The Characterization of Learning Environments and Program Structures of Instructional Programs Produced Using Logo

Mei Chen

A Thesis Submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Arts

Department of Educational Psychology and Counselling
McGill University, Montreal
November 1992

Shortened version of thesis title   Mei Chen

The characterization of learning environments for instructional solfware

ABSTRACT

A methodology was developed in this study for identifying the cognitive, pedagogical, and computational characteristics of computer-based learning environments. The characterization of the cognitive and pedagogical features was achieved by decomposing the learning environments into episodes which were composed of sequences of "views". Each "view" was described in terms of the different types of knowledge presented, the pedagogical strategies used to present the knowledge, and the forms and functions of user-computer interactions elicited. The computational characteristics were described in terms of modularity and other programming properties. The methodology was applied to characterizing the instructional programs produced by student teachers using Logo.

The results showed that this methodology can successfully identify the cognitive, pedagogical and computational characteristics of the learning environments. It can also clarify what can be learned in a microworld, especially the "powerful ideas" in Logo environments. In addition, the usability and constraints of learning environments in meeting the learners' cognitive needs during the learning process can be assessed. Several findings are particular important in this study. First, student teachers represented domain knowledge in a concrete and isolated manner. Second, some student teachers developed elegant pedagogical strategies such as "modeling", "scaffolding", and "exploration". However, the problematic representations of operating knowledge indicate that student teachers often failed to consider human factors in designing computer interface for system operation. Finally, this study indicated that the learning environments constructed by student teachers lacked interactions. In particular, these envi. aments provided insufficient interactions for learning domain knowledge and for providing learners with the flexibilities in chosing task activities, task complexity, and various assistance to meet the individual needs.

This methodology has implications for studies of instructional software interface and instructional software evaluation. It also has implications for Logo research and for expertise studies in developing instructional software and other teaching contexts. Suggestions are made for further research in instructional software development and the expertise related to such development.

Résumé

Cette thèse présente une méthode qui a été développée afin d'identifier les caractéristiques cognitives et pédagogiques d'environnements d'apprentissage informatisés. De plus, cette méthode permet d'identifier les caractéristiques computationnelles de ces environnements. Les caractéristiques cognitives et pédagogiques sont identifiées en décomposant hiérarchiquement les environnements d'apprentissage en différentes "vues". Chaque vue est analysée en terme des types de connaissances présentés, des stratégies pédagogiques employées pour présenter ces connaissances, et des interactions utilisateur-ordinateur qui sont favorisées. Les caractéristiques computationnelles sont décrites en termes de modularité et de fonctions de programmation. Cette méthodologie est appliquée à la description d'environnements pédagogiques produits par des maîtres en formation utilisant Logo. Les résultats indiquent que la méthode résussit à identifier les caractéristiques cognitives et pédagogiques des environnements d'apprentissage. Les résultats indiquent également qu'il existe une relation entre les caractéristiques pédagogiques et les caractéristiques computationnelles des environnements étudiés. De plus, les avantages et les inconvénients des environnements d'apprentissage pour rencontrer les besoins cogintifs des utilisateurs peuvent être diagnostiqués. Finalement, cette méthode a des implications pour l'étude des interfaces des logiciels pédagogiques et de l'expertise dans le développement de logiciels pédagogiques. Les avenues de recherches ultérieures sont discutées.

ACKNOWLEDGMENTS

I want to thank all people who showed me love, kindness and who helped me in various ways.

I would like to express my deepest love and gratitude to my parents and my sisters Chen Jie and Chen Tao for their unconditional love and support. I also would like to thank FengShia Hsu and her children Melinda Hall and Martin Hall for making their home as warm as my own.

In producing of this thesis, I would like to heartily thank my friends David Kaufman, Soibhan Harty, and Andre Kushniruk for being my proofreaders and providing invaluable editorial suggestion.

All wonderful teachers I have had deserve special recognition. I am particularly grateful to Dr. Alain Breuleux and Dr. Glenn F. Cartwright who spared no effort to help me complete this thesis after the tragic sudden death of my supervisor Dr. Guy Groen.

I would like to specially thank Prof. Bruce Shore and Ms. Pat Grafton for their care and help in solving the associated administrative problems caused by this tragedy. I would like to thank Erika Franz and Marion Barfurth for providing emotional support when it was much needed.

Also, I would like to thank all students who voluntarily participated in this study in Faculty of Education, McGill University.

Finally, I wish to express the deepest gratitude and greatest respect to my supervisor, Dr. Guy Groen, who inspired and supervised this study and worked with me enthusiastically until his untimely death just prior to the submission of this thesis. It is to Dr. Groen I owe the greatest debt not only in the production of this study, but also in my growth as a researcher

Dr Groen was like an experienced ocean explorer who knew all the secrets of the ocean However, he did not simply tell me what treasure he had found nor how I might find it. Instead, he let me delve into the sea and gave me the freedom to explore my interests. Whenever I became lost in the exploration, he was always there to give me direction. When I made a discovery, even if a tiny one, he always shared the excitement with me and encouraged me to go further and deeper. Today, when I look back on the two years of study supervised by Dr. Groen, I realize that he was practicing the theory he stated as that the best way to teach is to give the learner the opportunity to indulge in free problem-solving activities. The approach is to follow the model of the learner as a researcher and the teacher as a research director.

It was my great fortune to have Dr. Groen as my supervisor in his last years. My life has been enriched by experiencing his outstanding expertise, his serious attitude towards research, and his warmness, as well as the enthusiasm, encouragement and confidence he inspired. It is with the greatest respect and admiration that I dedicate this thesis to his memory.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

OVERVIEW OF THE STUDY

The purpose of this thesis is to develop a methodology for characterizing computer-based learning environments. Such a methodology is required in order to describe the important features that determine the effectiveness of instructional software and identify the characteristics of instructional software that distinguish expert programmers from novice programmers. Methods currently used in instructional software evaluation and in studying programming expertise are inadequate to address the wide range of cognitive and pedagogical issues involved. Therefore, this thesis focuses on identifying the cognitive and pedagogical, as well as the computational characteristics of instructional software.

The critical roles that computers play in modern school learning necessitate a closer examination of computer-based learning environments. It is equally important to assess whether such learning environments meet the user's cognitive needs in the learning process, and to explain what features of the learning environments promote success or failure. In addition, the study of programming expertise requires a methodology that enables us to look at the products, and not just the processes, of programming.

In order to characterize effectively instructional software, it is necessary to look at the programs from both the learning environment and program structure perspectives. A learning environment refers to the display (e.g., text, graphics, speech, and animation, etc.) and user-computer interactions elicited by a program for a specific educational purpose (e.g., learning subject matter knowledge, developing general problem-solving ability or motor skills, etc.). Learning environments play critical roles in the acquisition of knowledge and skills. This is because a user acquires knowledge and skills in interacting with the information provided by a program. It is the learning environment that a user comes into contact with and explores. Therefore, the cognitive and pedagogical characteristics of the learning environments determine the effectiveness of learning. In addition, in the domain of instructional software design, expert programmers differ from novice programmers not only in terms of the knowledge they possess, but also in terms of the way they convey that knowledge to the user through the medium of the computers, as well as the interactions they design to promote learning. The differential knowledge a programmer possesses and skills for conveying such knowledge will be reflected in the products of programming. Therefore, the methodology for studying programming expertise should allow one to examine the products of programming with respect to the types of knowledge presented to the user, the ways that knowledge is presented in terms of pedagogical strategies, and the kinds of interactions promoted in a learning environment.

Program structures refer to the computational characteristics of instructional software. The computational characteristics have a significant impact on the allocation of resources, data storage, and execution time and these factors constrain the efficiency of the instructional software. Program structures are particularly important for large and complicated systems. Furthermore, in the area of instructional software design, an expert programmer may differ from a novice by the way in which a program is structured. Therefore, the description of instructional software must include a characterization of the program structures.

This study attempts to achieve four major goals. The first goal is to develop a methodology for characterizing the learning environments provided by instructional software. This methodology is applied in the context of characterizing the instructional programs produced by student teachers using Logo. The second goal is to evaluate the usability and constraints of the learning environments in meeting the user's cognitive needs during the learning process. The third goal is to characterize the program structures and examine whether there is any relationship between the learning environments and the program structures. Finally, this study investigates how the characteristics of products of programming reflect programmers' differential knowledge and skills in instructional software design, in addition to those suggested in previous programming studies.

The characterization of a learning environment is achieved by decomposing a learning environment into *episodes* (i.e., sets of exercises or lessons) which often consist of sequences of views. A view refers to a screen display and the interactions associated with this particular screen display. Each view is then characterized with respect to the types of knowledge presented, the manner in which the knowledge is presented in terms of pedagogical strategies, and the forms and functions of the interactions. The usability and constraints of the learning environments are assessed in terms of the supports needed for exploratory learning.

The characterization of program structures is conducted by depicting the program structures as either linear or modular, with consideration of other programming properties (e.g., reusable subprocedures, conditional statements, variables, and recursion). This study also investigates if there is any relationship between the learning environments and the program structures of the programs produced by student teachers using a Logo-based application called LogoWriter™.

The reason for using Logo rather than another programming language as the context for characterizing the learning environments constructed by student teachers is that Logo, a dialect of LISP, is often regarded as a novice-oriented programming language. Logo has the flexibility for designing and expanding program structures. It can also be used to produce sophisticated programs and to engage high-level programming skills. Programming is a very complex and difficult activity and it takes lots of effort and time to produce a program. For novice programmers, like student teachers, the task is even more difficult. It is critical to choose a novice-oriented programming language to minimize the difficulties of constructing an executable or "runnable" program so that novice programmers can devote greater effort to "elaborate" the learning environments that this study attempts to characterize. Meanwhile, it is important to select a programming language that has the potential for eliciting high-level programming skills and to construct sophisticated programs. Logo is an appropriate programming language that satisfies these preconditions for developing a methodology for characterizing the learning environments constructed by programmers at various levels of programming skills.

Logo learning environments create the conditions under which powerful ideas can take root (Papert, 1980). The fact that the computer-based learning environment, or the notion of a microworld, has never been defined in a precise fashion, has hampered its usefulness (Groen, 1985). Groen suggested that a more precise definition may clarify the notion of a microworld and the powerful ideas that take place in such an environment.

In summary, developing a methodology to characterize the instructional programs produced by student teachers using Logo can serve several purposes. First, it can help us to understand the nature of computer-based learning environments and it can also help us to determine whether the characteristics of a learning environment meet the users' cognitive needs in the learning processes. Second, the results from the assessment can provide guidelines for developing instructional software. Third, the identification of the important features of instructional programs can help us to determine how the characteristics of final products of programming reflect the cognitive skills of programmers at various levels of expertise in the domain of instructional software design. Finally, the precise definition of the learning environment can provide a better understanding of the nature of Logo, and can also contribute to the theoretical discussion of what is learned in such an environment.

Chapter 2

REVIEW OF THE LITERATURE

The major goal of this study is to develop a methodology to identify precisely the characteristics of instructional programs produced by student teachers using Logo. In order to understand the characteristics of instructional software, it is necessary to consider at least the following three perspectives: a) the cognitive and pedagogical features, b) the human-computer interface, and c) the computational structure of the program. These three perspectives form the basic organization of the review of the literature presented in this chapter.

First, this chapter reviews the studies on instructional software evaluation. The issue is whether the current methods used are adequate for identifying the cognitive and pedagogical features of instructional software and determining its effectiveness. Second, the process of knowledge communication between the user and the computer is examined from the point of view of human-computer interaction research. This chapter presents the methods for evaluating the usability of human-computer interface and discusses whether the notion of human-computer interface needs to be modified in order to account for the special properties of instructional software. Third, this chapter summarizes the findings on Logo, and explains why it is necessary to characterize learning environments in Logo. Finally, this chapter reviews studies on programming expertise and argues that such studies should integrate analyses of both the cognitive *processes* that a programmer engages in when producing a program and the *products* that a programmer produces as a result of the programming processes.

Instructional Software

This section presents a general introductory definition of instructional software, reviews studies on instructional software evaluation, and argues that the methods used in most of these studies do not take into account the important features that contribute to the effectiveness of instructional software.

Traditional instructional software includes four primary categories (Criswell, 1989; Hannfin & Peck, 1988): a) tutorial; b) drill and practice; c) exploratory environment; and d) games and simulation. In tutorial environments, the computer provides instruction to teach the user new knowledge, whereas in drill and practice environments the computer provides exercises to the user as reinforcement so that the user can practice on what he or she has already learned and receive feedback. Exploratory environments allow the user to engage in relatively unconstrained problem-solving activities, and the user learns by doing and exploring. Games and simulations are computer environments that present attractive pictures, animation, and even simulate complex concepts and events. The user can play games or manipulate the simulation process by giving input.

The primary objective of using various educational techniques is to improve the effectiveness of learning. To assess the effectiveness of instructional software, formative and summative evaluations are often conducted. In addition, meta-analysis method is used to summarize the results of summative evaluations for different categories of instructional software and then compare them.

Formative Evaluation

Formative evaluation is conducted to identify features that require modification. Formative evaluation procedures are applied extensively in the ongoing process of program development. Issues ranging from design logic to selection of vocabulary, from clarity of graphics to branching execution, from the judging of student input to the clarity of the lesson text should be all considered (Hannafin & Peck, 1988).

Summative Evaluation

Summative evaluation is conducted to determine whether an educational product is effective after it has been built. The purpose of summative evaluation is to validate performance rather than to locate areas in need of improvement (Hannafin & Peck, 1988). Summative evaluation is often used in experimental comparison studies. In this type of study, typically, pretests and posttests on critical variables (e.g., accuracy and latency of students' response) are conducted. The performance level achieved by the treatment group which uses the software being evaluated is compared with that of a comparison group which uses another instructional method or of a control group which receives no treatment. The conclusion is based on the statistical analysis of the results of the tests. If there is a significant difference between the two groups and the treatment group performs better in the posttest than the comparison group or the control group, then the software being evaluated is considered effective.

Meta-analysis

Investigators have used meta-analysis to summarize various summative evaluation studies on the effectiveness of educational software (e.g., Kulik & Kulik, 1987; Roblyer, Castine & King, 1988). Meta-analysis studies attempt to determine whether a particular category of educational software is efficient, and with whom, how, and when. For example, investigators often try to determine if educational software can improve students' performance in basic skills, for specific grade levels, and in particular content areas. In addition, they try to determine what kinds of students profit most from using computers to learn, and may also address whether educational software improves students' attitudes toward school and learning.

Meta-analysis uses "effect size" (ES) as a criterion to evaluate learning effectiveness. Effect size is calculated by first subtracting the mean scores (differences between pretests and posttests) achieved by the non-treatment group from that achieved by the treatment group, and then dividing the results by the pooled standard deviation of the two groups. Then the individual studies in one area are compiled to determine overall effect size. ES is often used to quantify the amount of effect due to a given treatment and compare the effectiveness of different instructional software.

Findings from Educational Software Evaluation

The results from meta-analysis indicated that instructional software generally has significant effects on all kinds of skills within all content areas at all grades, regardless of the sample of students and the types of instructional software used (e.g., Kulik & Kulik, 1987). Such results are presented in detail in the sections below.

- For whom is the instructional software effective?

Meta-analysis conducted by Roblyer, Castine, and King (1988) showed significantly higher results for students using instructional software at college/adults levels than at elementary and secondary levels. The effects were fairly homogeneous in low-achiever and regular groups. Therefore, instructional software seemed to benefit college students more than elementary students and secondary students.

When types of instructional software were compared for different student characteristics, investigators (Roblyer et al., 1988; Kulik, 1981) found that tutorials seemed to benefit good students or older students, whereas drill and practice produced highest effect sizes in elementary school. However, other investigators (Burns & Bozeman, 1981) found that disadvantaged students achieved significantly better gains in performance in comparisons with advantaged students in tutorials, and achieved about as well as advantaged students in most drill and practice studies. The overall results concerning which students benefit most from different kinds of instructional software is not clear.

- In what content areas is instructional software more effective?

By comparing the effectiveness of instructional software achieved in different content areas, some researchers found that instructional software was much more effective for learning in science than either in mathematics, language, or general problem-solving skills (Roblyer, et al., 1988). These researchers suggested that science was an especially promising area for using instructional software. The relative effectiveness achieved in the areas of mathematics, language and problem-solving skills was comparable (Roblyer, et al., 1988). However, there have been divergent findings which indicated that instructional software in mathematics was more effective than in language areas (Vinsonhaler & Bass, 1972; Roblyer & King, 1983)

- What types of instructional software are more effective?

Some studies have indicated better results with tutorials than with drill and practice in mathematics, reading (Roblyer et al., 1988) and language arts (Burns, et al. 1981; Samson, Niemiec, Weinstein & Walberg, 1985) In contrast, other studies (Niemiec Samson, Weinstein & Walberg, 1987) have found that drill and practice was more effective than tutorial at the elementary level, and that it was particularly effective for mathematics computation skills.

- How can instructional software be used effectively?

Investigators (Roblyer, et al., 1988; Willett, Yamashita & Anderson, 1983) have found that simulated experiments in science were highly effective only when students were provided with the opportunity to interpret results and make decisions on the basis of the results.

In comparisons of supplement versus replacement roles for instructional software, the findings suggest that instructional software is more effective in supplemental than replacement uses (Roblyer et al., 1988), which suggests that teacher participation is necessary for the successful implementation of instructional software.

The Need for Developing a Methodology to Identify the Cognitive and Pedagogical Characteristics of Instructional Software

There are several problems in the evaluation studies reported in the previous section. First, ES used in meta-analysis is a comparative value and it depends on not only the effect in the treatment group, but also in the non-treatment group. The larger the effect in the non-treatment group, the lower the ES will be. Therefore, ES cannot provide an estimate of the effectiveness of a given software independent of the effectiveness of the comparison group(s).

Second, the inconsistent findings in these studies suggested that the effectiveness of instructional software was confounded with a number of factors such as students characteristics, teacher interventions, and the nature of the subject areas in which software was used. Results from these studies are difficult to interpret. According to Breuleux (1992) the difficulty is caused mainly by the fact that most reports of instructional software: a) do not present the assumptions that are implemented in the software; b) do not clearly explain how the assumptions are actually implemented; and c) do not systematically test alternative combinations of assumptions and implementations.

The third problem is that, these evaluations were based on categories of instructional software rather than on specific programs. The inconsistent findings on the effectiveness of the same types of instructional software may also indicate that one piece of instructional software is efficient whereas another is inefficient within the same category. It might be the characteristics of the individual software rather than the categories of instructional software that determined the differential effectiveness. However, previous evaluation of instructional software did not provide sufficient information regarding the characteristics of individual instructional software. Without precise identification and description of the cognitive and pedagogical characteristics of individual programs, it is impossible to examine their strengths and weaknesses. Consequently, it is difficult to determine what factors within the individual software promote success or failure and provide further useful information for good instructional software ɔign.

Intelligent Tutoring Systems

The promise of computer-assisted instruction is to provide learners with a rich learning environment that is tailored to the user's individual learning needs and objectives (Clancey & Soloway, 1990). However, traditional instructional software does not seem to have such capacity. Since the 1970s, researchers have applied artificial intelligence (AI) methods to create more sophisticated learning environments called *intelligent tutoring systems* (ITSs).

Intelligent Tutoring Systems (ITSs) are computer programs that use AI techniques for presenting knowledge and carrying out complex interactions with students (Sleeman & Brown, 1982). In current ITS research, many different architectural components are proposed and used in unique combinations and often with unique structures (Psotka, Massey & Mutter, 1988). In spite of the variety, the standard architecture of an ITS consists of three primary components: the student modeling module, the expert module, and the tutorial module (Clancey & Soloway, 1990; Frye, Littman & Soloway, 1988). Ideally, the student model involves a description of all aspects of the students' knowledge and behaviour pertinent to performance (Wenger, 1987). In an ITS, the expert module contains a representation of the domain knowledge to be communicated and also serves as a standard for evaluating student performance. The tutorial module embodies specific instructional goals such as, the remediation of particular misconceptions or the sequencing of material Much research effort goes into developing these modules since they form the core of ITS (Frye et al., 1988). Until recently, the idea that pedagogical knowledge could be explicitly represented in tutoring systems has received less attention than the representation of the subject matter (Wenger, 1987). The need for investigating interface design issues in instructional software has been underlined only in the more recent field of intelligent tutoring systems (e.g., Frye, Littman & Soloway, 1988) but there is a lack of specific research findings. Significant effort will need to be directed toward looking at interface design.

Human-Computer Interface

Research on human-computer interaction draws attention to the importance of interface in the design of software systems. Since there has been little research on the issues of instructional software interface, the area of human-computer interface research will also be reviewed to provide a better understanding of the interactive processes involved in computer-based learning environments.

According to Card, Moran, and Newell (1983), the defining notion of the human-computer interface is that the user and the computer engage in a communicative dialogue because both have access to the stream of symbols flowing back and forth to accomplish the communication; each can interrupt, query, and correct the communication at various points in the process.

This statement emphasizes two agents — the user and the computer in the communicative dialogue. Ravden and Johnson (1989) proposed a clear definition of human-computer interface:

> The user interface generally consists of information displayed to the user and facilities which allow the user to enter information into the computer, to manipulate information which is displayed, and to take control actions. It enables the user to access and make use of the tasks for which it has been designed. It provides the user with information about the system, about what it does, and about what the user can and should do. It enables the user to learn about the system and to build an understanding of how it works (p. 15).

Researchers generally consider that the human-computer interface consists of three components: the user, the computer, and the tasks. These three principal components represent the three major topics in the research on human-computer interface.

Research Approaches to Human-Computer Interface

Researchers in the fields of computer science and software engineering generally agree that the human-computer interface should and can be improved, although there is currently no consensus on exactly how to design a better human-computer interface. The promising approaches are dependent upon analyzing the dynamic interactions between computer systems, tasks, and users (Bennett, 1984; Eason, 1981; Shackel, 1991).

Computer systems

The research on computer systems from the perspective of human-computer has two foci. One focus is on the physical devices of computer system, another focus is on the cognitive factors related to computer systems.

Physical devices. The studies on physical devices are mostly related to display layout and input-output devices. Early studies of physical devices considered the physical quality of display (e g., luminance, contrast, regeneration rate, and resolution). More recently studies were concerned with display layout and development of input-output devices (e.g., mouse, light pen, handwriting input, touchscreen, voice synthesizer, picture processing, and video display terminals) (e g., Balzert, 1988, Bullinger, 1988).

Cognitive factors. Naturalness, feedback, and consistency are the cognitive factors generally investigated in human-computer interaction research. In addition, simplicity and individualization are often studied. It is frequently asserted that novices and unsophisticated users would find computer systems more congenial and easy to use if they could communicate with the computer using termii ology similar to natural language commands and queries (e g., Ledgard, Whiterside, Singer & Seymour, 1980). However, some researchers found that the use of an artificial data-base language resulted in faster performance than when natural language was used (Small & Welson, 1977). The effects of immediate or delayed feedback, and positive or negative feedback in the human-computer interface have been investigated (Corbett & Anderson, 1992; Shneiderman, 1980a). Consistency is regarded as an important aspect of the quality of user interface. Consistency refers to regularities in various aspects of the interactions or interface: the actions that the user has to perform in order to achieve a task, the feedback the system provides, the spatial layout of the screen, etc. (Schiele & Green, 1990). Consistent interfaces allow users to make generalizations on the basis of their current knowledge. This facilitates the learning process and the development of automated responses which can help reduce the user's working memory load (Schiele & Green, 1990).

Tasks

Computers have been widely used to perform tasks such as word processing, calculation, drawing, and accounting. Researchers in the field of human-computer interactions have studied the tasks of programming (Brooks, 1977), editing (Card, Moran & Newell, 1980), learning to use a word processor (Carroll & Mack, 1984), and fault diagnosis (Rouse, Rouse & Pellegrino, 1980). The typical approach is to decompose the task into hierarchical branches and analyze the behaviour of the user with the behaviour of the computer.

Users

The human factors considered in human-computer interactions are working memory load, long-term memory (LTM), and mental models of problem solving activities in the process of interacting with the computers Working memory load is considered to be how much immediate information the user has to keep in working memory whereas LTM is considered to be how easy is it for the user to recall information needed to accomplish a task (Card, Moran & Newell, 1983). Mental model is a theoretical construct that has been used to describe how individuals form internal models of systems from interacting with these systems (Norman, 1983). Researchers have begun to consider the user's mental models in human-computer interaction, investigating, for example, the user's mental models of tasks, how different types of representations affect the user's performance and how to apply what we know of the user's knowledge to design interface and train users (Carroll & Olson, 1987).

The Methods for Evaluating the Usability of Human-Computer Interface

Measuring usability means measuring the behaviour of a user and the system during the performance of a task. The usability of human-computer interface is measured by how easily and how effectively the computer can be used by a specific set of users, given particular kinds of training and user support to fulfill the specified range of tasks in a defined set of environments (Chapanis, 1991; Shackel, 1984, 1991).

There are three criteria usually suggested for evaluating usability (Shackel, 1991). The first criterion is the success rate in meeting a specified range of users, tasks, and environments. The second criterion is the ease of use as judged by the users (e.g., convenience, comfort, effort, and satisfaction). The last criterion is the effectiveness of human use in terms of performance (e.g., time, errors, number, and sequence of activities, etc.) in learning, relearning, and carrying out a representative range of operations.

Based on these criteria, the methods for evaluating the usability of an interface include task analyses, questionnaires, comparisons of a program against "standards" (e.g., checklists, specifications) and field tests or experiments.

Task analysis

Investigators have argued that task analysis was potentially the most powerful method in the field of human-computer interaction (HCI) either for evaluating systems or for producing requirement specifications (Card, Moran & Newell, 1983). A task analysis allows one to describe the cognitive and motor aspects of the tasks (Diaper, 1989).

According to Card, Moran, and Newell (1983), an assumption underlying task analysis is that, humans behave in goal-oriented ways, and within their limited perceptual and information-processing abilities, attempt to adapt to the task environments to attain their goals (p. 10). A task analysis models the behaviour of expert user performance by giving his or her goals, operators, methods, and selection rules for choosing among method alternatives.

The GOMS model (Goals-Operators-Methods-Selection rules) proposed by Card, Moran, and Newell (1983) describes the behaviour of a computer-user in a text editing task. In this model, the user's cognitive structure consists of four components: a) a set of goals, b) a set of operators, c) a set of methods for achieving the goals, and d) a set of selection rules for choosing among competing methods for goals.

Card, Moran, and Newell (1983) suggested some basic performance variables to be used as criteria for measure the ease and effectiveness of the human-computer interface by other researchers. These variables include functionality, time to learn to use the system, time to perform specific tasks, as well as types and number of errors made. The GOMS model can be used to predict the user behaviour sequence and the time required to perform particular task.

Subjective measures of usability: questionnaires

Subjective measures of ease of use, often combined with task analysis methods, are obtained by ratings on questionnaires that include questions on attitude (Zoltan & Chapanis, 1982; Shneiderman, 1987), user's acceptance (i.e., how the user subjectively rates the system) and enjoyability of the system (i.e., how much fun it is for the user).

Evaluating a program against a "standard"

Using a checklist is a practical method for evaluating the usability of the human-computer interface. The evaluator carries out the tasks for which the system is designed and evaluates the system according to the items listed in the checklist reflecting conventions shared by the field of computer system design or less frequently by the principles of human cognition. The checklist usually includes visual clarity, consistency, compatibility, informative feedback, explicitness, and appropriate functionality. Flexibility and control, error prevention and correction, user guidance and support are also often included in the checklist (Ravden & Johnson, 1989). For example, the checklist may suggest that "X percent of typical users should be able to read and understand the instruction in less than Y time", or "X percent of typical users should be able to diagnose and correct their errors in less than Y minutes."

Diagnostic evaluation

Diagnostic evaluation refers to the process of choosing a target user to perform the tasks for which computers are designed, and observing and analyzing the user's behaviour in great detail. Diagnostic evaluation is something like a physician diagnosing a disease: using the errors, difficulties, help requests, response times, and complaints as symptoms for diagnosing problems (Chapanis, 1981). By analyzing the user's performance frame by frame, the experimenter probes to find out whether the instructions were unclear, whether the information presented was inadequate, and whether the vocabulary was too difficult (Chapanis, 1981).

Experimental evaluations

Experimental evaluations refer to the tests that involve comparing particular features or functionality with more than one group of subjects or comparing several different products with similar subjects (Chapanis, 1991). When comparing some features with different subjects, the evaluator measures the users' performance in terms of time, questions, and errors. This method can answer whether the same features are easier for population A than for population B, but does not answer why the same features are easier for population A than for population B. When comparing different products with similar subjects, the evaluator measures the difficulty of different features of each program by mean percentages of "essentially correct" scores. This method can answer the question whether program A is easier than program B, but it does not answer why A is easier than B.

Most of the evaluation methods presented above are conducted after the development of whole systems has been completed. A disadvantage of such evaluations is that any problem detected will demand considerable modification when it may be too late to effect the desired change.

A cognitive walkthrough method

Polson, Lewis, Rieman, and Wharton (1991) developed a cognitive walkthrough method which was adapted from the design walkthrough techniques that have been used for many years in the software community. In a cognitive walkthrough evaluation, the author of a particular aspect of design presents to a group of peers a proposed design solution. The method involves hand simulation of the cognitive activities of a user. The peers evaluate the solution using an explicit set of criteria appropriate to the particular class of design issues. The criteria are focused on the cognitive processes needed by the users to successfully complete the tasks for which the system was designed. That is, first-time users can perform tasks with little or no formal instruction or informal coaching. They must learn to operate the system by using cues provided by the system rather than by using prior knowledge acquired through instruction.

During the walkthrough process, the reviewers step through the actions, considering the behaviour of the interface and its effect on the user, and diagnosing whether a typical use will succeed or fail. In particular, the reviewers must identify those actions that would be difficult for the average member of the target population to choose or execute, and analyze the causes of failures.

Application of Human-Computer Interface Approaches to Studying Instructional Software Interface

Studies on the human-computer interface have made a great contribution to understanding and improving the usability of computer systems. These studies have captured the fundamental components of the human-computer interface and have provided some theory-based or convention-based methods to evaluate the easiness and effectiveness of a human-computer interface. However, the study of human factors in human-computer interaction is relatively new and has not focused on computer uses for learning tasks (Frye et al., 1988). Therefore, the concepts and approaches taken in the area of human-computer interfaces need to be clarified and adapted in order to be used for studying instructional software interface.

The users in most studies of human-computer interfaces were either experts who displayed error-free behaviour or novices who had subject matter knowledge but did not know how to operate the computers. The typical users of instructional software are novices who have neither subject matter knowledge nor operating knowledge. Consequently, instructional software needs to be evaluated both from the point of view of the subject matter knowledge that is presented to the learner and the operating knowledge that the learner must use to operate the system. In terms of the operating knowledge, it is important to consider whether the computer environment provides cues for operating the system and for learning to operate the system by exploration. This kind of assessment can be used not only to detect and diagnose problems but also to find the strengths of the system so that a more complete picture of the instructional software can be provided.

The evaluations of human-computer interfaces in general were focused on "usability" which refers to easiness and effectiveness of performing the tasks rather than "learnability" which refers to easiness and effectiveness of learning subject matter knowledge. When the researchers in the area of human-computer interfaces used words such as "learning" or "learnability", they referred to learning how to use the computer rather than learning subject matter knowledge. In previous studies of human-computer interfaces, most tasks did not involve the learning of subject matter knowledge. Moreover, some studies only required the users to perform routine tasks (Card, Moran & Newell, 1983). Therefore, the tasks involved little learning about how to use computers. The evaluation of the instructional software needs to be concerned with both the usability of operating the system and the learnability of subject matter knowledge.

The validity of "standards"

Instructional software, particularly ITS, is a relatively new area of research, so the attributes of good instructional software are not known. Thus, there is no "standard" for good instructional software that is well established. Furthermore, the requirements for the instructional software interface may differ on the basis of the characteristics of learners, the nature of subject areas, and teaching approaches. This complexity presents considerable difficulty for establishing a "standard".

User's behaviour versus system's behaviour

Most methods for evaluating the usability of human-computer interface focus on evaluating the user's behaviour rather than the system's behaviour. For example, an evaluation conducted by Software Digest (1984) presented the numbers of tasks that a user can perform with the system (i.e., versatility), time of learning to operate the system, time of performing specific tasks, and error rate as measures of usability (Chapanis, 1991). The correlation of the above variables indicated that all measures, except versatility, are positively correlated. What this finding suggests is that the programs that were easier to use, easier to start up, easier to learn to use, and allowed users to perform tasks more quickly, were less versatile (Chapanis, 1991).

What can be learned from such evaluation? Does it mean that the versatility has to be reduced if the programs are to be easier to use, easier to start up, easier to learn to use, and allow the users to deal with errors more easily? The results of these evaluations are difficult to interpret and do not seem to provide sufficient information for improving the quality of the programs. In order to evaluate and compare the quality of instructional software, and provide further useful information for improving the system, it is necessary to identify and describe the system's behaviour in conjunction with the user's behaviour.

Computer-based environments

Studies of human-computer interfaces in the computer engineering community often viewed the computer-based environments in terms of the physical devices (i.e., input-output devices). It is more important to view a computer-based learning environment in terms of cognitive and pedagogical features (e.g., pedagogical strategies used to present various types of knowledge, interactions promoted). It is the cognitive learning environment that has the most significant effect on a user's learning when the user engages in the activities of learning subject knowledge. This emphasis does not imply that a cognitive environment is completely independent of physical devices. However, a good set of computer devices does not guarantee a good cognitive environment.

Logo Exploratory Learning Environments

The present study uses Logo as a tool for investigating the development of instructional software. Logo was originally designed for children and it is regarded as an exploratory learning environment in which children can learn by discovery and doing. The underlying rationale is adapted from Piagetian constructivism which asserts that learning takes place through the construction of mental models developed in exploration. Papert (1986) explained constructivist theory from two perspectives. First, from a psychological perspective, learning is considered as a reconstruction rather than a transmission of knowledge. Second, from an educational perspective, learning is particularly effective when it is embedded in an activity that the learners experience in constructing a meaningful product (such as a computer program) rather than acquiring knowledge and facts without a context which can be immediately used and understood. Logo programming requires the explicit definition of ideas, the reconstruction of the ideas or the development of computational models of the concepts, turning the children into epistemologists.

The Claims of Logo

The five most important claims made by Papert and other Logo advocates (Brooks, 1977; Nickerson, 1982) can be summarized as follows: First, Logo can serve as an object-to-think-with, a model for the notion of assimilation. It can be used as a tool to construct learning environments in which other learning can occur, and therefore it supports other school learning. Second, it is hypothesized that, through the processes of programming the computer to perform various tasks, Logo allows students to acquire certain cognitive capabilities which can be transferred to problem solving in many other contexts. Third, the experiences from Logo can bring about a more positive mindset in students as intellectual agents, increasing their self-esteem and making science and mathematics attractive to children. Fourth, the flexibility of Logo allows children to display and develop their creativity. Finally, Logo is claimed to be accessible with virtually no pre-requisites and to offer potential for unlimited development; therefore, it can be used by different populations with diverse characteristics.

The Tests of the Claims: Findings from Empirical Studies

Based on these claims, previous Logo research has focused on understanding the cognitive and social effects of children's experiences with Logo. Specifically, researchers tried to find out, first of all, what are the cognitive outcomes for children of programming the computer to perform various tasks? In particular, can Logo help students acquire certain cognitive capabilities, and can these capabilities be transferred to problem solving in other domains? Second, can Logo, as a programming language, be a general educational tool for constructing learning environments in which other learning can occur? In other words, does learning Logo facilitate learning other subjects? The third question is, what is the effectiveness of Logo with various instructional methods and for various target populations. In addition, there are some concerns related to the social and motivational impacts of learning Logo, such as on self-esteem and motivation to learn. Recent research focused on exploring the constructive attributes of Logo for mathematics learning These studies are discussed further on in this chapter.

- Logo is supportive of learning other subject knowledge

Programming is assumed to require the use of fundamental concepts such as variables and recursive structures, which are important in mathematics and physics. These concepts are difficult to learn in conventional teaching and the use of variables and recursion in the functional context of programming makes them more easily comprehended (Papert, 1980; Nickerson, 1982). Studies have shown that, generally, Logo is supportive of other school learning and is useful for communicating difficult abstract concepts, such as in mathematics (Feurzig, Papert, Bloom, Grant & Solomon, 1989; Howe, Ross, Johnson, Plane & Inglis 1982; Howe, O'Shea & Plane, 1979; Kurland, Pea, Clement & Mawby, 1986; Sutherland, 1992; Statz, 1973) and in particular, geometry (Abelson & diSessa, 1980; Lehrer, Randle & Sancilio, 1989).

Logo is best known for its applications in mathematics, but it has become fairly widespread and its applications go beyond mathematics (Wenger, 1987). Studies also indicate that Logo can be used as tools to learn physics (Briskman, 1984; Dale, 1984), languages (Bouchard & Emirkanian, 1984; Bull, 1983), as well as logical reasoning (Gorman & Bourne, 1983).

- Can Logo programming develop certain cognitive abilities and can these abilities be transferred to other domains?

Programming is a complex activity which demands various cognitive abilities. It is hypothesized that in the process of programming these cognitive abilities will develop. However, the answers to this question differ with different implementing methods and school settings and hence are controversial. Some studies showed that learning to program can have positive effects on thinking and problem solving skills (Feurzig, Papert, Bloom, Grant & Solomon, 1989; Kynigos, 1992; Mayer, Dyck & Vilberg, 1986), and debugging skills (Howe, Ross, Johnson, Plane & Inglis 1982, Howe, O'shea & Plane, 1979; Statz, 1973). The debugging skills acquired in Logo programming can be transferred to nonprogramming domains (Klahr & Carver, 1988). Investigators also found that Logo had an important effect on creativity (Clements & Gullo, 1984; Reimer, 1985).

Other studies, however, found little evidence that current approaches to teaching programming bring students to the level of programming competence needed to develop cognitive ability and the kinds of systematic, analytic, and reflective thought that is characteristic of expert adult programmers. These studies did not support that learning to program can help children develop a model of computer functioning that would enable them to write useful programs (Kurland, Clement, Mawby & Pea, 1986; Pea & Kurland, 1984; Rampy, 1984). Kurland, Pea, Clement and Mawby (1986) found that students were doing so-called *brute-force paragraph programming* in which they decided on sets of desired screen effects and then lined up commands to cause the screen effects. In this process, students did not engage in problem decomposition or use the powerful features of the language to structure a solution to the programming problem. In addition, a study found that very few children had a correct understanding of concepts such as flow of control, conditionals, or recursion (Kurland & Pea, 1985). Children's spontaneous projects often did not involve the use of variables and children had to be initiated to it (Hillel, 1992; Sutherland, 1992). As can be expected, since students had not developed the programming competence and cognitive abilities in the first place, the studies found little evidence of transfer of cognitive skills to other domains.

- Logo can bring about positive effects on students' self-esteem, motivation, and attitudes towards learning

Most studies indicated that experience with Logo has positive effects on students' self-esteem, motivation, and attitudes toward learning. Especially, Lego-Logo is highly motivational to young learners of both genders (Papert, 1986). However, Roblyer, Castine and King (1988) stated that no conclusions could be drawn about the impact of Logo on students' image of themselves on the basis of evidence available.

- The effectiveness of Logo with various instructional methods

Learning by exploration is recommended by Papert as the best way to use the Logo environment (1980). Papert strongly suggests to help children learn how to develop and debug their own theories rather than to teach them theories adults consider correct (Papert, 1972a, 1972b). Papert claimed that, without the imposition of adult authority and adult ideas, children can come to an understanding of the nature of fundamental programming concepts such as recursion. Newman (1986) argued that this is not true for programming because computer programming is seldom mastered by young children.

From a problem solving perspective, Groen (1978) argued that:

> The best way to use the Logo environment is to give the learner the opportunity to indulge in *free problem-solving activities* (e.g., inventing computer programs that do interesting things). The child selects a project and is free to do anything he or she wishes to accomplish it, subject to quite explicit constraints imposed by the Logo environment. The goal is to improve the learner's ability to articulate the working of his or her own mind and particularly the interaction between him/herself and reality in the course of learning and thinking. The approach is to follow the model of the child as researcher and the teacher as research director (p. 56-57).

Results from empirical studies showed that the effects of Logo differed according to implementing methods and school settings. In contrast to other views, certain studies found that a structured teaching method is more effective than an unstructured, discovery-oriented method (Littlefield, Delclos, Bransford, Clayton & Franks, 1989). Among various methods, asking children to design instructional software with assistance from the teacher seemed to be an effective method for learning both Logo and fractions (Harel, 1988). Recent research tended to emphasize that teacher's role is critical in building the bridge between Logo and mathematical task activities (Gurtner, 1992) and providing problems and information relevant to the constraints on programming context (Sutherland, 1992). Therefore, the results support that children can learn efficiently by exploration in Logo environment when being directed or assisted by the teacher.

- The nature of the Logo exploratory learning environments

Groen indicated that computer programs are structures that coordinate other structures (1978). He explicitly articulates some of the properties of the microworld and programming in the following quote:

> First, a programming language...can provide an introduction to mathematical formalism that is better coordinated with the natural structures of the child. Second, the process of writing a computer program encourages thinking about how one would perform the actions that are being embodied in the program. Third, and most importantly, the pupil may invent a grossly incorrect or "buggy" theory about the microworld. Computer-based microworlds are naturally self-correcting... The nature of the errors may yield additional information. If the cause is nontrivial, the task of debugging or discovering the cause of the error may lead to major modifications in the theory (p. 371).

These concerns seem to be the topics of recent studies on Logo. Researchers (Edwards, 1992; Hoyles and Noss, 1992; Loethe, 1992; Kynigos, 1992) attempted to determine the extent to which Logo provides a mathematical environment and whether there are properties of the Logo environment that are inherently mathematical. They also attempted to sketch an understanding of how Logo operates as a medium for children to express their mathematical ideas. The conclusions were that, first, the mathematical nature of Logo programming allows children to express geometrical ideas in a "natural" way (Loethe, 1992, Kynigos, 1992, Edwards, 1992). Second, Logo offers a means for students to accept and use abstract symbols (Sutherland, 1992). Third, the most importantly, the microworld provides meaningful, interpretable feedback that the learners can use to refine their understanding of the structure of the new mathematical entities they encounter.

Misconceptions can be corrected by students themselves through a process of conceptual "debugging" (Edwards, 1992). Therefore, there is considerable evidence that Logo provides a computational environment in which mathematics can, at some level, take place and that it can provide access to otherwise unattainable mathematical ideas (Hoyles and Noss, 1989). The rationale was derived from the way in which the Logo environment can provide pupils with an opportunity to engage in mathematical problem posing and solving during which they develop control over their own learning, and the use of computational tools which can potentially structure, amplify, and reorganize thinking (Noss & Hoyles, 1992). However, Noss and Hoyles (1992) suggested that the idea that Logo provides an "all purpose learning environment" has raised a range of unrealistic expectations concerning the development of general problem-solving skills.

In summary, studies showed that Logo supports other school learning and that experience with Logo has positive effects on students' self-esteem, motivation, and attitudes toward learning. However, the data on whether cognitive abilities can be developed from experience with Logo and further be transferred to other domains is controversial. Moreover, the studies which did not find positive cognitive effects in learning to program also indicated that children did not progress very far in programming skills, or in depth of understanding program concepts. More recent studies have confirmed the constructive nature of Logo environment for mathematics learning.

Limitations of Previous Logo Studies

The limitations of the previous research are created by three major factors: the research questions asked, the research methods, and the subjects used.

First, the nature of the Logo learning environment was not studied from the perspective of learning mathematics until recently by Noss and Hoyles (1992) and other researchers (Edwards, 1992; Loethe, 1992; Kynigos, 1992). Noss and Hoyles concluded that Logo provides an exploratory environment that is inherently mathematical rather than an all-purpose learning environment. In order to examine whether the Logo environment can provide a general computational representation for learning, it is necessary to investigate the use of Logo for learning other knowledge rather than mathematics. This study uses Logo as the medium for student teachers to develop instructional programs for teaching.

Second, in past Logo research, researchers who conducted empirical studies usually used either extensive observations or pretests and posttests to measure the cognitive outcome from the interaction with Logo. However, these methods cannot account for what the child learned in the Logo environment, which is the way of establishing a correspondence between the concrete world and one of abstract representations (Groen, 1984) and intellectual structures (Papert, 1980). Groen (1984) further presumed that more extensive use of empirical methods in cognitive science might be of considerable value in research on Logo, because this use could result in the emergence of a body of research in which theory and data are closely linked.

Therefore, this study uses a cognitive method to examine the structures constructed in Logo environment.

Finally, mostly children were used as subjects in previous Logo studies rather than university students or adults. Because children's maturity levels, cognitive abilities, and intellectual experiences differ greatly from adults, it is not surprising to find that children had not developed the kinds of cognitive skills or abilities in Logo programming that are the characteristics of expert adult programmers. In order to determine whether learners can develop the kinds of cognitive skills or abilities in Logo programming that are the characteristics of expert adult programmers, this study will use university students whose maturity levels, cognitive abilities, and intellectual experiences are similar to those of professional programmers.

Children's programming is emphasized as a way of building intellectual structures; professional adult programming, however, has been extensively studied as a cognitive skill in the area of cognitive science. The findings from empirical studies of programming are presented below.

Empirical Studies on Programming Expertise

In order to understand the characteristics of instructional software, it is necessary to take into account the structure of the program, from a computational perspective. Research relevant to this aspect of the description comes from empirical studies on programming expertise. These studies have also investigated the cognitive processes in which programmers engage, the content and organization of programming knowledge, and related cognitive abilities. These findings provide a basis for understanding what is required from the designer in developing efficient software, including instructional software. Because the products and processes in programming are closely interrelated, they need to be studied in an integrated way. A second reason for reviewing the research on programming expertise is to understand the methods used to compare experts and novices in programming and to determine how these methods can be improved. The current methods used in studies of programming expertise for describing program structures are very few and incomplete.

Programming is a complex configuration of various activities oriented toward developing a product consisting of a series of instructions that direct a computer to accomplish some tasks (Pea & Kurland, 1984). Programming consists of such activities as understanding the problem to be solved, designing a solution, coding the solution using a programming language, comprehending the written program in order to debug, testing the program's correctness, and evaluating usability for target population. These activities require cognitive skills such as systematic planning, procedural, and conditional reasoning (Brooks, 1977; Jeffries, Turner, Polson & Atwood, 1981; Nickerson, 1982; Pea & Kurland, 1983; Pennington, 1982). Programming also demands knowledge of subject matter and knowledge of programming languages and computer architecture. In addition, knowledge of design strategies is also required (Adelson & Soloway, 1988; Pennington, 1987). Successful software design involves the coordination of the activities in which goals and operators interact with various skills and knowledge.

The basic issue addressed in programming expertise studies is similar to that asked in other areas. That is, what distinguishes outstanding individuals in a domain from less outstanding individuals in that domain, as well as from people in general (Ericsson & Smith, 1991). To capture the essence of programming expertise and the related abilities of programming, two types of tasks are often used: the representative programming task, such as recursive programming and tasks that measure a related function or ability, such as, recall of programming codes. Methods such as thinking-aloud, observing task performance, and explanation are often used to gain understanding of the cognitive processes and the strategies employed in performing some representative tasks, as well as the content and organization of knowledge the subjects utilized in their problem solving. The measure of related functions and abilities, such as memory tests are often used to make inferences regarding expertise.

Research on programming skills has focused on the programming processes which coordinate and display the various knowledge and skills by comparing experts and novices. Some findings from expertise studies are similar to findings in other domains, whereas others are different. For example, studies in other domains consistently show that experts use forward reasoning (see Groen & Patel, 1988, 1990), but studies in programming indicated both experts and novices use backward reasoning (Adelson, Soloway, 1985; Jeffries, Turner & Polson, 1981). Also, research in other domains showed that experts display better memory performance (deGroot, 1966; Chase & Ericsson, 1982), but the results in programming suggested that this is not always true (Adelson, 1984). Therefore, the findings from programming provide a unique perspective to look at expertise. The following section will present findings from programming expertise studies that focused on the programming processes and related cognitive factors.

Findings from Expert-Novice Studies in Programming

The studies of programming expertise have analyzed almost every aspect of the programmer's behaviour displayed in the processes of programming and related cognitive abilities using novice-experts comparisons Results from expert-novice studies have revealed certain characteristics of programming expertise. This section focuses on findings concerning process, representations, memory performance, validation of programs, and program structures.

1. Process

Decomposition of complex tasks into more manageable subtasks is essential to successful software design. Researchers consistently found that both novice and expert programmers use a top-down decomposition to reduce the complexity of tasks in programming design (e.g , Jeffries, Turner & Polson, 1981). That is, starting from a global statement of a problem, a programmer decomposes the initial problem into subproblems, then further into subproblems, until the problem is solved by implementation of programming code. As decomposing proceeds from the top to the bottom the abstract solution become more concrete, until the solution can be implemented in program codes (Adelson & Soloway, 1985; Jeffries, Turner & Polson, 1981).

The difference between novices and experts is that novice code the first part of a solution until the first part can be implemented in program codes, they then code the next part of the solution, and so on. This process is called depth-first decomposition. In contrast, experts use a top-down, breadth-first decomposition strategy. They develop the solutions for all elements at the same level equally and all information about the current state of the design is at the same level of abstraction so that all elements can interact with each other. Therefore, both novices and experts use top-down decomposition strategy, but novices decompose depth-first whereas experts decompose breadth-first (e.g., Jeffries et al., 1981) However, when expert programmers solve problems in an unfamiliar domain they create the partial solutions and combine them to form a full solution (Adelson & Soloway, 1985). This strategy is similar to the ones used by novices

- Expert programmers devote a great deal of effort to understanding a problem and its constraints before breaking it into subproblems.

In addition to strategies used in the decomposition process, expert and novice programmers also differ in other aspects of the decomposition programming process. Similar to the findings of research in other domains (Paige & Simon, 1966; deGroot, 1966), expert programmers are found to devote a great deal of effort in understanding a problem before attempting to break it into subproblems. They clarify constraints on the problem, derive their implications, explore potential interactions, and relate this information to real-world knowledge about the task. Novices, on the other hand, show little inclination to explore aspects of subproblems before proposing a solution. This has serious consequences for both the correctness and the efficiency of their design (Jeffries et al., 1981).

- Experts tend to decompose the problem based on known solutions, efficiency and aesthetics, whereas novice programmers do not show such a tendency.

Experts decompose the problem into manageable, minimal, and interacting parts in order to reach the point where the subproblems have known solutions. In contrast, novices are much less effective in their use of this iterative decomposition method. They seem to lack the more subtle aspects of the decomposition. In addition, experts state alternative solutions and select among them based on the hypothesized efficiency and aesthetics, whereas novices seldom consider more than one possible solution to any subproblem. Even in the few cases in which novices choose among alternatives, they base their decisions on programming convenience rather than on efficiency or aesthetics (Jeffries et al., 1981).

2. Representations

Research has shown that expert programmers have effective representations of programming knowledge at both the abstract and concrete levels (e.g., Adelson, 1985), whereas novice programmers only have concrete representation (Jeffries et al., 1981; Linn, 1985; Sheiderman & Mayer, 1979; Soloway, 1984). These studies indicated that expert programmers represent programming problems in terms of the general concepts, the underlying structures of broad classes of problems, the solution strategies which crosscut many types of problems, and routinized plans (Soloway, 1984) or templates (Linn, 1985). In addition, expert programmers' mental representations of programs are based on procedural (flow control) rather than functional (goal hierarchy) relations (Pennington, 1987).

In contrast, novice programmers have difficulties representing knowledge effectively. Even if novice programmers begin to develop an understanding of the programming language and write relatively sophisticated programs, they may still represent problems in terms of the surface codes, format, and syntactic properties of the language (Samurcay, 1985). These findings mirror the results of expert-novice comparisons in domains, such as physics (Chi, Feltovich, & Glaser, 1981), which found that experts represent problems according to abstract principles, whereas novices tend to rely on surface structures to organize their representation of problems.

- Experts have superior recognition abilities for identifying the class of relevant solutions and the conditions of applicability.

Experts have templates which include the critical features of the problem, relevant solutions and the conditions of the applicability. Expert programmers simply retrieved the appropriate template from memory and applied it when they solve problems in familiar domain. They are also able to retrieve a known solution in a novel context and adapt the solution to the particular context of a design problem (Adelson & Soloway, 1985; Jeffries et al., 1981)

Novice programmers operated on the partial template which could be retrieved from their memory or the texts (Anderson, Farrell & Sauers, 1984; Pirolli, 1986; Pirolli & Anderson, 1985). They showed no evidence of recognizing the applicability of information in a novel situation comparable to situations they had learned previously. In addition, the information they generated in the course of solving the problem was often not available when it was most needed, and when it was available, they did not attempt to alter the previous solution to the current problem (Jeffries et al , 1981).

- Expert and novice programmers interpret the same concepts differently.

Novices have inadequate understanding of many of the basic concepts of computer science. The same technical computer science terms do not have the same meaning for the novices as they do for experts (Jeffries, et al., 1981). Studies have consistently indicated that expert and novice programmers have different understandings of recursion[1] (Jeffries, 1982; Kahney, 1982; Kahney & Eisenstadt, 1982; Kurland & Pea, 1985).

- Expert programmers have a well developed design schema of programming knowledge.

According to Jeffries et al. (1981), a *design schema* is a template for developing programming structures that is independent of its content. This has impact on almost on every facet of the programmer's behaviour in software design. It directs the programmer's behaviour in an efficient way. A programming schema is complex and it is developed in stages as a result of experience with software design. The mature design schema facilitates the refinement of understanding, retrieval of known solutions, generation of alternatives, and critical analysis of solution components. Experts are assumed to possess such a design schema, whereas novices programmers

[1]Recursion refers to a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminatied ones. This is the model of the recursive process that experts have, whereas novices have a "looping" model of recursion. That is, novices view a recursive procedure as a single object instead of a serial of new instantiations.

have a less developed design schema. This explains why their behaviour is less efficient.

Another idea related to the design schema is the *plan schema* proposed by Rist (1986, 1989) and Spohrer, Soloway, and Pope (1985). A plan is a set of stereotyped sequences of actions that expert programmers know and that can be adapted to the current situation. Some researchers regard programming plans as the most important characteristic of advanced programming skills (Adelson, 1981; Bonar & Soloway, 1985; Dalbey, Tourniaire & Linn, 1986; Detiennne & Soloway, 1989; Kurland, Mawby & Cahir, 1984; Shneiderman, 1976; Soloway, Adelson & Ehrlich, 1988; Spohrer, Soloway & Pope, 1985; Rist, 1986). It is assumed that experts do not only develop a greater range of these plans than novices, but also know the "rules of programming discourse" that govern the valid application of plans in particular circumstances (Soloway & Ehrlich, 1984).

3. Memory performance

Experts possess chunks that represent functional units in their respective domains, whereas novices do not possess such chunks as demonstrated in performance on recall tasks (Adelson, 1981; McKeithen, Reitman, Rueter & Hirtle, 1981; Shneiderman, 1980b). Results from Shneiderman's studies further showed that experts were able to chunk lines of code together into meaningful configurations which allowed them to achieve better memory performance, whereas less experienced users were less able to form such chunks so they recalled fewer stimuli.

Adelson (1984) reported findings that contradict the notion that experts have superior memory performance. She indicated that novices had better memory for the details of code than did experts. The explanation appeared to be that experts focused more on the overall goal structure of the programming task than on the actual code because it is easier for them to solve a programming task than to memorize a detailed solution whereas it was the reverse for novices. Therefore, expert programmers do not always display superior memory performance.

4. Validation of programs

Novices and experts differ in their skill in testing designs and programs. Experts have well-developed knowledge of debugging strategies associated with their programming templates and they are good at designing tests for revealing potential problems. In contrast, novices often test only the obvious or usual forms of input and may systematically fail to test all of the codes (Kurland et al., 1986; Mandinach & Linn, 1989).

5. Program structures

One of the differences between the study of cognitive skills in computer programming and those in most domains is that the tasks in programming often involve constructing products. It is reasonable to assume that, besides the process of programming, expert programmers also differ from novices in the ways they design the final products of programming. Previous studies of programming skills did find differences between experts and novices in terms of program structures, in particular in ways to construct recursion.

- Expert programmers construct modular programs whereas novice programmers construct linear ones.

Effective programs require modular structures so that large systems can be divided "naturally" into coherent parts that can be separately developed and maintained (Abelson, Sussman & Sussman, 1985) Most studies on learning to program distinguish between a linear program and a modular program. A linear program empha ›izes the generation of effects without any consideration or understanding of the inner structure of the code (Soloway, 1984). A modular program, however, is considered as emphasizing elegant and efficient programming, and is accompanied by a higher-level understanding of programming (Carver, 1987; Kurland, Clenment, Mawby & Pea, 1987). The cognitive demands for modular programming are different from those for linear programming. Researchers indicated that expert programmers tend to construct modular programs while novices tend to construct linear ones.

- Experts and novices differ in the way they construct recursion.

An essential aspect of recursive programming is related to how one exits the recursive cycle. Novice programmers often construct a cycle which permits them to exit from the middle of the cycle, whereas experts construct exit points from the top or the bottom a cycle, a strategy that is believed to be superior (Soloway, Bonar & Erihlich, 1983).

In summary, the results from studies of programming expertise showed that, first, both experts and novices use top-down decomposition strategy. The differences between experts and novices in decomposition are that experts devote more effort to analyze the problem, and they decompose the problem based on known solutions, efficiency, and aesthetics using breadth-first strategy. In contrast, novices use depth-first strategy to solve problems and they do not show a tendency to consider efficiency or aesthetics. Expert programmers select a solution among alternatives based on the hypothesized efficiency and aesthetics whereas novice programmers select a solution based only on convenience.

Second, expert programmers have well developed representations of programming knowledge whereas novice programmers only have low-level representations of programming knowledge. The well developed knowledge representation is often called as a design schema. This schema impacts almost every aspect of the programmer's behaviour in software design and it directs the programmer's behaviour in an efficient way. Expert programmers have such a design schema so they are able to retrieve and modify a known solution to fit a current problem whereas novices do not have such a design schema so they are unable either to retrieve the solution or to adapt it to a novel problem.

The results do suggest experts demonstrate an enhanced ability to chunk meaningful stimuli but do not necessarily remember more details of code than novices. Expert programmers also differ from novice programmers in the ways they construct the program structures.

The Limitation of Previous Studies on Programming Expertise

Previous studies of programming have provided a great deal of understanding of the content and organization of programming knowledge, the general strategies for solving problems, and the related cognitive abilities.

However, a limitation of the these studies is that they only focused on the programming process and did not examine the important features of the products in order to see if and how they distinguish expert from novice programmers.

Programming is a complex configuration of activities oriented toward developing a product. Different types of knowledge, skills and abilities interact in a very intricate way in the programming process and it is very difficult to assess the roles played by each factor in achieving outstanding performance. For example, it is not convincing to say that the designer who uses top-down and breadth-first strategies will definitely produce a better program than the one who uses top-down and depth-first strategies. In fact, just like an expert runner is distinguished from novice runner by how fast he or she arrives the goal, an expert programmer may be distinguished from novice programmers by how well he or she can produce a program. Therefore, expertise or outstanding achievements in programming may be identified by the products that a programmer produced However, in many domains in which experts produce complex products as texts, it is difficulty to analyze such products in order to identify the measurable aspects capturing the superior quality of the product. Therefore, researchers focused on systematic characteristics of the cognitive process in order to differentiate experts from novices (Ericsson & Smith, 1991).

Similarly, the methods available in current expertise research are unable to identify the measurable aspects capturing the expertise embedded in the final products of programming. In addition, the previous studies of programming expertise have not yet covered programming expertise in designing instructional software. Therefore, to develop a methodology to characterize instructional software will have considerable value.

The identification of programming expertise from the programming product does not imply that it can replace the studies of the cognitive processes of programming Instead, the emphasis is based on the assumption that the different knowledge, skills, and abilities that the programmers possess will be displayed not only in the processes, but also in the final products of programming. The better understanding of programming expertise can be achieved only when the components of the superior performance displayed in both the processes and products can be described and identified.

Summary of the Chapter

This chapter presented a review of literature to develop the rationale underlying this study. The review considered perspectives of instructional software evaluation, human-computer interaction, the Logo approach, and the studies of programming expertise. This section presents the key points for developing a methodology to identify the cognitive, pedagogical, and computational characteristics of instructional programs produced by student teachers using Logo.

First, the previous evaluation of instructional software may indicate only whether instructional software is efficient. However, it does not identify the cognitive and pedagogical characteristics or give any other information regarding the strengths and weaknesses of the instructional software that determine the effectiveness of instructional software. Therefore, it is difficult to distinguish one program from another and further to compare them. Consequently, the results from these evaluations do not provide sufficient guidelines for developing efficient instructional software. This study suggests that previous evaluation methods should be complemented by the characterization of the important features of instructional software that determine effectiveness.

Second, the process of knowledge communication between a user and a computer is studied as human-computer interface, without any consideration of the instructional properties of the software. Therefore, the concepts and approaches in the study of human-computer interface have to be modified in order to effectively study instructional software interface.

Third, the previous studies on Logo did not explore the nature of Logo environment until recently. Recent studies have confirmed that Logo provides a computational environment that is inherently mathematical. For example, the mathematical nature of Logo programming allows children to express geometrical ideas in a "natural" way. However, the computational application of Logo is limited to learning mathematics. The present study will explore whether the suitably constructed, computational nature of Logo environment can be used for learning other knowledge that requires computational representations, such as learning how to teach in a computer-based medium.

Finally, previous studies of programming expertise have provided a great deal of understanding of programming as a problem solving activity, however, they did not, or were unable to account for the different programming expertise embedded in the final products of programming. Furthermore, the previous studies of programming have not yet considered the expertise involved in designing instructional software interface. Therefore, the development of a methodology to identify the cognitive, pedagogical, and computational characteristics of instructional software produced by student teachers using Logo may prove to have significant research value in several related areas of investigation.

Chapter 3

METHODOLOGY

The methods for characterizing computer-based learning environments and program structures are developed in the context of student teachers using Logo to produce instructional programs. This chapter describes the framework and methods for characterizing the learning environments and program structures.

To effectively characterize instructional software, it is necessary to distinguish between the learning environment and the program structures, which are two different aspects of a program. A primary goal of this research is to characterize the learning environments constructed by student teachers in developing instructional programs using Logo. The characteristics of the learning environments constructed by student teachers are assessed in terms of the usability and constraints in meeting the user's cognitive needs during the learning process.

As mentioned previously, LogoWriter™ is a Logo-based application incorporating unique program structures and screen layouts. This study also investigates how student teachers structure pages and procedures, as well as use program properties.

Subjects

Subjects were 18 university students (14 females and 4 males), between 23 and 35 years of age, participating in a one semester, undergraduate, introductory Logo course. All subjects were majoring in the elementary and secondary teaching programs in the Faculty of Education at McGill University. None had any previous experience with computers.

Materials

Computer Hardware

A laboratory equipped with 24 Apple Macintosh LC microcomputers and colour monitors, four Apple ImageWriter printers, and one Apple LaserWriter laser printer was used during the classroom sessions for completing assignments and projects. All microcomputers were connected to a local network server, and all printers were connected to the microcomputers by AppleTalk links so that student teachers could print from any of the computers.

Student teachers had free access to the laboratory for completing assignments and projects during the period of the course.

Software

The software used in this course was LogoWriter™ produced by Logo Computer Systems Inc., for Apple Macintosh computer systems. Four features distinguish LogoWriter™ from Logo. The first feature is that LogoWriter™ has the capacity to execute more than one page[2] easily in a program, with or without the user's interactions. The second feature is that, using a mouse, the user can drag the turtle around and use it as a pen to draw pictures on the front page

[2] A page in LogoWriter™ has a front side and a flip side. The front side is divided into two parts: front page and command center. The front page can display the screen effects of the procedures, whereas the command center can be used to type the commands. The flip side is used to write procedures or a program.

(screen). The third feature is that there is a word processor in LogoWriter™. The last feature is that LogoWriter™ has a special screen layout which divides the screen into a front page and a command center so the user can see the procedures and their effects at the same time.

Readings

Student teachers were required to use a reference book (Le Gallais, Shapiro & van Gelder, 1988). This book explains some of the basic concepts and skills used in Logowriter™, such as drawing graphics, writing procedures, using variables, recursion, and structuring procedures. Each chapter provides an explanation of specific concepts and primitives followed by a series of practice examples and suggested activities. The chapters also discuss common problems encountered by the learners and present suggestions for teaching Handouts on Macintosh computers and Logo programming were distributed to the students at the beginning of some sessions. In addition, Papert's *Mindstorms* (1980) was recommended reading for the student teachers.

Other Materials

Student teachers also used paper and pencils in the class.

Data Source

The data used in the present study consisted of the final projects completed by the student teachers at the end of the semester as part of the course requirements. In order to situate these projects, it is necessary to describe briefly the overall structure of the course.

The teaching method used in this course can be characterized as "project-driven" learning in which students were required to produce a sequence of working samples, two projects, and a term paper These requirements are described in more detail below. There were 24 semi-weekly classroom sessions of three-hour duration during which the students worked on the exercises or on their projects individually and at their own pace. Meanwhile, the instructor and an assistant observed the students' learning and provided help when it was needed In addition, explanations about Macintosh microcomputers and Logo programming were given in 15 minute sessions at the beginning of approximately 10 of the sessions

Working Samples and Sharing

Students were required to replicate the exercises in the reference book or expand creatively on these exercises. They were instructed to submit these working samples to their individual computer "folders" on the server so that they could look at each other's work.

Students worked individually during this phase, but they could discuss and help each other in class. It was clearly indicated that the working samples would not be graded but that they had to be handed in to complete the course.

Midterm Project

After six weeks, students were required to complete a midterm project. The objective was to show how creative students could be within the Logo environment and grades were based on the extent to which students deviated from the book. After the midterm projects were graded, the ten best projects were put in a display folder in the server so that all students could look at them.

Final Project and Short Paper

Toward the end of the term, all students were asked to plan and design a final project individually or in groups. The students were required to use Logo procedures to develop a program with which a user could interact in an interesting way. In addition, students were encouraged to use a modular programming style to break down a problem into small units as explained in the reference book. The grade was based on the interest and effectiveness of the instructional strategy developed.

At the same time, students were asked to write a short paper to indicate how they would use Logo for instructional purposes. It was explained that the paper should be an idea paper rather than a reading assignment or optionally, students could combine the term paper with the project. Thus, the paper would be a description and justification of the final project.

The thirteen projects were submitted at the end of the term included nine individual projects and four group projects. These projects constituted the data for this research.

Data Analysis

The present study distinguishes between learning environments and program structures. A *learning environment* refers to the display (e.g., text, graphics, animation, and speech) of instructional software and the user-computer interactions the software promotes, which is characterized in terms of the types of knowledge presented, the pedagogical strategies used to present this knowledge, and the forms and functions of the interactions. *Program structures* refer to the computational construction of the program units, such as pages or procedures, as well as other programming properties.

The hierarchical organization of a learning environment is described graphically in Figure 1 A learning environment in this study is characterized by dividing it into *episodes* which are composed of sequences of *views*, with task descriptions at each level An episode refers to a lesson or a set of exercises developed by student teachers for specific instructional purposes, whereas a view refers to the display on a screen and the interactions the screen display elicits. A view is changed when there is a significant effect on the screen.

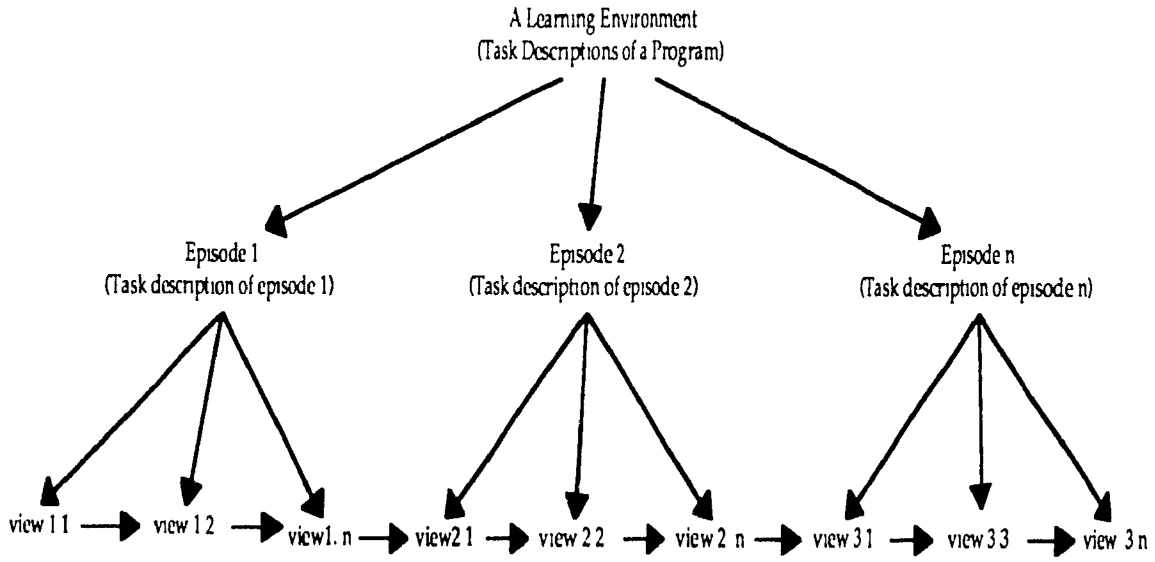

A Learning Environment
(Task Descriptions of a Program)
Episode 1
(Task description of episode 1)
Episode 2
(Task description of episode 2)
Episode n
(Task description of episode n)
view 1 1
view 1 2
view1. n
view2 1
view 2 2
view 2 n
view 3 1
view 3 3
view 3 n

Figure 1. The hierarchical organization of a learning environment.

Each view consists of *view space* and *command space* . The view space refers to the static attributes of a view — the *types of knowledge* presented and the *pedagogical strategies* used to present the knowledge. The command space refers to the dynamic attributes of a view which includes *automatic operators* and *manual operators* (see Figure 2). Automatic operators refer to the cases in which a program does not require any input from the user to execute a procedure whereas manual operators necessitate user input (e.g., user-computer interactions). The manual operators are characterized in terms of the forms and functions.

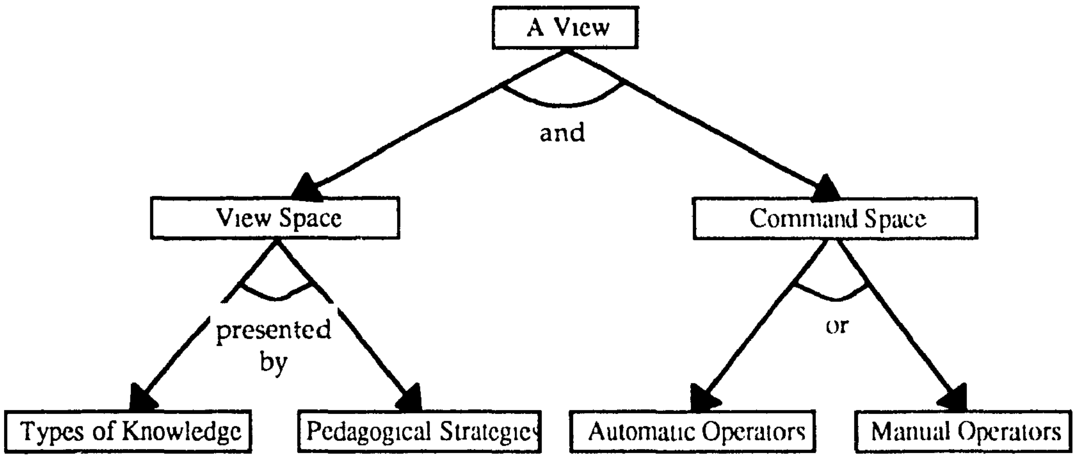


Figure 2. The view space and the command space in a view.

Program structures are characterized in terms of *modularity* that refers to *linear* or *modular structures.* A linear structure refers to a unit (it can be a page or a procedure) which employs subunits in a linear sequence. A modular structure refers to a unit that can be divided "naturally" into coherent parts that can be developed and maintained separately.

In characterizing program structures, some specific *program properties* are considered, such as reusable procedures, conditional statements, variables, and recursion. These properties are described in detail in a subsequent section.

Characterization of the Learning Environments

A learning environment is hierarchically decomposed into views and each conceptual unit (e.g., a statement, a question, or a configuration) and interactions associated with a particular view is characterized with respect to the types of knowledge presented, pedagogical strategies used to present this knowledge, and the forms and functions of the interactions.

Types of knowledge

In order to have a clear picture of the types of knowledge presented in a learning environment, the knowledge is categorized as domain knowledge, operating knowledge, affective knowledge, or implementation knowledge.

Instructional software is always used for specific instructional purposes, such as for teaching art or mathematics. Such knowledge that a program is designed to teach is referred to as *domain knowledge*. Successful execution of a program by the user requires knowledge of the features of the program (i.e., what a program can do), as well as knowledge of how to manipulate the program (e.g., how to retrieve a page). This type of knowledge is referred to as *operating knowledge*. Both domain knowledge and operating knowledge are sub-categorized as either declarative knowledge (describing facts, events, concepts, principles or relationships) or procedural knowledge (explaining actions or conditions under which the actions can be taken). In addition, studies on learning have suggested several other subcategorical knowledge be important in learning situations. These studies suggested that, first, learning and strategy acquisition occurs at impasses (Siegler, 1989; VanLehn, 1988, 1990). Second, learners have certain misconceptions and they exist in all kinds of learning, such as physics (diSessa, 1988), chemistry (Albert, 1978; Erickson, 1979). In addition, it is indicated that problem-solving strategies and learning strategies can be taught, and there are many methods for doing so (Collins, Brown & Newman, 1989; Mason, Burton & Stacey, 1982; Schoenfeld, 1985)

Based on these research findings, it is reasonable to assume that efficient teaching programs might present so called "bug problems" that the learners often make mistakes on, indicate misconceptions and the origins of the misconceptions, as well as provide various strategies for problem solving or learning, such as heuristic strategies, control strategies, and learning strategies (Collins, Brown & Newman, 1989). Therefore, the characterization of the learning environments should include these sub-categorical knowledge This is particularly crucial for describing domain knowledge.

Because the program is manipulated by human beings rather than by machines, the program might present the knowledge that has the social and affective impacts on the users. All knowledge related to emotion, motivation, or self-esteem is referred to as *affective knowledge* In addition, a program may indicate what the program is designed for, who can use it, and how to use it. This type of knowledge is referred to as *knowledge for implementation* about content area, target populations and methods.

This study concerns the global nature of the knowledge presented in the learning environments and the pedagogical strategies used to present the knowledge. Therefore, this study is not preoccupied with the questions of whether the complexity of the domain knowledge is appropriate to the characteristics of its target population, or whether the content of domain knowledge is organized logically and systematically, or whether the affective knowledge has positive effects on retaining the user's motivation and self-esteem.

Pedagogical strategies

Conventional instruction usually involves pedagogical strategies such as *setting goals*, providing *instructions, explanations, demonstrations*, and *presenting tasks* as well as *asking questions*. *Evaluation and feedback* is also an important pedagogical strategy. These categories are used to determine the ways that student teachers present various types of knowledge in the learning environments. However, in the process of interacting with a computer, a user can only give input when a corresponding working space is provided by the program. Therefore, pedagogical strategies for characterizing computer-based learning environments should include the *provision of working spaces*. Brief definitions of seven pedagogical strategies used in this study are presented below.

1. *Setting goals*. An instructional designer informs a learner of the new knowledge or skills he or she is expected to acquire when a program or a learning episode (a lesson or a set of exercises) is finished. For example, a stated goal can be to teach children geometry. Since human activities are goal-oriented, the knowledge of goals or objectives can help a learner to organize and direct his or her behaviour effectively. Therefore, the goals or objectives should be specified at the beginning of a program or an episode.

2. *Instructions*. This refers to the uninterrupted presentation of any type of knowledge. For example, the instruction can be that a rectangle requires two inputs, length and width.

3. *Explanations*. This refers to any type of knowledge which is explicitly provided to the user in anticipation of potential sources of confusion. For example, following a demonstration, the program informs the user on how to

generate the procedures used to perform the demonstrations.

4. *Demonstrations.* Demonstrations are the processes by which a program shows a user how to perform a particular task by illustration. For example, a program can exhibit the screen effects after displaying a set of procedures

5. *Presenting tasks.* The tasks that users are demanded to perform are presented through text or graphics. For example, a student is directed to find points in a grid.

6. *Asking questions.* The users are presented questions and they have to give specific answers to these questions. For example, a user is asked: "Which the correct answer? "

7. *Providing working spaces.* After a task is presented, a user is provided the space to work on the screen. An example would be after the program asks a question, the program waits for an input from the user.

8. *Evaluation and feedback.* A user's performance is evaluated and the feedback is provided accordingly. For instance, a user may be told "Very good! You got the right answer! ".

In characterizing learning environments, the concern is not only with what types of knowledge are presented to the user and how they are presented, but also what kind of interaction the user has with the program. In order to address the latter, the dynamic attributes of a view must be considered.

Interactions

As mentioned above, the command space includes the operators required to change a view or produce any effect on the screen. These operators are categorized as automatic operators or manual operators

Automatic operators. Using automatic operators, a program does not require any input from the user to execute the program. There are several types of automatic operators in Logo. One type is where the program uses the CT or CG primitives in its procedures to clear the text or graphics from the screen without any input. Another type of automatic operator is the scrolling effects that are produced when a program uses the PRINT or TYPE primitive to print more information than can be displayed on a screen. The last automatic operator is when a program uses GETPAGE in a procedure so that the execution of retrieving a page changes the view

Manual operators. In most cases, a program requires input from the user to execute the program Such inputs are called manual operators and they are grouped into seven categories in this study:

1. Pressing a letter or a number;
2 Pressing <enter> when a procedure appears on the screen;
3. Typing a command;
4. Typing a command and a variable;
5. Using primitives;
6. Typing a word in response to a question;
7. Typing a sentence.

These manual operators can accomplish several functions. These functions are listed in the below:

1. To chose a type of tasks or activities (e.g., pressing a key to chose a type of activities: "+" for addition, "-" for subtraction, "x" for multiplication, and "÷" for division).

2. To chose task complexity (e.g., pressing a key to chose a level of multiplication: "1" for one-digit problems, "2" for two-digit problems, and "3" for bug problems).

3. To answer questions (e.g., Question: how many provinces in Canada?

Answer: ten).

4. To select answers for multiple choice questions (e.g. e.g , Question: how many provinces in Canada? a) 5, b) 10, and c) 9. Answer: b).

5. To perform tasks (e.g., using the commands provided by the program to find the points on a grid).

6. To chose various assistance (e.g., receiving a correct answer; receiving an explanation of the origins of a mistake; receiving a suggested strategy to solve the current problem).

7. To operate the system (e.g , pressing the enter to continue).

In order to characterize the types of knowledge presented, the pedagogical strategies used to present them, and the forms and functions of interactions, as well as the relationship among them, a framework was developed. This framework is described in the following section.

Procedures for characterizing the learning environments

Figure 3 shows the framework developed in this study for characterizing the learning environments. This framework also consists of view space and command space. View spaces include the different types of knowledge a program provides and the pedagogical strategies used to present the knowledge. Command spaces include the automatic and manual operators required by the computer system. Notice that there are three dimensions in this framework and each dimension is coded by two or three-character codings. Therefore, conceptual units composed of the learning environments are described by seven-character codings.

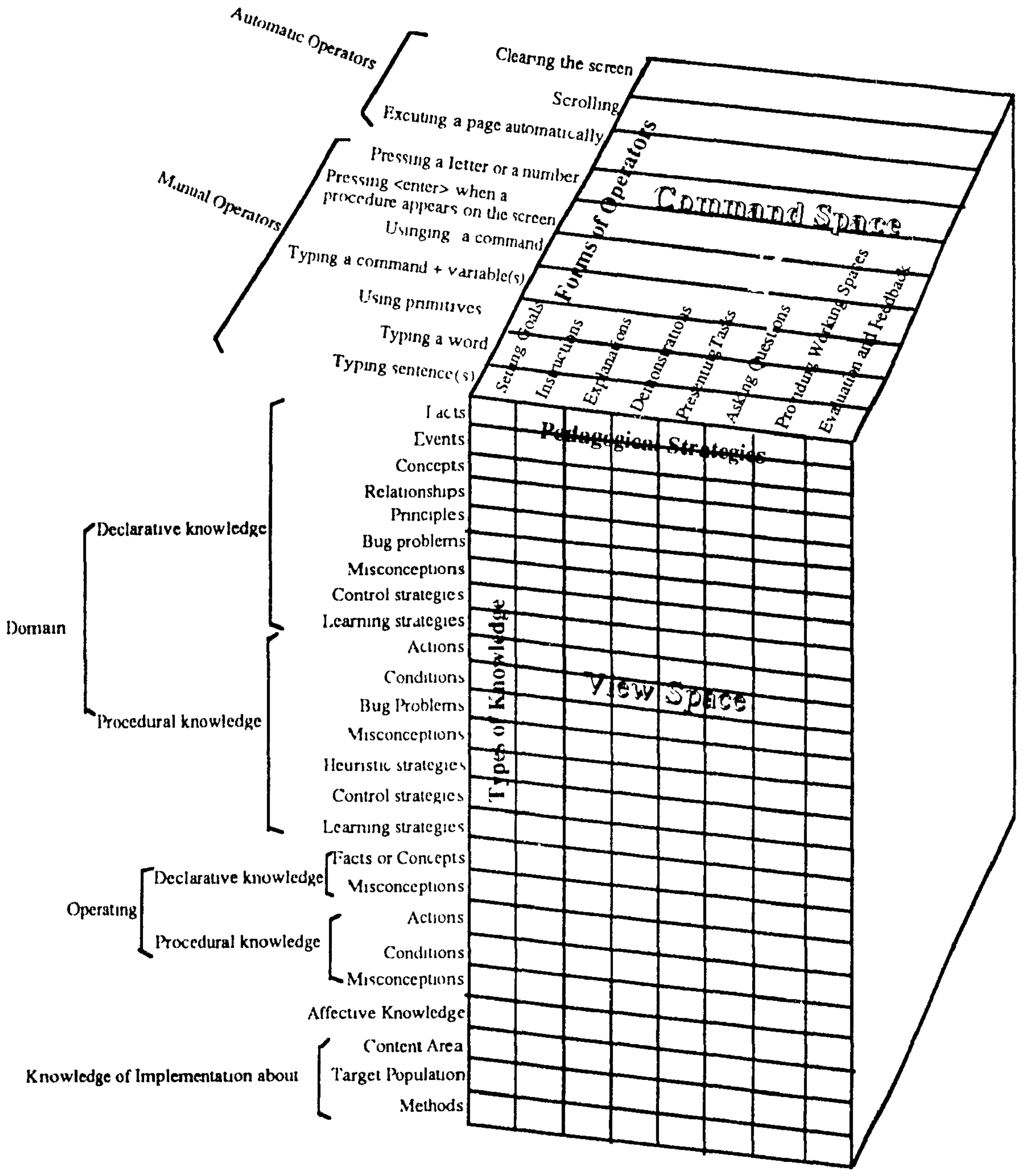


Figure 3. A framework for characterizing the learning environments.

The rows on the left of the view space include four major types of knowledge, which include subcategories of knowledge. The different types of knowledge are often represented by one to three digits, which are placed at the beginning of the seven-character codings. To the right of view space, the columns include eight pedagogical strategies which are represented by uppercase letters, and their necessary sequences are indicated by lowercase letters such as a, b, and c and so on These two letters are located in the middle of the codings. The dimension at the top of the view space is the command space which represents the operators designed in a view. The forms and the functions of the operators are represented by the last two-digit codings. The format of the codings for each conceptual unit is:

digit {(digit) digit} LETTER {(letter)} {digit (digit)}

Note that the letter or digit in braces indicating the subcategories of knowledge, the necessary sequence for pedagogical strategies, and the operators required for execution, respectively are optional. For example, presenting a task for taking the action of operating the system is coded as 2(2)1En, and providing working space and requiring the user to press a letter to operate the system is coded 2(2)1 Gn 4(7), while "n" indicating the sequence of pedagogical strategies in a program.

All final projects produced by student teachers were executed using the cognitive walkthrough method developed by Polson, Lewis, Rieman and Wharton (1991). As mentioned previously, the learning environment was divided into episodes, and these episodes often consist of sequences of views. Each conceptual unit in a view was examined and coded by the framework developed in this study which describes the types of knowledge a view presents, the pedagogical strategies used to present the knowledge, and the forms and functions of interactions. During the walkthrough process, similar to what

Polson et al. (1991) did, the reviewer stopped at each action and considered the strengths and weakness of the environment in terms of the effects on the typical user, and diagnosed whether a user would succeed or fail in the exploratory learning processes[3] Some descriptions and comments were made when each view was examined. The coding for each conceptual unit and each operator was recorded The codings for each project were summarized in terms of the types of knowledge, pedagogical strategies, and the forms and functions of the operators. The overall data for all projects were then analyzed to attain a global picture of the learning environments constructed by all student teachers. Further, the data in each project were compared in order to identify the characteristics of each project.

Characterization of Program Structures

In LogoWriter™, the ways of structuring the pages and procedures are very flexible and they may have an impact on the interactions of a program. Therefore, it is necessary to take the interaction issue into consideration when characterizing the program structures developed with LogoWriter™. In this study, the program structures include page structures, procedure structures, and other program properties.

The program structures are characterized in terms of single-level, linear, modular or fragmented structures. Program properties such as reusable procedures, conditional statements, variables, and recursion are also considered.

[3] The exploratory learning here refers to the process by which the first-time user can learn system operation using cues provided by the system and the novice learner can learn subject matter knowledge using the supports provided by the system, rather than receiving instruction or coaching from the teacher. This process can be called as guided exploratory learning, whereas the term "exploratory learning" used in Logo can be called as open exploratory learning in which learners can construct or invent products.

In addition, the operators used to link pages and procedures were examined

The program structures

Traditionally, program structures are categorized as either linear or modular. However, there might be single-level and fragmented structures in the programs produced by student teachers in LogoWriter™ because the easy-retrieval feature of LogoWriter™ enables student teachers to design a slide-like program (where a program consists of several pages and all pages are executed automatically one after another) which does not necessarily involve either linear or modular structures. Therefore, the program structures refer to single-level, linear, modular, and fragmented ones.

As mentioned earlier, *a linear structure* refers to a unit which employs subunits in a linear sequence and *a modular structure* refers to a unit that can be divided "naturally" into coherent parts that can each be separately developed and maintained. *A single-level structure,* of course, has only a one level procedure. What distinguishes a single-level structure from *a fragmented structure* is that a single-level structure explicitly indicates how it should be used, while a fragmented structure does not.

Program properties in this study refer to reusable procedures, conditional statements, variables and recursion. They are listed below:

1. *A reusable procedure* refers to the procedure that is used as a subprocedure by more than one superprocedure.

2. *A conditional statement* refers to a procedure that consists of a conditional evaluation and is executed if a condition is met.

3. *A variable* has a name and a value. Using variables, it is possible for a procedure to operate on different data each time it is invoked, but the pattern of

what the procedure does with the data remains constant.

4. *A recursion* is a procedure which uses itself as a subprocedure.

The operators used to link pages and procedures

When more than one page is designed in a program or more than one procedure is designed on a page, the execution requires operators. The categories of operators used in characterizing program structures are the same as those used in characterizing the learning environments. That is, the operators are also categorized as automatic and manual operators. Automatic operators refer to the cases in which all pages are retrieved by GETPAGE or GETTOOLS primitives in a (STARTUP) procedure on the first page, or all procedures are executed in one procedure. Manual operators refer to the cases in which a user's input are required in linking the pages or the procedures. Because interactions depend on the manual operators rather than automatic operators, the characterization of the program structures is only on the forms of manual operators and their functions.

Procedures for characterizing program structures

In order to identify the program structures, all pages and procedures in each project were drawn as diagrams using the symbols shown in Figure 4. The page structures, procedure structures, and the program properties used by each project were then summarized. To find whether the ways that the student teachers structure the pages and procedures have an impact on the interactions in the learning environments, the distribution of the manual operators is indicated as the manual operators between pages, between procedures, and within procedures. The functions for the manual operators located in different places are compared

In order to explore if there is any relationship between the pedagogical strategies and the program structures, the projects which used appropriate pedagogical strategies are separated from those which did not These two kinds of projects are further compared with the characteristics of the program structures.

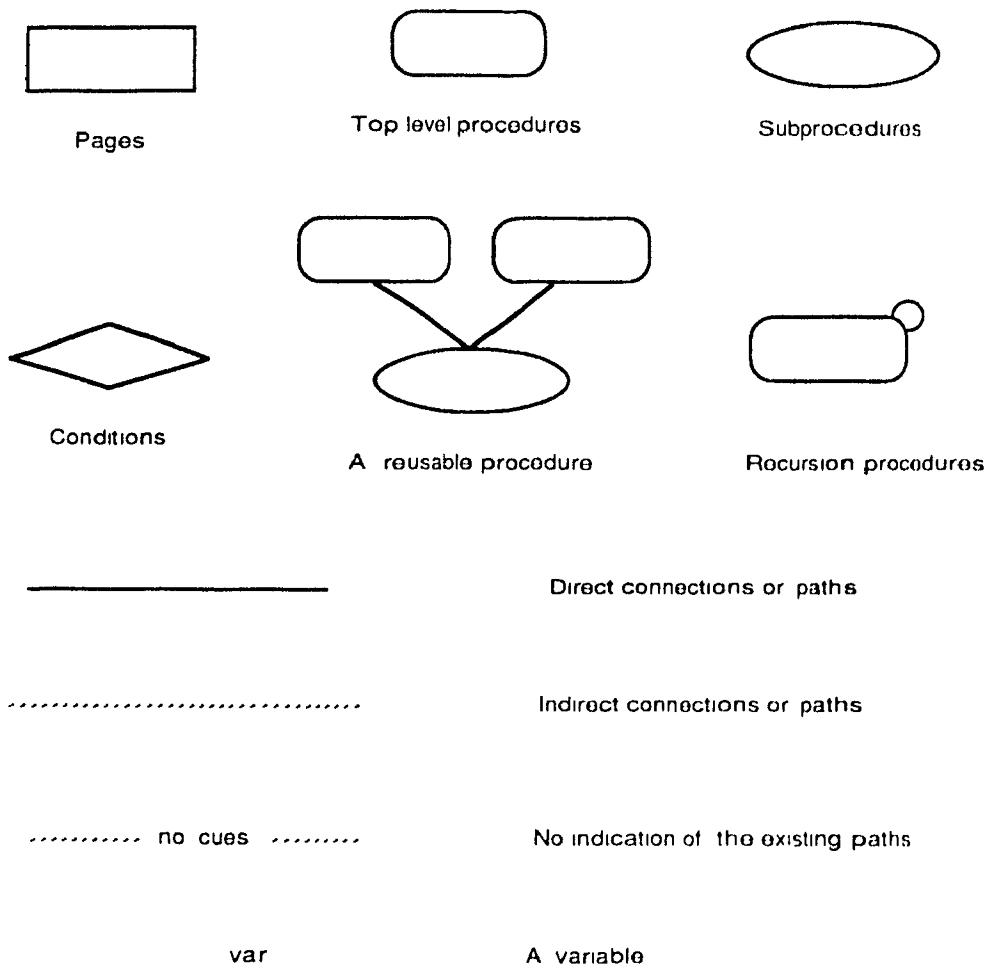


Figure 4. Symbols used in the diagrams.

In summary, the methodology in this study consists of characterization of the learning environments and program structures. A learning environment was broken down hierarchically into episodes and these were further broken down into views. Each conceptual unit in a view was then examined and coded by the framework developed in this study for characterizing the types of knowledge, the pedagogical strategies used to present this knowledge, and the interactions elicited in a view During the walkthrough process, the reviewers diagnosed whether a typical user would succeed or fail in exploratory learning processes, based on the strengths and weaknesses of the learning environment. The program structures, however, were characterized by single-level, fragmented, linear, and modular structures, and the operators used to link the structures, as well as programming properties. Finally, the characteristics of program structures were compared with the appropriateness of the pedagogical strategies used in the projects.

Chapter 4

RESULTS AND DISCUSSION

The results from the characterization of the learning environments and program structures are presented and discussed in three major sections. First, this chapter presents the results from encoding the learning environments in order to identify the characteristics of the learning environments constructed by student teachers. The strengths and weaknesses of the learning environments were assessed in the walkthrough processes. Second, this chapter presents the results from characterizing the program structures of LogoWriter™ and discusses their attributes. Finally, this chapter examines the relationship between the characteristics of the program structures and the learning environments (e.g., pedagogical strategies).

Characteristics of the Learning Environments

There are several major issues in characterizing a learning environment provided by instructional software. The first concerns the types of knowledge presented to the user and the consequences of lack of an important type of knowledge, such as the knowledge required to operate the system. The second issue is how the knowledge is presented to the user That is, what pedagogical strategies does a designer use to convey the knowledge to the user, and whether the pedagogical strategies support learning domain knowledge and system operation. The third issue is whether the input required from the user facilitate the user's learning and whether the user has the freedom to choose activities and task complexities, and to seek various assistance according to his or her needs.

In order to address these issues, first, the frequencies of the codings representing the different types of knowledge and pedagogical strategies were categorized and summarized. Second, the tasks, questions, and working spaces, as well as evaluation and feedback were matched for each action (e.g., each manual operator); their sequence and appearance for each action was examined. The frequencies of tasks, questions, and working spaces, as well as evaluation and feedback for all actions were presented. Third, the different types of manual operators and their functions were indicated. Finally, the overall results were presented in tables identifying the different types of knowledge, pedagogical strategies used to present this knowledge, as well as the forms and the functions of interactions. These data in each project were also presented and compared.

Knowledge Presented to the Users

Table 1 shows the frequencies of the codings representing the different types of knowledge and the pedagogical strategies used to present such knowledge in the learning environments constructed by student teachers. The rows indicate four major categories of knowledge presented, whereas columns indicate eight pedagogical strategies used to present the knowledge. Data from this table show that domain knowledge represented the primary knowledge (73%) and that operating knowledge was the second most important (18%). In addition, affective knowledge (4%) and the knowledge for implementation about content area, target population, and methods (5%) was also presented. Currently, no conclusion can be drawn regarding a reasonable percentage of various types of knowledge. However, from a qualitative view point, the characterization of the four major categories of knowledge indicated some incoherence in representing knowledge. For example, the codings from student

Table 1

Frequencies of different types of knowledge and pedagogical strategies for all projects.

Types of Knowledge		Setting Goals	Instructions	Explanations	Demonstrations	Providing Tasks	Asking Questions	Providing Working Space	Evaluation and Feedback	Total	Percent
Domain	Declarative Knowlegdge	15	117	25	45	57	28	37	44	368	73%
	Procedural Knowledge	50	218	41	79	71	21	57	6	543	
Operating	Declarative Knowledge		17	2						19	18%
	Procedural Knowledge		46	2		56		97		201	
Affective	Knowledge		38						17	55	4%
Implementing Knowledge of	Content Area	2	20							22	5%
	Target Population		5							5	
	Teaching Methods	2	20			14				36	
Total		69	481	70	124	198	49	191	67	1249	
Percent		5%	39%	6%	10%	16%	4%	15%	5%		

teachers' projects sometimes display sequences of codings like 1(2)Ea, 1(2)Ba, 2(2)Eb, 2(2)Gb, 1(2)Ga In these kinds of patterns, the second type of knowledge is prematurely introduced before the representation of the previous one is ended appropriately So it seems that various types of knowledge were sometimes represented incoherently in the learning environments constructed by student teachers.

Further analysis about the subcategories of the knowledge indicated several features of domain knowledge presented in the learning environments constructed by student teachers. First, the knowledge about facts, events, concepts, and actions forms the major parts of the environments whereas the knowledge about principles, conditions under which the actions could be taken was seldom involved. Second, most programs presented the isolated elements such as facts, concepts, events, and actions. When the programs sometimes presented the elements as a whole, only temporal, partial, or identical relationships were involved. The programs usually did not present the more critical relationships such as causal and conditional relationships. Third, the programs did not employ so called "bug problems" that the learners often make mistakes on; they did not indicate the learner's misconceptions, or any strategies for efficient problem solving or learning. In short, the instructional programs produced by student teachers only presented relatively simple knowledge such as facts, events, concepts, and actions.

Frequencies of the codings representing the different types of knowledge in each project, shown in Table 2, indicate that all projects presented domain knowledge in the learning environments, but only 62% of them presented operating knowledge. Seventy-seven percent of the projects presented affective knowledge and 62% of the projects presented the knowledge for implementation about the content area, target population, and teaching methods.

Missing operating knowledge

As mentioned, successful execution of the programs by the users may require knowledge of the program features and knowledge of how to operate the program. However, the fact that 38% of the projects constructed by student teachers did not present any operating knowledge motivates an examination of whether operating knowledge is necessary and of the consequences of this absence.

- Is operating knowledge necessary in successful execution of an instructional program?

Table 2 shows that Projects 1, 6, 9 and 11 did not present any operating knowledge in the learning environments. In order to determine the consequence of the absence of operating knowledge in these projects, as well as in other projects which partially lack operating knowledge, the codings from these projects were analyzed in detail.

The analysis reveals three findings. First, it was found that the execution of a program did not demand any operating knowledge when all episodes or views in a program were linked by automatic operators. For example, in Project 9, the first episode of a program (it is a page in most cases) linked the rest of the episodes in a startup procedure with the result that all episodes could be executed automatically without interaction of the user. Therefore, the program required neither manual operators nor operating knowledge to link the episodes or views.

Second, it was found that executing a program did not require any operating knowledge when a manual operator for choosing activities, answering

Table 2

Frequencies of different types of knowledge presented in each project.

Project No	Domain		Operating		Affective	Implementing Knowledge of		
	Declarative Knowledge	Procedural Knowledge	Declarative Knowledge	Procedural Knowledge	Knowledge	Content Area	Target Population	Teaching Methods
1	28	6			10			
2	2	23	5	11		1		2
3	5	12	1	20	3			3
4	97	20	1	68	20	2		
5	6	17		17	3	2		5
6	5	108						
7	56	2		8	9			
8	22	58		1	3			
9	40	1						
10	2	6	1	9	1	1		1
11		27			5	3	1	2
12	48	14		22	10	1	2	
13	57	248	11	45	1	3	3	21
Percent of the Projects	100%		62%		77%	62%		

questions, or choosing assistance within domain knowledge played the role of linking the next episode or view. For example, in Project 7, the program presented the following question to a user:

```
Do you know what time it is?  (Showing a clock)
Type in a letter and press RETURN.
A) 4: 30
B) 6: 00
C) 8: 15
```

The correct answer, which is represented by B, will change the view to a happy face with feedback. After a few seconds, the happy face is replaced by a new question. In this case, the manual operator plays the double roles of answering a question in a domain and linking the current view to the next one. Despite the fact that operating knowledge is not required for executing the program, the program must still indicate how the user is supposed to answer the domain question. Otherwise, the user's answer will be invalid for operating the system.

Finally, it was found that a program must present operating knowledge for its successful execution when the program requires a manual operator to link the episodes or views. Otherwise, the user would be unable to figure out what to do next. For example, both Projects 1 and 11 consisted of fragmented pages and required manual operators to link these pages during program execution. However, the designers did not present any knowledge of how to link these episodes and views, and consequently the user did not know what to do when a view or an episode was finished.

To summarize, a program does not need to present any operating knowledge when the episodes or views are linked by automatic operators, or when the manual operators for domain knowledge play the role of linking episodes or views. However, a program must present operating knowledge to the user when a manual operator is required to link to an episode or a view. In this case, lack of operating knowledge in the learning environments will create difficulties for the novice user to execute the programs. These difficulties will be illustrated in detail in the following section.

- What is the consequence of lacking operating knowledge or incomplete representation of operating knowledge?

The absence of operating knowledge when a manual operator is required to link the current episode or view to the next one is the most common problem in the learning environments constructed by student teachers There are several situations in which a learning environment lacks operating knowledge. The first situation is that student teachers designed paths to the next episode or view, but they did not always indicate these paths to the user so the path remained hidden. For example, in Project 13, the designer designed one path to the next page by pressing the N key and another path to the previous page by pressing the P key in all lessons On the first view of episode 1, the designer indicated that the user can always press N to see next page and press P to see previous page. The designer assumed that the user would always be able to remember these two simple, clear and easily-memorized commands, so she did not indicate that the user needed to press N or P key in subsequent episodes and views. In the third view of episode 2, the designer explained what the user was supposed to do in the rest of the lesson. At the end of that view, the following text was presented:

```
When you are ready to colour, type "colour" into the
command center.  All four turtles will appear.  Assign
at least one very dark colour and another very light
one.  HAVE FUN!
```

The designer assumed that the user would press N or P at that moment. However, it was more probable that the user would type "colour" at the current view since there was no operating knowledge presented at the end of the view. Unfortunately, the working space for the command "colour" was located in another view so there was no corresponding working space available to perform this task at the current view. As expected, a bug occurred when the user typed in "colour". Therefore, the absence of operating knowledge can result in difficulties and even bugs for the user in executing the program This example also illustrates that operating knowledge is not only required in a program, but also in each view when a manual operator is required to link episodes or views The lack of operating knowledge in an episode or a view when manual operators were required for system operating were found in most of the projects.

The second situation is where student teachers neither present operating knowledge nor design the path to the next view when a manual operator is required to link the current view to the next. For example, on the tenth view of episode 10 in Project 4, the designer presented only an open-ended question which was not accompanied with evaluation and feedback. There was neither operating knowledge nor a path to the next episode when the user needed to move on to the next episode. As a consequence, the user encountered an impasse and had to quit the program and restart it in order to choose other branches.

A learning environment must present operating knowledge whenever a manual operator is required to link an episode or a view Moreover, this operating knowledge must correspond to each view in which a working space for operating the system is provided. Otherwise, even if the program has presented consistent, simple, meaningful and easily-memorized commands at the beginning, the user may experience difficulty or encounter impasses during the execution of the program.

In addition to lacking operating knowledge when manual operators are required for operating the system, some other problems in presenting operating knowledge were detected in the walkthrough and encoding processes. Codings from student teachers' projects indicate that in 85% of the projects there were problems in presenting operating knowledge. The other 15% of the projects which did not have any problem presenting operating knowledge were those which needed neither manual operators nor operating knowledge to link the episodes and views In other words, all projects which required manual operators for linking episodes or views had problems with providing adequate operating knowledge. Other problems, besides missing operating knowledge encountered in the learning environments constructed by student teachers, are presented below.

Mismatch between tasks and working spaces for operating knowledge

Sometimes the difficulty experienced in executing a program is caused by a mismatch between tasks and working spaces for operating knowledge. That is, the designer does present operating knowledge for linking one view to another at a certain point, but does not present it at the right place. For example, in the second episode of Project 2, the designer presented a task for operating knowledge three views ahead of its working space. In other cases student teachers presented the tasks for operating the system first, and then presented the domain tasks for the user to perform; after the user had made a great deal of effort to perform the domain tasks, the designer presented the working space for operating the system without indicating the tasks for using that working space, based on the assumption that the user would remember the task for operating the system which was presented before the user performed the domain tasks.

The mismatch between tasks and working spaces for operating systems is indicated by coding patterns like 2(2) Ea, 2(2) Ba, 1(2) Eb, 1(2) Gb, 2(2) Ga. Two problems can be discovered in such patterns. One is that the indicators of necessary sequence "a" and "b" are not in alphabetical order; another is that the types of knowledge are mixed up.

There are two consequences of such mismatch. The first consequence is that it increases the users' working memory load when they are performing domain tasks or processing domain knowledge The second consequence is that users will have difficulty in providing input to link the views if they cannot remember the operating knowledge when they finally get to the working space after performing the domain tasks.

Incomplete instruction for operating knowledge

Student teachers often skipped important components of the procedures when they presented operating knowledge. For example, they might not indicate the page name or show the required quotation mark when they asked the user to use GETPAGE or GETTOOLS primitives, or they might forget to indicate that the user needs to press the enter key when instructing the user to type a command. As a consequence, novice users would become confused and frustrated because the procedures did not conform to the instruction.

Misconceptions of the operating knowledge

Two types of misconceptions of operating knowledge were found often in student teachers' instructions. One is misconception of key functions For example, one student teacher instructed the user to use the return key, the arrow keys, and/or the space bar to move the cursor from the command center to the blanks on the front page for answering questions. However, no matter how hard the user tried, it never worked because what the user needed to do was to hold the command key and press U to move the cursor up, or click the mouse in the appropriate place.

The second misconception concerns operating procedures. For example, a student teacher presented the following instruction to the user:

```
If you need to see what you just read, TYPE   stop
STARTUP.
```

The misconception in this instruction is that, "stop" can only be used in a procedure and it is not used as a command to be typed in. Furthermore, even if "stop" could be used as a command, the computer could not respond to command "stop" when it was executing the program. When the computer had finished its execution, there was no point in stopping the execution any more When these misconceptions occur in instructions, a novice user may become extremely confused and frustrated, and finally give up.

To summarize the above findings on operating knowledge, successful execution of the program requires presenting sufficient operating knowledge in the corresponding view when a manual operator is required to link an episode or a view. Any problems of operating knowledge, such as lack of operating knowledge, mismatch between operating tasks and working spaces, and ignorance of important component of operating knowledge, as well as the designer's misconceptions on operating knowledge will create difficulties for the user to execute the program.

Besides domain knowledge and operating knowledge, other types of knowledge, such as affective knowledge and knowledge for implementation on content areas, target population and methods are also important in users' learning. However, this study did not consider the characteristics of other knowledge and their relative impacts on user's learning. Instead, this study focussed on domain knowledge and operating knowledge which is more important in determining the nature of the learning environments. Based on the fact that the domain knowledge presented by student teachers is relatively simple, this thesis did not analyze the subcategories of domain knowledge and pedagogical strategies used to presented them, in order to simplify the data analysis. The following section presents pedagogical strategies used to convey declarative and procedural knowledge for both domain knowledge and operating knowledge, and analyzes their strengths and weakness.

The Characteristics of Pedagogical Strategies

Table 1 shows the overall pedagogical strategies used by student teachers, and the knowledge that each strategy presents. These data indicate that instruction was the major pedagogical strategy used by student teachers (39%), providing task was second (16%), followed by providing working spaces (15%), demonstrations (10%), and explanations (6%). The least-used strategies were setting goals (5%), providing evaluation and feedback (5%), and asking questions (4%).

Although it is difficult to draw general conclusions regarding the reasonable expected proportions of different types of pedagogical strategies without considering the types of CAI and the nature of the content areas, as well as the learning approaches that the designer taken, it is necessary to match the task, working space, and evaluation and feedback for each action and examine their sequence and appearance. This is because the user cannot perform the tasks or answer the questions if there are no working spaces to do so. In addition, studies have indicated that immediate evaluation and feedback is critical in user's success in a computer-based learning environment (e.g., Corbett & Anderson, 1991). Therefore, it is necessary to look at whether the tasks and questions are accompanied with corresponding working spaces and whether the evaluation and/or feedback corresponds to the user's performance and answers

Because the codings are based on conceptual units of the text and graphics of the display in each view and the operators the view promotes, rather than actual numbers of tasks, working spaces, and evaluation and feedback, it is possible that student teachers use several conceptual units to present the same tasks. Therefore, the tasks, working spaces, and evaluation and/or feedback were investigated in terms of each action. In addition, their sequence and appearance for each action was indicated by the lowercase letters. The actual frequencies of the tasks, working spaces and evaluation and feedback are shown in Table 3.

The balance between providing working spaces, tasks and questions, and evaluation and/or feedback

Results in Table 3 indicate that the total number of tasks and questions are not equal to those of working spaces. The breakdown of types of knowledge reveals that: a) for declarative domain knowledge, all tasks and questions are provided with working spaces, b) for procedural domain knowledge, 10% of the tasks and questions are not provided with working spaces, and c) for operating knowledge, all tasks have working spaces (in fact there are 1.7 times more working spaces than there are tasks and questions).

Results presented in Table 3 also show that the evaluation and feedback is much less than the tasks and questions. For declarative domain knowledge, 74% of the task and questions are provided with evaluation and/or feedback, whereas only 23% of the tasks and questions are provided with evaluation and/or feedback for procedural domain knowledge. None of the tasks and

Table 3

Frequencies of tasks, working spaces and evaluation and feedback.

Types of Knowledge		Presenting Tasks	Asking Questions	Providing Working Spaces	Evaluation and Feedback
Domain	Declarative Knowlegdge	48	9	57	42
	Procedural Knowledge	98	5	93	24
Operating	Declarative Knowledge	0	0	0	0
	Procedural Knowledge	55	0	93	0
Total		201	14	243	66

questions is accompanied by feedback and/or evaluation for operating knowledge.

The overall data on pedagogical strategies reveals several obvious problems in the learning environments constructed by student teachers. First, some learning environments constructed by student teachers provided insufficient working spaces for performing the tasks and answering questions for procedural knowledge. As a result, the user would fail to perform the tasks or answer questions. Further examination of the working spaces designed for performing tasks and answering questions for domain knowledge indicates that student teachers often designed ill-structured working spaces. For example, they designed spaces on the screen so that the users could type their answers in the boxes, or type in answers for open-ended questions on the screen. However, there was no interaction between the user and the program. As a consequence, no evaluation and feedback could be provided in these cases.

Second, the fact that working spaces were 1 7 times more frequent than tasks on operating knowledge indicates that the designers sometimes did not present tasks for operating the system even if they had designed working spaces for do so. Thus, a user often became unable to continue at the end of an episode or a view due to his or her lack of understanding a task, even if there were a path to the following episode or view. This finding corresponds to that emerging from the characterization of types of knowledge in the previous section.

Finally, the results showed that only 74% of the task and questions were provided with evaluation and/or feedback for domain declarative knowledge, whereas only 23% of the tasks and questions were provided with evaluation and/or feedback for domain procedural knowledge. This suggests that there was a serious shortage of evaluation and feedback in the learning environments constructed by student teachers. This shortage of evaluation and feedback may be due to the lack of working spaces and the unreadable input in ill-structured working spaces. On the other hand, the fact that no feedback and evaluation was provided for operating knowledge is not regarded as a problem since the Logo program itself can provide feedback on operating knowledge.

The coherence of pedagogical strategies

In addition to the above findings, the indicators of necessary sequences from codings reveal two problems in the sequence of pedagogical strategies. The first problem concerns the coherence of presenting tasks and providing working spaces. Incoherences between providing tasks or instructions on how to perform the tasks and working spaces were often found in the learning environments constructed by student teachers. It occured when, in the course of presenting tasks and instructing the user on how to perform the tasks, the student teacher interrupted one presentation with another irrelevant presentation, or when the tasks or instructions were provided a few views ahead of the working spaces. Such incoherence is indicated by the sequences of lowercase letters that are in alphabetical disorder. The second problem in the sequence of pedagogical strategies concerns the coherence of task representations An incoherent task presentation occurs when the text and the pictures that present the same task are separated in different views.

There are several consequences of incoherent pedagogical strategies. First, they increase the user's working memory load. Second, they increase the difficulty for the user in understanding the instructions and tasks. The users might even be unable to continue the execution if they forget the instructions or the tasks by the time they get to the working spaces. Finally, the incoherence between instructions, tasks, and working spaces would cause bugs when the user performs tasks that do not have corresponding working spaces in the current view.

The distribution of strategies across projects

In order to further investigate how student teachers used pedagogical strategies in the learning environments they constructed, the pedagogical strategies used in each project are shown in Table 4. Data in Table 4 indicate that all projects used three pedagogical strategies: providing instructions, providing tasks, and providing working spaces. Sixty-nine percent of the projects employed setting goals as a strategy, and the same proportion of projects employed the explanation strategy. In addition, 62% of the projects used demonstration strategies, 54% of projects designed evaluation and/or feedback, and 35% of projects used the strategy of asking questions.

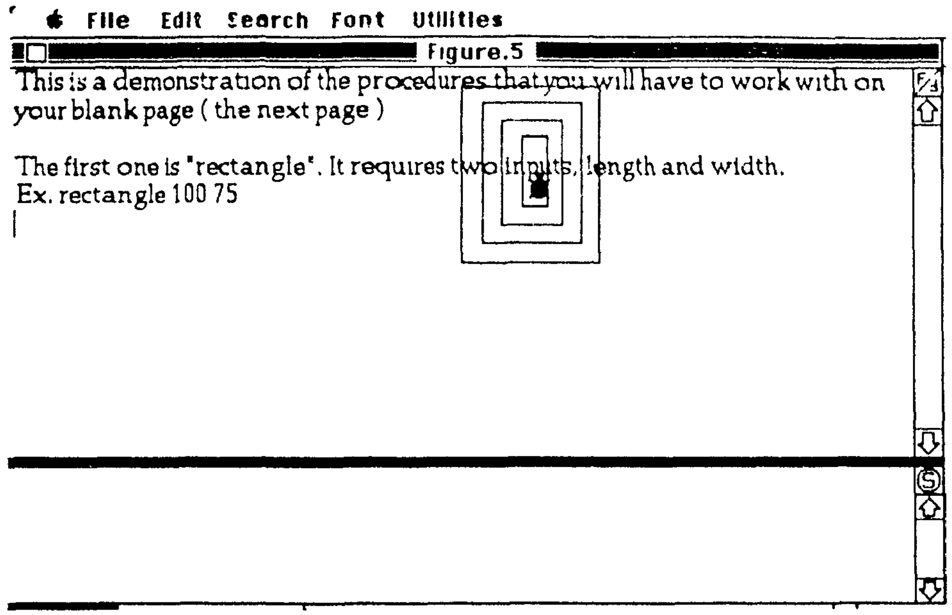
Table 4

Frequencies of pedagogical strategies used in each project.

Project No.	Setting Goals	Instructions	Explanations	Demonstrations	Presenting Tasks	Asking Questions	Providing Working spaces	Evaluation and Feedback
1		7			10		10	10
2	2	27	2		6		7	
3	2	22	1	5	6		8	
4		64			58		53	20
5	3	26	2	2	10		7	
6	2	45	2	20	14	1	29	
7		17	8	1	16	8	8	17
8	9	19	36	13	3	1	2	1
9		20		19	1		1	
10	1	8	1		5		4	2
11	1	29			5		2	1
12	5	39	3	3	18	8	12	9
13	44	158	13	61	46	18	49	
Percent of the Projects	69%	100%	69%	62%	100%	38%	100%	54%

What these data indicate is that instructions, providing tasks, and providing working spaces are three basic strategies used by all student teachers. In addition, demonstrations, explanations, and asking questions were also used in some projects as more advanced strategies. For example, in Project 13, the designer first provided instructions on the content area in which the program was supposed to be used, the target population, and implementation methods. Later, the designer provided instructions on the structures of the lessons. Through the first two views of instructions, it was clear to the user the purpose of the program, who could use it, and how to use it. Furthermore, the designer set the goals and objectives at the beginning of each lesson so that the user knew in advance what he or she was supposed to do.

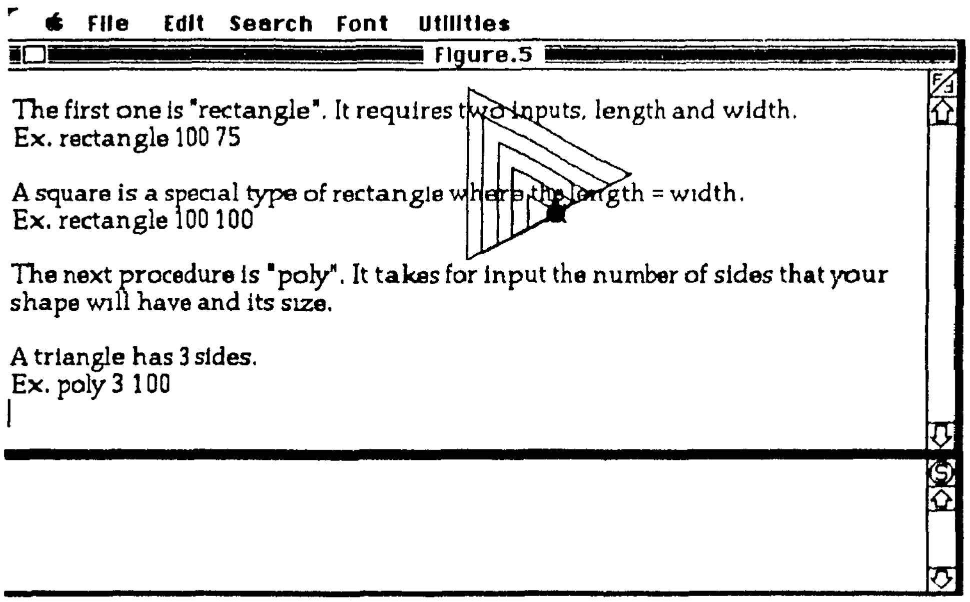
In the rest of the program, Project 13 used a combination of pedagogical strategies similar to the one used in Projects 6, 8 and 9. These projects showed consistent coding patterns. Such patterns are composed of instructions, demonstrations, and explanations in an elegant way so that the user could see the procedures needed for performing particular tasks, and the screen effects these procedures produced, such as in Project 13 (See Figure 5) and Project 6 (See Figure 6) Slightly different from Projects 6 and 13, Project 8 used demonstrations, explanations, and instructions intensively to tutor the user on the nature of a grid, how to make a grid, and how to find points on a grid (See Figure 7) These combinations of pedagogical strategies helped the user visualize abstract concepts so that the domain knowledge was efficiently conveyed to the user.



This is a demonstration of the procedures that you will have to work with on your blank page (the next page)

The first one is "rectangle". It requires two inputs, length and width.
Ex. rectangle 100 75

Figure 5 a



The first one is "rectangle". It requires two inputs, length and width.
Ex. rectangle 100 75

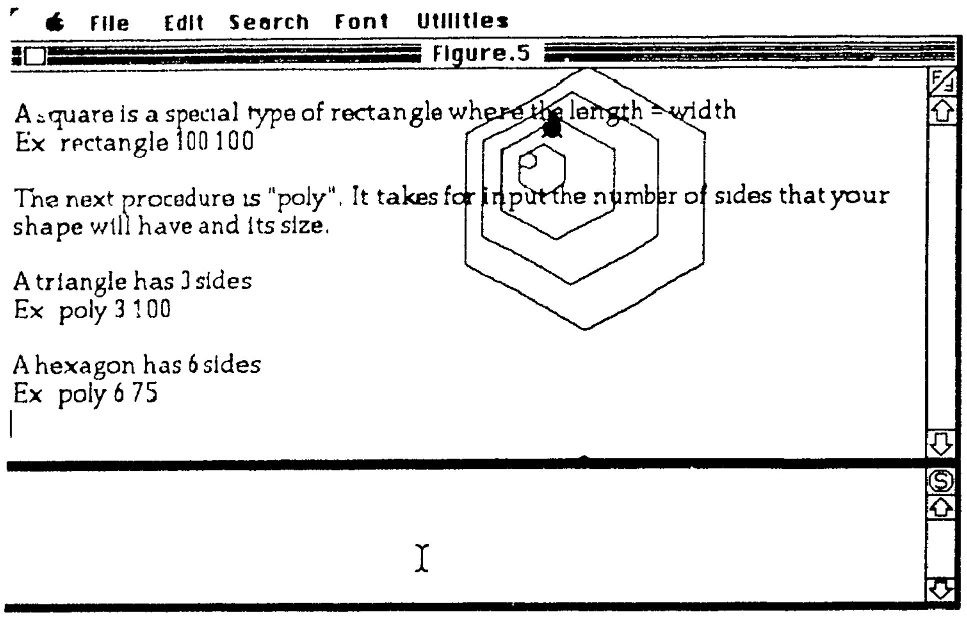
A square is a special type of rectangle where the length = width.
Ex. rectangle 100 100

The next procedure is "poly". It takes for input the number of sides that your shape will have and its size.

A triangle has 3 sides.
Ex. poly 3 100

Figure 5 b

(continued on next page)



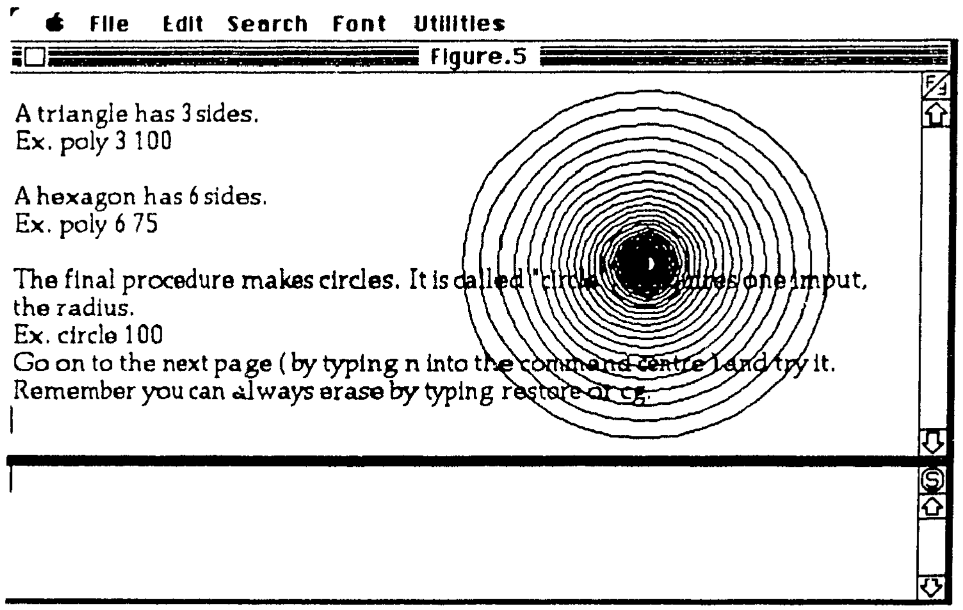
A square is a special type of rectangle where the length = width
Ex rectangle 100 100

The next procedure is "poly". It takes for input the number of sides that your shape will have and its size.

A triangle has 3 sides
Ex poly 3 100

A hexagon has 6 sides
Ex poly 6 75

Figure 5 c



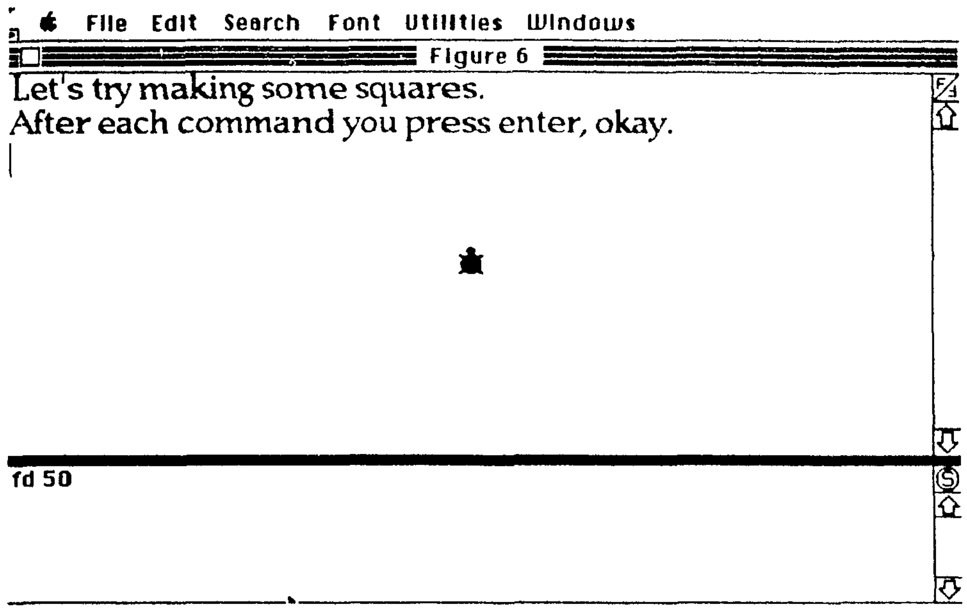
A triangle has 3 sides.
Ex. poly 3 100

A hexagon has 6 sides.
Ex. poly 6 75

The final procedure makes circles. It is called "circle" ... requires one input, the radius.
Ex. circle 100
Go on to the next page (by typing n into the command centre) and try it.
Remember you can always erase by typing restore or cg.

Figure 5 d

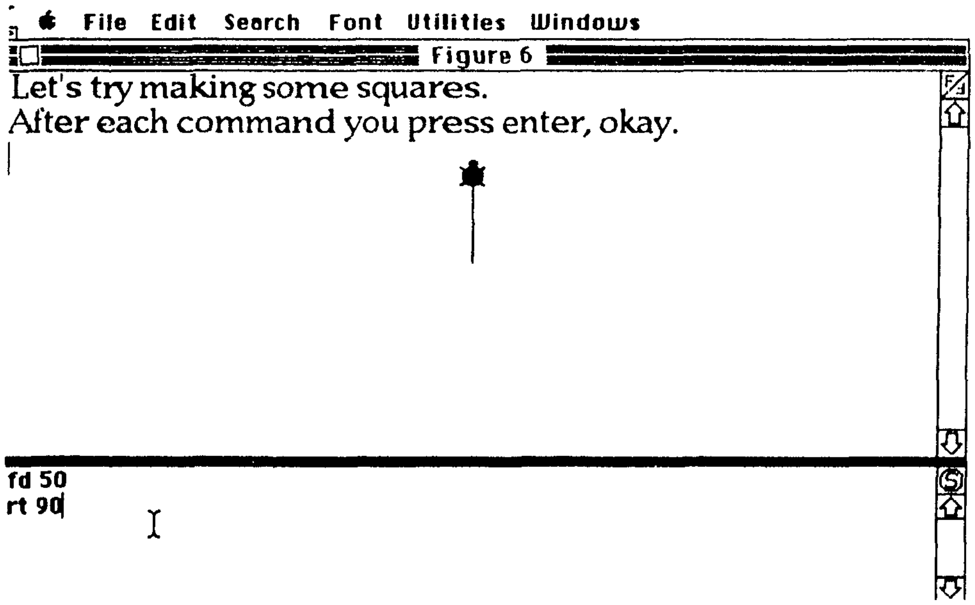
Figures 5 a-d. A combination of instruction and demonstration with scrolling effects.



Let's try making some squares.
After each command you press enter, okay.

```
fd 50
```

Figure 6 a

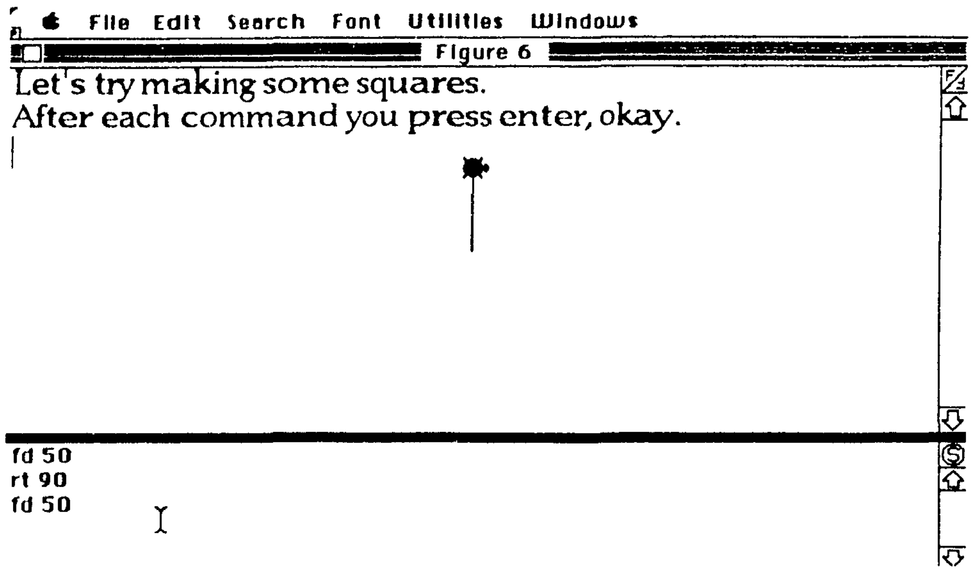


Let's try making some squares.
After each command you press enter, okay.

```
fd 50
rt 90
```

Figure 6 b

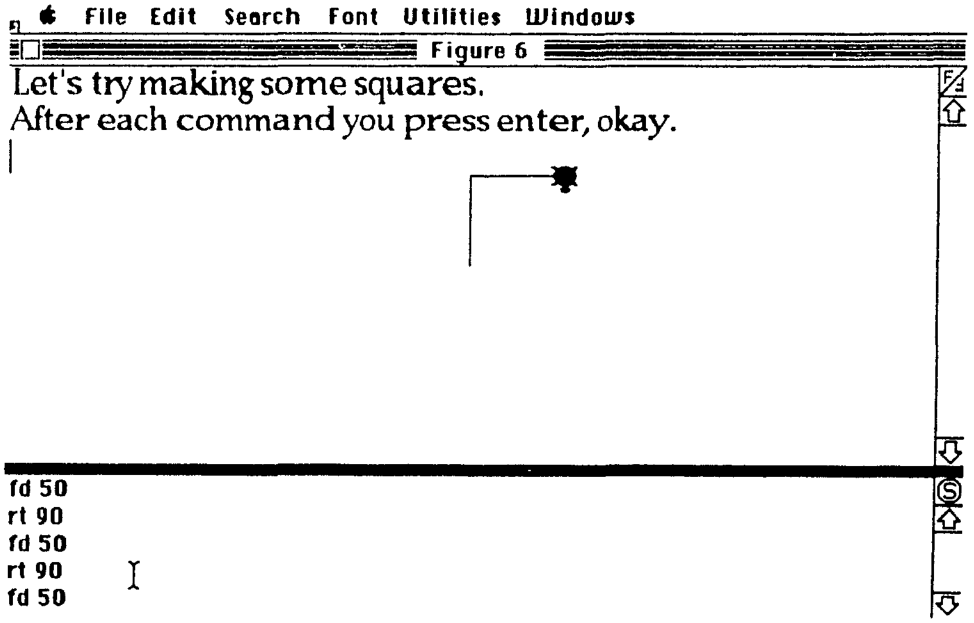
(continued on next page)



Let's try making some squares.
After each command you press enter, okay.

```
fd 50
rt 90
fd 50
```

Figure 6 c

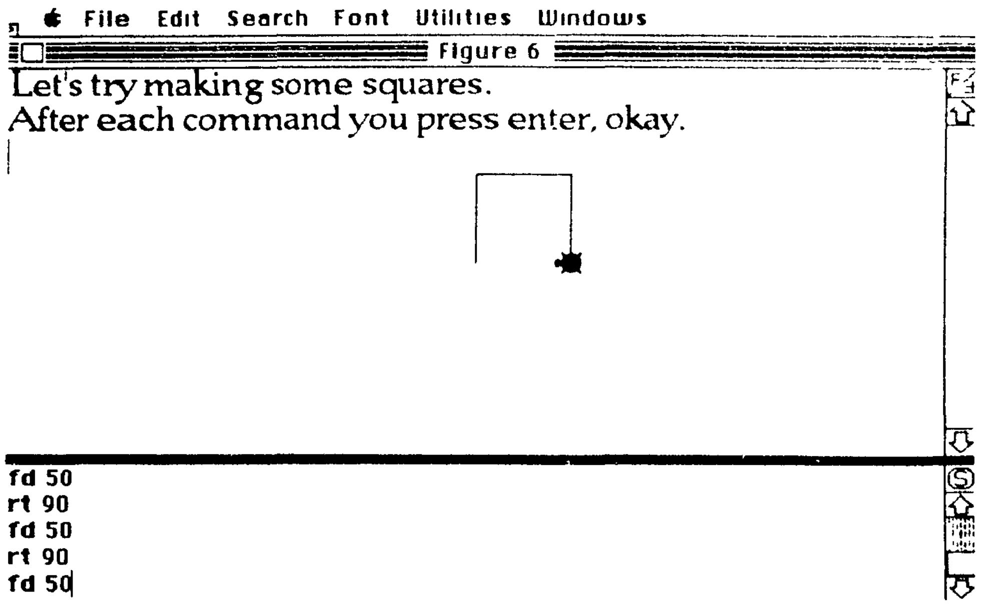


Let's try making some squares.
After each command you press enter, okay.

```
fd 50
rt 90
fd 50
rt 90
fd 50
```

Figure 6 d

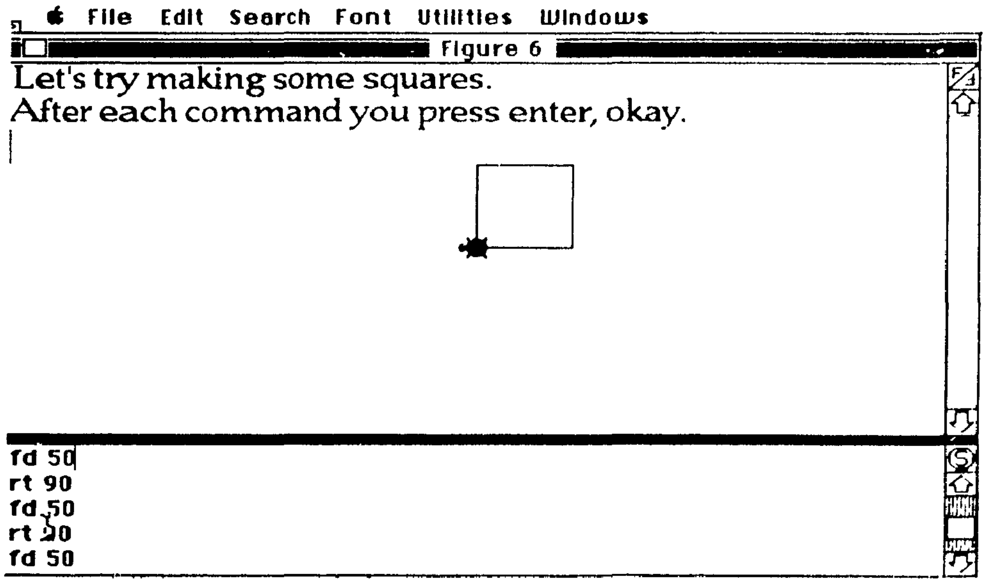
(continued on next page)



Let's try making some squares.
After each command you press enter, okay.

```
fd 50
rt 90
fd 50
rt 90
fd 50
```

Figure 6 e

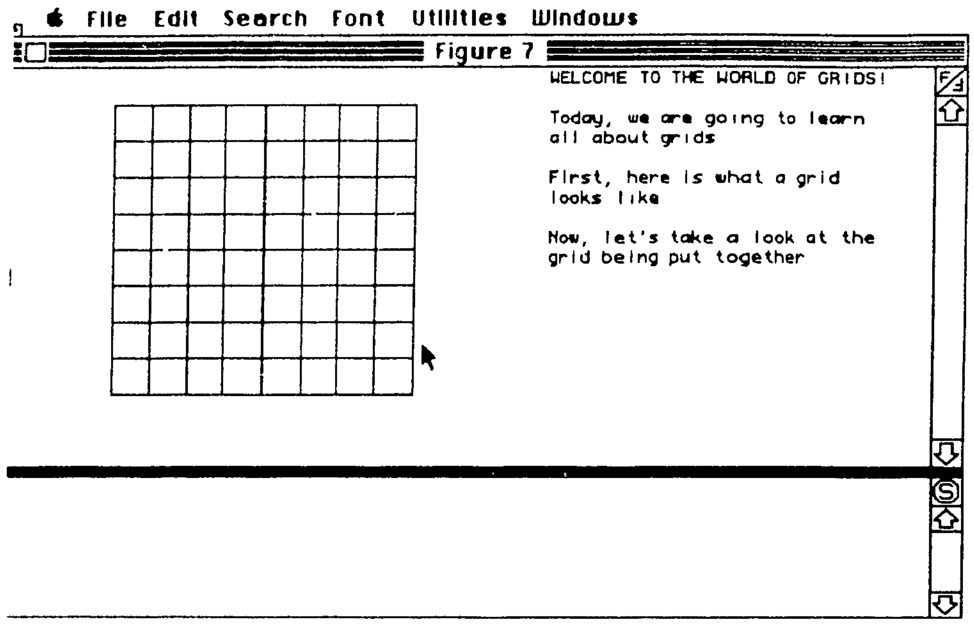


Let's try making some squares.
After each command you press enter, okay.

```
fd 50
rt 90
fd 50
rt 90
fd 50
```

Figure 6 f

Figures 6 a-f. A combination of instructions and demonstration user controlled by pressing the enter key.



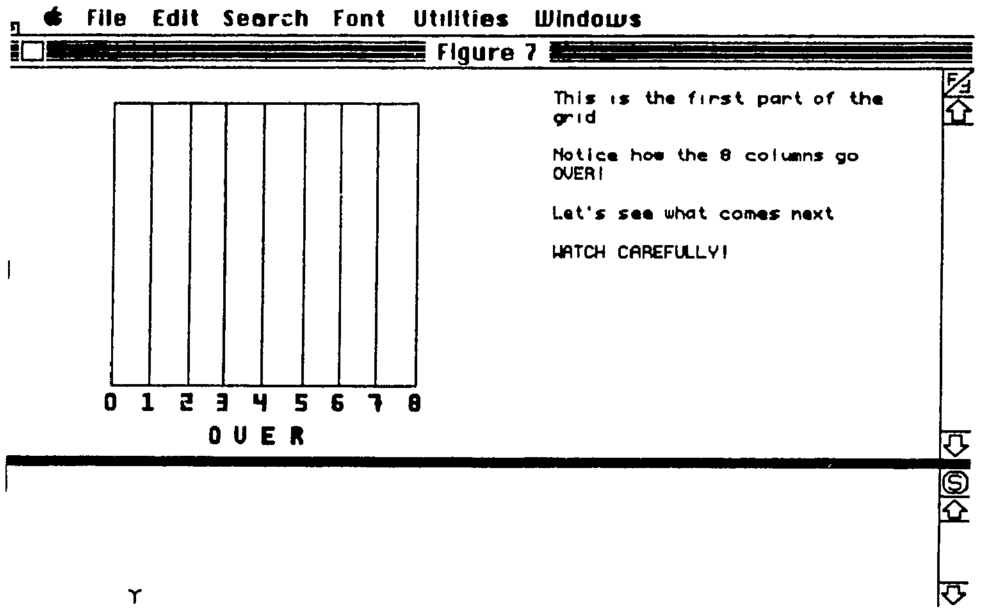
WELCOME TO THE WORLD OF GRIDS!

Today, we are going to learn all about grids

First, here is what a grid looks like

Now, let's take a look at the grid being put together

Figure 7 a



This is the first part of the grid

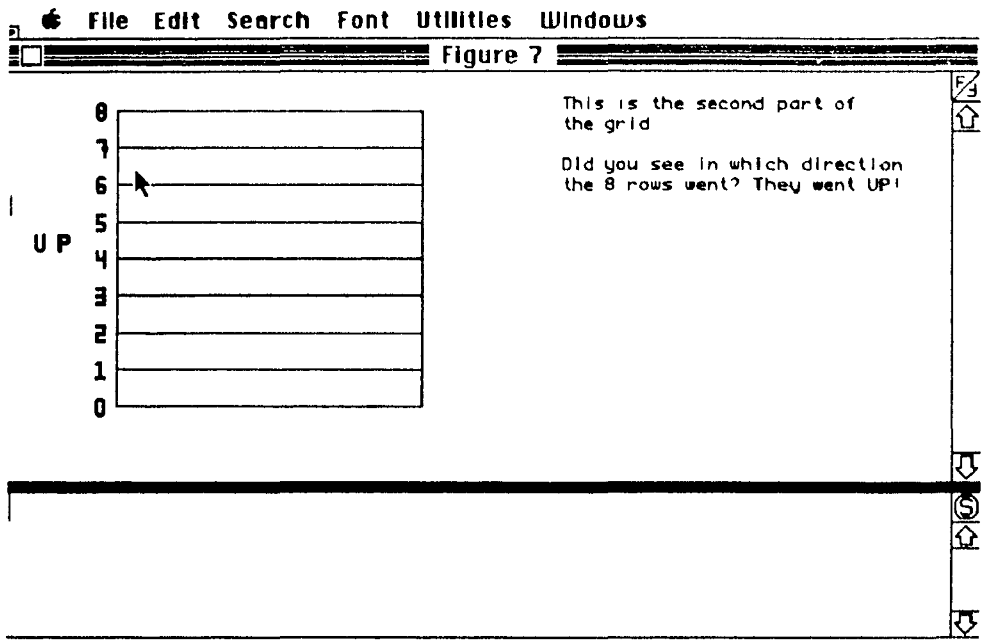
Notice how the 8 columns go OVER!

Let's see what comes next

WATCH CAREFULLY!

Figure 7 b

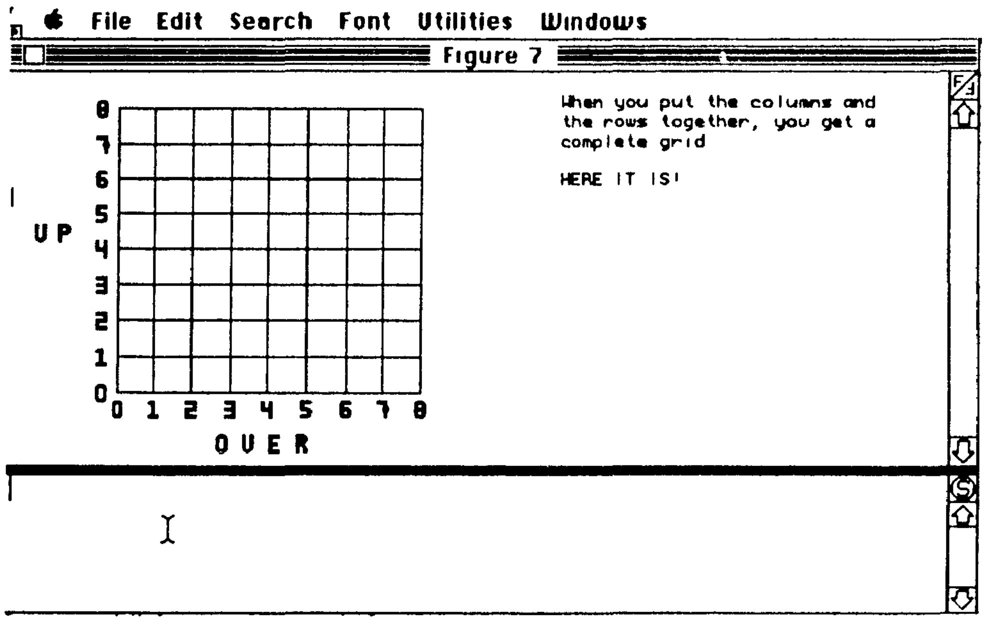
(continued on next page)



This is the second part of the grid

Did you see in which direction the 8 rows went? They went UP!

Figure 7 c

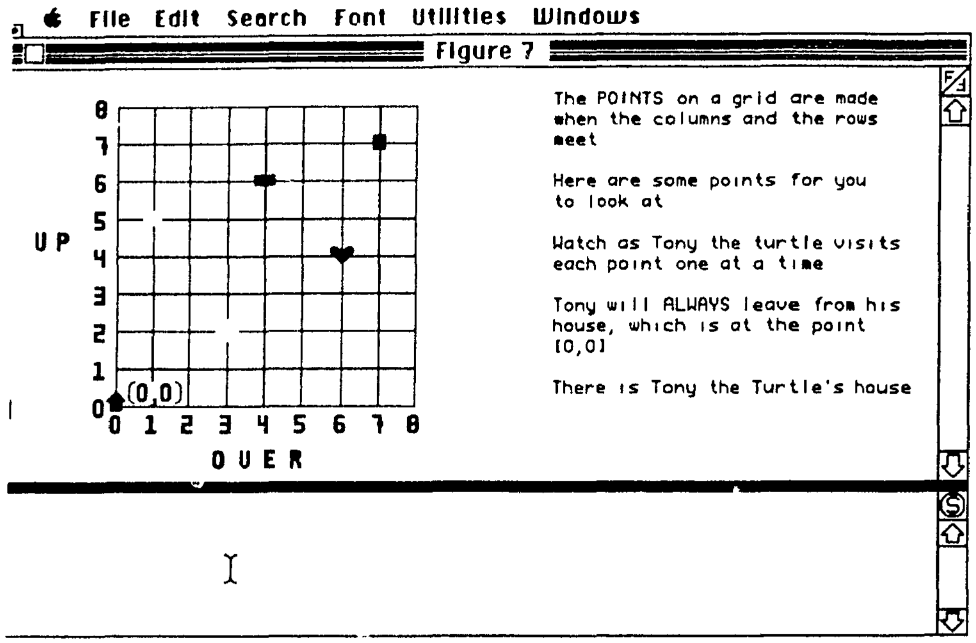


When you put the columns and the rows together, you get a complete grid

HERE IT IS!

Figure 7 d

(continued on next page)



The POINTS on a grid are made when the columns and the rows meet

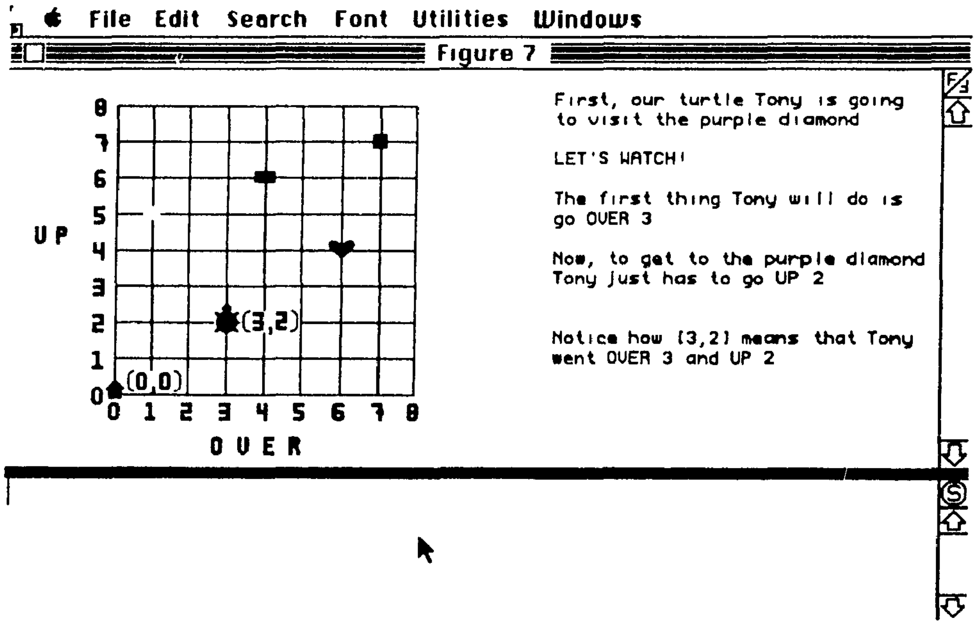
Here are some points for you to look at

Watch as Tony the turtle visits each point one at a time

Tony will ALWAYS leave from his house, which is at the point [0,0]

There is Tony the Turtle's house

Figure 7 e



First, our turtle Tony is going to visit the purple diamond

LET'S WATCH!

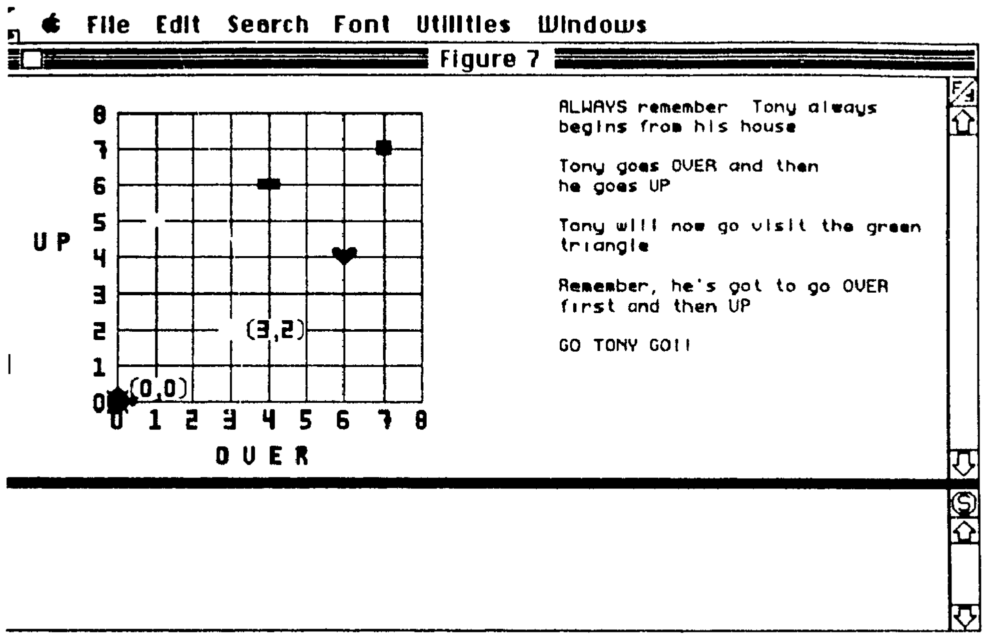
The first thing Tony will do is go OVER 3

Now, to get to the purple diamond Tony just has to go UP 2

Notice how [3,2] means that Tony went OVER 3 and UP 2

Figure 7 f

(continued on next page)



ALWAYS remember Tony always begins from his house

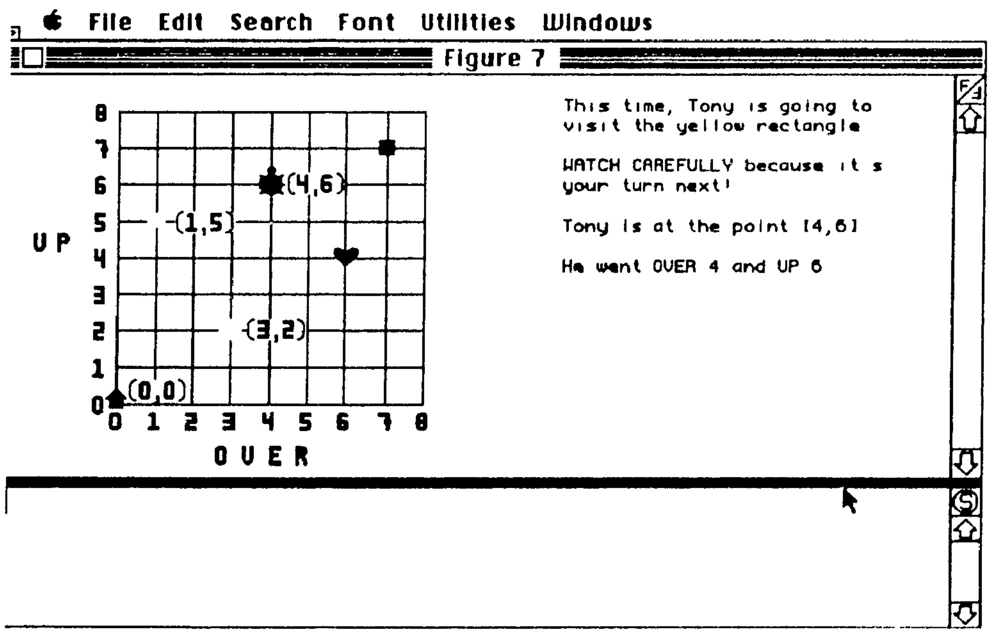
Tony goes OVER and then he goes UP

Tony will now go visit the green triangle

Remember, he's got to go OVER first and then UP

GO TONY GO!!

Figure 7 g



This time, Tony is going to visit the yellow rectangle

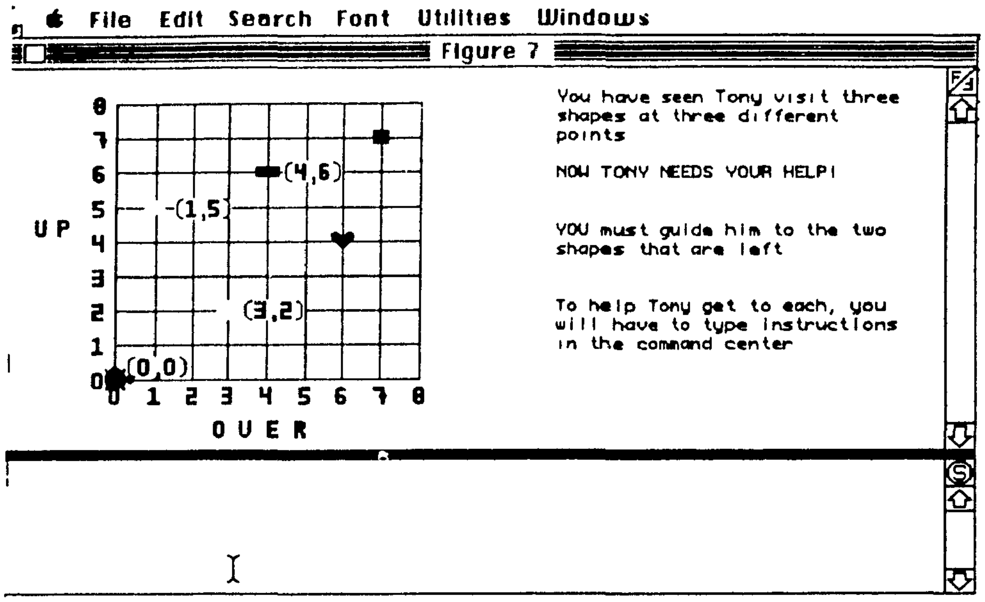
WATCH CAREFULLY because it s your turn next!

Tony is at the point [4,6]

He went OVER 4 and UP 6

Figure 7 h

(continued on next page)



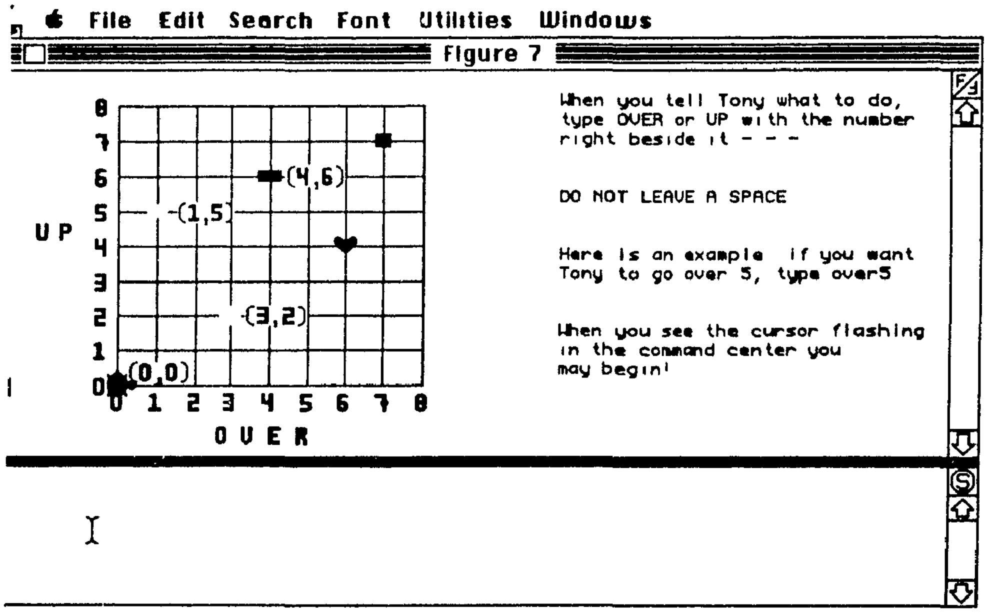
You have seen Tony visit three shapes at three different points

NOW TONY NEEDS YOUR HELP!

YOU must guide him to the two shapes that are left

To help Tony get to each, you will have to type instructions in the command center

Figure 7 i



When you tell Tony what to do, type OVER or UP with the number right beside it - - -

DO NOT LEAVE A SPACE

Here is an example if you want Tony to go over 5, type over5

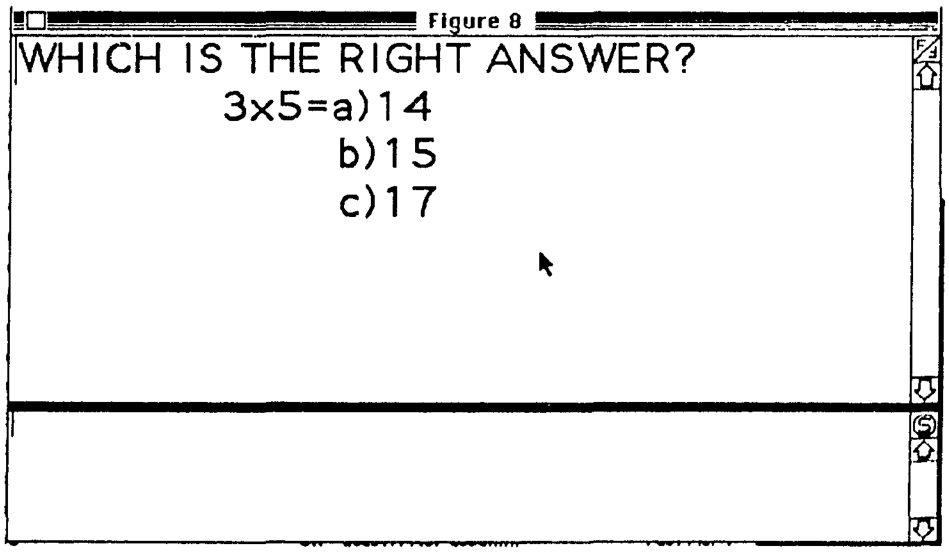
When you see the cursor flashing in the command center you may begin!

Figure 7 j

Figures 7 a-j. A combination of demonstrations, explanations, and instructions with clearing text and clearing graphics.

Another common feature in these four projects was that tasks and working spaces were provided for users to practice or explore what they had learned right after the tutorial was finished. In addition, the designers provided the users with important elements of the procedures required for performing tasks, or the means to access these elements while the tasks are being performed. Moreover, these projects enabled the users to control the flow of execution. In Project 6, users could control the speed of producing the screen effects of each procedure by pressing the enter key, while in Project 13, the designer allowed the user to go back to the previous page or to move on to the next one by pressing "n" or "p" key.

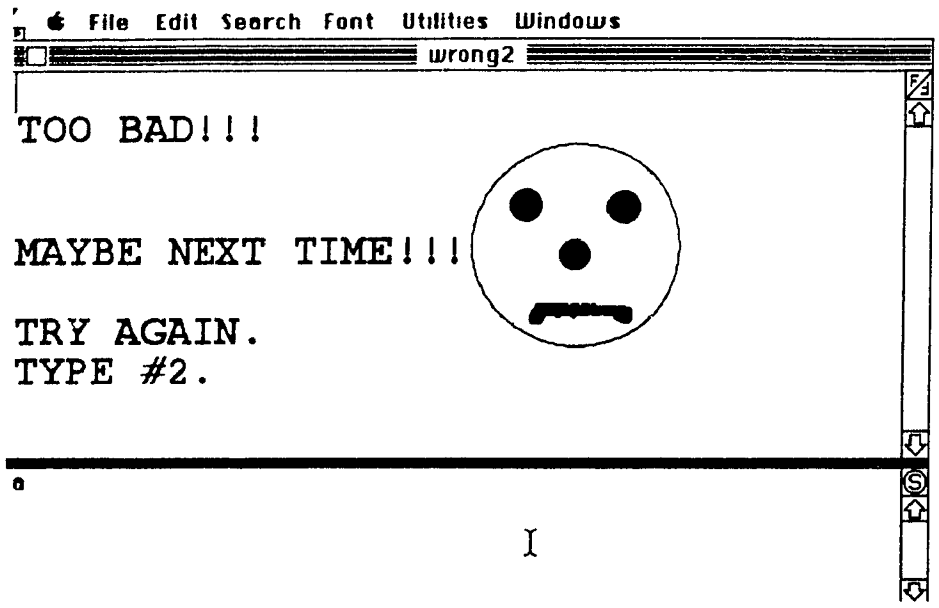
There was also a pattern of pedagogical strategies for drill and practice programs. Projects 4 and 7 and some episodes of Project 1 used the combination of presenting tasks, working spaces, and immediate evaluations and feedback (see Figure 8a-8d).



WHICH IS THE RIGHT ANSWER?
3x5=a)14
b)15
c)17

Figure 8 a

(continued on next page)

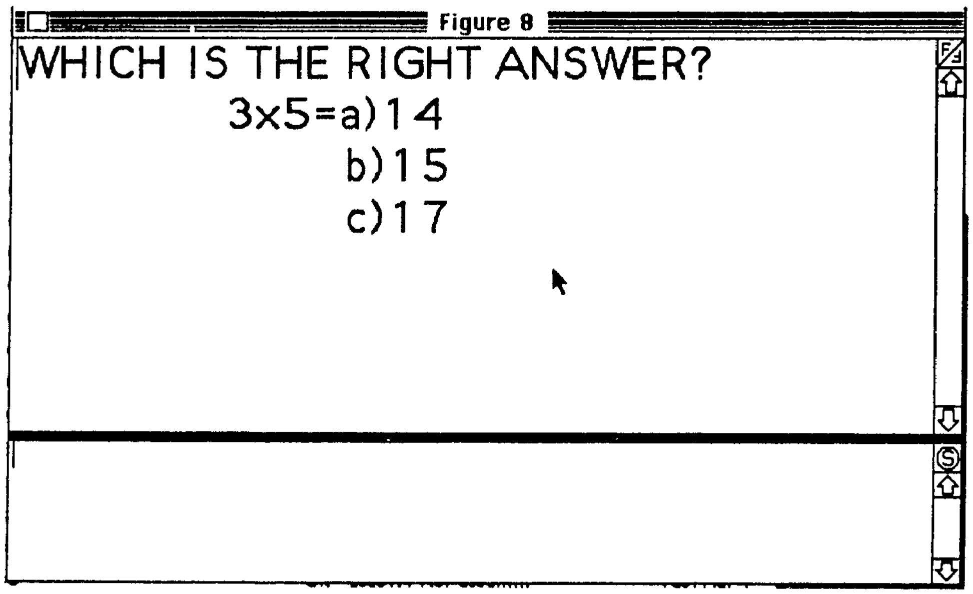


TOO BAD!!!

MAYBE NEXT TIME!!!

TRY AGAIN.
TYPE #2.

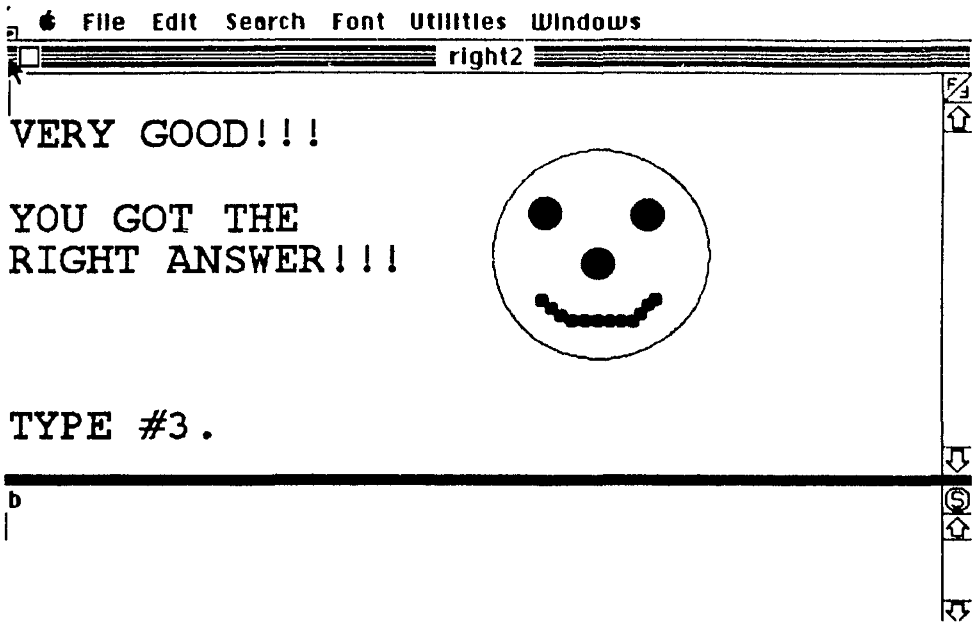
Figure 8 b



WHICH IS THE RIGHT ANSWER?
3x5=a)14
b)15
c)17

Figure 8 c

(continued on next page)



VERY GOOD!!!

YOU GOT THE
RIGHT ANSWER!!!

TYPE #3.

Figure 8 d

Figures 8 a-d. A combination of task presentation, working spaces, and evaluation and feedback.

The findings from characterizing pedagogical strategies are that, on the one hand, student teachers have developed some pedagogical strategies to convey the knowledge efficiently. On the other hand, there were still some problems with the pedagogical strategies used. First, the learning environments provided insufficient working spaces, or ill-structured working spaces which did not promote interactions for performing tasks and answering questions in domains. The second problem was that the learning environments presented working spaces for operating knowledge without any indication of these working spaces. The third problem was that some sequences of pedagogical strategies led to incoherent presentation of tasks and incoherent instructions, tasks, and working spaces. Finally, there was a serious shortage of evaluation and feedback in the learning environments constructed by student teachers.

The characterization of knowledge and pedagogical strategies has shown both the advantages and limitations of the learning environments constructed by student teachers. From a human-computer interaction perspective, the knowledge presented in the learning environments is regarded as the output of the computer whereas the manual operators from the user are regarded as the input of the computer. It is necessary to determine what kind of input a computer requires from the user within the context of interaction. From the perspective of learning, the characterization of a learning environment should consider whether a learning environment provides sufficient user-computer interactions (i.e., learning activities) and whether such interactions facilitate learning. Furthermore, it is necessary to investigate whether the learning environment provides the freedom for the user to choose activities, task complexities, and various types of assistance according to his or her individual needs. The following section will attempt to discuss these issues by characterizing the interactions.

Interactions

The interactions in the learning environments were characterized by the attributes of the operators required to execute programs, which are either automatic or manual. *Automatic operators* refer to the cases where the execution of a program does not require any input from the user. In contrast, manual operators refer to the cases when the execution of a program requires the user's input. Through manual operators, a user may be able to perform tasks, answer questions, or operate the system. In addition, it is possible for the user to select activities, choose task complexity, or choose various types of assistance according to his or her needs.

Ratio of manual and automatic operators

Tables 5 and 6 show the overall operators and their functions designed in the learning environments constructed by student teachers. Student teachers designed 58% automatic operators and 42% manual operators The ratio of automatic operators suggests that the users did not have the freedom to control the flow. It also suggests that these learning environments might not provide sufficient interactions which are critical in the learning process and which enable the user the flexibility to select activities or task complexity, as well as various types of assistance. These results will be further examined by analyzing the manual operators designed for different purposes.

Characteristics of manual operators

In the learning environment constructed by student teachers, data from Table 5 indicate that 39% of the manual operators were designed for operating the systems, and 55% of the manual operators were designed for performing tasks and answering questions. Only three percent of the manual operators were designed for choosing assistance, and the same percent of the manual operators were designed for choosing activities. There were no manual operators for choosing task complexity.

Table 5

Frequencies of different types of manual operators and their functions designed for all projects.

	Choosing Tasks or Activities	Choosing Task Complexity	Answering Questions	Answering Multiple Choice Questions	Performing Tasks	Choosing Assistance	Operating the System	Total	Percent
Pressing a letter or a number				19			43	62	27%
Pressing enter after a procedure					16			16	7%
Typing a command	6			2	4	8	41	61	26%
Typing a command + a variable					28			28	12%
Using the primitives					22		7	29	12%
Typing a word according to a question			17	13			1	31	13%
Typing sentence(s)			6					6	3%
Total	6	0	23	34	70	8	92	233	
Percent	3%	0%	10%	15%	30%	3%	39%		100%

The ratio of manual operators for performing tasks and answering questions vs. operating the system .

The numbers of manual operators for performing tasks and answering questions for domain knowledge are only 1 38 times greater than those tor operating the systems. This ratio suggests that the learning environments constructed by student teachers lack the interactions that promote task performance and learning. In addition, the fact that the manual operators were rarely designed for choosing the activities, types of assistance, or task complexity indicates that the learning environments provided the users with very limited control over the system and that they did not have the flexibility to meet the individual's needs in the learning process.

The forms of manual operators for performing tasks, answering questions, or operating the system.

In order to determine whether the input required from the user supported the user in learning the domain knowledge and in operating the system, manual operators were categorized further according to their forms. Data in Table 5 shows that there were four types of manual operators for performing tasks for domain knowledge: typing a command, typing a command with a variable, using the Logo primitives, and pressing the enter key when the procedures appear on the screen. The manual operators for answering questions consisted of typing a word or a sentence, or pressing a letter or a number for multiple choice questions. The manual operators for operating the systems mainly included typing a command, pressing a letter or a number, or using primitives

What was observed from these data was that the student teachers tended to use pressing a letter or a number for operating the system and typing a command or command with a variable to perform the tasks in domains. It seemed that they tried to minimize the difficulty of operating the system by simplifying the input required from the users. On the other hand, for performing the tasks, they used the input which required more understanding and gave users more flexibility. Therefore, the manual operators they designed for performing tasks in domains could support learning the domain knowledge and operating the systems.

Distribution of manual operators in projects.

The data in Table 6 indicate that 85% of the projects designed manual operators for performing tasks or answering questions, 46% of the projects designed the manual operators for operating the systems, and 31% of the projects designed the manual operators for the users to choose the activities or tasks. Only 15% of the projects designed the manual operators for choosing types of assistance and none of the projects designed the manual operators for choosing task complexity.

These data suggest that, even though only two percent of the manual operators were used for choosing activities, one-third of the student teachers had considered providing such flexibility to the users. However, the fact that 15% of projects did not include any manual operators for performing tasks or answering questions for domain knowledge further confirmed that the learning environments constructed by some student teachers lacked the interactions that promoted task performance and learning for domain knowledge.

Table 6

Frequencies of different functional operators in each project.

Project No	Choosing Tasks or Activities	Choosing Task Complexity	Answering Questions	Answering Multiple Choice Questions	Performing Tasks	Choosing Assistance	Operating the System
1			7	3			
2					2		4
3	1						6
4	1		6	16			31
5	1				3		4
6				1	20	8	
7				9			
8					20		
9			10				
10						1	3
11					2		
12	3			5	4		
13					39		43
Percent of the Projects	31%	0%	23%	38%	61%	15%	46%

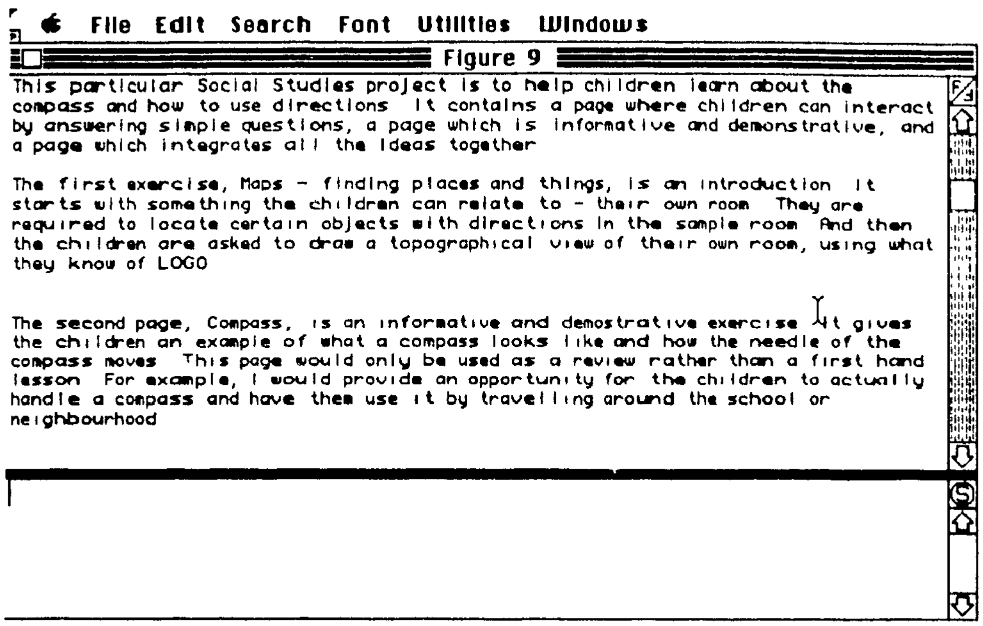
Characteristics of automatic operators

The data in Table 7 show that the most frequently used automatic operators in the learning environments constructed by student teachers were scrolling effects (86%). The disadvantages of scrolling, including other automatic operators are that, first, the user's interactions which are critical in learning processes are rarely involved in executing a program so the computers only present movable text or pictures, and their unique potential for interacting with users is not utilized. Second, the text and graphics are prearranged through automatic operators so the user did not have any choice in the learning processes. Finally, the user could not control the execution of the program, so it becomes a serious problem when the text is difficult to understand, poorly formatted, and presented at an inappropriate speed. One example of such disadvantages of using automatic operators can be seen in Episode 1 of Project 12. This episode involved a high proportion of scrolling effects to present text that was complex and crowded so it was difficult to read and understand (See Figure 9).

Table 7

Frequencies of automatic operators designed in all projects.

Automatic Operators	General	Clearing Texts	Clearing Graphics	Total	Percent
Clearing	9	19	3	31	9%
Scrolling	260		38	298	86%
Automatically Executing pages	19			19	5%
Total	288	19	41	348	100%

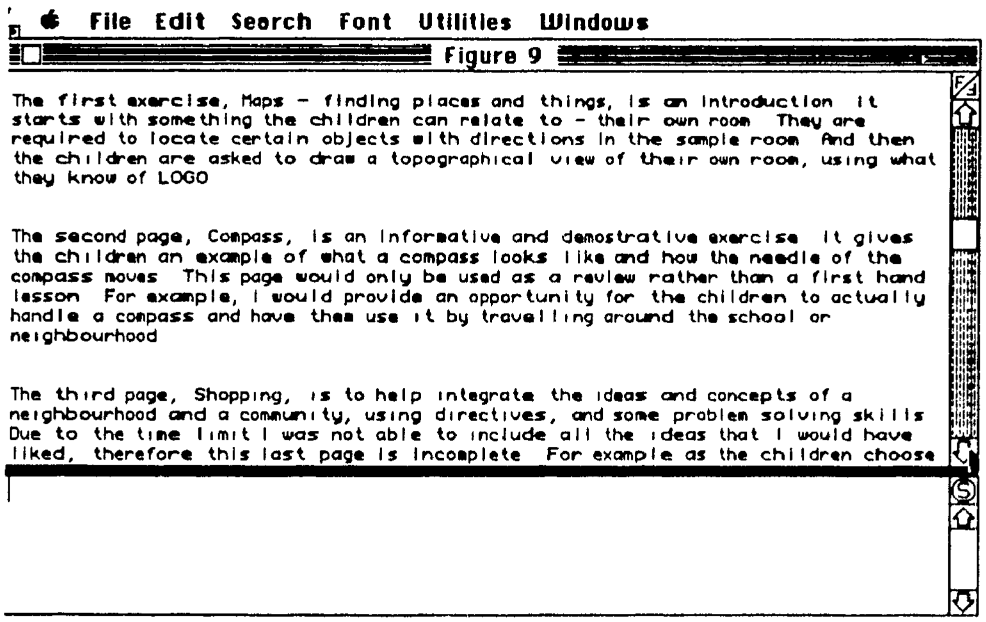


This particular Social Studies project is to help children learn about the compass and how to use directions It contains a page where children can interact by answering simple questions, a page which is informative and demonstrative, and a page which integrates all the ideas together

The first exercise, Maps - finding places and things, is an introduction It starts with something the children can relate to - their own room They are required to locate certain objects with directions in the sample room And then the children are asked to draw a topographical view of their own room, using what they know of LOGO

The second page, Compass, is an informative and demostrative exercise It gives the children an example of what a compass looks like and how the needle of the compass moves This page would only be used as a review rather than a first hand lesson For example, I would provide an opportunity for the children to actually handle a compass and have them use it by travelling around the school or neighbourhood

Figure 9 a



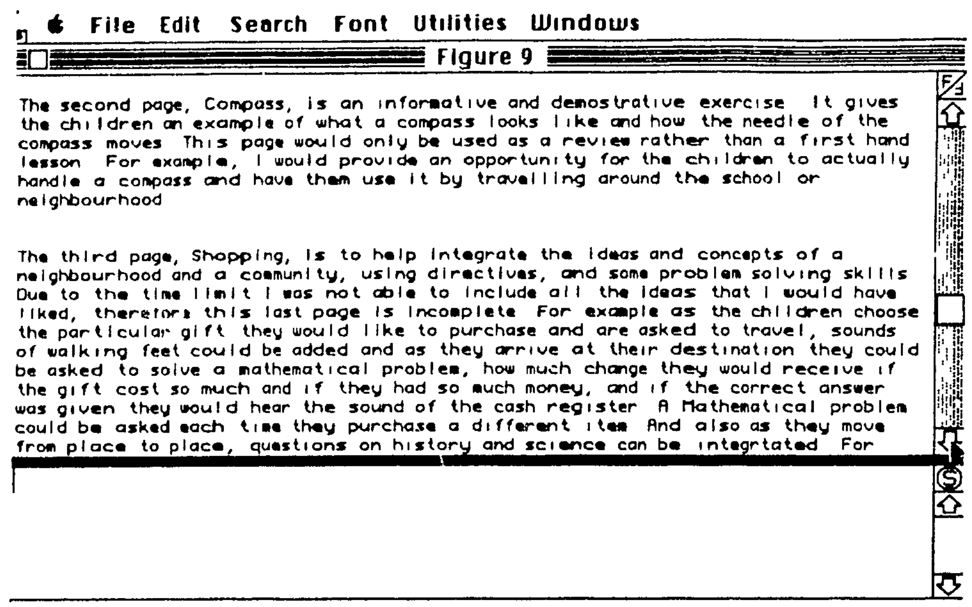
The first exercise, Maps - finding places and things, is an introduction It starts with something the children can relate to - their own room They are required to locate certain objects with directions in the sample room And then the children are asked to draw a topographical view of their own room, using what they know of LOGO

The second page, Compass, is an informative and demostrative exercise It gives the children an example of what a compass looks like and how the needle of the compass moves This page would only be used as a review rather than a first hand lesson For example, I would provide an opportunity for the children to actually handle a compass and have them use it by travelling around the school or neighbourhood

The third page, Shopping, is to help integrate the ideas and concepts of a neighbourhood and a community, using directives, and some problem solving skills Due to the time limit I was not able to include all the ideas that I would have liked, therefore this last page is incomplete For example as the children choose

Figure 9 b

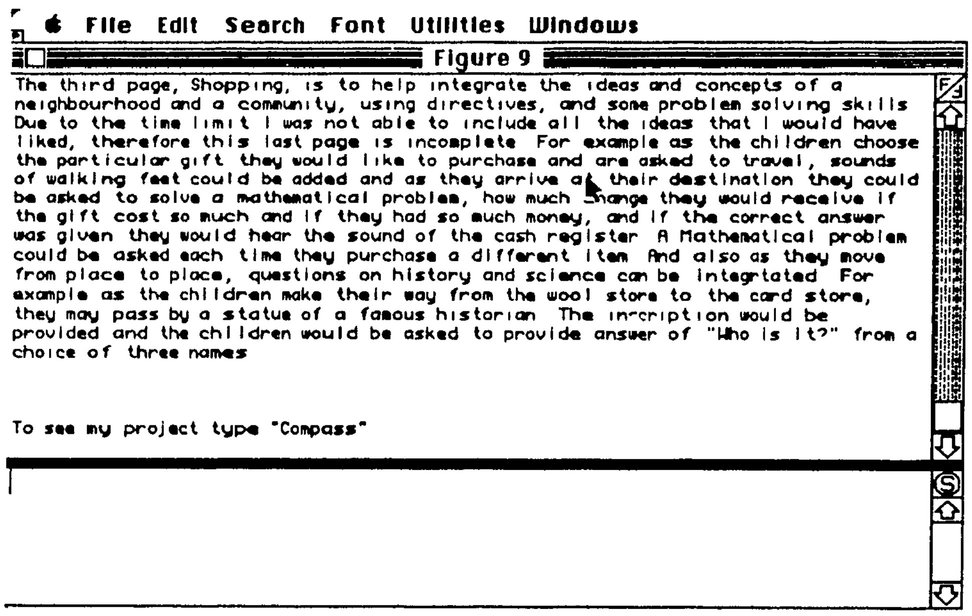
(continued on next page)



The second page, Compass, is an informative and demostrative exercise It gives the children an example of what a compass looks like and how the needle of the compass moves This page would only be used as a review rather than a first hand lesson For example, I would provide an opportunity for the children to actually handle a compass and have them use it by travelling around the school or neighbourhood

The third page, Shopping, is to help integrate the ideas and concepts of a neighbourhood and a community, using directives, and some problem solving skills Due to the time limit I was not able to include all the ideas that I would have liked, therefore this last page is incomplete For example as the children choose the particular gift they would like to purchase and are asked to travel, sounds of walking feet could be added and as they arrive at their destination they could be asked to solve a mathematical problem, how much change they would receive if the gift cost so much and if they had so much money, and if the correct answer was given they would hear the sound of the cash register A Mathematical problem could be asked each time they purchase a different item And also as they move from place to place, questions on history and science can be integrtated For

Figure 9 c



The third page, Shopping, is to help integrate the ideas and concepts of a neighbourhood and a community, using directives, and some problem solving skills Due to the time limit I was not able to include all the ideas that I would have liked, therefore this last page is incomplete For example as the children choose the particular gift they would like to purchase and are asked to travel, sounds of walking feet could be added and as they arrive at their destination they could be asked to solve a mathematical problem, how much change they would receive if the gift cost so much and if they had so much money, and if the correct answer was given they would hear the sound of the cash register A Mathematical problem could be asked each time they purchase a different item And also as they move from place to place, questions on history and science can be integrtated For example as the children make their way from the wool store to the card store, they may pass by a statue of a famous historian The incription would be provided and the children would be asked to provide answer of "Who is it?" from a choice of three names

To see my project type "Compass"

Figure 9 d

Figures 9 a-d. An example of complex and crowded text with scrolling effects.

On the other hand, the appropriate use of automatic operators may reduce the difficulty that the users confront in using manual operators due to problematic representations of operating knowledge Also, the user's working memory load may be reduced because the program automatically presents the texts or pictures and the user does not need to worry about how to operate the system. Therefore, users may work more smoothly if the texts are easy to understand, well formatted, and presented at a reasonable pace for the users

Project 13 also involved a high proportion of scrolling. Different from Project 12, the texts in Project 13 were short, easy to understand, and well formatted. In addition, the text on how to use procedures to draw various shapes was combined with demonstrations of the screen effects produced by those procedures (See Figure 5). Not only could the user see the procedures needed to perform the tasks and their screen effects without any interruption, but also the speed of the scrolling could be slowed down. Thus, the user could work smoothly with automatic operators in this program.

To summarize the findings from characterizing the operators, the results of this study suggest that the learning environments constructed by student teachers lacked manual operators and overused automatic operators. The learning environments particularly lacked manual operators which promoted task performance and provided flexibility for users to meet their individual needs. However, the types of manual operators designed by student teachers seemed to support the user in learning domain knowledge and system operation. In addition, the automatic operators sometimes displayed advantage when they were used appropriately. Therefore, good instructional software should combine the advantages of both automatic and manual operators so that users cannot only execute the program smoothly, but also have the opportunity to interact with the computer, as well as the freedom to choose the tasks or assistance to meet their individual needs.

Summary of the Characteristics of the Learning Environments

There are three major findings which emerged from the overall characterization of the learning environments constructed by student teachers. First, the learning environments presented domain knowledge, operating knowledge, affective knowledge, as well as knowledge for implementation about content areas, target population, and methods. Domain knowledge was the first major concern of all projects, whereas operating knowledge was the second major concern Domain knowledge presented by student teachers was mostly about facts, events, concepts, and actions and it was seldom involved principles or conditions under which the actions can be taken. Sometimes student teachers presented temporal, partial, or identical relationships, but they did not present causal and conditional relationships, or indicate "bug problems", the learner's misconceptions, or efficient problem-solving strategies that might be employed by advanced instructional programs. Further analysis of operating knowledge indicated that a program did not need to present any operating knowledge when the episodes or views were linked by automatic operators or when the domain manual operators were used to link episodes or views. However, a program must present operating knowledge to the user when a manual operator is required to link to an episode or a view. Lack of operating knowledge when a manual operator is required to link an episode or a view is the main reason for a user becoming stuck.

The second finding was that instructions, providing tasks, and providing working spaces are three major pedagogical strategies used by all student teachers. More advanced pedagogical strategies which integrated *instructions,* *demonstrations,* and *explanations,* as well as hinds for assisting task performance were also designed by some student teachers Furthermore, a few student teachers took the advantage of Logo exploratory learning environment and enabled their target learners to learn domain knowledge by exploration On the other hand, data also show that the learning environments constructed by the student teachers lacked working spaces to perform tasks or answer questions, and there was a serious shortage of evaluation and feedback on domain knowledge. In addition, insufficient tasks were presented for operating the system when relevant operating working spaces were designed The consequences of these problems were that, first, the lack of working spaces in domains led to failure to perform tasks, and second, insufficient indication of operating working spaces created difficulties for the user or even resulted in failure in program execution. The insufficient working spaces and the input to which the computer did not respond were the sources that the learning environments lacked evaluation and feedback, while the lack of evaluation and feedback in turn indicated that the learning environments constructed by student teachers were weak in assisting user's learning.

The last finding was that the learning environments constructed by student teachers lacked manual operators and overused automatic operators. Moreover, the learning environments lacked manual operators for performing tasks, for providing flexibility to meet the user's individual needs, and for providing assistance in learning.

Characteristics of the Program Structures

LogoWriter™ has special features which can execute more than one page in a program and more than one procedure on a page with or without the user's interaction. The program structures in this study, therefore, refer to both page structures and procedure structures. The program structures are also characterized by the static attributes and dynamic attributes. Static attributes here refer to single-level, linear, modular, or fragmented structures. In addition, programming utilities such as reusable procedures, conditional statements, variables and recursion are also considered as static attributes. Dynamic attributes of the program structure refer to the manual operators required in linking procedures or pages. The categories and functions of manual operators are the same as those used in characterizing the learning environments. Table 8 shows the program structures in each project produced by the student teachers, including page structures, procedure structures, and programming utilities.

Page Structures

Data in Table 8 indicate that 69% of projects produced by student teachers used linear page structure, 23% of the projects used modular page structure and 23% used fragmented pages. It is clear that there is consistency in the way the student teachers structure their pages, based on the fact that 85% of the projects used only one page structure and only 15% used two page structures.

Table 8

Page structures, procedure structures and programming utilities used in each project.

Project No.	Pages Structures			Procedure Structures				Programming Utilities			
	Linear	Modular	Fragmented	One Level	Linear	Modular	Fragmented	Reusable Procedures	Conditional Statements	Variables	Recursions
1			x	x	x	x	x		x		x
2	x			x			x				
3		x					x				
4	x	x		x							
5	x			x		x					
6		x		x	x	x	x		x	x	x
7	x			x	x	x					
8	x		x	x		x			x		
9	x			x	x	x		x			
10	x			x	x		x				
11			x				x				
12	x			x	x	x		x		x	
13	x			x	x			x	x	x	x
Percent of the Projects	69%	23%	23%	77%	54%	54%	46%	23%	31%	23%	23%

Linear page structures

The most common page structure in the programs produced by student teachers was linear. There were several versions of linear page structures due to different ways of linking the pages. The simplest version was that all pages are retrieved by the GETPAGE primitive in the first page of the program so the program did not require any inputs to link the pages, as shown in Figure 10. Another version was that the user was instructed to use the GETPAGE primitive and the page name to ret [illegible]'e the page when the previous page was finished, as shown in Figure 11. The third version of linear structures was that the user could press one key to move to the next page when the current page was finished. Once the user was on the next page, then he or she could access the previous one by pressing another key (see Figure 12). The final version of linear structure combined an automatic operator with a manual operator so that the last procedure in the previous page could automatically retrieve the following one that presented a question. When the question was presented, the program waited for a manual operator, which was a correct answer from the user. Once the correct answer was typed in, the following page was linked and a new question was presented (See Figure 13).

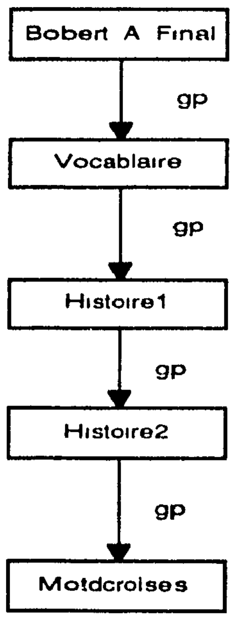


Figure 10. A linear page structure linked by automatic operators.

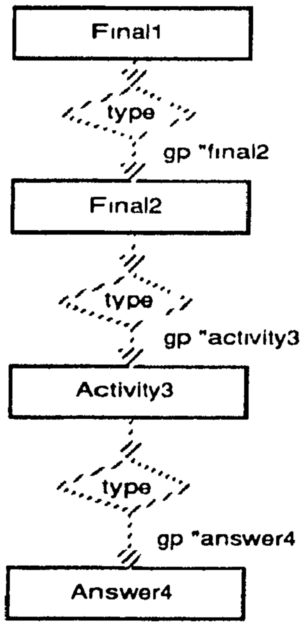


Figure 11. A linear page structure linked by manual operators.

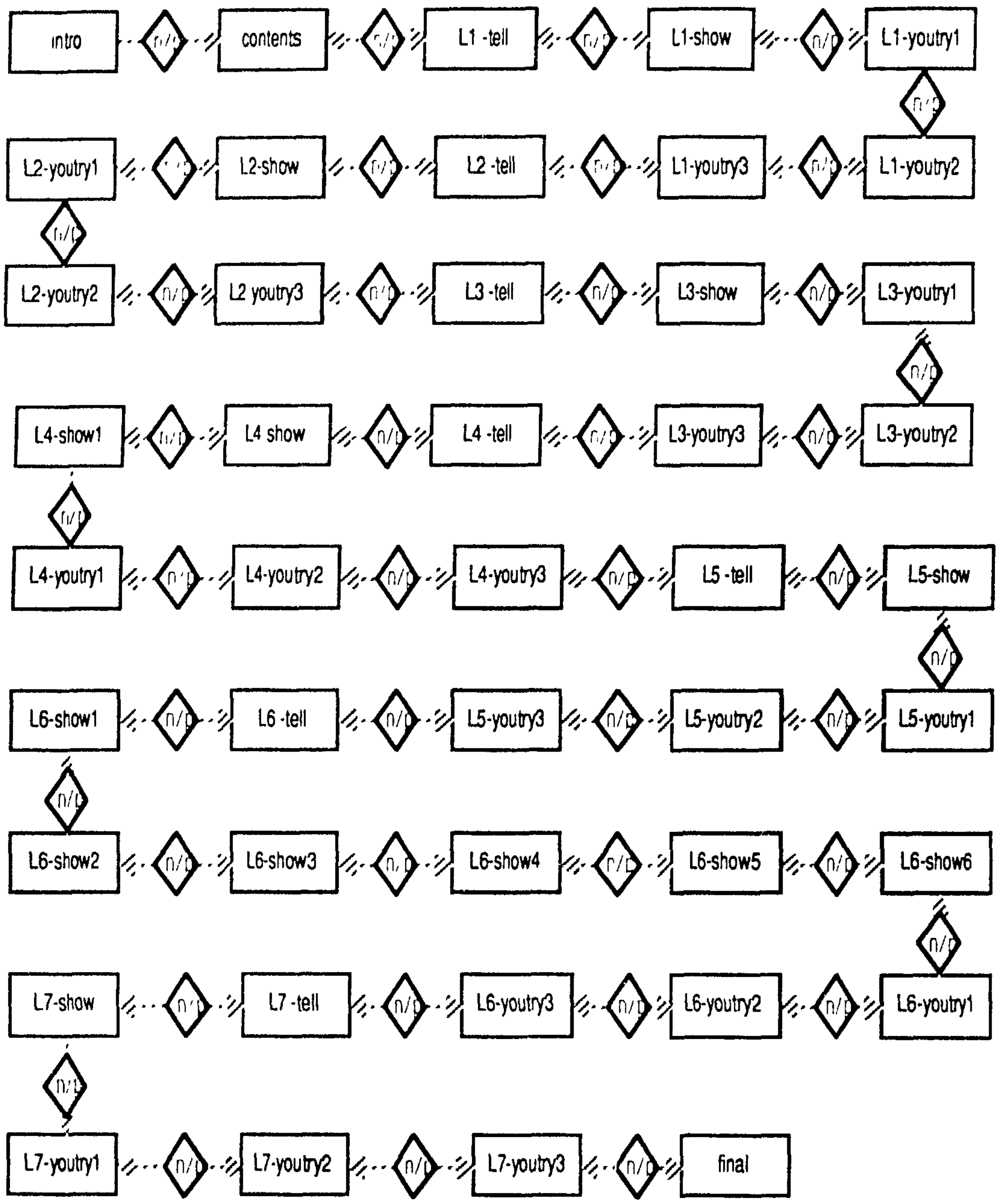


Figure 12. A linear page structure in which the direction of operation is controlled by pressing a key.

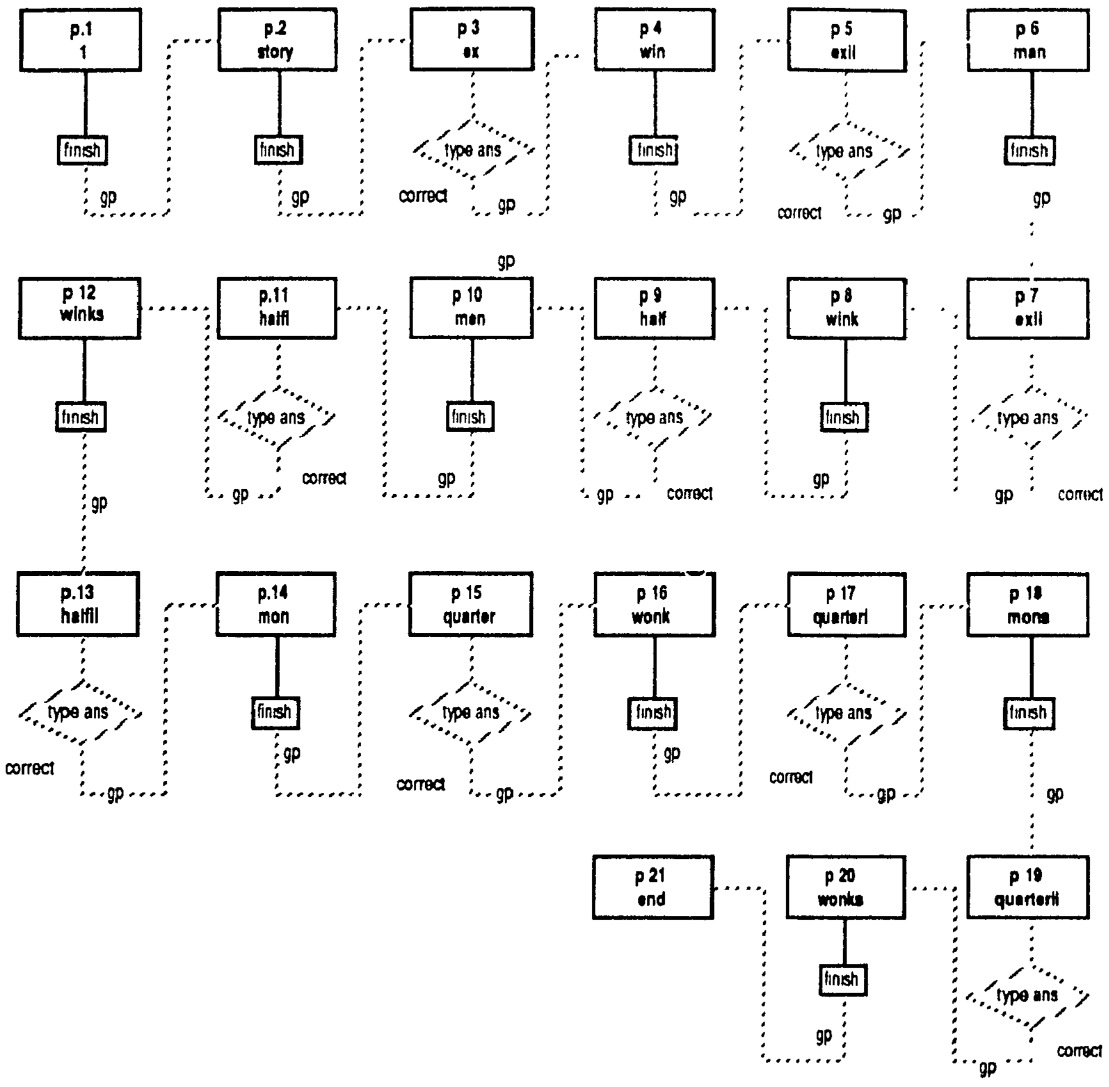


Figure 13. A linear page structure combining automatic and manual operators.

Modular page structures

The modular page structure, which is supposed to have more advantages than the linear one, had two versions in the programs produced by student teachers. In one version, the pages were structured at two main levels and the bottom ones were parallel (See Figure 14). In the second version, the main frame was modular, while the rest of the pages could be either modular or linear (See Figure 15).

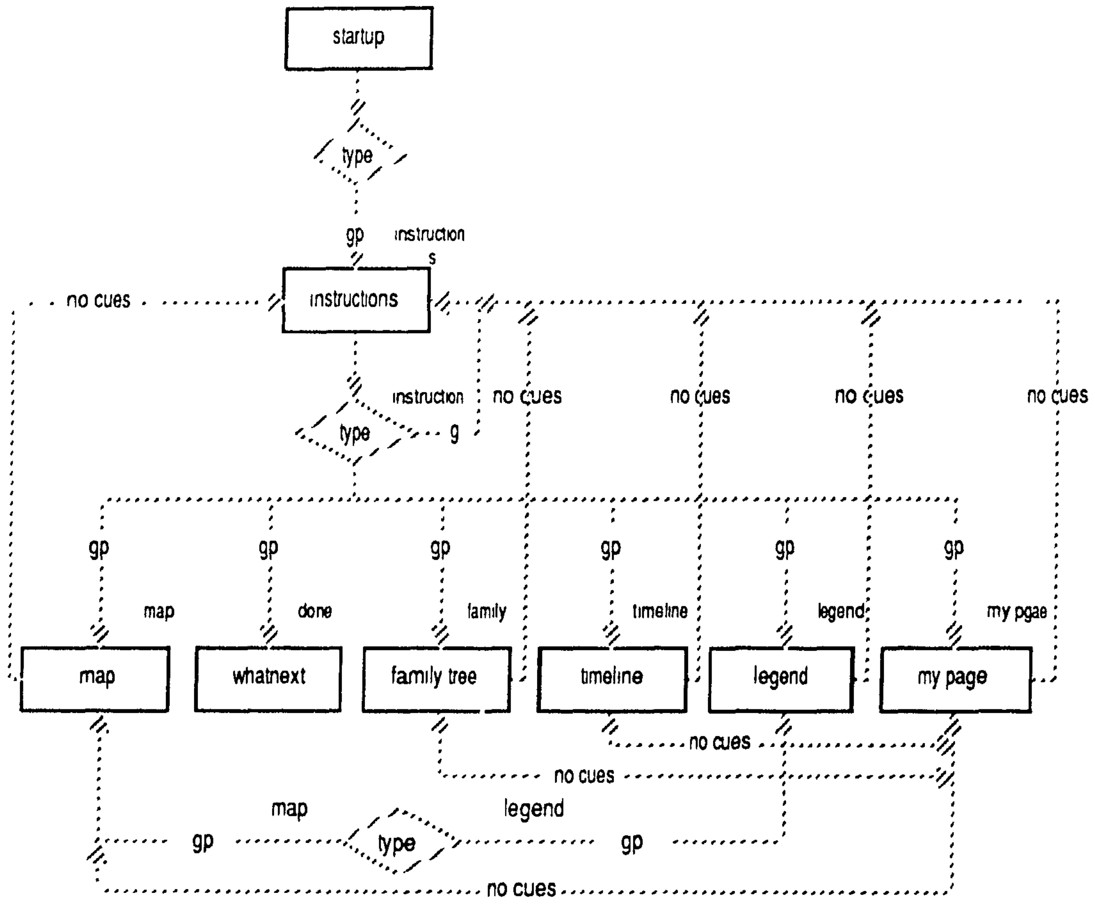


Figure 14. A modular page structure consisting of two main levels with the bottom ones parallel.

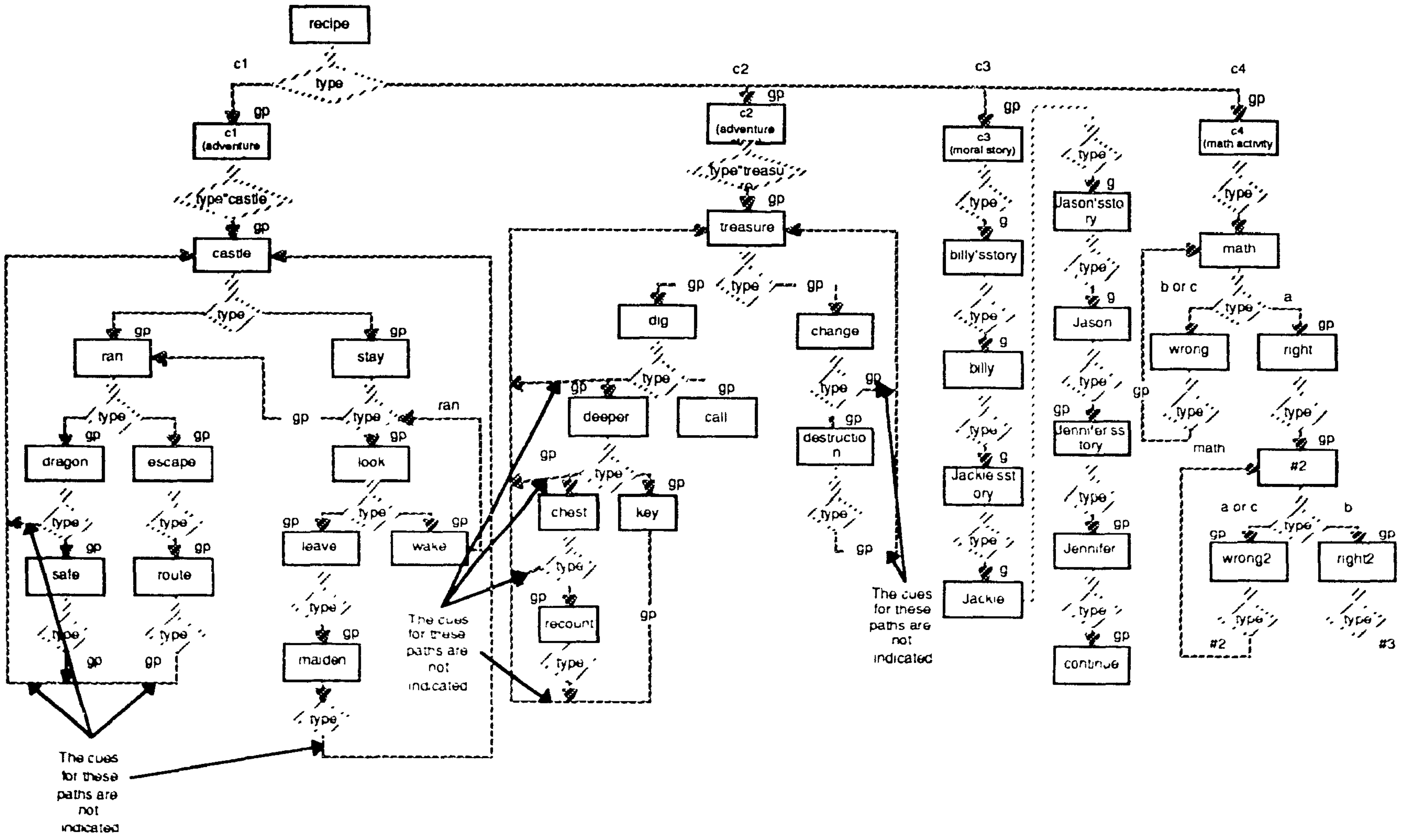


Figure15. A modular page structure

Two problems were frequently observed in the modular page structures designed by student teachers. One problem was that some bottom branches did not have paths leading to another branch, so that users had to quit the program and restart it again in order to access other branches. Another problem was that in many cases the designers did not indicate some important paths to the users even though they had designed these paths. So it seemed that student teachers experienced many problems in designing smooth modular page structures. Nevertheless, the modular page structure did enable the user to choose alternatives in execution and therefore still has some advantages over the linear one.

Fragmented page structures

Not surprisingly, some programs designed by student teachers consisted of fragmented pages in which there was no indication of how to link one page to another. Hence, the user had difficulty in executing these pages as a program.

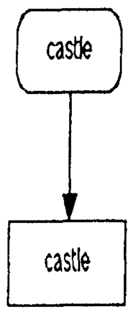
To sum up the findings from page structures, student teachers preferred to design linear page structures and they encountered less difficulties in doing so than in designing modular page structures. Some of them have developed consistent and systematic linear page structures which enabled users to answer questions and control the direction of operations. In addition, some student teachers developed modular page structures that promoted alternatives in execution so that users could choose tasks or activities. However, student teachers seemed to have trouble in designing the paths from one branch to another and further in indicating these paths to the user. The fragmented pages were either a result of uncooperative group work or an inability to design alternative page structures.

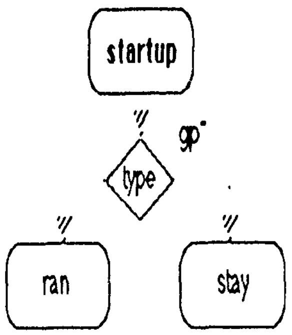
Procedure Structures

There were four procedure structures designed by student teachers. These procedure structures were single-level, linear, modular, and fragmented The data in Table 8 show that 92% of the projects involved single-level procedure structures and 54% of the projects involved either linear structures or modular ones, or both. Forty-six percent of the projects had fragmented procedures at some points.

Single-level procedure structures

The projects produced by student teachers made heavy use of single-level procedures. The simplest single-level procedure is shown in Figure 16a, where the procedure named "castle" would retrieve a page with the corresponding name. However, student teachers used single-level procedures in flexible ways with the result that several single-level procedures could accomplish complex tasks. For example, they used single-level procedures for answering multiple choice questions (Figure 16b), choosing activities (Figure 16c), and performing tasks. However, in most cases, the student teachers used more than one procedure structure. Therefore, single-level procedures were sometimes only loose-ends attached to another main procedure structure (See Figure 17), for the purpose of performing tasks, providing feedback, or retrieving a page, or were used as a tool to present the lesson.





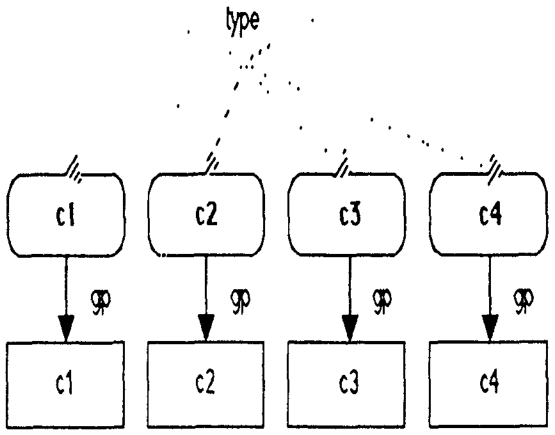


Figure 16 a **Figure 16 b** **Figure 16 c**

Figure 16. Single-level procedures used to link a page (16a), to answer a question (16b), or to choose an activity (16c).

Linear procedure structures

Figure 17 shows a linear procedure structure used by a student teacher. In this type of procedures, the subprocedures were structured in a linear way so that the information was processed in sequences. Most linear procedures structured by student teachers were combined with single-level procedures and modular procedures. Some of them also involved fragmented procedures.

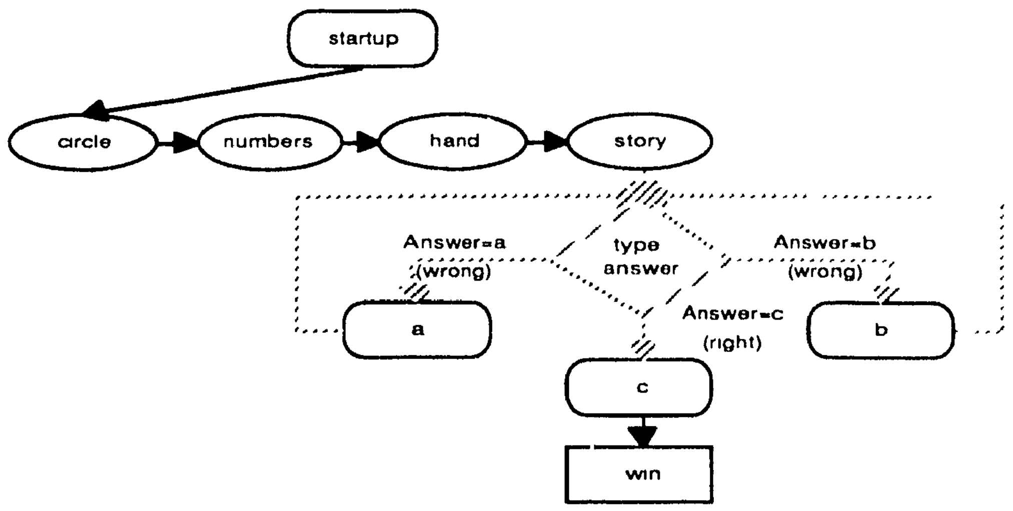


Figure 17. A linear procedure attached by single-level procedures.

Modular procedure structures

In addition to linear procedures, student teachers constructed modular procedure structures by decomposing procedures into parts, and further decomposing these parts into other parts (See Figure 18). One command phenomenon found in such a modular procedure structure is that student teachers did not design alternatives in execution so that the execution of the the was still in a sequence. For example, in Project 1 (See Figure 18), a startup procedure consisted of three procedures: "hello", "runstuff" and "goodbye". Runstuff can be decomposed into getanswerA, getanswerB, getanswerC and getanswerD, but these procedures were executed in a sequence and there was no alternative in execution. Furthermore, the modular structures constructed by student teachers usually had only three levels. Therefore, even though student teachers had developed the ability to use a decomposition technique to produce modular procedures, these procedures were still linear in their execution.

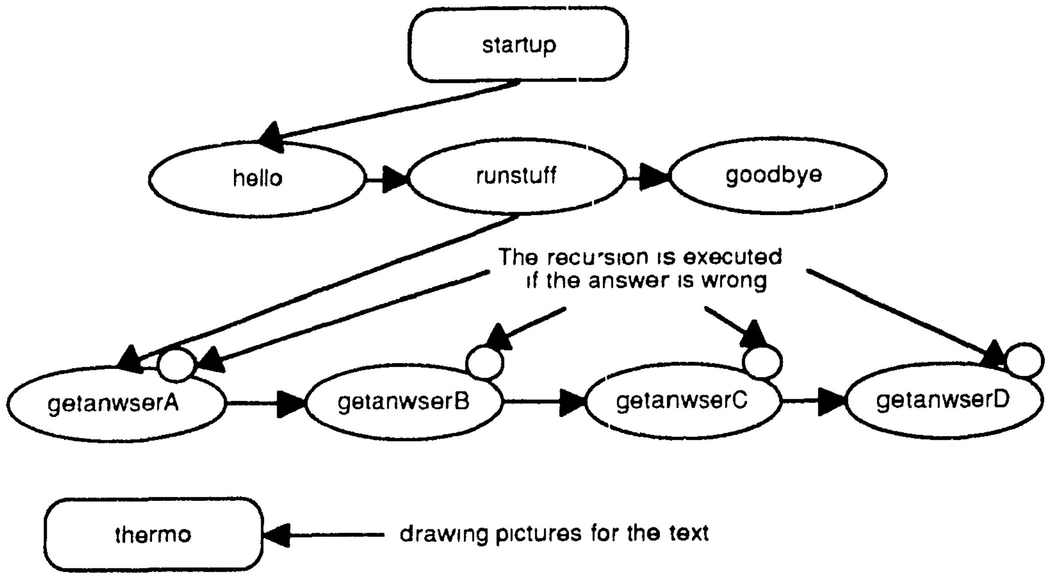

startup
hello
runstuff
goodbye
The recursion is executed if the answer is wrong
getanwserA
getanwserB
getanwserC
getanwserD
thermo
drawing pictures for the text

Figure 18. A modular procedure structure with recursion.

Fragmented procedure structures

The fragmented procedures found in the programs constructed by student teachers seemed to be some leftovers from other activities in the learning phase, so the existence of fragmented procedures is not considered a problem.

To sum up, the procedure structures found in student teacher's projects were frequently single-level procedures which were used in a flexible way so that they could accomplish complex tasks. In addition, these single-level procedures were often attached to a linear or a modular procedure as loose-ends. The modular procedures produced by student teachers showed that student teachers developed decomposing techniques to produce modular procedures, but that these modular procedures did not allow alternatives in execution and that they are linear in logic.

Programming Properties

Even though all student teachers learned programming utilities such as reusable procedures, conditional statements, variables and recursion in the learning phase and were able to use them in their exercises, data in Table 8 showed that student teachers did not frequently use programming utilities in their final projects. Only 23% of the projects used either variables or recursion or both in their final projects. In addition, another 23% of the projects used reusable procedures and 31% of the projects used conditional statements. Since the fragmented procedures were leftover from the exercises, even though 23% of the projects used variables in fragmented procedures, these projects were not regarded as using programming utilities. Therefore, 54% of the projects did not use any programming utilities in their final projects It seemed that the designers for these projects had not developed the ability to design programming utilities in their final projects. In other words, they may have been able to use programming utilities in their exercises, but they were unable to apply these techniques in a flexible manner to their own projects.

As mentioned before, LogoWriter™ has the capacity to run more than one page in a program and more than one procedure in a page, and the pages and procedures require links in executing them as programs. Therefore, interaction issues between pages and between procedures cannot be avoided. The following section will present and discuss the results from characterizing the manual operators used to link pages or procedures in terms of the types of manual operators and their functions.

Manual Operators

Table 9 shows the types and functions of manual operators located between pages and between procedures, as well as within procedures. The data in this table indicate that 47% of the manual operators designed by student teachers were located between pages, 21 5% of the manual operators were located between procedures, and only 31.5% of the manual operators were located within procedures Such high proportion of manual operators located between pages and between procedures indicate that the interactions in the learning environments constructed by student teachers in LogoWriter™ depended heavily on the ways designers structured the pages or procedures. In order to find out what these manual operators were, and for what they were used, the following sections will analyze the types and functions of the operators at different locations.

Manual operators between pages

The results in Table 9 show that 36% of the manual operators located between pages were used for operating the systems, 10% were used for answering questions, and only two percent were used for choosing activities or tasks. None of the manual operators between pages was used to perform tasks or choose assistance. The manual operators for operating the system were all located between pages. In other words, operating knowledge was needed for linking pages but not for linking procedures. In addition, most of the manual operators for choosing tasks/activities and answering questions were also located between pages.

Table 9

The distribution of the manual operators.

Distribution	Choosing tasks or activities	Choosing Task Complexity	Performing Tasks	Answering Questions	Answering Multiple Choice Questions	Choosing Assistance	Operating the System	Percent
Between Pages	1 50%	0	0	3 50%	6%	0	36%	47%
Beteen Procedures	0 50%	0	11%	4%	2%	4%	0	21 50%
Within Procedures	0	0	25%	1 50%	5%	0%	0%	31 50%
Percent	2%	0%	36%	9%	13%	4%	36%	100%

Manual operators between procedures

Eleven percent of the manual operators located between procedures were used for performing tasks, four percent were used for choosing assistance, and six percent were used for answering questions, only 0.5% for choosing tasks or activities. With respect to choosing assistance, all the manual operators for this were located between procedures.

Manual operators within procedures

Twenty five percent of the manual operators within procedures were used for performing the tasks and 6.5% were used for answering questions. None of the operators within procedures were used for choosing activities or assistance, or operating the system. The manual operators located within procedures were mainly used for performing tasks and answering questions.

To sum up the findings from the characterization of the program structures produced by student teachers, first, student teachers designed single-level, linear, and modular structures at both the page level and the procedure level. Among these structures, the most frequently used page structure was linear and the most frequently used procedure structure was single-level. Second, some student teachers developed consistent and systematic page structures in which various procedure structures were combined and functionally expanded. Third, student teachers often failed to build and later indicate the paths from branches to branches in modular page structures. When modular procedure structures were designed, student teachers always failed to design ones which would permit alternatives in execution. Finally, most student teachers were unable to use the programming utilities in their project design, even though they had all used them in the exercises.

Furthermore, the programs produced by student teachers usually had more than one procedure on a page and more than one page in a program When the manual operators were considered, most of them were located between procedures and between pages Further analysis indicates that the manual operators located at different places played different functional roles so that the programs could accomplish more complex tasks that could not be accomplished by a single procedure or a single page produced by student teachers. Therefore, the functions in the programs produced by student teachers in LogoWriter™ depended not only on the individual procedure structures and programming utilities, but also on the way the pages and procedures were structured.

The Relationships between Program Structures and the Use of Pedagogical Strategies

The characteristics of the program structures of the projects which used appropriate pedagogical strategies and those which did not were compared in order to determine whether there was a relationship between the characteristics of the program structures and the pedagogical strategies As mentioned before, Projects 6, 8, 9 and 13 had integrated instructions, demonstrations, and explanations, accompanied by tasks and working spaces and even assistance for performing the tasks, whereas Projects 4 and 7, as well as some episodes in Project 1 used a combination of providing tasks, working spaces, and immediate evaluation and feedback. Among the final projects designed by student teachers, 50% of them employed appropriate pedagogical strategies. The results in Table 8 indicate that 77% of the projects which employed appropriate pedagogical strategies used programming utilities at some points, whereas only 15% of the projects which did not use appropriate pedagogical strategies used programming utilities. Projects 6, 8 and 13 which employed appropriate pedagogical strategies all had used conditional statements, and two of them used variables and recursion. These three projects comprised 62% of the overall programming utilities for all projects. Furthermore, the results in Table 10 indicate that the projects which employed appropriate pedagogical strategies all used conditional statements and recursion. In addition, they consisted of most modular procedure structures (71%), variables (67%), and reusable procedures (67%), as well as modular page structures. Therefore, well structured programs and the use of programming utilities seemed to be associated with the use of appropriate pedagogical strategies On the other hand, the appropriate pedagogical strategies could be attained by using single-level procedures in a flexible way, as in Project 4.

Table 10

The program structures and the use of pedagogical strategies.

Pedagogical Strategies		Patterns	No Patterns
Page Structures	Linear	55%	45%
	Modular	67%	33%
	Fragmented	50%	50%
Procedure Structur	Fragmented	33%	67%
	Single-level	59%	41%
	Linear	57%	43%
	Modular	71%	39%
Programming Utiliti	Reusable Procedures	67%	33%
	Conditional Statements	100%	0
	Variables	67%	33%
	Recursion	100%	0

To sum up, the characteristics of learning environments can be described in terms of the types of knowledge presented, the pedagogical strategies used to present the knowledge, and the forms and functions of interactions. The pedagogical strategies and interactions are related to the ways that programs are structured and to that programming utilities are used in the programs. When the programs are well structured and employ the programming utilities, the designers can provide more sophisticated learning environments. However, the designers can also construct a smooth program without using sophisticated programming skills when the user's cognitive learning needs in the learning processes are considered. Without the consideration of human factors in the design of the instructional programs, the designers may produce the program with which the user encounters a lot of difficulty and even failure in interacting.

Chapter 5

CONCLUSION

This chapter first presents a summary of the findings from this study and discusses the implications for instructional software development. Next, the implications for studying expertise in the domains of instructional software design and human teaching are presented and several issues pertaining to studying Logo environments are raised. Finally, some possible directions for future research are considered.

Summary of the Research Findings

This study was concerned with developing a methodology for identifying the cognitive, pedagogical, and computational characteristics of computer-based learning environments. The methodology developed provides precise descriptions of these features of the learning environments. By considering the features of the learning environments and their effects on the user, a diagnostic evaluation can be made of the usability and constraints of a given system.

The methodology developed in this study allows for the investigation of *different types of knowledge* presented in learning environments, the *pedagogical strategies* used to present this knowledge, and *the forms* and *functions* of *interactions* that the learning environments elicit (e.g., the task activities of the user). In addition, this study characterized the computational characteristics of programs in terms of *single-level, linear,* and *modular structures*, as well as other programming properties.

Three major types of problems were identified in creating representations for knowledge. The first type of problems was related to domain knowledge. That is, the programs presented neither causal or conditional relationships that are important components of propositions and schemata of domain knowledge, nor principles that reflect the nature of the domains. In addition, these programs provided the user with neither the difficult learning tasks that they often make mistakes on nor the efficient strategies for problem solving. Therefore, the domain knowledge conveyed by student teachers was simple, concrete, and isolated. The second type of problems related to operating knowledge. The results showed that certain programs partially lacked operating knowledge, or contained incomplete or inaccurate instructions. The problems in presenting operating knowledge created impasses for the users to operate the system. The third type of problems is that incoherent, long, and ambiguous text was sometimes used in representing knowledge. When the text in a view was poorly formatted and linked by automatic operators without providing the user any control, it was particularly problematic.

In terms of pedagogical strategies, the learning environments constructed by student teachers employed three basic pedagogical strategies: giving instructions, presenting tasks, and providing working spaces Some student teachers often elegantly integrated instructions, demonstrations, and explanations, showing the user not only the procedures needed to perform a particular task, but also the screen effects produced by each procedure and some potential problems. In addition, these programs sometimes provided reminders to assist the learner to perform the tasks. Furthermore, a few designers took the advantage of Logo exploratory learning environment and enabled the learners to construct and invent new products for learning domain knowledge. Such combinations are called modeling, scaffolding, and exploration by Collins, Brown and Newman (1989), as characteristics of ideal learning environments.

Several problems were detected in the pedagogical strategies used in the learning environments developed by the student teachers. These problems can be interpreted in terms of difficulties in representing the perspective of the learner. The first problem was that some learning environments lacked sufficient working spaces for performing tasks and answering questions. Consequently, the user would not be able to perform tasks or answer questions. The second problem was that insufficient tasks were presented for using the working spaces for operating knowledge. In this case, the user would often encounter an impasse in executing the program. The third problem was that there was a serious shortage of evaluation and feedback provided to the user. Finally, there was a lack of congruence in the pedagogical strategies used in the learning environments constructed by student teachers. This was reflected by the lack of continuity between tasks or questions and working spaces, and inconsistencies between the instructions about how to perform tasks and relevant working spaces. The consequences of these problems in using pedagogical strategies are that they could increase the user's working memory load, and thus make it difficult for the user to understand and remember the instructions or tasks. These problems greatly reduced the efficiency of the learning environment.

In terms of *interactions* two findings are important. On the one hand, there were insufficient user-computer interactions. In particular, the programs lacked interactions which are used for learning domain knowledge, or for providing the user with flexibility in controlling the learning process. On the other hand, student teachers provided interactions that seemed to support learning of domain knowledge and of the system operation. That is, the types of manual operators designed by student teachers for operating the system were easy to use, those for performing domain tasks provided flexibility for task performance, and those for answering questions promoted understanding.

In terms of *program structures*, student teachers did not construct many complex structures (i.e., modular procedures or pages), or did not construct them successfully. For example, when they designed modular page structures, they often failed to design all the paths needed between branches and further to indicate these paths to the user Student teachers preferred simple structures (e.g., single-level procedures, linear pages), but achieved a high level of consistency and systematicity. The single-level procedures were used in a very flexible way, for example combining them with linear or modular procedures. Combined procedures appear to enhance greatly the program's function. Student teachers designed single-level, linear, and modular procedure structures. Single-level was the most commonly designed procedure.

Only a few subjects used the programming utilities, such as conditional statements, recursion, variables in their project design even though they had all learned how to use these utilities in the course. This study found that modular program structures and use of programming utilities, in particular recursion and conditional statements, are strongly related to the use of effective pedagogical strategies. This phenomenon will be discussed in a later section.

The findings from this study indicate that this methodology has the potential to identify the cognitive, pedagogical, and computational characteristics of the learning environment. The task activities (i.e., the manual operators) can be analyzed in great detail in terms of the types and functions of the operators, and the user's cognitive needs for performing these tasks are also considered in the context of the dynamics of the learning environments. Therefore, the individual evaluating the instructional software can diagnose the strengths and weaknesses of a learning environment and determine whether and why the system is well suited for system operation and for promoting learning subject matter knowledge.

Implications for Instructional Software Development

The research presented in this thesis has implications in two areas of instructional software development: the study of instructional software interface and instructional software evaluation.

Implications for Studying Instructional Software Interfaces

This research is related to the study of human-computer interaction because the method presented describes precisely the instructional software interface and its effects on the learner for performing tasks elicited by the system In particular, the method reveals the different types of interactions promoted in a learning environment, as well as the functions of these interactions.

The methodology developed in this research differs in two significant ways from the methods that are used in research on human-computer interaction. The first difference is that it puts more emphasis on the system behaviour (for example the types and functions of different manual operators) than on the user's behaviour (for example the process of selecting among different manual operators). The second point of departure is that the usability and constraints of the system are characterized in terms of cognitive and pedagogical content rather than in terms of the measures such as time required to perform tasks or number of errors made by the user.

In characterizing the cognitive, pedagogical and computational features of computer-based learning environments, this study has proposed a framework for precise description of instructional software interface This interface consists of three principal components: learning environments, tasks, and users. This approach adopted the idea that in order to investigate human-computer interaction, one needs to analyze the dynamic interaction among users, tasks and computer systems (Bennett, 1972, 1979; Card, Moran & Newell, 1983; Chapanis, 1991; Eason, 1981; Shackel, 1991). Moreover, this study has taken into consideration of the special properties of instructional software.

Computer-based learning environments

The computer systems are regarded as physical devices used to provide learning environments. This study proposed that computer-based learning environments have three types of attributes: cognitive and pedagogical, computational, and physical attributes. The *cognitive and pedagogical attributes* are reflected by the display of a program (e.g , text, graphics, speech and animation) and user-computer interactions promoted These attributes were described in terms of different types of knowledge, pedagogical strategies used to present the knowledge, and the forms and functions of interaction. The cognitive and pedagogical attributes of a learning environment have the most significant impact on learning subject matter knowledge.

Computational attributes refer to the characterization of structures in a microworld in the sense proposed by Groen (1984). The structure consists of a set of states and transformations between states. A well constructed microworld has the following three properties: a) the transformations should be modular; b) a transformation can be undone to go back to the previous state; and c) the transformations and transformational structures have representations analogous to operations and procedures in the real world. The computational attribute in this study was described in terms of the modularity of the program units (i e , structures of pages and procedures) and the transformations between them (i.e, links between pages and procedures). In addition, programming utilities were also considered. Computational attributes determine the ease of constructing or inventing products, such as producing a computer program

Physical attribute is reflected by physical devices such as input-output devices (i.e., a light pen, handwriting input, a touchscreen, a voice synthesizer, and video display terminals, etc). Physical attributes can be described in terms of text, sound, pictures, colours, and so on. Physical devices differ in their compacity in providing displays or accepting input, thus the physical attribute may have also an impact on user's learning.

In terms of the *hierarchical organization* of a learning environment, a learning environment consists of a set of episodes (i.e., a sequence of lessons or a set of exercises) which are composed of sequences of views. Each view consists of view space and command space. The view space refers to the static attributes (i.e., types of knowledge presented, the pedagogical strategies used to present this knowledge) and the command space refers to the dynamic attributes of the information (i.e., the interactions prompted). The decomposition of a learning environment permits a fine-grained characterization of important features of a learning environment and a precise diagnosis of its usability and constraints.

Tasks

What differs a computer-learning environment from other computer environments (e.g., using a wordprocessor, spreadsheet, or drawing program) is that the tasks are strongly constrained by what is promoted by the learning environment. The user's task activities in computer-based learning environments involve learning of subject matter knowledge and system operation, whereas in other computer environments the user needs only to learn how to use the computer to perform tasks.

The learning activities or tasks were referred to the manual operators promoted by the learning environment. There are three levels of task descriptions for these activities: the top level refers to the global goal that the user is supposed to achieve; the next level is the tasks that a user is supposed to perform in a lesson or a set of exercises; the bottom level, which is referred to the manual operators, corresponds to the sequence of task activities that the user needs to perform. This study categorizes the types of the operators and their functions and assesses whether the operators are easy to use in operating the system and whether they are supportive in promoting learning of subject matter knowledge and providing the learner flexibility to control the learning process.

Learners

This study considered the users' cognitive needs in terms of their basic cognitive resources (e.g., working memory), the cues needed for system operation (e.g., the information that the novice users need for executing the program), and the supports required for learning domain knowledge (e.g., pedagogical strategies that facilitate learning; manual operators that promote understanding and flexibility).

Instructional software interface needs to adapt to the following conditions: the characteristics of the population (e g., children vs. adults), the subject matter (e.g , arts, science, etc.), the subjects' stage of learning (i.e., novices, intermediates), and the types of the tasks (e.g., learning subject knowledge or constructing products). Further studies are needed to identify the kinds of interface required for these differential needs.

Implications for Instructional Software Evaluation

This research has implications for the evaluation of educational software. The method presented allows one to identify the cognitive and pedagogical characteristics of instructional software. Such descriptions are required in order to evaluate instructional software from the point of view of the knowledge that it implements, in particular by identifying the possible factors that account for the effectiveness of a given learning environment. The application of this method should allow researchers in the field to provide evaluations of instructional environments that are more precise and, therefore, can serve to improve the design of future environments.

The methodology developed in this research allows us to identify the significant features of the learning environment that affect learning. This provides a basis for determining the effectiveness of the learning environment for performing a set of the tasks. Other methods such as experiment comparisons and meta-analysis (e.g., Kulik & Kulik, 1987;

Roblyer, Castine, & King, 1988), are more limited because they only provide information regarding whether the instructional software is effective but do not identify the factors that determine the effectiveness.

This method direct one's attention towards the educational properties of instructional software that are not normally accounted for by most measures of usability of human-computer interfaces. More importantly, this method identifies the cognitive and pedagogical characteristics which are important in improving instructional software. The results from previous research suggest that the typical measures of success rate, time, and error do not present sufficient information for improving instructional software (e.g., Chapanis, 1991; Shackel, 1981, 1991).

This study suggests that evaluation should look at not only the content and the representation of the subject matter knowledge, but also the appearance of the operating knowledge and pedagogical strategies used to present various types of knowledge. The problems in operating knowledge will create difficulty or even failure for the user executing the program.

Another important criterion for evaluating the usability of instructional software is the quantity and quality of the interactions. Guidelines for instructional software design and evaluation usually suggest that a good system should maximize the interactions, without indicating what kind of interactions should be maximized. This study specifically indicates that efficient instructional software should maximize the interactions that promote the user's understanding and development of cognitive skills, as well as provide the user with flexibility to perform the tasks and enable the user to choose activities, task complexity and various types of assistance. However, the software should minimize the number and complexity of interactions required for operating the system.

Implications for Studying Expertise in Instructional Software Design and Human Teaching

In order to investigate the differences between experts and novices, it is necessary to examine both the cognitive processes that a programmer goes through in producing a program and the program that the programmer devoted all his knowledge and skills to produce. The identification of the cognitive and pedagogical characteristics of the instructional software provides a means of studying expertise in the domain of instructional software design. This method can also be modified for studying expertise and for addressing a wide range cognitive and pedagogical issues involved in human teaching. Therefore, there are two major implications of this research for instructional design· the study of expertise in instructional software design and, more generally, the study of instruction.

The Knowledge and Skills Reflected in the Final Products of Programming

Previous research has focused on the programming processes which coordinate and display various knowledge and skills. These studies found that novice programmers differ from expert programmers in various ways, such as the representation of programming knowledge (Adelson, 1981, 1984; Jeffries, Turner, & Polson, 1981; Linn, 1985; Samurcay, 1985; Schneiderman & Mayer, 1979; Soloway, 1984), the strategies used in programming (Adelson & Soloway, 1985; Jeffries, et al., 1981), and other cognitive abilities (Schneiderman, 1976, 1980).

The present study found that there are cognitive, pedagogical, and computational characteristics evident in the final products of programming. These characteristics can be summarized as below:

- Student teachers represented subject matter knowledge in a concrete and isolated manner. The overall representation of various types of knowledge was sometimes incoherent

Even student teacher had shown some knowledge in the domain they were trying to teach through programming, most of them only presented isolated facts, events, and concepts. Only a few student teachers introduced temporal, partial and identical relationships, but they did not include causal or conditional relationships that are more important in developing propositions and schemata. In addition, the programs provided the learners with neither the difficult learning tasks that the learners often make mistakes on nor efficient strategies for problem solving. Therefore, the domain knowledge found in the learning environments constructed by student teachers was concrete and isolated.

In presenting various type of knowledge, student teachers often prematurely introduced one type of knowledge before the previous one was ended appropriately. Consequently, the overall knowledge sometimes lacked coherence. Furthermore, this study indicates that student teachers had considerable operating knowledge but could not effectively apply this knowledge. For example, they sometime did not present the operating knowledge in the view when the user needed to have the cues to operate the system, although they were able to present cues in other views Such inconsistency in presenting operating knowledge may be due to the failure in representing the user's perspective.

- Some student teachers developed the skills to combine the pedagogical strategies to convey domain knowledge efficiently whereas others still lack such skills.

Certain student teachers have developed the skills to combine pedagogical strategies to explicitly convey subject matter knowledge to the user. These skills might be a kind of characteristics of expert behaviour in instructional software design. On the other hand, some student teachers have not developed the skills to use appropriate pedagogical strategies. For example, they provided insufficient working spaces for performing domain tasks or answering questions, and providing insufficient tasks for using working spaces for system operation. In addition, there was a noticeable incoherence between tasks or questions and working spaces, inconsistency between the instructions about how to perform the tasks and the working spaces, and a lack of continuity in the presentation of the tasks. There was insufficient evaluation and feedback. The inability to use appropriate pedagogical strategies greatly reduces the usability of the learning environments.

- Not all student teachers seemed aware of the ease and effectiveness of the interactions.

Some student teachers have designed the types of interactions which promote understanding of domain knowledge and which provide more flexibility for the user to perform tasks. Several student teachers attempted to reduce the complexity involved in operating the system. These findings indicate that certain student teachers have developed knowledge about the usability and learnability of human-computer interface.

Many of the problems found in the learning environments constructed by student teachers are partially due to their lack of consideration of the user's task activities and the related cognitive needs in performing these tasks Since LogoWriter™ is a relatively simple environment, the student teachers may fail to consider the user's needs for learning how to operate a system. For example, programs frequently lacked cues for system operation although student teachers indicated an ability to design such cues. These problems reduced the efficiency of the program and were avoidable.

- The modular structures designed by student teachers lacked paths from one branch to another or alternatives in program execution.

Previous studies (Carver, 1987; Kurland, Clement, Mawby & Pea, 1986; Soloway, 1984) often indicated that students did not engage in problem decomposition and only produced linear programs. This study showed that student teachers designed a high ratio of single-level and linear structures but some student teachers had also developed the ability of decomposing and designing modular structures. However, the problem was that they often failed to design all paths needed from one branch to another and to indicate these paths to the user when modular page structures were implemented. Similarly, they did not design alternatives in the execution when the modular procedure structures were used. This finding indicated that although student teachers have developed the ability for decomposing and designing modular program structures, they were unable to interrelate the decomposed parts as a whole.

- The inability to apply to their projects the programming utilities that they used in exercise phases implies that student teachers possess inert knowledge.

Although all student teachers used conditional statements, reusable procedures, variables, and recursion in their exercises, only a few applied them in their projects This suggests that student teachers might know how to design these programming utilities, however, they did not learn the conditions under which the programming utilities can be applied. Therefore, the knowledge of the programming utilities still stays "inert" when the conditions for applying such knowledge are provided.

- Is there a balanced development for student teachers in constructing program structures and designing good pedagogical strategies?

This study also found that modular structures of programs and the use of programming utilities, in particular recursion and conditional statements were related to the use of good pedagogical strategies. The knowledge required to design modular structures, recursion, and conditional statements is programming knowledge, whereas the knowledge required for designing good pedagogical strategies is teaching knowledge. How can we account for this finding? There are three explanations that can be made. The first explanation is that there is a parallel development for student teachers in constructing program structures and designing good pedagogical strategies. Adelson and Soloway (1988) indicated that balanced development between domain-specific knowledge in particular application and domain-independent design model was frequently found in experts behaviour. If the parallel development found in novice behaviour is what was called balanced development by Adelson and Soloway, this finding implies that the novices have begun to develop a kind of expert behaviour at a certain point. The second explanation is that the design of superior pedagogical strategies requires relevant program structures and this prompts student teachers to apply a wider range of programming techniques. The last explanation is that the transformations and transformational structures have natural representations as operations and procedures in the real world (Groen, 1984). Recursion and conditional statements have numerous analogs in the real world. In the case of instructional design, the natural representations are pedagogical strategies.

The Study of Teaching Expertise

The method presented in this research can be extended to the study of a wider range of instructional environments or contexts, including human teaching and more traditional materials.

The methodology developed in this study can be modified for identifying the cognitive and pedagogical characteristics of human teaching processes. The sharing of the same research method in studying expertise in ITS and natural teaching could promote a promising collaboration in these two areas. That is, the findings from the study of expertise in human teaching can be directly applied to developing efficient ITSs, whereas the design of ITSs provides a computational model to test and improve the teaching theories developed in the contexts of human teaching.

Implications for Logo studies

The primary function of Logo is as a learning environment. Papert argued that Logo is an instrument that can be used by teachers and learners. It can be used in many different ways and it can have very different effects, depending on how it is used (Papert, 1986). This study applied the primary function of Logo to student teachers constructing other learning environments in which children can learn subject matter knowledge, and further characterized what was constructed by using this tool. The results indicate that Logo is a unique learning tool by which student teachers can develop teaching skills in the processes of designing instructional programs, and that Logo can also be used as a research tool for testing theoretical hypotheses.

A Learning Tool

By characterizing the instructional programs, this study identified some relatively sophisticated pedagogical strategies developed by student teachers using Logo. These pedagogical strategies, which are called modeling, scaffolding, and exploration by Collins, Brown and Newman (1989) as characteristics of ideal learning environments provide good supports for children learning subject matter knowledge. For example, modeling strategy (i e., the pedagogical strategies integrated instructions, demonstrations, and explanations) can help the children visualize the abstract concepts and build conceptual model. Scaffolding strategy (i.e., the designer provides assistance or access to the assistance when children perform tasks) can minimize the difficulty that children might face in performing task, and the exploration facilities enable children to construct or invent products (e g., computer programs, drawing) Therefore, the characterization of the learning environments does not only provide a clear description of the cognitive, pedagogical, and computational features of instructional program produced by student teachers, but also clarifies what can be learned in Logo environments. It provides evidence that the users of Logo can develop the type of cognitive skills that might be the characteristic of expert behaviour in instructional software design. Therefore, the findings from this study do not support the conclusion that the subjects cannot develop the kinds of cognitive skills in Logo programming that are the characteristics of expert programmers, or develop a model of computer function that would enable them to write useful programs (Kurland, Clement, Mawby, & Pea, 1986; Pea & Kurland, 1984; Rampy, 1984). Instead, the findings support the claim that Logo environments create the context where other learning can take place (Papert, 1986)

Furthermore, this study indicates that Logo provides a computational environment which is not only inherently mathematical as Hoyles and Noss (1992) indicated, but its easily-decomposing computational representations and debugging facilities also enable student teachers to develop the skills of teaching. Therefore, Logo is appropriate for a wider range of learning and learners.

A Research Tool

In this study, Logo was not only used as a medium for learning purposes, it was also used as a research tool for several research purposes. In particular, it was used to develop a methodology for characterizing the cognitive, pedagogical, and computational characteristics of the learning environment. It was also used as a tool to build a framework for understanding instructional software interface. In addition, it was used to test the hypothesis that the final products of programming can provide insight into the designer's knowledge and skills pertaining to the cognitive and pedagogical characteristics in instructional software design. The results showed that these expectations of Logo were achieved.

Implications for Providing Instruction in the Development of Instructional Software

The characterization of learning environments and program structures has revealed both strengths and weaknesses of the instructional programs produced by student teachers. This has implications for providing instruction in the development of educational software. Such instruction can take into account the common problems that novices have (e.g., not emphasizing causal and conditional relationships; not taking into consideration of the user's perspective), and support students in their efforts to focus on these difficult aspects of instructional software development. Students should be informed of both the typical problems and the elegant patterns found in instructional programs. This could help students avoid the problems in presenting knowledge and develop the ability to design efficient instructional programs.

Limitations of This Study

There are numbers of limitations to this research. The most significant one is that the instructional programs used as data source in this study were relatively simple. Such simplicity of the programs limites this study to display the potential of the methodology developed for charactering learning environments. Another deficiency is related to the composition of the sample which limites a comparison of the characteristics of the programs produced by programmers at various levels; the group used was inexperienced in both Logo and computers and there was no contrasting alternative group (e.g., more advanced instructors or programmers). In addition, the focus of this research was exclusively on the final products developed by student teachers.

Further Research

The methodology developed in this study is complementary to most methods previously used in studies of instructional software effectiveness and investigations of programming expertise Therefore, further research can concentrate on integrating these different approaches to achieve different objectives. For example, the further study on programming expertise can examine both the cognitive processes and the products of programming. In addition, the "good" patterns identified in the learning environments can be tested by experimental studies. The patterns that are validated can then serve as a basis for developing guidelines for evaluating and designing programs

There are several directions for this research. Briefly, this methodology can be applied to evaluating various types of instructional software or to conducting novice-expert studies in developing instructional software in order to identify the cognitive and pedagogical characteristics of good instructional software or expertise in instructional software design. This method can also be applied to the study of expertise in human teaching. The information regarding efficient human teaching or computer instructional programs can be used for evaluating and improving CAI or ITS environments. It is reasonable to assume that the refinements will be needed to apply this methodology to the study of expertise and instructional software evaluation.

REFERENCES

Ad^elson, B. (1981). Problem solving and the development of abstract categories in programming languages. *Memory and Cognition, 9,* 422-433.

Adelson, B. (1984). When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Leaning, Memory and Cognition, 10,* 484-495.

Adelson, B. (1985). Comparing natural and abstract categories: A case study from computer science. *Cognitive Science, 9,* 417-430.

Adelson, B., & Soloway, E. (1985). The role of domain experience in software design, *IEEE Transactions on Software Engineering, 11* (1 1), 1351-1360.

Adelson, B., & Soloway, E. (1988). A model of software design. In M. T. H. Chi, R. Glaser, & M. J. Farr (Eds.), *The nature of expertise* (pp. 185-128). Hillsdale, NJ: Lawrence Erlbaum.

Abelson, H., Sussman, G. J., & Sussman, J. (1985). *Structure and interpretation of computer programs.* Cambridge, MA: MIT Press.

Anderson, J. R., Farrell, R., & Surers, R., (1984). Learning to program in LISP. *Cognitive Science, 8,* 87-129.

Balzert, H. (1988). Input-output devices for human-computer interaction. In H.J. Bullinger, & R. Gunzenhauser (Eds.), *Software ergonomics: Advances and applications.* Chichester, England: Ellis Horwood.

Bennett, J.(1972). The user interface in interactive systems. *Annual Review of Information Science and Technology, 7,* 159-196.

Bennett, J. L. (1984). Managing to meet usability requirements: Establishing and meeting software development goals. In J. Bennett, D. Case, J. Sanelin, & M. Smith (Eds.), *Visual display terminals* (pp. 164-184). Engelwood Cliffs, NJ: Prentice-Hall.

Bonar, J, G., & Soloway, E. M. (1985). Programming knowledge: A major source of misconceptions in novice programmers. *Human-Comupter Interaction, 1* (2), 133-161.

Bouchard, L. & Emirkanian, L. (1984). Use of Logo in the teaching of french. In R. J. Sorkin (Ed.), *Proceedings of the Logo 1984 National Conference* (pp. 158-158). Cambridge, MA: Massachusetts Institute of Technology.

Breuleux, A. (1992). *Educational and psychological assumptions in computer-based learning environments* Unpublished manuscript. Montreal, McGill University, Laboratory of Applied Cognitive Science.

Briskman, D. (1984). Logo and physics. In R. J. Sorkin (Ed.), *Proceedings of the Logo 1984 National Conference* (pp. 158-158). Cambridge, MA: Massachusetts Institute of Technology.

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

Bull, G. (1983). Talking with Logo: Logo in speech, hearing, and language. In R. J. Sorkin (Ed.), *Proceedings of the Logo 1984 National Conference* (pp. 158-158). Cambridge, MA: Massachusetts Institute of Technology.

Bullinger, H. J. (1988). Principles and illustrations of dialogue design. In H. J. Bullinger, & R. Gunzenhauser (Eds.), *Software ergonomics: Advances and applications* (pp. 13-25). Chichester, England: Ellis Horwood.

Burns P. K., & Bozeman, W. C. (1981). Computer-assisted instruction and mathematics achievement: Is there a relationship? *Educational Technology, 21* (10), 32-39.

Card, S. K., Moran, T. P., & Newell, A. (1980). Computer text-editing: An information-processing analysis of a routine cognitive skill. *Cognitive Psychology, 12,* 32-74.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human-computer interaction.* Hillsdale, NJ: Lawrence Erlbaum.

Carroll, J. M., & Mack, R. L. (1984). Learning to use a word processor: By doing, by thinking, and by knowing. In M. Schneider (Ed.), *Human factors in computer systems* (pp. 13-52). Norwood, NJ: Ablex.

Carroll, J. M., & Olson, J. R. (Eds.) (1987). *Mental models in human-computer interaction: Research issues about what the user of software knows* Washington, DC: National Academy Press.

Carver, S. M. (1987). *Transfer of Logo debugging skill: Analysis, instruction and assessment* Unpublished doctoral dissertation, Carnegie-Mellon University, Pittsburgh, PA.

Chapanis, A. (1981). Interactive human communication: Some lessons learned from laboratory experiments. In B. Shackel (Ed.), *Man-computer interaction Human factors aspects of computers and people.* Rochville, Maryland: Sijthoff and Nordhoff.

Chapanis, A. (1991). Evaluating usability. In B.Shackel, & S.J. Richardson (Eds.), *Human-factors for informatics usability* (pp. 21-38). Cambridge, England: Cambridge University Press.

Chase, W. G., & Ericsson, K. A. (1982). Skill and working memory. In G. Bower (Ed.), *The psychology of learning and motivation.* (pp. 2-58). New York, NY: Academic Press.

Chi, M. T. H., Feltovich, P. & Glaser R. (1981). Categorisation and representation of physics problems by experts and novices. *Cognitive Science, 5,* 121-152.

Clancey, W. J., & Soloway, E. (1990). Artificial intelligence and learning environments: Preface. *Artificial Intelligence, 42,* 1-6.

Clements, D. H., & Gullo, D. F. (1984). Effects of computer programming on young children's cognition. *Journal of Educational Psychology, 76,* 1051-1058.

Collins, A., Brown, J. S., & Newman, S. E. (1989). Cognitive apprenticeship: Teaching the craft of reading, writing, and mathematics. In L. B. Resnick (ed.), *Knowing, learning, and instruction.* Hillsdale, NJ: Lawrence Erlbaum.

Corbett, A. T., & Anderson, J. R. (1991). LISP intelligent tutoring system: Research in skill acquisition. In J H Larkin, & R W Chabay (Eds.), *Computer-assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches* (pp 73-108). Hillsdale, NJ: Lawrence Erlbaum.

Criswell E. L. (1989) *The design of computer-based instruction.* New York, NY: Macmillan.

Dalbey, J., Tournaire, F., & Linn M. C. (1986). Making programming instruction cognitively demanding: An intervening study. *Journal of Research in Science Teaching, 23.*

Dale, E. (1984). Logo as a tool for studying physics. In R. J. Sorkin (Ed.), *Proceedings of the Logo 1984 National Conference* (pp. 160-160). Cambridge, MA: Massachusetts Institute of Technology.

deGroot, A. (1966). Perception and memory versus thought: Some old ideas and recent findings. In B. Kleinmuntz (Ed.), *Problem solving* (pp. 19-50). New York, NY: Wiley.

Detienne, F., & Soloway, D. (1989). Program understanding as an expectation driven activity. In G. Salvendy & M. J. Smith (Eds.), *Designing and using human-computer interfaces and knowledge-based systems.* Amsterdam: Elsevier.

Diaper, D. (1989). Task analysis for knowledge descriptions (TAKD); the method and an example. In D. Diaper (Ed.). *Task analysis for human-computer interaction* (pp. 108-159). Chichester, England: Ellis Horwood.

Eason, K. D. (1981). A task-tool analysis of manager-computer interaction. In B. Shackel (Ed.), *Man-computer interaction: Human factors aspects of computer and people* (pp. 289-307). Rochville, Maryland: Sijithoff and Noordhoff.

Edwards, L. D. (1992). A Logo microworld for transformation geometry In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp. 127-155). Cambridge, MA: MIT Press.

Ericsson, K. A., & Smith, J. (1991). Prospects and limits of the empirical study of expertise: An introduction. In K. A. Ericsson, & J. Smith (Eds.). *Toward a general theory of expertise* (pp. 1-38). Cambridge, London: Cambridge University Press.

Feuzeig, W., Papert, S., Bloom, M., Grant, R., & Soloman, C. (1989). *Programming languages as a framework for teaching mathematics* (Report, No., 1899). Cambridge, MA: Bolt, Beranek, and Newman.

Frye, D., Littman, D. C., & Soloway, E. (1988). The nest wave of problems in ITS: Confronting the "user issues" of interface design and system evaluation. In J. Psotka, L. D. Massey, & S. A Mutter (Ed.), *Intelligent tutoring systems* (pp. 451-478). Hillsdale, NJ: Lawrence Erlbaum.

Gorman, H., & Bourne, L. E. (1983). Learning to think by learning LOGO: Rule learning in third grade computer programmers. *Bulletin of the Psychonomic Society. 21,* 165-167.

Gould, J. D. (1968). Visual factors in the design of computer controlled CRT displays. *Human Factors, 10,* 359-376.

Groen, G. (1978). The theoretical ideas of Piaget and educational practice. In P. Suppes (Ed.), *Impact of research on education: Some cases studies*. Washinton, DC: National Academy of Education.

Groen, G. (1984). The theories of Logo. In R. J. Sorkin (Ed.), *Proceedings of the Logo 1984 National Conference* (pp. 49-54). Cambridge, MA: Massachusetts Institute of Technology.

Groen, G. (1985). *The epistemics of computer based microworlds* Paper presented at 2nd international conference on Artificial Intelligence and Education University of Exeter, England.

Groen, G., & Patel, V. (1988). The relationship between comprehension and reasoning in medical expertise. In M. Chi, R. Glaser, & M. J. Farr (Eds), *The nature of expertise* (pp. 287-310). Hillsdale, NJ: Lawrence Erlbaum.

Gurtner, J. L. (1992). Between Logo and mathematics. A road of tunnels and bridge. In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp. 247-268). Cambridge, MA MIT Press

Hannafin, M., & Peck, K. L. (1988). *The design, development, and evaluation of instructional software* New York, NY: Macmillan.

Harel, I. (1988). *Software design for learning: Chilren's construction of meaning for fractions and Logo programming*. Unpublished doctoral dissertation, Cambridge, MA: The Media Technology Laboratory, Massachusetts Institute of Technology.

Hayes, J. R., & Flower, L. S. (1980). Identifying the organization of writing processes. In L. W. Gregg, & E. R. Steinberg (Eds.), *Cognitive processes in writing.* Hillsdale, NJ: Lawrence Erlbaum.

Hillel, J. (1992). The notion of variable in the context of turtle graphics. In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp. 11-36). Cambridge, MA: MIT Press.

Howe, J. A. M., O'Shea, T., & Plane, F. (1979). Teaching mathematics though Logo programming: An evaluation study. In R. Lewis & E. D. Tagg (Eds.), *Computer-assisted learning—scope, progress and limits.* Amsterdam: North-Holland.

Howe, J. A. M., Ross, P. M., Johnson, K. R., Plane, F., & Inglis, R. (1982). Teaching mathematics through programming in the classroom. *Computers in Education, 6,* 85-91.

Hoyles, C. & Noss, R. (1989). The computer as a catalyst in children's proportion strategies. *Journal of Mathematical Behaviour, 8,* 53-75.

Jeffries, R. (1982). *A Comparison of debugging behaviour of expert and novice programmers.* Paper presented at the annual meeting of the American Educational Research Association.

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.

Just, M. A., & Carpenter, P. A. (1980). A theory of reading. From eyes fixations to comprehension. *Psychology Review, 87,* 329-354.

Kahney, H. (1982). What do novices programmers know about recursion? (Technical Report No. 5). Human Computer Research Laboratory.

Kahney, H., & Eisenstadt, M. (1982). Programmers' mental models of their programming tasks: The interaction of real-world knowledge and programming knowledge. *Proceedings of the fourth annual conference of the Cognitive Science Society*, Ann Arbor, MI.

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

Kulik, J. A. (1981). *Integrating findings from different levels of instruction* Paper presented at the annual meeting of the American Educational Research Association, Los Angeles, CA.

Kulik, J. A., & Kulik, C. C. (1987). Review of recent research literature on computer-based instruction, *Contemporary Educational Psychology, 12,* 222-230.

Kurland, D. M., & Mawby., & Cahir, N. (1984). *The development of programming expertise.* Paper presented at the annual meeting of the American Educational Research Association, New Orleans, LA.

Kurland, D. M., Clement, C. A., Mawby, R., & Pea, R. D. (1987). Mapping the cognitive demands of learning to program. In R. D.Pea, & K. Sheingold (Eds.), *Mirrors of minds Patterns of experience in educational computing* (pp. 103-127). Norwood, NJ· Ablex.

Kurland, D. M., Pea, R. D. (1985). Children's mental models of recursive Logo problems. *Journal of Educational Computing Research, 1,* (2), 235-243.

Kurland, D. M., Pea, R. D., Clement, C., & Mawby, R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research, 2,* (4), 429-459.

Kynigos, C. (1992). The turtle metaphor as a tool for children's geometry. In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp. 97-126). Cambridge, MA: MIT Press.

Ledgard, H. F., Whiteside, J. A. Singer, A., & Seymour, W. (1980). The natural language of interactive systems. *Communications of the ACM., 23,* 556-563.

Le Gallais, J., Shapiro, M., & van Gelder, S. (1988). *A teacher's tutorial for LogoWriter™.* Unpublished manuscript, McGill University, Faculty of Education, Montreal.

Lehrer, R., Randle, L., & Sancilio, L. (1989). Learning preproof geometry with Logo. *Cognition and Instruction, 6,* 159-184.

Linn, M. C. (1985). The cognitive consequences of programming instruction in classroom. *Educational Researcher, 14,* 9-16.

Littlefield, J., Delclos, V. R., Bransford J. D., Calyton, K. N., & Franks, J. J. (1989). Some prerequisites for teaching thinking: Methodological issues in the study of Logo programming. *Cognition and Instruction, 6* (4), 331-366.

Loethe, H. (1992). Conceptually defined turtles. In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp.55-95). Cambridge, MA: MIT Press.

Mandinach, E., & Linn, M C. (1989). Cognitive consequences of programming · Achievements of experienced and talented students. *Journal of Educational Computing Research.*

Mayer, R. E., Dyck, J. L., & Vilberg, W. (1986). Learning to program and learning to think: What's the connection? *Communications of the ACM, 29* (7), 605-610.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, C. (1981). Knowledge organization and skill deferences in computer programmers. *Cognitive Psychology, 13,* 305-325.

Newell, A. (1980). *Reasoning, problem solving, and decision processes: The problem space as a fundamental category* Hillsdale, NJ: Lawrence Erlbaum.

Nickerson, R. S. (1982). Computer programming as a vehicle for teaching thinking skills. *Thinking: The Journal of Philosophy for Children, 4,* 42-48.

Niemiec, R., Samson, G., Weinstein, T., & Walberg, H. J. (1987). The effects of computer based instruction in elementary schools: A quantitative synthesis *Journal of Research on Computing in Education, 20* (2), 85-103

Norman, D. A. (1983). Some observations on mental models. In D. Gentner & A. L. Stevens (Eds.), *Mental models.* (pp. 7-14). Hillsdale, NJ: Lawrence Erlbaum.

Noss, R., & Hoyles, C. (1992). Looking back and looking forward. In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp.431-468). Cambridge, MA: MIT Press.

Paige, J. M., & Simon, H. A. (1966). Cognitive processes in solving algebra word problems. In B. Kleinmuntz (Ed.), *Problem solving* (pp. 119-151). New York, NY: Wiley.

Papert, S. (1972a). Teaching children thinking. *Programmed Learning and Educational Technology, 9* (5), 245-255.

Papert, S (1972b). Teaching children to be mathematicians versus teaching about mathematics. *International Journal of Mathematical Education in Science and Technology 3*, 249-262. London: Taylor Francis.

Papert, S. (1980). *Mindstorms children, computers, and powerful ideas.* New York, NY. Basic Books Inc.

Papert, S. (1986). *Constructionism: A new opportunity for elementary science education.* A proposal to the national science foundation. Cambridge. MA: The Media Technology Laboratory, MIT.

Pea, R. D., & Kurland, D. M. (1983). *Logo programming and the development of planning skills* (Technical report No. 16). New York, NY: Bank Street College of Education.

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

Pennington, N. (1982). *Cognitive components of expertise in computer programming: A review of the literature* (Technical report No. 46). Ann Arbor, MI: University of Michigan, Center for Cognitive Science.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs *Cognitive Psychology, 19,* 295-341.

Pirolli, P. L.(1986) A cognitive model and computer tutoring for recursion. *Human-Computer Interaction, 2,* 319-335.

Pirolli, P. L., & Anderson, J. R. (1985). *Problem solving by analogy and skill acquisition in the domain of programming.* Unpublished doctoral dissertation, Carnegie Mellon University.

Polson, P G., Lewis, C., Rieman, J., & Wharton, C. (1991). *Cognitive walkthroughs: A method for theory-based evaluation of user interfaces* (Technical report). University of Colorado.

Psotka, J., Massey, D., & Mutter, S. A. (1988). *Intelligent tutoring systems Lessons learned* Hillsdale, NJ: Lawrence Erlbaum.

Rampy, L. M. (1984). *The problem solving style of fifth graders of using Logo.* Paper presented at the meeting of American Educational Research Association, New Orleans.

Ravden, S. J., & Johnson, G. I. (Eds.). (1989). *Evaluating usability of human-computer interfaces*. Chichester, England: Ellis Horwood.

Rist, S. R. (1986). Plans in programming: Definition, demonstration, and development In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers.*

Rist, S. R. (1989). Schema creation in programming. *Cognitive Science, 13,* 389-414

Roblyer, M. D., Casting, W. H., & King, F. J. (1988). Assessing the impact of computer-based instruction: A review of recent research. *Computers in the Schools*, *5,* (3/4), 1-149.

Roblyer, M. D., & King, F. J. (1983). *Reasonable expectations for computer-based instruction in basic reading skills* Paper presented at the annual conference of the Association for Educational Communications and Technology New Orleans, LA.

Rouse, W. B., Rouse, W. B., & Pellegrino, S. J (1980) A rule-based model of human-problem solving performance in fault diagnosis tasks. *IEEE Transactions on System, Man, and Cybernetics,* SMC-10, 366-376.

Samurcay, R. (1985). The concept of variable in programming: Its meaning and use in program-solving by novice programmers. *Educational Studies in Mathematics, 16,* (2), 143-161.

Samson, G. E., Niemiec, R. Weinstein, T., & Walberg, H. J. (1985). *Effects of computer-based instruction on secondary school achievement A quantitative synthesis.* Paper presented at the annual meeting of the American Educational Research Association.

Schiele, F., & Green, T. (1990). HCI formalisms and cognitive psychology: The case of task-action grammar. In M. Harrison, & H. Thimbleby (Eds.), *Formal methods in human-compute interaction* (pp. 9-62). Cambridge, UK: Cambridge University Press.

Shackel, B. (1984). Designing for people in the information age. In B. Shackel (Ed.), *Human-computer interaction-INTERACT'84* (pp. 9-18). Amsterdam: North-Holland.

Shackel, B. (1991). Usability-context, framework, definition, design and evaluation. In B. Shackel, & S. J. Richardson (Eds), *Human-factors for informatics usability* (pp. 21-38). Cambridge, UK: Cambridge University Press.

Shneiderman, B. (1976). Exploratory experiments in programmer behaviour. *International Journal of Computer and Information Science, 5,* 123-143.

Shneiderman, B. (1980a). System message design guidelines and experimental results. In A. Badre & B. Shneiderman (Eds.), *Directions in human-computer interaction.* Norwood, NJ: Ablex.

Shneiderman, B. (1980b). *Software psychology· Human factors in computer and information systems* Cambridge, MA: Winthrop.

Shneiderman, B (1987). *Designing the user interface· Strategies for effective human-computer interaction* Reading, MA: Addison-Wesley.

Shneiderman, B., & Mayer, R. (1976). Syntactic/semantic interactions in programmer behaviour: A model and experimental results. *International Journal of Computer and Information Science, 7,* 219-239.

Siegler, R. S. (1989). *How children discover new strategies.* Hillsdale, NJ: Lawrence Erlbaum.

Small, D. W., & Weldon, L. J. (1977). *The efficiency of retrieving information from computers using natural and structured query languages* (Report SAI-78-655). Arlington, VA: Science Applications.

Sleeman, D., & Brown J. S. (1982). *Intelligent tutoring systems.* London: Academic Press.

Soloway, E (1984). *From problems to problems via plans The content and structure of knowledge for introductory LISP programming* (Technical report No. 21). Cognition and programming project, New Haven, Connecticut: Yale University, Department of Computer Science.

Soloway, E., Adelson, B., & Ehrlich, K (1988) Knowledge and processes in the comprehension of computer programs. In M. T. H. Chi, R. Glaser, & M. J Farr (Eds.), *The nature of expertise* (pp. 185-128) Hillsdale, NJ: Lawrence Erlbaum.

Soloway, E. Bonar, J., Ehrlich, K. (1983). Cognitive strategies and looping constructs: An empirical study. *Communications of ACM, 26,* (11),. 853-861.

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

Spohrer, J. C., Soloway, E., & Pope, E. (1985). A goal plan analysis of buggy Pascal programs. *Human-Computer Interac'ion, 1* (2), 163-207.

Statz, J. (1973). *Problem solving and Logo* (Final report of Syracuse University Logo project). Syracuse University, New York.

Sutherland, R. (1992). What is algebraic about programming in Logo? In C. Hoyles, & R. Noss (Eds.), *Learning mathematics and Logo* (pp. 37-54). Cambridge, MA: MIT Press.

VanLehn, K. (1988). Toward a theory of impasse-driven Learning. In H. Mandle & A. Lesgold (Eds.), Learning issues for intelligent tutoring systems. New York, NY: Springer Verlag.

VanLehn, K. (1990). Minds bugs: The origins of procedural misconceptions Cambridge, MA: MIT Press.

Vinsonhaler, J. F., & Bass, R. K. (1972). A summary of ten major studies on CAI drill and practice. *Educational Technology,* 29-32.

Wenger, E. (1987). *Artificial intelligence and tutoring systems.* Los Altos, CA: Morgan Kaufmann.

Willett, J. B., Yamashita, J. M., & Anderson, R. D. (1983). A meta-analysis of instructional systems applied in science teaching. *Journal of Research in Science Teaching, 20* (5), 405-417.

Zoltan, E., & Chapnis, A. (1982) What do professional persons think about computers? *Behaviour and Information Technology, 1,* 55-68.