTANGLE } <310.8>
```

Figure 6: The agenda model

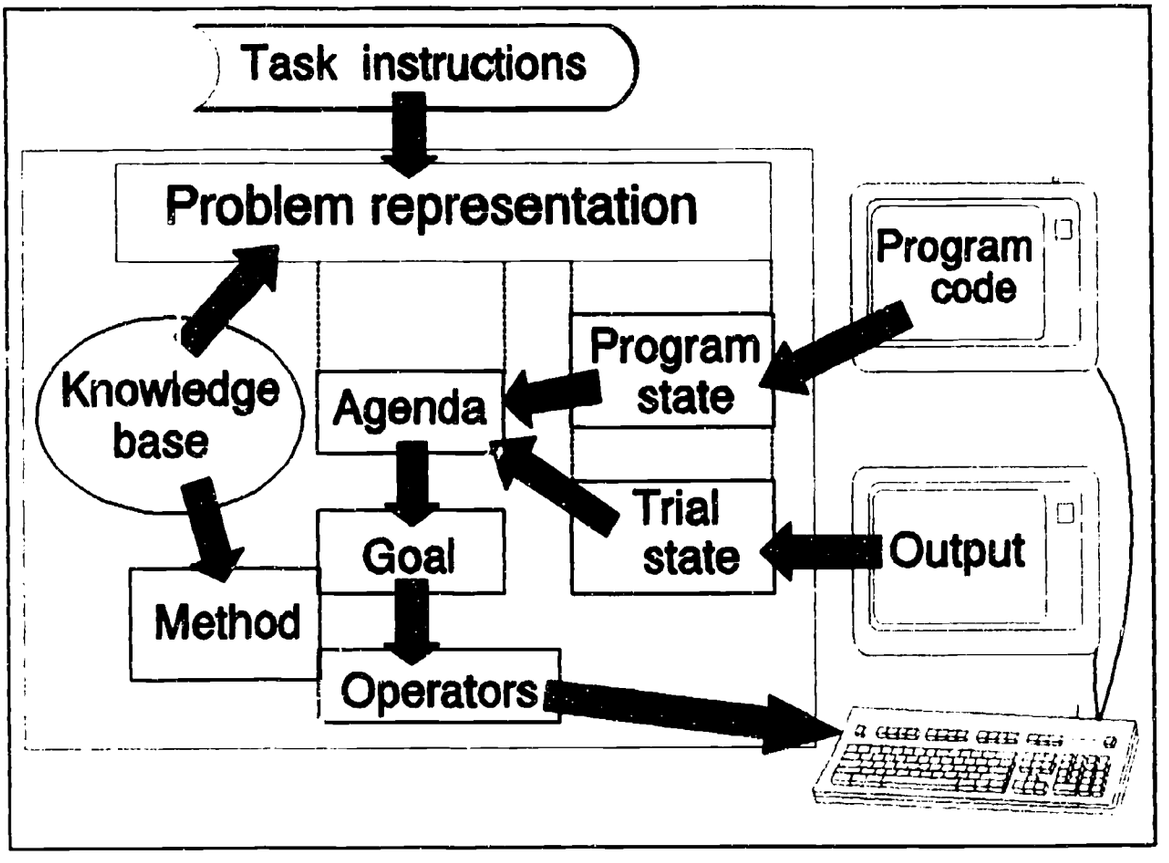
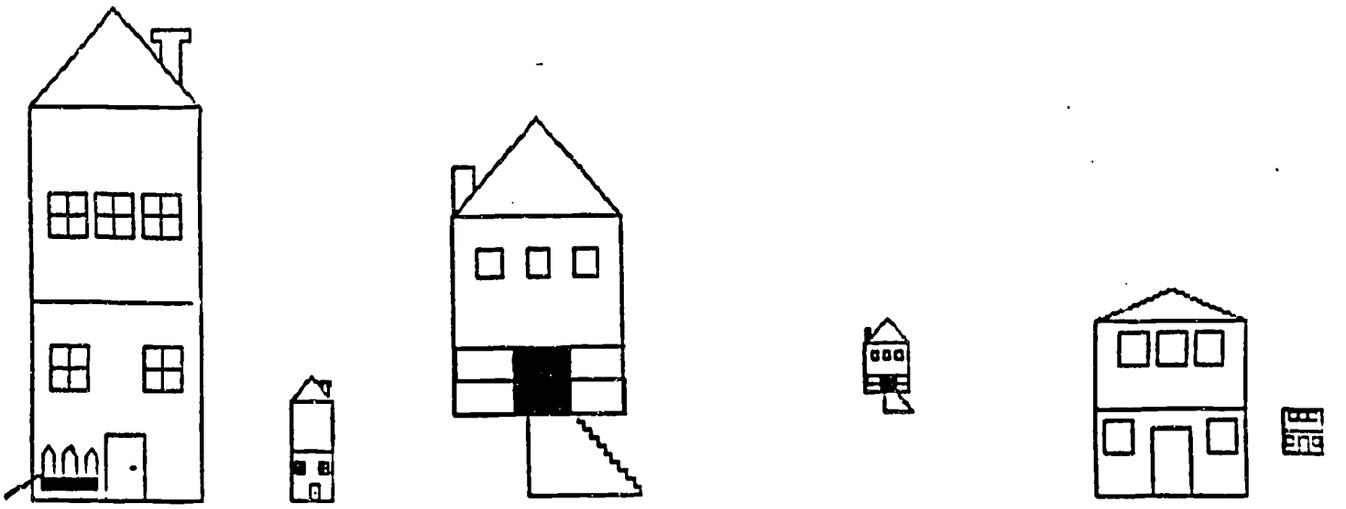
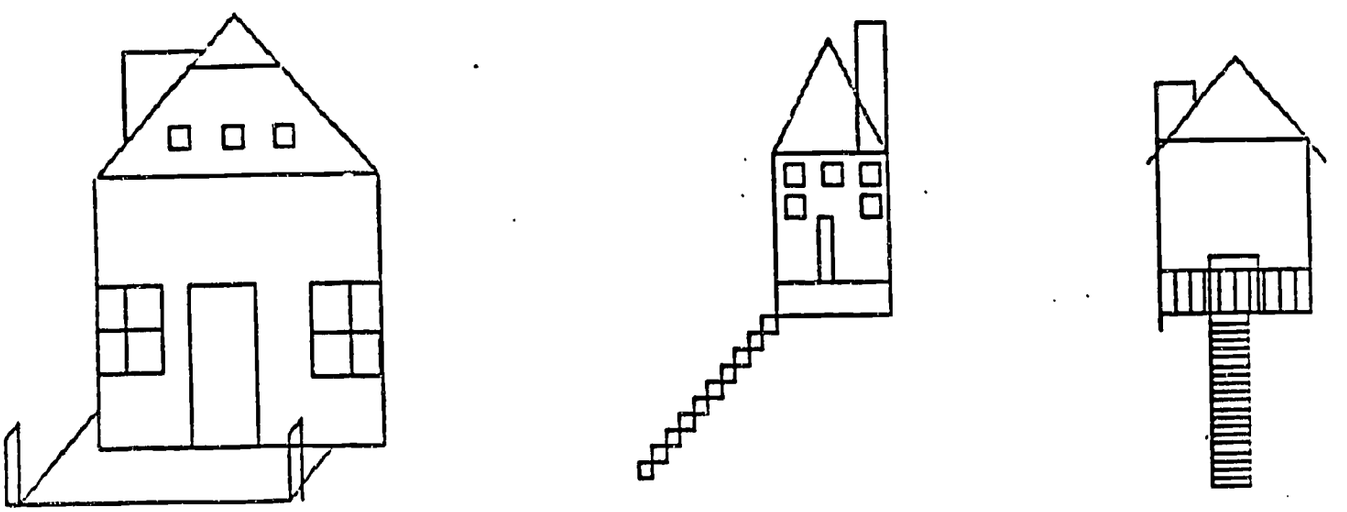


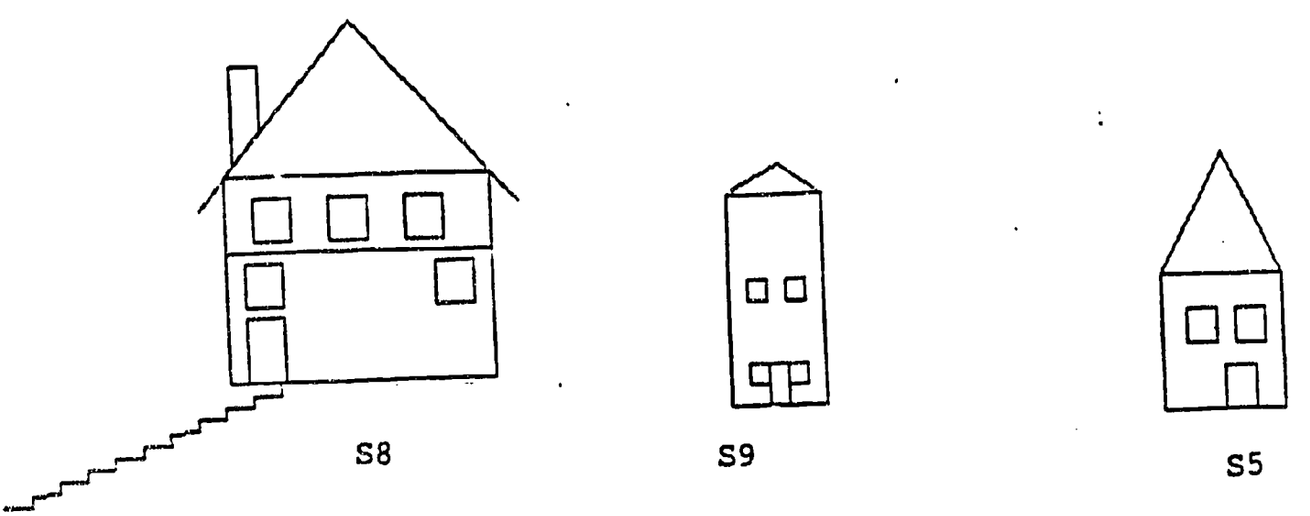
Figure 7: The output of the House-Playhouse programs



S1 S3 S6



S2 S4 S7



S8 S9 S5

Figure 8: The structure of the House-Playhouse programs

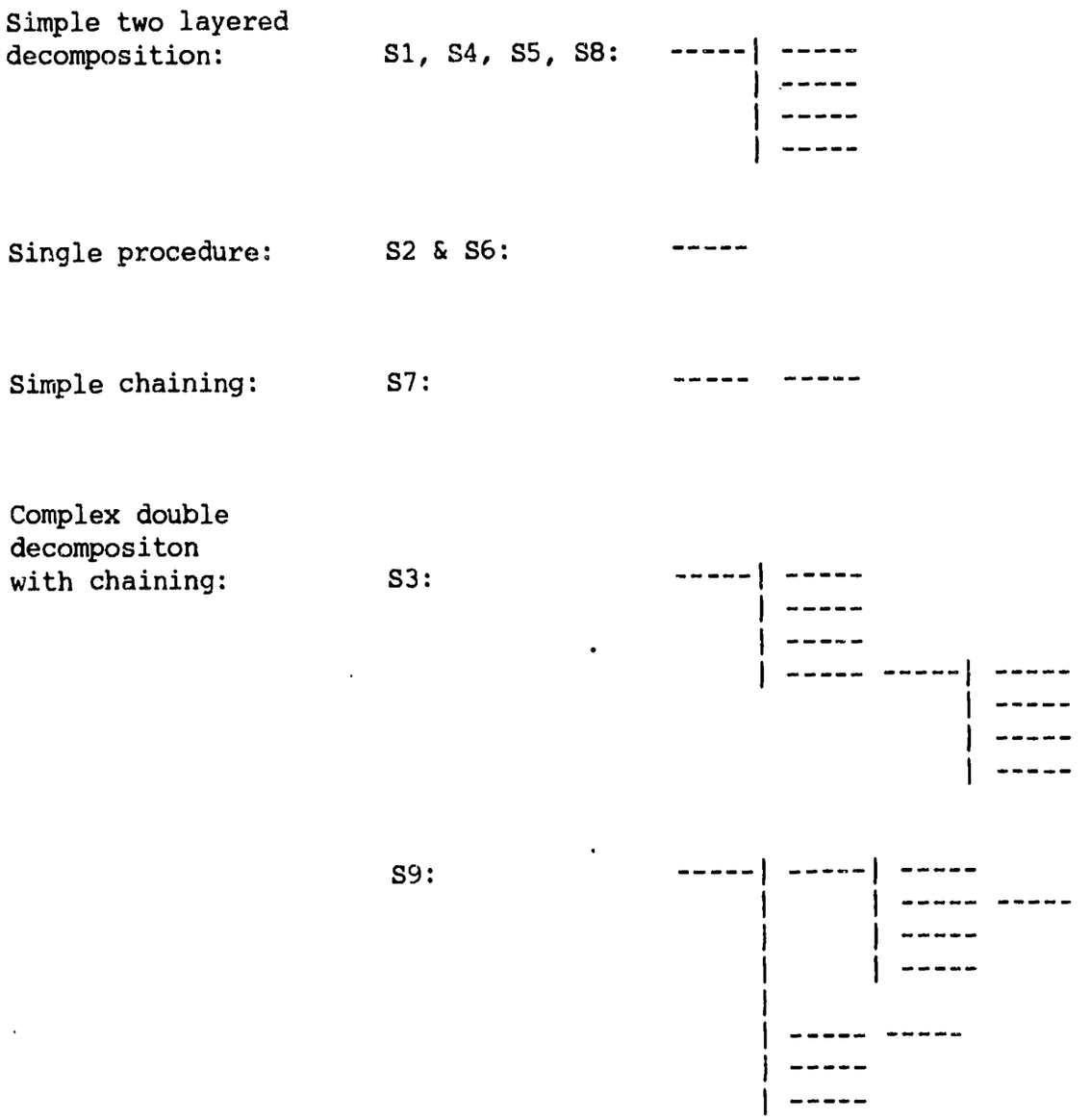


Figure 9: Example 1

Subject 2, episode 1, task 1

PROGRAM SPACE

```
<1> TO HOUSE
>PU
>RT 90
>FD 50
>END

<3> TO HOUSE
...
FD 100
{ FD 50 }
END

<6> TO HOUSE
...
LT 90
PD
FD 100
END

<10> TO HOUSE
...
LT
FD 150
END

<13> TO HOUSE
...
LT 90
FD 150
END

<16> TO HOUSE
...
LT 90
FD 175
END

<18> TO HOUSE
...
LT 90
FD 150
LT 90
FD 100
END
```

PROBLEM BEHAVIOUR GRAPH

```
G: square

    G: define HOUSE (def) < 1 > { HOUSE }
    G: make start (app) < 1 > { HOUSE } < 113 >

    G: test SQUARE (invoke) < 2 > { HOUSE } < 6.6 >

    S: make start
        G: more right (del w/in, ins w/in) < 3 > { HOUSE } <27.7>

    G: test SQUARE
        G: clearscreen (invoke) < 4 > { CS } < 1.5 >
        G: test it (invoke) < 5 > { HOUSE } < 6.1 >

    G: left, up (app) < 6 > { HOUSE } < 41 >

    G: test SQUARE
        G: clearscreen (error) < 7 > { CAS } < 4.1 >
                ( I DON'T KNOW HOW TO CAS )
            G: try again (invoke) < 8 > { CS } < 2.4 >
        G: test it (invoke) < 9 > { HOUSE } < 3 >

    G: left, across top (app) < 10 > { HOUSE } < 32.5 >

    G: test SQUARE
        G: clearscreen (invoke) < 11 > { CS } < 3.3 >
        G: test it (error) < 12 > { HOUSE } < 3.1 >
                ( NOT ENOUGH INPUTS TO LEFT )

    S: left, across top
        G: put in angle (ins) < 13 > { HOUSE }

    G: test SQUARE
        G: clearscreen (invoke) < 14 > { CS }
        G: test it (invoke) < 15 > { HOUSE }

    G: left and down (app) < 16 > { HOUSE }

    G: test SQUARE
        G: clearscreen (invoke) < 17 > { CS } < 2.5 >
        G: test it (invoke) < 18 > { HOUSE } < 2.7 >

    G: around to start (app) < 19 > { HOUSE } < 38.6 >

    G: test SQUARE
        G: clearscreen (invoke) < 20 > { CS } < 1.3 >
        G: test it (invoke) < 21 > { HOUSE } < 3.9 >
```

TRIAL SPACE

<2> HOUSE

<4> CS

<5> HOUSE

<8> CS

<9> HOUSE

<11> CS

<12> HOUSE

<14> CS

<15> HOUSE

<17> CS

<18> HOUSE

<20> CS

<21> HOUSE

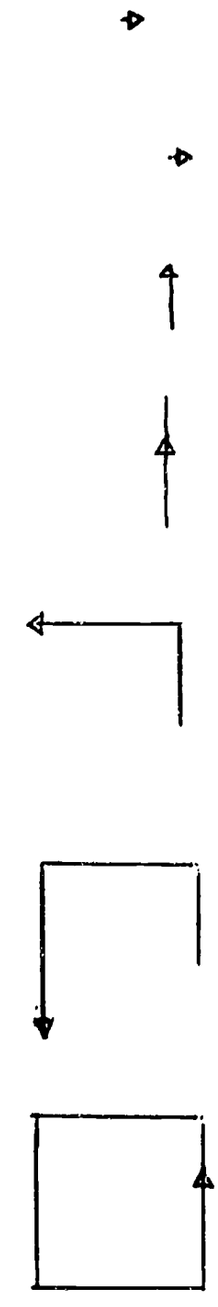


Figure 10: Example 2

Subject 3, task 1, episode 1

PROGRAM SPACE

```
<1>

TO HOUSE
>CS
>PU
END

<3>

TO HOUSE :SIDE
CS
PU
BKEND
END

<9>

TO HOUSE :SIDE
CS
PU
BK 80
LT 90
FD 120
PD
REPEAT 4 [FD :SIDE RT 90]
END

<12>

TO HOUSE :SIDE
CS
PU
BK 80
LT 90
FD 120
RT 90
PD
REPEAT 4 [FD :SIDE RT 90]
END
```

PROBLEM BEHAVIOUR GRAPH

G: square

- G: define HOUSE (def) < 1 > { HOUSE }
- G: set up to start (app) < 1 > { HOUSE } < 185.9 >
- G: edit HOUSE (error) < 2 > { ED"HOUSE } < 7.1 >
  (I DON'T KNOW HOW TO ED"HOUSE)
- G: use variable (ins w / in) < 3 > { HOUSE }
- G: get in position (app) < 3 > { HOUSE } < 46.2 >
  - G: see what to do next
    - G: try this (invoke) < 4 > { BK 40 } < 4.8 >
    - G: try this (invoke) < 5 > { BK 40 } < 4.5 >
    - G: try this (invoke) < 6 > { LT 90 } < 5.2 >
    - G: try this (invoke) < 7 > { FD 100 } < 6.1 >
    - G: try this (invoke) < 8 > { FD 20 } < 4.8 >
  - G: use info (ins w / in, app) < 9 > { HOUSE }
- G: make a square (app) < 9 > { HOUSE } < 90.6 >
- G: test HOUSE (error) < 10 > { HOUSE } < 4.4 >
  (NOT ENOUGH INPUTS TO HOUSE)
  - G: use an input (invoke) < 11 > { HOUSE 30 } < 6.9 >
- S: get in position
  - G: reposition (ins) < 12 > { HOUSE } < 25.2 >
- G: test SQUARE (invoke) < 13 > { HOUSE 100 } < 8.9 >

TRIAL SPACE

<4>

BK 40

<5>

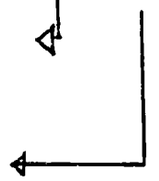
BK 40

<6>

LT 90

<7>

FD 100

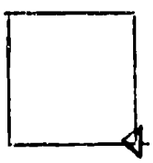


<8>

FD 20

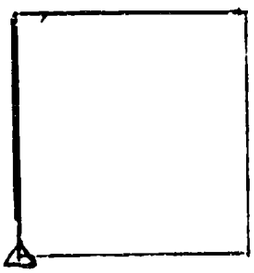
<11>

HOUSE 30



<12>

HOUSE 100



50

Figure 11: Example 3

Subject 1, episode 1, task 1

PROGRAM SPACE

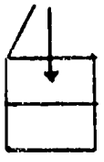
```
<1>  TO STRUCTURE
CS PU RT 180 FD 100
RT 90 FD 50 RT 90 PD
REPEAT 2 [REPEAT 4 [FD
100 RT 90] FD 100]
RT 45 FD 75 RT 135 FD 75

<3>  TO STRUCTURE
...FD 125 {FD 100}
RT 90 FD 50 RT 90 PD
REPEAT 2 [REPEAT 4 [FD
100 RT 90] FD 100]
RT 45 FD 75 RT 90
{RT 135} FD 75

<5>  TO STRUCTURE
...FD 120 {FD 125} RT 90
FD 50 RT 90 PD REPEAT 2
[REPEAT 4 [FD 90 {FD 100}
RT 90] FD 90 {FD 100}] RT
45 FD 75 RT 135 FD 75

<15>  TO STRUCTURE
...FD 50 {FD 75}
RT 90 FD 50 {FD 75}

<22>  TO STRUCTURE
...FD 60 {FD 50}
RT 90 FD 60 {FD 50}

<24>  TO STRUCTURE
...FD 65 {FD 60}
RT 90 FD 65 {FD 60}

<26>  TO STRUCTURE
...FD 63 {FD 65}
RT 90 FD 65
END
```

PROBLEM BEHAVIOUR GRAPH

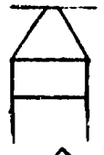
```
G: outline house
    G: define STRUCTURE ( def ) < 1 > { STRUCTURE }
    G: make outline ( app ) < 1 > { STRUCTURE } < 275.4 >
    G: test STRUCTURE ( invoke ) < 2 > { STRUCTURE } < 7.5 >
    G: get in frame ( rep w/in ) < 3 > { STRUCTURE }
    G: fix peak angle (rep w/in) <3> {STRUCTURE} <48.8>
    G: test STRUCTURE ( invoke ) < 4 > { STRUCTURE } < 4.4 >
    S: get in frame
        G: not so low (rep w/in) <5> {STRUCTURE}
        G: smaller (rep w/in) <5> {STRUCTURE} <81.2>
    G: test STRUCTURE ( invoke ) < 6 > { STRUCTURE } < 5.1 >
    G: make roof meet
        G: find way to meet
            G: try this ( invoke ) < 7 > { BK 75 } < 33.7 >
            G: try this ( invoke ) < 8 > { RT 135 } < 5.5 >
            G: try this ( invoke ) < 9 > { LT 135 } < 13.4 >
            G: try this ( invoke ) < 10 > { LT 135 } < 8.3 >
            G: try this ( invoke ) < 11 > { RT 45 } < 4.8 >
            G: try this ( invoke ) < 12 > { BK 75 } < 11.3 >
            G: try this ( invoke ) < 13 > { FD 50 } < 4.1 >
            G: try this ( invoke ) < 14 > { FD 5 } < 11.1 >
        G: closer (rep w/in) <15> {STRUCTURE} <31.1>
    G: test STRUCTURE ( invoke ) < 16 > { STRUCTURE } < 4.9 >
    S: make roof meet
        S: closer
            G: measure
                G: try this ( invoke ) < 17 > { LT 45 } < 11.9 >
                G: try this ( invoke ) < 18 > { HT } < 4.7 >
                G: try this ( invoke ) < 19 > { FD 10 } < 4.7 >
                G: try this ( invoke ) < 20 > { FD 5 } < 3.4 >
                G: try this ( invoke ) < 21 > { FD 5 } < 5.5 >
            G: more (rep w/in) <22> {STRUCTURE} <22>
    G: test STRUCTURE ( invoke ) < 23 > { STRUCTURE } < 3.4 >
    S: make roof meet
        S: closer
            S: more
                G: and more (rep w/in) <24> {STRUCTURE} <30.4>
    G: test STRUCTURE ( invoke ) < 25 > { STRUCTURE } < 7.2 >
    S: make roof meet
        S: closer
            S: more
                S: and more
                    G: bit more (rep w/in) <26> {STRUCTURE}<26>
    G: test STRUCTURE ( invoke ) < 27 > { STRUCTURE } < 2.5 >
```

TRIAL SPACE

<2> STRUCTURE



<4> STRUCTURE



<6> STRUCTURE

<7> BK 75

<8> RT 135

<9> LT 135

<10> LT 135

<11> RT 45

<12> BK 75

<13> FD 50

<14> FD 5

<16> STRUCTURE

<17> LT 45

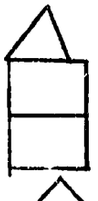
<18> HT

<19> FD 10

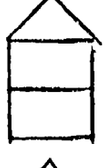
<20> FD 5

<21> FD 5

<21> STRUCTURE



<25> STRUCTURE



<27> STRUCTURE

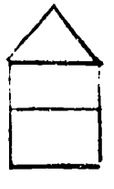


Figure 12: Example 4

Subject 4, episode 1, task 1

PROGRAM SPACE

```
<1>  TO BEGIN
>REPEAT 5 [SQUARE]

<2>  TO SQUARE
REPEAT 4 [FD 40 RT 90]

<3>  TO BEGIN
SQUARE
{REPEAT 5 [SQUARE]}
TRIANGLE

TO SQUARE
REPEAT 5 [FD 40 RT 90]
{REPEAT 4 [FD 40 RT 90]}

TO TRIANGLE
LT 90
RT 18
FD 20
LT 36
FD 20

<8>  TO TRIANGLE
RT 90
{LT 90}
RT 18
FD 20
LT 36
FD 20

<9>  TO BEGIN
CS
SQUARE
TRIANGLE

<11>  TO SQUARE
REPEAT 4 [FD 40 RT 90]
{REPEAT 5 [FD 40 RT 90]}

<13>  TO SQUARE
REPEAT 6 [FD 40 RT 90]
```

PROBLEM BEHAVIOUR GRAPH

```
G: square with roof

    G: define BEGIN (def) < 1 > { BEGIN }

    G: square
        G: call SQUARE repeatedly (app) < 1 > { BEGIN } < 132.6 >

        G: define SQUARE ( def ) < 2 > { SQUARE }
        G: make square ( app ) < 2 > { SQUARE } < 59.4 >

        G: remove multiple calls ( del ) < 3 > { BEGIN }

    G: roof
        G: call TRIANGLE ( app ) < 3 > { BEGIN }

    G: get to top of SQUARE ( del w/in, ins w/in ) < 3 > { SQUARE }

    S: roof
        G: define TRIANGLE ( def ) < 3 > { TRIANGLE }
        G: make triangle ( app ) < 3 > { TRIANGLE } < 120.5 >

        G: test TRIANGLE
            G: clearscreen ( invoke ) < 4 > { TRIANGLE } < 2.1 >
            G: test it ( invoke ) < 5 > { TRIANGLE } < 6.7 >

    G: test BEGIN
        G: clearscreen ( invoke ) < 6 > { BEGIN } < 2.7 >
        G: test it ( invoke ) < 7 > { BEGIN } < 3.7 >

    G: get roof on top ( del, ins ) < 8 > { TRIANGLE } < 19.2 >

    G: ease tests ( ins ) < 9 > { BEGIN } < 8.1 >

    G: test BEGIN ( invoke ) < 10 > { BEGIN } < 3.4 >

    G: get house upright ( del, ins ) < 11 > { SQUARE } < 19.4 >

    G: test BEGIN ( invoke ) < 12 > { BEGIN }

    S: get house upright
        G: get house inverted ( del, ins ) < 13 > { SQUARE }

    G: test BEGIN ( invoke ) < 14 > { BEGIN } < 2 >
```

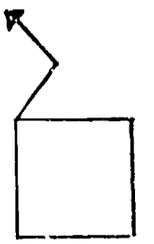
TRIAL SPACE

<4> CS

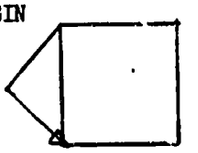
<5> TRIANGLE

<6> CS

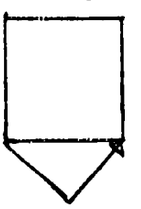
<7> BEGIN



<10> BEGIN



<12> BEGIN



<14> BEGIN

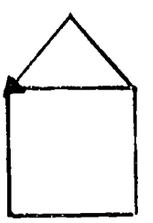


Figure 13: The structure of the High-Low programs

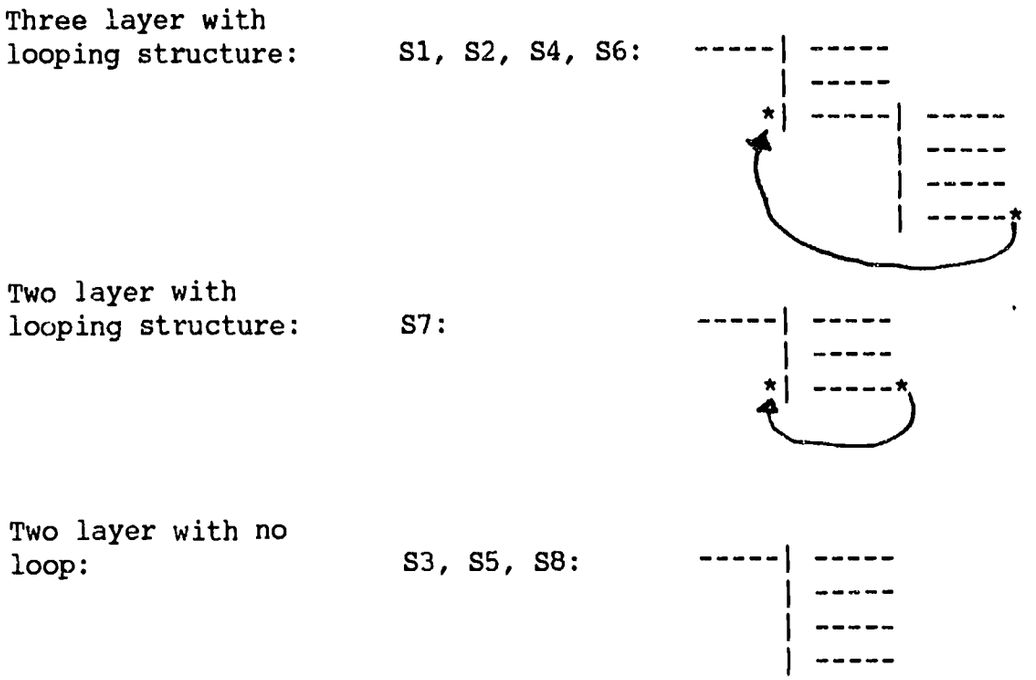


Figure 14: Example 5 (episode 1, task 2, S1)

```
G: basic game
    G: define PLAY ( def ) < 1 > { PLAY }
    G: gather list of guesses
        G: initialize guesses list ( app ) < 1 > { PLAY }
    G: get value for name
        G: make variable for OP of ASK.NAME ( app ) < 1 > { PLAY }
    G: get value for random number ( app ) < 1 > { PLAY }
    G: game loop
        G: call GAME ( app ) < 1 > { PLAY }
    S: get value for name
        G: define ASK.NAMES ( def ) < 1 > { ASK.NAMES }
        G: get value for name for OP to PLAY ( app ) < 1 > { ASK.NAME }
    S: game loop
        G: define GAME ( def ) < 1 > { GAME }
    G: get valid guesses
        G: make variable for OP of ASK.GUESS ( app ) < 1 > { GAME }
    S: gather list of guesses
        G: create list of moves ( app ) < 1 > { GAME }
    G: check if won
        G: create variable by call to CHECK.CORRECT ( app ) < 1 > { GAME }
    G: informing function
        G: call WIN and stop if won ( app ) < 1 > { GAME }
    G: give feedback on guesses if not win
        G: call ANSWER ( app ) < 1 > { GAME }
    S: game loop
        G: recurse to GAME and stop ( app ) < 1 > { GAME }
    S: get valid guesses
        G: define ASK.GUESS ( def ) < 1 > { ASK.GUESS }
        G: get guesses for output to GAME ( app ) < 1 > { ASK.GUESS }
            G: get input ( app ) < 1 > { ASK.GUESS }
            G: error check for out of range ( app ) < 1 > { ASK.GUESS }
    S: check if won
        G: define CHECK.CORRECT ( def ) < 1 > { CHECK.CORRECT }
        G: if won output to GAME ( app ) < 1 > { CHECK.CORRECT }
    S: informing function
        G: define WIN ( def ) < 1 > { WIN }
        G: inform by name if won ( app ) < 1 > { WIN }
        G: inform how many moves ( app ) < 1 > { WIN }
        G: inform what moves
            G: call PMOVES ( app ) < 1 > { WIN }
            G: define PMOVES ( def ) < 1 > { PMOVES }
            G: print moves on separate lines ( app ) < 1 > { PMOVES }
    S: give feedback on guesses if not win
        G: define ANSWER ( def ) < 1 > { ANSWER }
        G: give feedback ( app ) < 1 > { ANSWER } < 1278.9 >
    G: test PLAY ( error ) < 2 > { PLAY } < 11 >
          ( MOVE has no value in ANSWER )
    G: make variables consistent ( del w / in, ins w / in ) < 3 > { ANSWER } < 30 >
    G: test PLAY ( error ) < 4 > { PLAY } < 23.7 >
          ( NUM has no value in WIN )
    G: create counting function ( app ) < 5 > { GAME }
        G: initialize NUM for counting moves ( app ) < 5 > { GAME }
        G: provide for incrementing NUM ( app ) < 5 > { GAME } < 47.1 >
    G: test PLAY ( invoke ) < 6 > { PLAY } < 17.8 >
    G: take out CS ( del ) < 7 > { ASK.NAMES } < 42.3 >
    G: test PLAY ( invoke ) < 8 > { PLAY } < 29.3 >
```

Figure 15: Example 6 (episode 1, task 2, S2)

```
G: basic game
    G: define START ( def ) < 1 > { START } < 28.6 >
    G: edit ( empty ) < 2 > { } < 5.6 >
    G: introduction
        G: call INTRO ( app ) < 3 > { START }
    G: get name function
        G: call GET.NAME ( app ) < 3 > { START }
    G: record guesses
        G: call MAKE.LIST ( app ) < 3 > { START }
    G: game loop
        G: call PLAYER.LOOP ( app ) < 3 > { START }
    G: get it to go again
        G: call TRY.AGAIN ( app ) < 3 > { START }
    G: end game function
        G: call END.GAME ( app ) < 3 > { START } < 49.4 >
    S: game loop
        G: define PLAYER.LOOP ( def ) < 4 > { PLAYER.LOOP }
    G: erase message
        G: call ERASE.MESSAGE ( app ) < 4 > { PLAYER.LOOP }
    G: random number function
        G: call RANDOM.NUM ( app ) < 4 > { PLAYER.LOOP }
    G: get guess
        G: call GR.O.GN ( app ) < 4 > { PLAYER.LOOP }
    G: error checking
        G: call LEGAL? ( app ) < 4 > { PLAYER.LOOP }
    G: feedback
        G: call WIN? ( app ) < 4 > { PLAYER.LOOP }
    S: game loop
        G: recurse to PLAYER.LOOP ( app ) < 4 > { PLAYER.LOOP }
    S: get name function
        G: define GET.NAME ( def ) < 5 > { GET.NAME }
        G: get name ( app ) < 5 > { GET.NAME } < 49.9 >
        G: test GET.NAME ( error ) < 6 > { GET.NAME } < 1.8 >
              ( I don't know how to RS )
        S: get name
            G: substitute RW ( del w / in, app ) < 7 > { GET.NAME } < 147.6 >
        G: test GET.NAME ( invoke ) < 8 > { GET.NAME } < 4.6 >
        G: give instructions ( ins ) < 9 > { GET.NAME }
        G: print name of player ( app ) < 9 > { GET.NAME } < 61.3 >
        G: test GET.NAME ( error ) < 10 > { GET.NAME } < 3.4 >
              ( I don't know what to do with FDSD )
        S: print name of player
            G: fix name printing ( del w / in, ins w / in ) < 11 > { GET.NAME } < 20.7 >
        G: test GET.NAME ( invoke ) < 12 > { GET.NAME }
        S: print name of player
            S: fix name printing
                G: put on different lines ( del w / in, ins ) < 13 > { GET.NAME } < 15.7 >
        G: test GET.NAME ( invoke ) < 14 > { GET.NAME } < 3.9 >
```

Figure 16: Example 7 (episode 1, task 2, S4)

```
G: basic game
    G: define BEGIN ( def ) < 1 > { BEGIN }
    G: random number function
        G: call FIND# ( app ) < 1 > { BEGIN }
    G: process guess
        G: call GUESS ( app ) < 1 > { BEGIN }
    G: check number function
        G: call CHECK ( app ) < 1 > { BEGIN }
    S: random number function
        G: define FIND# ( def ) < 1 > { FIND# } < 336.2 >
        G: test RANDOM command ( invoke ) < 2 > { } < 4.1 >
        G: test RANDOM command ( invoke ) < 3 > { } < 6.1 >
        G: test RANDOM command ( invoke ) < 4 > { } < 1.4 >
        G: generate random number ( app ) < 5 > { FIND# } < 64.5 >
    G: test BEGIN ( error ) < 6 > { BEGIN } < 4.6 >
           ( not enough inputs to FIND# )
    S: random number function
        G: get rid of number variable in command line ( del ) < 7 > { FIND# } < 26.7 >
    G: test BEGIN ( error ) < 8 > { BEGIN } < 11.3 >
           ( NUM has no value )
    S: random number function
        G: verify number ( error ) < 9 > { } < 9.9 >
               ( NUM has no value )
    G: get name function
        G: call GET.NAME ( ins ) < 10 > { BEGIN }
    G: abandon CHECK ( del ) < 10 > { BEGIN }
    S: random number function
        S: generate random number
            G: change punctuation of variable ( del w / in, ins w / in ) < 10 > { FIND# }
    S: process guess
        G: define GUESS ( def ) < 10 > { GUESS }
    G: get guess ( app ) < 10 > { GUESS }
    G: win function
        G: if win, call WIN ( app ) < 10 > { GUESS }
    G: feedback function
        G: if not win, call HINT ( app ) < 10 > { GUESS }
    S: get name function
        G: define GET.NAME ( def ) < 10 > { GET.NAME }
        G: get name ( app ) < 10 > { GET.NAME } < 306.2 >
    G: test BEGIN ( error ) < 11, 12 > { BEGIN } < 8.2 >
           ( I don't know how to AVID. get first letter, then rest )
    S: get name function
        G: get all of name
            G: see what name is ( app ) < 13 > { GET.NAME } < 25.4 >
    G: test BEGIN ( error ) < 14 > { BEGIN } < 4 >
           ( not enough inputs to MAKE in FIND#. just gets first letter of name )
    S: random number function
        S: generate random number
            G: correct syntax of MAKE ( del, ins w / in ) < 15 > { FIND# } < 104.5 >
    G: test BEGIN ( error ) < 16 > { BEGIN } < 8.6 >
           ( I don't know how to HINT )
    S: get name function
        S: get all of name
            G: isolate GET.NAME ( ins ) < 17 > { BEGIN } < 27.7 >
    G: test BEGIN ( error ) < 18, 19 > { BEGIN } < 6.1 >
           ( I don't know how to AVID. get first letter, then rest )
    S: get name function
        S: get all of name
            G: switch to RW ( del, ins ) < 20 > { GET.NAME } < 16.4 >
    G: test BEGIN ( invoke ) < 21 > { BEGIN } < 5.3 >
```

Figure 17: Example 8 (episode 1, task 2, S3)

```
G: basic game
    G: define START ( edit error ) < 1 > { START } < 23.9 >
        G: try again ( def ) < 2 > { START }
    G: setup
        G: call BEGIN ( app ) < 2 > { START }
    G: get name function
        G: call ENTERNAME ( app ) < 2 > { START }
    G: get random number
        G: call CHOOSENUMBER ( app ) < 2 > { START }
    G: get guesses
        G: call ENTERNUMBER ( app ) < 2 > { START }
    G: error check
        G: call LEGAL ( app ) < 2 > { START }
    G: give feedback on guesses
        G: call TOOHIGHLOW
    G: check if won
        G: call CHECKWIN ( app ) < 2 > { START }
    G: inform if won
        G: call WIN ( app ) < 2 > { START }
    G: inform number of guesses
        G: call NUMBERGES ( app ) < 2 > { START }
    G: inform what guesses used
        G: call GEGUSED ( app ) < 2 > { START } < 199.4 >
    G: examine names of procedures ( invoke ) < 3 > { } < 2.2 >
    G: examine START ( unchanged ) < 4 > { START } < 14.1 >
    S: setup
        G: define BEGIN ( 5 ) < def > { BEGIN }
        G: tell what the game is ( 5 ) < app > { BEGIN } < 71.5 >
    S: get name function
        G: define ENTERNAME ( def ) < 6 > { ENTERNAME }
        G: wait and instruct ( app ) < 6 > { ENTERNAME }
        G: get name ( app ) < 6 > { ENTERNAME } < 270.4 >
    G: test START ( error ) < 7 > { START } < 7.4 >
            ( I don't know what to do with SDF )
    S: get name function
        S: get name
            G: intialize RW ( app ) < 8 > { ENTERNAME } < 61.9 >
    G: test START ( error ) < 9 > { START } < 5 >
            ( I don't know what to do with GHFGH )
    S: get name function
        S: get name
            G: remove initialization ( del w / in ) < 10 > { ENTERNAME }
            G: use MAKE with RW ( ins w / in ) < 10 > { ENTERNAME } < 234.5 >
    G: test START ( error ) < 11 > { START } < 4.4 >
            ( I don't know how to CHOOSENUMBER )
    S: get name function
        G: verify got name ( invoke ) < 12 > { } < 22.6 >
    G: examine procedures ( unchanged ) < 13 > { START ENTERNAME } < 19.1 >
```

Bibliography

Adelson, B. (1981). Problem solving and the development of abstract categories in programming languages. Memory and Cognition, 9(4), 422-433.

Adelson, B. (1984a). When novices surpass experts: the difficulty of a task may increase with expertise. Journal of Experimental Psychology, 10(3), 483-495.

Adelson, B. (1984b). Constructs and phenomena common to semantically-rich domains. (Report no. 10, Cognition and Programming Project). New Haven, CT: Yale Computer Science Department.

Adelson, B. & Soloway, E. (1985). The role of domain experience in software design. IEEE Transactions on Software Engineering, 11(11), 1351-1360.

Bateson, A. G., Alexander, R. A., & Murphy, M. D. (1987). Cognitive processing differences between novice and expert computer programmers. International Journal of Man-Machine Studies, 649-660.

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

Brown, J. S. & Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. In D. Sleeman & J. S. Brown (Eds.), Intelligent tutoring systems (pp. 157-183). London: Academic Press.

Card, S. K., Moran, T. P., & Newell, A. (1983). The Psychology of Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum Associates.

Carver, S. M. (1987). Transfer of LOGO debugging skills: Analysis, instruction, and assessment. Ann Arbor, MI: University Microfilms International.

Carver, S. M. & Klahr, D. (1986). Assessing children's debugging skills with a formal model. Journal of Educational Computing Research, 2(4), 487-525.

Curtis, B. (1988). Five paradigms in the psychology of programming. In M. Helander (Ed.), Handbook of Human-Computer Interaction (pp. 87-105). North-Holland: Elsevier Science Publishers.

Davies, S.P. (1990a). Plans, goals and selection rules in the comprehension of computer programs. Behaviour and Information Technology, 9 (3), 201-214.

Davies, S.P. (1990b). The nature and development of programming plans. International Journal of Man-Machine Studies, 32, 461-481.

Delclos, V. R. & Kulewicz, S. J. (1985). The teacher as mediator in computer-based problem solivng. Paper presented at the annual meeting of the American Educational Research Association, Chicago, IL.

Delclos, V. R., Littlefield, J., & Bransford, J. D. (1985). Teaching thinking through Logo: the importance of method. Roeper Review, 7, 3, 153-156.

Dunbar, K. & Klahr, D. (1989). Developmental differences in scientic discovery processes. In D. Klahr & K. Kotovsky (eds.), Complex Information Processing (pp. 109-143). Hillsdale, NJ: Lawrence Erlbaum Associates.

Emihovich, C. & Miller, G.E. (1986). Verbal mediation in LOGO instruction: Learning from a Vygotskian perspective. Paper presented at the annual meeting of the American Educational Research Association, San Francisco, CA.

Ericsson, K.A. & Simon, H.A. (1984). Protocol Analysis. Cambridge, MA: MIT Press.

Fisher, C. (1986). How do programmers program: Coping with complexity. Unpublished manuscript.

Fisher, C. (1988). Advancing the study of programming with computer-aided protocol analysis. In G. M. Olson, E. Sheppard, & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop (pp. 198-216). Norwood, NJ: Ablex.

Gilmore, D.J. & Green, T.R.G. (1988). Programming plans and programming expertise. Quarterly Journal of Experimental Psychology, 40A (3), 423-442.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. International Journal of Man-Machine Studies, 7, 151-182.

Gray, W. D. & Anderson, J. R. (1987). Change-episodes in coding: When and how do programmers change their code? In G.M. Olson, S. Sheppard, E. Soloway, Empirical Studies of Programmers: Second Workshop (pp. 185-197). Norwood, NJ: Ablex.

Hoc, J.M. (1977). The role of mental representation in learning a programming language. International Journal of Man-Machine Studies, 9, 87-105.

LOGO Computer Systems, Inc. (1984). LOGO. Bacon Roca, Florida: IBM.

James, J. M., Sanderson, P. M., & Seidler, K. S. (1990). SHAPA Version 2.0. [Computer program and manual]. Urbana, IL: Department of Mechanical and Industrial Engineering, University of Illinois at Urbana-Champaign.

Jeffries, R., Turner, A. A., Polson, P. G., Atwood, M. E. (1981). The processes involved in designing software. In J. R. Anderson (Ed.), Cognitive Skills and Their Acquisition (pp. 255-283). Hillsdale, NJ: Lawrence Erlbaum Associates.

Kant, E. & Newell, A. (1984). Problem solving techniques for the design of algorithms. Information Processing and Management, 20(1-2), 97-118.

Khayrallah, M. A. & Van Den Meiraker, Maud. (1987). Logo programming and the acquisition of cognitive skills. Journal of Computer-Based Instruction, 14, 4, 133-137.

Kinzer, C., Littlefield, J., & Delclos, V. R. (1985). Different Logo learning environments and mastery: Relationships between engagement and learning. Paper presented at the annual meeting of the American Educational Research Association.

Klahr, D. & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. Cognitive Psychology, 20, 362-404.

Klahr, D. & Dunbar, K. (1988). Dual space search during scientific reasoning. Cognitive Science, 12, 1-48.

Kowalski, B. & VanLehn, K. (1988). Cirrus: Inducing subject models from protocol data. Program of the Tenth Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Lawrence Erlaum Association.

Krasnor, L. R. & Mitterer, J. O. (1984). Logo and the development of general problem-solving skills. Alberta Journal of Educational Research, 30, 2, 133-144.

Kull, J. A. (1986). A Brunerian approach to teaching and learning Logo. Paper presented at the annual meeting of the American Educational Research Association, San Franciso, CA.

Lehrer, R. (1986). Logo as a strategy for developing thinking? Educational Psychologist, 21(1&2), 121-137.

Lehrer, R. & Smith, P. C. (1986). Logo learning: Is more better? Paper presented at the annual meeting of the American Educational Research Association, San Francisco, CA.

McAllister, A. P. (1985). Assisted LOGO learning. Paper presented at the annual meeting of the Ontario Educational Computing Organization, Toronto, Ontario.

McKeithen, K., Reitman, J. S., Rueter, H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers Cognitive Psychology, 13, 307-325.

Michayluk, J. O. (1986). Logo: More than a decade later. British Journal of Educational Technology, 17(1), 35-41.

Newell, A. (1980). Reasoning, problem solving, and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), Attention and Perfromance, Vol. 7 (pp. 693-718). Hillsdale, NJ: Lawrence Erlbaum Associates.

Newell, A. & Simon, H.A. (1972). Human Problem Solving. Englewood Cliffs, NJ: Prentice Hall.

Newell, A. & Simon, H.A. (1976). Computer science as empirical inquiry: Symbols and search. Communications of the ACM, 19(3), 113-126.

Norman, D.A. (1986). Cognitive engineering-- cognitive science. In D.A. Norman & S.W. Draper (Ed.), User Centered System Design. Hillsdale, NJ: Lawrence Erlbaum Associates.

Ohlsson, S. & Langley, P. (1985). Identifying solution paths in cognitive diagnosis. Pittsburgh, PA: Robotics Institute, Carnegie-Mellon University.

Pea, R. D. & Kurland, D. M. (1984). On the cognitive effects of learning computer proramming. New Ideas in Psychology, 2(2), 137-168.

Perkins, D. N. & Salomon, G. S. (1989). Are cognitive skills context bound? Educational Researcher, 18(1), 16-25.

Redmond, R. T. & Gasen, J. B. (1988). PAST: Viewing the programming process. Behavior Research Methods, Instruments, & Computers, 20(5), 503-507.

Redmond, R. T. & Gasen, J. B. (1989). Measuring change in the programming process. International Journal of Man-Machine Studies, 30, 697-711.

Rist, R. S. (1986). Plans in programming: Definition, demonstration, and development. In E. Soloway (Ed.), Empirical Studies of Programming (pp. 28-45). Norwood, NJ: Ablex Publishing.

Rist, R. S. (1989). Schema creation in programming. Cognitive Science,

13, 389-414.

Saloman, G. & Perkins, D. N. (1989). Rocky roads to transfer: Rethinking mechanisms of a neglected phenomenon. Educational Psychologist, 24(2), 113-142.

Sanderson, P. M., James, J. M., & Seidler, K. S. (1989). SHAPA: an interactive software environment for protocol analysis. Ergonomics, 32(11), 1271-1302.

Sheil, B. A. (1981). The psychological study of programming. Computing Surveys, 13, 1, 101-120.

Shneiderman, B. (1976). Explorary experiments in programmer behavior. International Journal of Computer and Information Sciences, 5, 123-143.

Shneiderman, B. (1977). Measuring computer program quality and comprehension. International Journal of Man-Machine Studies, 9, 465-478.

Shneiderman, B. & Mayer, R. (1979). Syntactic/semantic interactions in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 8, 219-238.

Simon, H.A. (1978). Information-processing theory of human problem solving. In W. K. Estes (Ed.), Handbook of Learning and Cognitive Processes: Human Information Processing, Vol. 5. Hillsdale, NJ: Lawrence Erlbaum Associates.

Simon, H. A. (1981). Studying human intelligence by creating artificial intelligence. American Scientist, 69, 300-309.

Simon, H. A. & Lea, G. (1975). Problem solving and rule induction: A unified view. In L.W. Gregg (Ed.), Knowledge and Cognition (pp. 105-128). Hillsdale, NJ: Lawrence Erlbaum Associates.

Soloway, E. & Ehrlich, K. (1984). Empirical studies of programming knowledge. IEEE Transactions on Software Engineering, 5, 595-609.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In B. Shneiderman & A. Badre (Eds.), Directions in human-computer interaction (pp. 87-122). Norwood, NJ: Ablex Publishing.

VanLehn, K. & Garlick, S. (1987). Cirrus: An automated protocol analysis tool. In P. Langley (Ed.), Proceedings of the Fourth Machine Learning Workshop (pp. 205-217). Los Altos, CA: Morgan-Kaufman., 205-217.

Vessey, Iris. (1985). Expertise in debugging computer programs: a process analysis. International Journal of Man-Machine Studies, 23, 459-494.

Vessey, I. (1986). Expertise in debugging computer programs: An analysis of the content of verbal protocols. IEEE Transaction on Systems, Man, and Cybernetics, 16(5), 621-637.

Waterman, D.A. & Newell, A. (1972). Preliminary Results with a System for Automating Protocol Analysis. Pittsburgh, PA: Departments of Psychology and Computer Science, Carnegie-Mellon University.

Appendix A: Key to the Problem Behavior Graphs

In the graphs, top level goals appear on the far left (G: square). Indentations represent the formation of subgoals and the preceeding goal immediately to the left is the superordinate goal. When a goal is formed which is a subgoal of a goal that is not adjacent to it in the graph, the associated superordinate goal appears (S: square). The goals are described in as abbreviated a form as possible either in terms of the component or, in the case of debugging, in terms of the purpose of the change. Operators are indicated in parentheses in abbreviated form: "def" for define, "app" for append, "ins" for insert, "del" for delete, and so on for program space operators. When a whole instruction is not the object of the operator, the change is indicated as "w/in" for within ("del w/in, ins w/in" usually indicates a simple replacement such as "FD 40" for "FD 60"; the abbreviation "rep" is used instead of the longer form in example 3). Trial space operators are indicated by "invoke." Operators that result in errors are indicated as "error." The numbers immediately following the operator indicate the line number. The information in brackets after the line number indicates the initiating action. In the case of errors, the error message follows in parenthesis. The time from the last action to the completion of the action is given for the last set of operators for the line.

The set of examples for the first task (Figures 9 to 12) represents states of the two problem spaces to the left and the right of the problem behavior graphs. In order to conserve space, not all of the programs are shown; in example 1, for instance, after the first program space state, only the changes to the program and the instructions that surround them are shown.

The set of examples for the second task (Figures 14 to 17) only includes the problem behavior graphs themselves; the size of the programs precludes including them on the same page as the graphs.

Appendix B: Instructions for the House-Playhouse task

Write a program in LOGO which will draw a front view of a 2 story house. The house should have a door and 2 symmetrically placed windows on the first floor, 3 evenly spaced windows on the second floor, a gable roof, and a chimney. At the first floor level at the front of the house, there should be a deck with a picket-type fence around it. There should be a set of stairs from the deck with 10 steps going off ot one side. Next to the house there should be a play house which is an exact replica of the house at 1/4 scale.

You should produce this drawing exactly as described. Once you have drawn the house and the play house as described, you may add whatever adornments to them that you wish.

You have 75 minutes to complete the drawing. You may not be able to finish it all. Do as much of it as you can.

Appendix C: Instructions for the High-Low task

Write a program in LOGO in the form of a game for guessing numbers. The game is usually called "HILO". In this game, there is only one player. The computer chooses a number randomly from between 1 and 100, and the player must guess the number.

You should write this program in the form of an interactive game. In other words, after the computer has randomly choosen a number, the player will enter numbers from the keyboard and the game will keep going until the player has correctly guessed the number.

The computer should ask for the name of the player at the outset. There should be error-checking so that only numbers between 1 and 100 will be accepted as valid input to the game. The computer should inform the player after each guess whether the number is too high or too low or if it is the right answer. If the guess is the same as the random number, (1) the computer should inform the player by name that he or she has won, (2) print out the number of valid guesses it took to win, and (3) list the guesses that were used.

There is no need to use any screen formatting instructions for the game. You should avoid adding any more features to the game than are specified unless you have time left over after completing all the requirements for the task.

You have 75 minutes to complete the drawing. You may not be able to finish it all. Do as much of it as you can. I may intervene if I feel you are not making progress.