Order Number 9218270

A framework for Proactive Interactive Adaptive Computer Help

Selker, Edwin Joseph (Ted), Ph.D.

City University of New York, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A Framework for Proactive Interactive Adaptive Computer Help

by

Edwin J. (Ted) Selker

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1992

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

Miriam Tausner

December 30, 1991
Date

Miriam Tausner
Chair of Examining Committee

January 9, 1992
Date

Stanley Habib
Executive Officer

Robert Campbell

Virginia Teller

Marsha Moroh

Supervisory Committee

The City University of New York

Abstract

A Framework for Proactive Interactive Adaptive Computer Help

by

Edwin J. (Ted) Selker

Adviser: Professor Miriam Tausner

User interfaces can be difficult to master. Typically, when problems occur, computers respond with generic, difficult-to-interpret feedback. This thesis presents Proactive Interactive Adaptive Computer Help (PIACH), a framework for a new style of computer help to alleviate these problems. The framework includes a scenerio and an architecture for realizing that framework. In this scenario, a *proactive* interface attempts to anticipate the needs of a user and responds *interactively* as the user is typing. It records and analyzes user actions to *adapt computer* responses to the individual, offering useful *help* information even before the user requests it.

The approach utilizes dynamic models of both the user and the domain the user is learning. These models facilitate and guide user goals. The feasibility of a system that can adapt to a user's level of proficiency during a computer session has been demonstrated by an implementation called COgnitive Adaptive Computer Help (COACH). COACH was first used to demonstrate PIACH for the domain of writing Lisp programs. The PIACH architecture was designed to facilitate developing and studying adaptive help systems; the help given the user, the domain in which the user is being coached, even the way the system adapts to the user, are represented in frames and controlled by rules which can be changed. A PIACH system for the UNIX command language domain was created with COACH to demonstrate its use as a shell for developing adaptive help systems. A study showing that an adaptive computer interface can improve user confidence and productivity validates the effectiveness of the approach.

This thesis is dedicated to

Alan Selker and to the memory of Lisa Ursell Selker

whose dreams for me and a world without subjugation

endure.

Acknowledgments

The work and writing of this thesis could not have occurred without the support and encouragement of many people. I think of first of my sister Diane Selker, sitting with me through the toughest rewriting efforts of the process. Diane is only the last of a stream of support which was crucial to the successful completion of this thesis. There was Dr. Abe Peled who assiduously convinced me to come to IBM to create the COACH system. My manager for three years, Marc Donner, encouraged me to continue the work. Ellen Shay was instrumental in helping me focus on the project seriously enough to endure the troubles. I am deeply indebted to Tom Wesselkamper for his acceptance and commitments which allowed me to integrate my Adaptive User Model (AUM) project research into a Ph.D. program. My advisor, Miram Tausner, has been supportive, thoughtful and hardworking; she worked through so many rough drafts I can not count them. She patiently pointed out important gaps from which, when I worked through them, I learned so much. Certainly there have been others, notably the students who have been part of the Adaptive User Model Project at one time or another: Matt Kamerman, Kevin Goroway, Matthew

Schoenblum, Brian Lett, Susan Williams, Chris Frye and Frank Linton. My friend, Sylvia Saltztein, and my father Alan Selker added their happiness to mine when I kept my promise to myself to work on my Ph.D. Other people helped and encouraged me through this process as well, particularly Ephraim Feig, who introduced me to, and convinced me to go to CUNY, and Ashok Chandra and Larry Carter, whose encouragement and criticism have been invaluable.

I have also been blessed with many collegial friends who have encouraged me in this work over the years: Oliver Selfridge, Henry Lieberman, Scott Penberthy, Jeffery Bonar, Joe Rutledge, Phil Agre, Harlan Baker, Randy Smith, Michael Lowery, Jennine Myer, Robert Campbell, and Anne Paulson stand out in my mind at this moment. Over the years the ideas in this thesis have fueled countless wonderful conversations with other exciting and supportive individuals not mentioned here; I hope no one is slighted by my memory.

This started with a fun adaptive kaleidoscope agent I built in the late 1970's with Jim Hoffman. I am grateful to the colleagues with whom I have worked on topics and projects which have been part of the process over the years from the first dreams I shared with Dave McDonald at U Mass

Amherst, to the enthusiastic support I received from Tom Binford at Stanford, to the exciting discussions and work with Alan Kay and the research staff at the Atari "ASR" lab, to my wonderful CS-122 students at Stanford who, in the fall of 1984, built inspiring adaptive help projects, to the discussions in context of COLAB I had when working with Danny Bobrow, Mark Stefik and Ken Kahn at Xerox, to all the people who have been involved in my adaptive user model project at IBM.

Contents

List of Figures

List of Tables

1 Objectives of this Thesis

Since the invention of computers, designers have been striving to better match the computer's actions to user expectations. The goal is to create systems which give the user the feeling of a useful tool that is immediately understood. The idea of a tool as an object that allows workers to concentrate on their task, rather than on the tool, was at the heart of Martin Heidegger's concept of "readiness-to-hand" [Heidegger, 1977; Winograd and Flores, 1986].

Still, computer users find themselves needing classes, tutoring, help, and reference materials in order to be able to accomplish even the simplest of tasks with a computer. Terry Winograd and Fernando Flores's book [Winograd and Flores, 1986] discusses "breakdown" of readiness-to-hand in terms with which we are all familiar – a computer becoming the focus of attention because the user does not know how to proceed. Having to seek aid outside the system frustrates the user and prolongs the process of becoming proficient. Attempts to computerize teaching aids have created an active research field. The impact and acceptance of computers in teaching roles, however, continues to be elusive.

Creating computer interactions so natural that they re-

quire no outside learning (the "walk up and use" ideal) would allow all user effort to be focused on the primary task. This being so far unattainable, the Proactive Interactive Adaptive Computer Help (PIACH) scenario is concerned instead with giving the most effective assistance possible to users while they try to focus on their work. Winograd [Winograd and Flores, 1986] describes the process of managing the "breakdown conversations" as the way to progress. When the productive conversation a user is having with the computer to accomplish a task breaks down, it spawns new conversations about what the computer can do. PIACH's goal is to manage these conversations when the computer's "un-readiness-to-hand" is slowing progress.

Most uses of computers in education focus on a single teaching objective specified by the designer. This thesis focuses instead on teaching which will facilitate a user's own objectives in a work session. Is it possible for a computer teaching system to address users' individual educational needs? Can computers demonstrate adaptation to users' educational goals *while* users are working during a productive session? Can a computer decide when to interact productively to advise a user? These questions drive this thesis.

The following are the major objectives explored in this work:

- To illustrate differences between an automated adaptive help system and a standard passive help system.
- To demonstrate that machine-learning mechanisms can be employed to shift computer education paradigms away from a pre-structured format to concentrate instead on users' individual needs. A PIACH scenario moves students toward an apprenticeship, or learn-while-doing, approach.
- To demonstrate the feasibility of a computer interface which uses Artificial Intelligence (AI) reasoning and learning techniques to guide computer reactions without introducing delays. Proactive Interactive Adaptive Computer Help (PIACH) is a scenario for adapting help to a user that runs concurrently with the computer program in use. Researchers have suggested that an interactive adaptive teaching system was not feasible [Zissos and Witten, 1985]. COgnitive Adaptive Computer Help (COACH) is an implementation designed to demonstrate the PIACH scenario for text based interfaces. The COACH

system demonstrates that an interactive adaptive teaching system is indeed feasible.

- To create a tool for enabling research in adaptive educational scenarios. The PIACH framework includes an architecture which allows researchers to describe and test ideas about adaptation in education and to develop adaptive pedagogical approaches. The COACH implementation allows researchers to build working adaptive help systems for text-based interfaces.

This thesis includes the following sections:

- **Objectives** sets forth the scientific questions addressed in this dissertation by the Proactive Interactive Adaptive Computer Help (PIACH) framework.
- **The Scenario** introduces the framework for PIACH by taking the reader through hypothetical user work sessions.
- **Review of Literature** discusses previous work related to the research presented in this thesis.
- **Requisites For Experimenting with PIACH** describes the need for a PIACH architecture, an educational research tool that allows researchers to experiment with a system's response to students and with various teaching strategies.
- **Technical Considerations for Creating PIACH** presents technology for classifying user problems, tracking their proficiency and classifying help to give them.
- **Architecture** describes the central functional concept of an Adaptive User Model (AUM) and describes modules sufficient to create such a PIACH interaction style.

- **An Adaptive User Model (AUM) PIACH Shell** shows that the COACH system, which instantiates the scenario, can be used as a tool for creating adaptive help systems.

- **Evaluation** presents experimental results and evaluation of the implementation.

- **Future Research Goals** describes research directions which could extend this thesis' work.

2 The PIACH Scenario

Users need help while working on solutions to problems in a curriculum or attempting to do productive work using a computer. The Proactive Interactive Adaptive Computer Help scenario presented in this dissertation aids the user in the mechanics of using the computer.

The computer creates a record in an *Adaptive User Model* (AUM) of a user's experience and expertise. Machine-learning and reasoning techniques adapt the help provided to the needs of the particular user. Such help is said to be provided proactively when the computer anticipates user needs to present help before it is requested. Both the user and the computer can initiate help, in a *mixed initiative* interaction.

The paradigm is introduced by three hypothetical users, working at different levels of experience and proficiency, learning the interpreted Lisp language.

2.1 A Novice Lisp Programmer

Freshman Bill is taking his first programming class. He has attended two classroom sessions and is sitting down for the first time in front of the computer. His assignment requires him to use Lisp s-expressions to make arithmetic calculations.

The computer screen with which he works is segmented into four panes: a user input pane to type and edit work, two help presentation panes (one general and one more specific) and an output pane (Figure 1). To help Bill get started using the computer to do his work, the system encourages him to type an open parenthesis to begin an s-expression or to type a defined word (see Figure 2). Examples show him this. He types (. The help changes, telling him that he must type a function name, and gives an example. "function", "s-expression" and "defined word" are concepts in the system's domain knowledge network. Bill remembers these words but does not quite remember what they mean. An example of the use of a function is displayed to get him started. Bill types ADD. The help window tells him that no known function starts with AD, and suggests that he press the "rubout" key. He could press the mouse button to browse available functions, but PLUS, not ADD, is the function he now remembers from his class and he types it (see Figure 3). As Bill types a space after PLUS, an example of using PLUS, together with a simple description and a simplified syntax, is presented on the help pane.

The context-dependent help allows Bill to avoid the usual

startup stalemate in which a user does not quite know what to type to get started. Novice programming problems such as mixing syntax with ideas have also been averted.

2.2 A Student Programmer

Sophomore Harry is trying to write a program. He types **DEFUN**. The help pane reminds him that he must name the function being defined and then give it an argument list. The system gives him an abbreviated syntax, omitting the difficult argument types (optional arguments, keyword arguments, etc.). He types **TIMES-2 (I) (PLUS**. The system shifts its focus to helping him with the PLUS function. An example of a use of PLUS he has already made is displayed. He realizes that he did not really mean to add numbers. He back-spaces and types **TIMES I 2)**. The system changes its focus of help to TIMES as Harry is typing it, and back to DEFUN when he is done with TIMES.

Intermediate programmers like Harry often have problems keeping track of the context and appropriateness of program pieces. The PIACH scenario works to keep this type of programmer oriented by providing context-sensitive help and user-examples. An instance of user-example help is shown

in Figure 5; the last correct use of TIMES-2, (TIMES-2 4), was presented when the user forgot to include an argument in the function call.

2.3 An Expert Programmer

Expert programmer Connie is working on an internal part of the Update-Rule computer program. A model of her expertise allows the system to know that it need present very little help. When she types (SETF, the system shows the very complex argument list syntax for the SETF function. If she uses a function for which no help has been written, the system reaches into that function's definition to present an argument list for it. If she makes an error (e.g., wrong argument type), the system changes its view of her expertise slowly at first. If she keeps making errors, it will change its opinion of her more quickly and begin to provide more help; it will show her examples and remind her of things which are related to the constructs she is using and the language concepts involved.

Expert programmers must be aware of anomalous, as well as simple, relationships between parts of a computer language. Even if they do not memorize them, experts are likely

to use sophisticated syntactic features. If Connie were using function or variable names which she had not yet defined, the system would put these names on a list of undefined functions. A menu would allow Connie to select from those names to aid her in remembering to define them later. In other words, PIACH would focus experts on the delicate anomalous parts of Lisp without bothering them with introductory information.

The above vignettes illustrate a PIACH model tracking users' needs to teach them what they need to know about their computer environment while they are engaged in their own work. A videotape [Selker, 1991] demonstrates the PIACH scenario in a working system called COACH.

The following overview of relevant work describes the current state of Computer Aided Instruction (CAI) and inspirations for the PIACH approach.

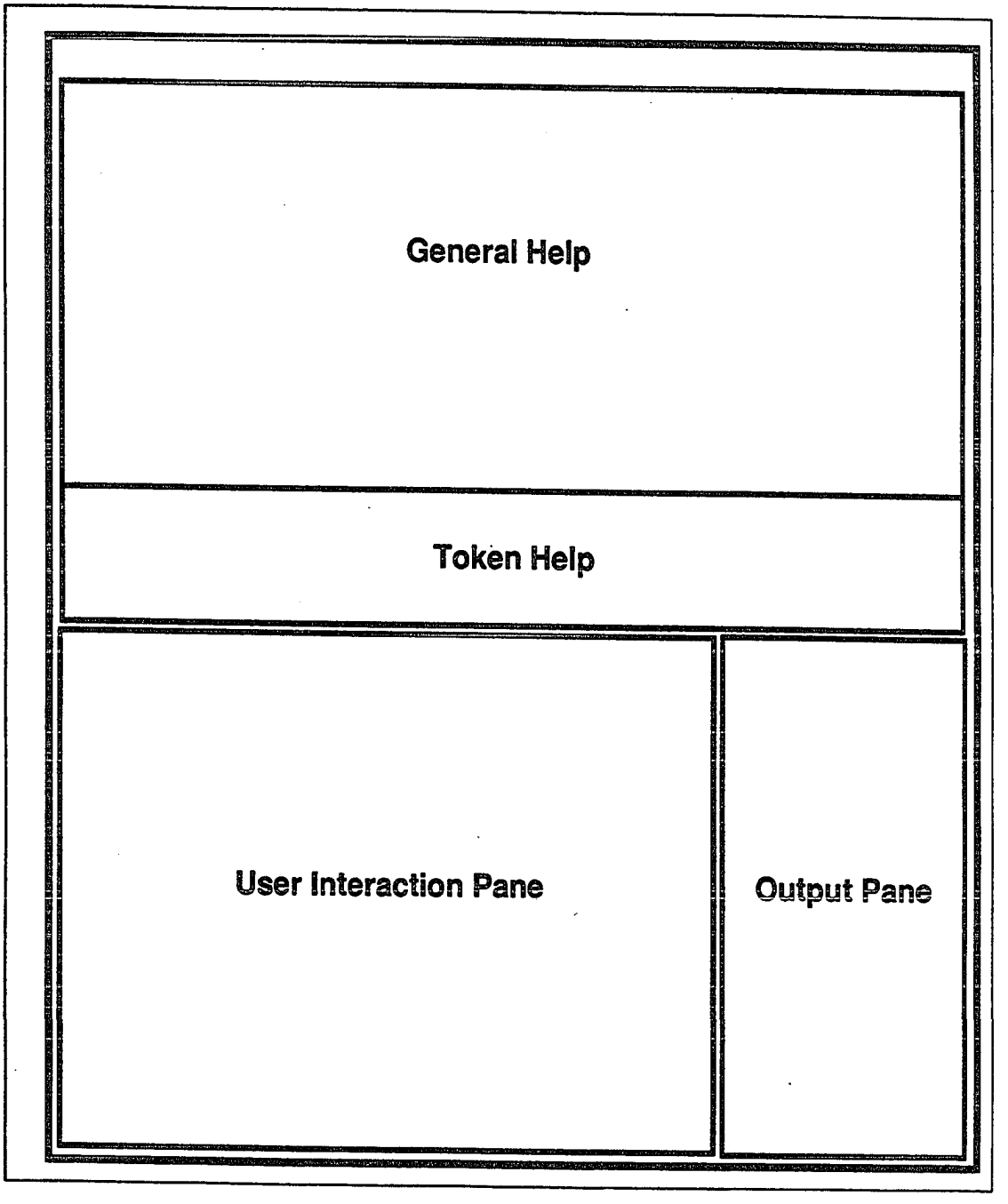

Figure 1: A user interface demonstrating the PIACH scenario.

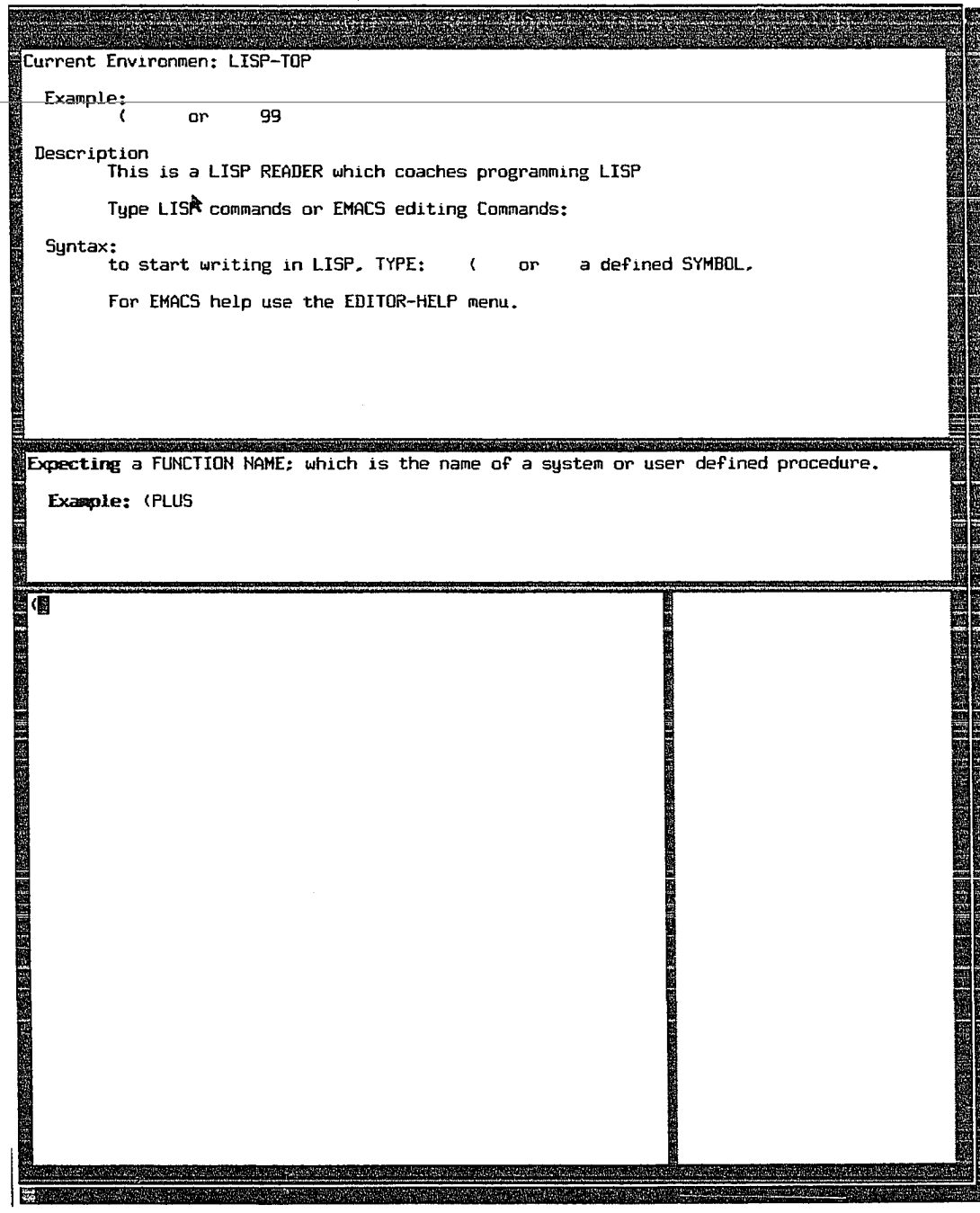

Figure 2: A PIACH interface after one character is typed.

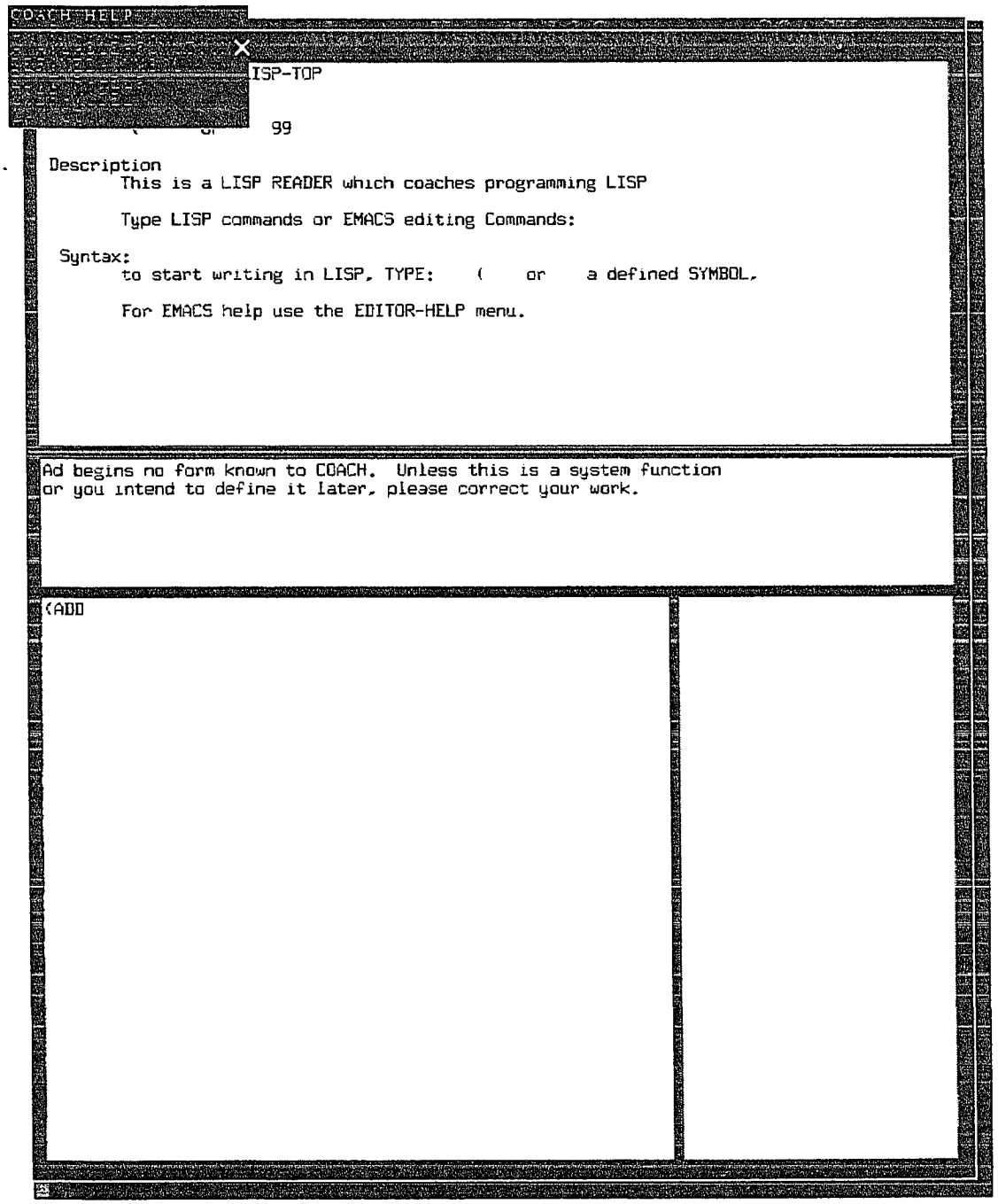

Figure 3: A PIACH interface during a simple error situation.

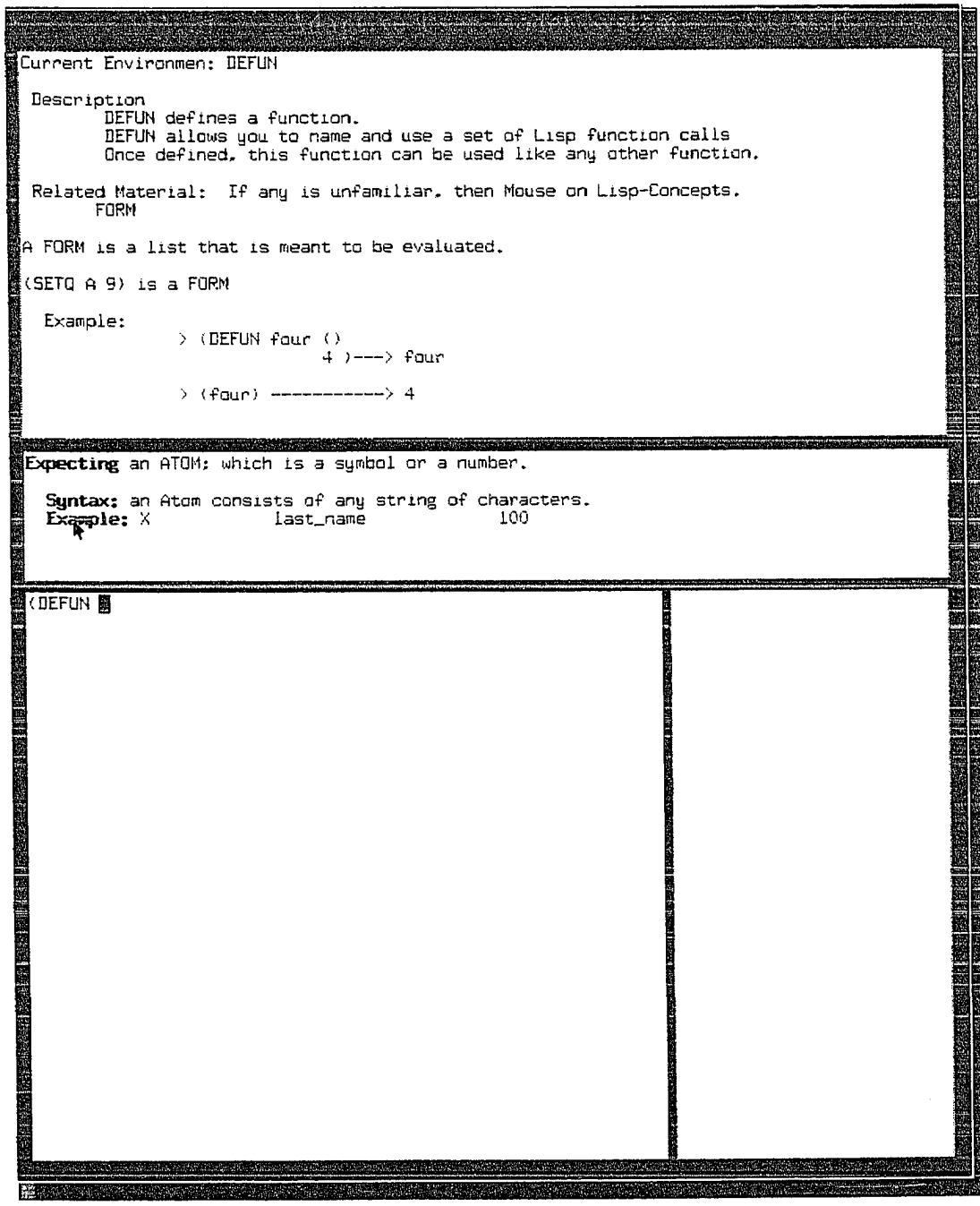

Figure 4: A PIACH interface supporting learning about a specific Form and the idea of form.

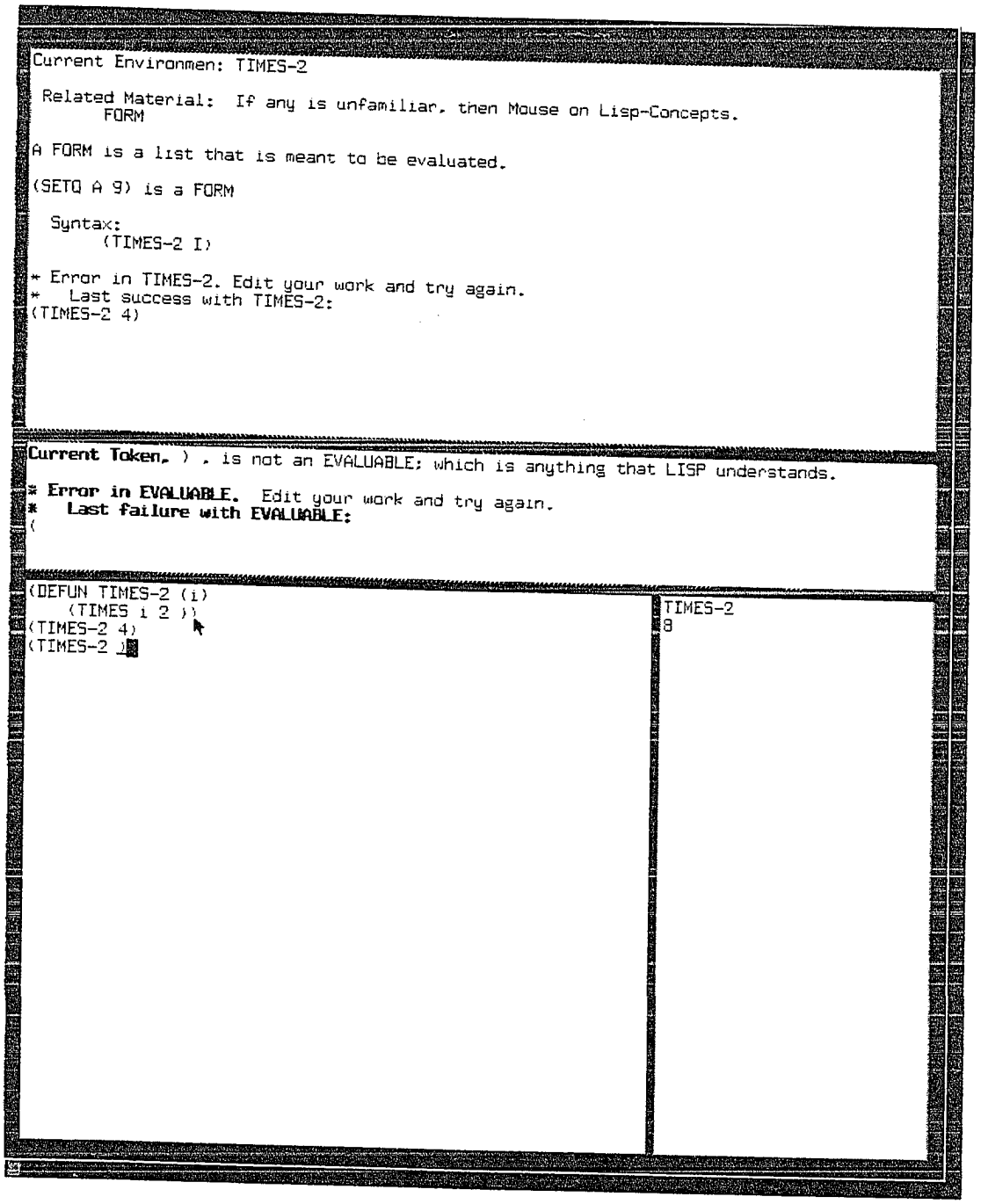

Figure 5: A PIACH interface using automatically accumulated knowledge help for a user defined form.

3 Review of Literature: On-line Computer Training

The framework presented in this thesis is a vehicle for research in human computer interaction and AI as applied to education.

Teaching styles impact students' roles in their learning tasks. Tom Mastaglio [Mastaglio, 1989] described a continuum of interaction styles for computer teaching: from a tutor who prescribes what a user should do — to a coach who kibitzes with a student while the student is trying to do something — to a critic who reviews work after it is completed. In these three teaching styles the point at which the system intervenes is varied relative to a student's phases of work: design, construction or evaluation.

The use of computer systems for teaching has been called Computer Aided Instruction (CAI), Intelligent Computer Aided Instruction (ICAI), Artificial Intelligent Computer Aided Instruction (AICAI), or Intelligent Tutoring Systems (ITS). These names have been created by their proponents to reflect the technological and research progress through the years. A common component of all such work is a prescribed educational goal to which students are guided with subgoals and

tasks. The curriculum can take the form of text with comprehension tests or problem sets or educational games. The term CAI will be used in this paper to refer to all such systems.

Research in CAI has included experiments using Artificial Intelligence (AI) representation, reasoning and machine-learning techniques to direct a tutoring session [Reiser *et al.*, 1985; Sleeman and Brown, 1982]. This section outlines progress in CAI, highlighting the roles AI has played. CAI teaching styles can be broken into the following classes:

- *tutoring* systems which include a syllabus or courseware schedule of what a user should know when, and how to learn it. Such systems use generic models of what a student needs to learn.

- *help* systems which answer questions a user asks. These have no syllabus or model of what a student needs to learn and only interact with a user when the user requests assistance.

- *coaching* systems which remark on user problems and successes as they occur. Like help systems, coaching systems help users while they are working on problems. Unlike help systems, coaching systems offer unsolicited

advice. A coaching system might also have mini-syllabi useful for specific situations in which users find themselves.

- *critic* systems suggest or perform changes to completed user solutions. The original Lisp Critic [Fischer *et al.*, 1985] worked in this way, giving criticism and making improvements to work students brought to it. A critic allows users to think and solve problems on their own, only giving them advice for a completed attempt. This can be compared to the grading and feedback phase in a traditional classroom course format, or the project review phase common to design projects.

A major distinction among these styles concerns motivation for, and timing of, teaching feedback.

3.1 Tutoring Research

Tutoring systems instantiate the classic theory of classroom learning, that students should learn things in a stepwise fashion. These systems present a session based on an analysis previously made by a courseware designer of what a student needs to learn and when. Such an approach is not generally responsive to the particular user. Coaching and help systems

respond at the moment the user's problem arises. A critic does not provide suggestions until the user has completed a solution.

Current theories of Instructional Design [Reigeluth, 1983] focus on important issues of syllabus design [Aronson and Briggs, 1983], student motivation and teaching students how to learn [Collins and Stevens, 1983]. Dennis Aronson and Leslie Briggs, for example, developed a widely referenced list of "instructional events" centered on steps of teaching a topic [Aronson and Briggs, 1983]:

1. Gaining (the learner's) attention
2. Informing the learner of the objective
3. Stimulating recall of prerequisite learning
4. Presenting the stimulus material
5. Providing learning guidance
6. Eliciting the performance
7. Providing feedback about performance correctness
8. Assessing the performance
9. Enhancing retention and transfer

Such a list encourages an educator to consider the *educator's* goals for the student and use these objectives to guide interaction with the student.

How should educational approaches themselves be judged? The educational goals differ in varying approaches. Glen Snelbecker [Snelbecker, 1983] created a list of nine educational issues that may be used to evaluate the priorities of a particular teaching model. The teaching model might emphasize the importance of student *preparation* for a topic or approach, gaining and keeping student *attention*, quality teaching *presentation*, timely *response*, appropriate *feedback*, teaching for *retention and use*, presenting material to aid student *understanding*, encouraging students to learn to use their *creativity*, and/or the model might focus on *management* of teaching situations. A teacher's goals may be met by choosing materials based on a model which emphasizes the desired aspects. With the exception of student preparation, the PIACH teaching model's adaptive approach allows it to concentrate on all of the above issues as appropriate.

Tutoring Systems depend on syllabi to guide students. Computer tutors most often guide lessons by branching on students' responses to yes/no, multiple choice or word fill-

in questions. Other *tutoring* work uses more sophisticated techniques to help guide a user through a lesson plan.

CAI work dates back to the early 1960's. Suppes [Suppes, 1967] may have been the first to describe the classic CAI method of mechanizing the programmed textbook. In place of a programmed textbook, a user reads text and questions from a computer screen. The user works through the text by typing word or number responses to problem questions. Most commercial *tutoring* systems use this mechanized programmed textbook method of presenting information to a student. Even early *tutoring* systems varied their responses relative to a user's answers, something that currently available commercial *help* systems fail to do.

CAI research has taken the syllabus approach to learning much farther than the initial programmed textbook efforts. John Anderson's Lisp Tutor has a sophisticated way of presenting the lesson questions as programs for a student to write [Reiser *et al.*, 1985]. Exercises require students to write programs designed to teach about a particular concept or tool. The student's solution can vary from the teacher's prototype in the naming of variables, but cannot vary the functions used; for example, a student cannot use the "IF"

function where the system expects the "COND" function. The student answers questions and writes programs; the system guides the student through the syllabus. The system certifies a student as a learned programmer relative to the problems in the syllabus that have been completed correctly.

Anderson's system improves the learning abilities of Lisp students. His system expands on the CAI programmed textbook style syllabus. A student's progress is guided by knowledge formalized in "if, then" rules, and the word or number answers of early CAI systems are replaced with programs the user must write.

In one experiment [Corbett and Anderson, 1989], the Lisp Tutor was modified to allow users to use the Lisp interpreter to experiment with Lisp statements. Students could explore the Lisp environment if they desired to "try things out" without leaving the *tutoring* environment. Curiously, in this experiment, student progress was retarded by allowing them to work on things other than the solution to the current *tutoring* problem. In this otherwise controlled learning environment, the flexibility of allowing exploration seemed to distract the student [Corbett and Anderson, 1989]. Educational settings in which students thrive on exploration do, however, exist.

Seminal work in applying AI in the field of education is typified by John Seely Brown and Richard Burton's productive collaboration. Brown and Burton's "Debuggy" [Burton, 1978] introduced knowledge representation and reasoning into CAI. In grade school studies, they showed Debuggy could teach a student about long subtraction with carrying, understanding the student's mistakes better than a teacher. Their approach to teaching subtraction included cataloguing the one hundred twenty or so possible types of mistakes a student can make while doing a subtraction problem. The system used a static sub-skill lattice to characterize what skills the student might be lacking which generated particular errors in an answer to a problem. For each possible mistake, the system had knowledge describing underlying missing concepts which could be responsible for the mistake. Debuggy's sophisticated representation of the problem domain enabled it to use a reasoning approach to evaluate bugs in student subtraction strategies. Pre-analyzing the entire solution and error domains gave the system the ability to explain all incorrect subtraction algorithms.

The reasoning approach which Brown and Burton used in Debuggy required them to *completely* describe and analyze

all possible subtraction errors. Many domains of interest are much larger than subtraction; identifying all possible mistakes in them is usually impractical. In fact, Brown and Burton found teachers seldom understood subtraction in the detail that the Debuggy approach used to reason about potential gaps in student understanding.

The syllabus approach to teaching has been validated as useful in some situations. For example, the order in which students learn two kinds of loop constructs can determine how easily they can master them both. Jeff Bonar [Bonar and Soloway, 1985] ran a large scale study to test this. Specifically, he taught some students the `REPEATUNTIL` construct in PASCAL before the `DOWHILE` construct and some after. The students did better when they learned `REPEATUNTIL` first. The order in which new skills are introduced can be important, even when the learning of one skill is not a prerequisite to the learning of another. Evaluations of the sort done by Jeff Bonar are especially instructive. Unfortunately, because various educators (and learners) may have different goals, not all issues of educational approach can be definitively resolved.

3.2 Help Systems

While *tutoring* systems present a curriculum through which a student travels, *help* systems at the other extreme allow motivated users to ask questions of the system. Many students are not motivated to follow the didactic lesson plan of a *tutoring* system. Many computer users come to an unfamiliar computer system with relevant experience and do not need to learn everything about it as though they were novice computer users. Their reason for using the new system may be that they have a particular task they want to perform that requires that system. Students absorb information when they have a use for it. Rather than provide a generic syllabus for learning the entire system, it is preferable to center the aid users will receive from the system on their specific needs related to that task.

Unlike *tutoring* systems, *help*, *coach* and *critic* systems work with the students in productive situations. Systems that support student goals can allow the student flexibility. They can also provide user support more easily than systems that give students simplified so-called "training-wheel" *tutoring* outside of their work environment. Training-wheel *tutoring* systems protect students from a realistic work sit-

uation, but must be left behind when a student is ready to experiment or begin a real project.

Systems which respond to user inquiries in work situations are termed *help* systems. Unlike other CAI research, most research on *help* systems has focused on the quality of information available to aid the user and has not extensively explored the use of AI or other strategies for delivering the information appropriate to the situation [Borenstein, 1985].

Nathaniel Borenstein [Borenstein, 1985] performed behavioral experiments showing that *help* systems are more effective when they are available from within (integrated in) the computer program than when a separate *help* system must be consulted. His studies also showed that *help* systems are improved when they give users context-dependent responses, basing the information users receive on the part of the computer program with which they are interacting. Borenstein also observed that the quality of *help* text and its relevance to the particular situation are more important than other usability issues, such as graphic design or the ease of asking for help. That is, content matters more than form.

3.3 Coaching Systems

A computer aid for using or learning a body of knowledge may be called a *coach* when, like a human coach, the computer trains, reprimands, gives aid for a personal weakness, and tries to provide a needed idea or fact when appropriate. As well as being extravagant, human coaches can be wrongly perceived. There is evidence that continuous human guidance provided during students' writing activity is often seen as ill-intended, leading students to reject it [Brehm and Brehm, 1981]. Computer-based guidance does not arouse such a reaction [Zellermayer *et al.*, 1991]. However, until now such *coaching* systems have relied on what users are doing (context) or whether they have made an error, without using any representation of the users' actual performance or users' understanding of the system.

To be able to respond to the user's actual level of proficiency, the system needs to learn from the user's actions. If a *coach* system had this ability, it could be said to have an Adaptive User Model (AUM). The system could build the adaptive model by asking a user questions. Elaine Rich's GRUNDY [Rich, 1983] system is well known for using a simple nonadaptive user model to choose books for users. A

user filled out a "form" which her system used to create a user model. GRUNDY consulted this user model "stereotype" to select library books which might be interesting to that user. An adaptive version might include feedback and followup questions. This thesis, instead, explores user models which are built by watching the user's actions [Selker, 1989].

An important issue is whether unsolicited feedback is intrusive, derailing and frustrating users, or whether it can offer welcome advice [Bereiter and Scardamalia, 1984]. Michael Zellermayer [Zellermayer *et al.*, 1991] performed a study which gave mixed results concerning unsolicited computer-presented advice. A system called "The Writing Partner" cued students with so-called "metacognitive" questions, concerned with higher level information than the writing itself, such as planning and organizing. The system attempted to help the students plan and organize their papers by asking questions such as: "Do you want your composition to persuade or describe?" In a comparison of three groups of students writing essays, one group received no guidance, one group received metacognitive guidance when they solicited it from "The Writing Partner", and one group received unsolicited metacognitive guidance from "The Writing Partner".

Many people have the intuition that unsolicited computer advice would intrude and slow a user down. And, indeed, while using "The Writing Partner" during the training period, the students in the group that received unsolicited advice took longer to accomplish their work and did not show an improved essay writing ability. While this might seem to corroborate the impeding, intrusive advisor hypothesis, those same students who had been continuously advised on the metacognitive aspects of their writing tasks were able in essays written two weeks later (on paper, not using "The Writing Partner") to write better essays than students in the other groups.

This provocative study shows how a *coach* offering unsolicited help can teach a person a new skill. Unfortunately, it also indicates that learning this new skill (essay organization and planning) with this type of unsolicited help has a cost. Happily, the PIACH study in the evaluation chapter of this dissertation provides new evidence that unsolicited help can shorten rather than prolong a task.

The idea of an explicit model or expectation of a user's performance is not new. Burton and Brown's electrical circuit trouble-shooting learning environments (SOPHIE 1, 2

and 3) [Burton and Brown, 1982] give important results concerning user modelling. SOPHIE 3 included an evaluation strategy which compared students' performances in designing circuits with those of expert circuit designers. The system reasoned about differences between novices and experts; in so doing, it attributed problems encountered by the novices to bugs in the novice user's otherwise expert approach. SOPHIE research promoted user exploration of the domain as a way of improving the task relevance of a syllabus. Since either the system or the user could control the session, these systems can be said to have provided *mixed initiative* interaction.

One important conclusion which this work (and others e.g., [Feldman, 1980; Genesereth, 1982]) put forward was the fallacy of the novice having an expert user model with some missing parts. Burton and Brown, instead, concluded that novices have a qualitatively different model of a domain than experts. Because of this, a novice user cannot easily be evaluated relative to an expert model. An expert has understanding which does not necessarily follow the procedural and simplistic analysis of a beginner.

Educational systems like SOPHIE or Burton and Brown's

WEST [Sleeman and Brown, 1982] involve users in a little world in which they can explore, learn, and try things out. Games with simulation are widely used in CAI [Sleeman and Brown, 1982]. They are particularly appropriate in *coaching* systems. The feedback and integration of such environments is natural for the *coaching* paradigm. Game scenarios often include a consistent simulated environment referred to as a microworld. Microworlds and other game teaching approaches have the advantage of addressing student motivation as an explicit goal.

3.4 Critic Systems

A *critic* system criticizes or evaluates work at a specific point, either when requested by a user or at the end of a session. Only when a user has come to this point does the *critic* system offer its aid. This approach has often been employed in order to allow the computer to analyze student work off-line.

The advantage of a *critic* system is that it gives the user time to reflect upon the system's suggestions and create a solution without interference. However, the disadvantage is that the advice is not available at the time the problems arise.

Adrian Zissos and Ian Witten [Zissos and Witten, 1985]

built a prototype adaptive *critic* system which could analyze transcripts of EMACS text editor usage after a user session. ANACHIES, as it was called, could decide how to improve a person's use of EMACS editor commands. Zissos and Witten's paper offers a pessimistic view of the feasibility of having a system react as the user needs assistance. Their research convinced them that the computational requirements for using adaptive AI techniques in interactive applications could not feasibly be met with the computers available in the foreseeable future, a cynicism which research presented in this dissertation demonstrates to have been unwarranted.

This chapter has reviewed CAI teaching styles in terms of their responsiveness to users and how their educational goals are chosen (Figure 6). *Critic* systems, for example, offer batch responses, while *coaching* environments respond interactively to a user.

While much work has been done on *tutoring* environments, demonstrations of real time adaptive teaching environments have not been convincing. Researchers still question the reasonableness of real time adaptation. Until now, neither the utility of adaptive interfaces of any type nor the possibility of unintrusive unsolicited help have been shown. The PIACH

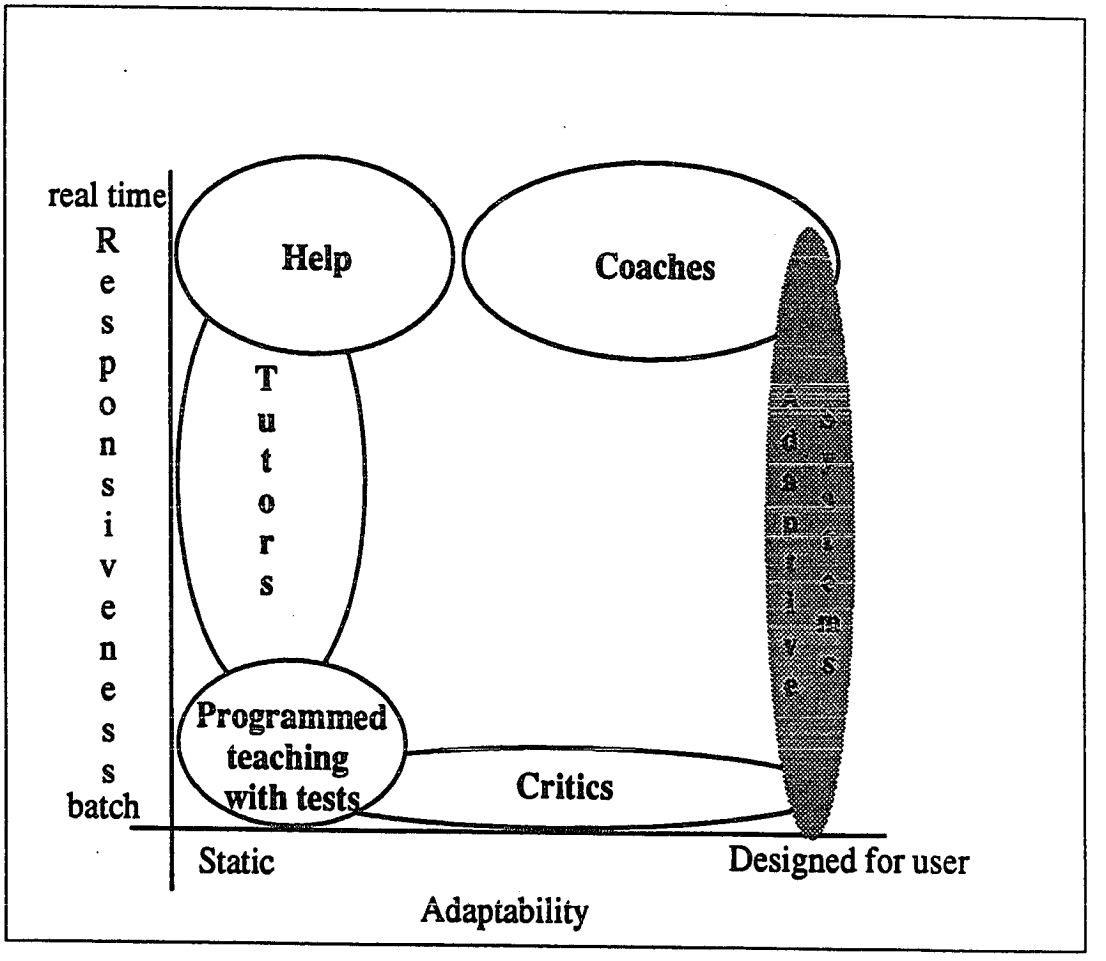

Figure 6: An illustration of the responsiveness and educational goal differences which characterize different computer learning environments. The horizontal dimension, adaptability, refers to the system's ability to be changed for a situation or user. The shaded ellipse indicates where systems which automatically change or adapt to a user's goals would lie in the illustration.

framework offers results addressing these research concerns. The following sections describe the PIACH alternative, concentrating on student motivated teaching scenarios.

4 Requisites for Experimenting with PIACH

Tools for building Computer Aided Instruction Systems (CAI) are usually referred to as CAI authoring systems. Tools for managing a rule base and providing a ready-made production system that can run these rules are often referred to as expert system "shells", connoting their ability to be hard containers that can store knowledge or representations [Barr and Feigenbaum, 1984]. The earliest described rule system, PLANNER [Hewitt, 1972], with its implementation, MICRO-PLANNER [Sussman *et al.*, 1970], could be described as a shell for developing AI applications. It was quite a general system in which "theorems", which can be generally thought of as rules, could be specified as being useful for forward or backward chaining. Filters specified classes, which can be loosely thought of as rule sets. More widely available systems like EMYCIN [Clancy, 1986] and Intellicorp's KEE [Fikes and Keeler, 1985] contain many tools to represent and reason about a domain. As well as providing an authoring system in which teaching information could be changed, a PIACH-based system would be a shell in which reasoning about the teaching process itself could be modified.

An AI practitioner using an expert system shell can utilize

the shell's representation and reasoning machinery without having to build it from scratch. Shells are designed to leverage AI practitioners' efforts by allowing them to create expert systems by merely describing the rules relevant to the task or skill domain; the practitioner may then use the reasoning tools provided in the shell. The challenge to building an AI application is understanding the domain knowledge that is to be embodied in the application, understanding the reasoning relationships in the knowledge, formalizing these, and converting them into the AI shell's representation and reasoning formalisms.

A system based on the adaptive automated help framework called PIACH, which is described in detail in the following two chapters, would be a shell for adaptive coaching. It would provide machinery for formalizing both teaching and domain knowledge for coaching scenarios. PIACH is designed to allow a curriculum designer to encode a domain to be taught. It is also designed to allow an educational researcher to encode theories of when and how to present information to a student. Additionally, PIACH allows the cognitive scientist to encode approaches to gathering user information and methods for altering treatment of students based on their

responding behavior.

The courseware developer's process of converting the system to teach in a new skill domain requires the following steps:

1. Identify a task or skill domain. PIACH is designed to work with textual interpreted interfaces.

2. Identify any delimiters and other token types that the system does not already have.

3. Write token handling function methods for the domain's token types not already supported.

4. Change the token table for parsing delimiters.

5. Identify commands in the skill domain for which help initially will be made available.

6. Describe the syntax of the language being taught in the PIACH formalism.

A primitive help system would then exist for the skill domain. The system would be able to check user syntax, look for undefined functions and variables, learn about user examples and add help for new and system functions. It could also increase and decrease help and change levels of assistance.

The system would not yet know about relationships between syntactic units, concepts, basis sets or required knowledge (see Section 6.3.1). As described in Chapter 6, these create a representation from which the PIACH creates adaptive help which can remind a user of related material and concepts in the skill domain when appropriate. Identifying the relationships between parts of the domain then allows the developer to add deeper knowledge about the skill domain. This can be accomplished with the following steps:

1. Identify skill domain concepts.
2. Identify basis sets in the skill domain.
3. Identify required knowledge for skill domain parts.
4. Write description, syntax and example text for different expertise levels of each skill domain part.

Just as AI shells have simplified AI application development, a shell for testing and extending the adaptive help framework simplifies experimentation and development of PIACH applications. As an expert system shell requires a practitioner to formally understand the reasoning that is included in an AI system, so an adaptive help shell requires the courseware designer to understand the formal syntax of

the language for which the system is to produce help and the relationships between its parts [Selker, 1989]

For as long as computers have existed, people have talked of the possible value of intelligent computer assistants. Unfortunately, the architectures suggested have not been entirely successful. Either they cannot run interactively with a user, or they have not been shown to improve the user's performance [Zissos and Witten, 1985; Gentner, 1986; Waters, 1982]. The PIACH framework, demonstrated in the COACH implementation (see Appendix A), has neither of those problems; it has been demonstrated to run interactively and improve user performance.

5 Technical Considerations for Creating PIACH

After years of work, researchers came to the conclusion that proactive interactive adaptive computer interfaces were not feasible [Zissos and Witten, 1985; Gentner, 1986; Waters, 1982]. This dissertation challenges that conclusion and demonstrates a viable structure for such a system. The enterprise relies crucially on an understanding of the constraints and requirements of real time response. To satisfy these constraints and requirements, the following questions must be addressed. What kinds of domains are interesting and possible to work with? What kinds of errors do users make? In what ways can these errors be addressed? What learning techniques can be used without undue computational overhead? This chapter addresses these issues. Chapter 6, which describes the PIACH architecture, addresses language theory issues that pertain to evaluating user work (see Section 6.4).

5.1 Suitable Domains for Demonstrating PIACH

Demonstrating that PIACH is a viable approach requires showing both that it will work for an important class of interfaces and that it will provide valuable improvements over current help systems. The class of interface chosen was text;

the first domain chosen was Lisp.

Text entry is probably the most common interactive technique used to communicate with computers. Even in the age of graphical languages, many operating system command languages, computer application interfaces, text markup languages and programming languages are interpreted text interfaces. Text was chosen for PIACH because text is computer parsable and used in so many interfaces. Also, learning a text-based interface forces a user to confront difficult educational issues, and to work with incomplete and inconsistent knowledge.

Computer interfaces often include commands which are too complicated or too seldom used to be easily remembered. To show that PIACH will alleviate such problems, the representative demonstration domain should contain many commands with complex syntax.

Computer interfaces contain many interdependent commands and concepts. Thus, a representative demonstration domain should also include complex interdependencies.

Computer interfaces often allow more than one way of doing things. Therefore, a demonstration domain should permit redundancies and alternative solutions.

The information the users must master often changes as time goes on. Therefore, a demonstration domain should be extensible.

Educational domains are often too large to enumerate or analyze fully. The domain should be larger than the implementation can fully represent.

To demonstrate the PIACH framework, a domain was sought which would show several strengths of the approach: the ability to cope with a large and changing domain, the ability to extend help to include additions made to the domain by the user, the ability to permit solutions to have a complex structure, and the ability to coach in domains relying upon many difficult concepts.

Lisp programming is a domain which provides all of these challenges. Many pedagogical tools are designed for limited or toy domains. Lisp is extensible, requiring a help system that will work even when the skill domain changes and enlarges. Working with a domain that changes and enlarges (i.e., an "open system" [Hewitt, 1985]) is a test of the robustness of the PIACH approach. Lisp is a complex open system that allows demonstration coaches to work in a realistic domain. Lisp has redundant ways of doing things that

require the system to act in ambiguous situations. Lisp is also a domain for which other people have built intelligent tutoring tools. This allows the PIACH framework to be compared to and bench-marked against their work.

5.2 Classifying Knowledge Deficits

A user is trying to learn a skill domain, e.g., Lisp. PIACH is designed for skill domains that require interpreted text input. Users must remember keywords, delimiters, syntax and their previous input to effectively use text based programming languages and computer command line interfaces. People forget; even experts are always working with gaps in their knowledge. These gaps may be classified into three types:

- *Issue* – the focus of what a person is trying to learn. This is the material users are aware of and *know they do not know.* On-line help system queries are useful for learning *issues.* The PIACH architecture would show such information automatically.

- *Incomplete* – the knowledge a person *does not know exists.* These are actual holes in the particular user's knowledge. PIACH would point out general knowledge when a user is having difficulty in a particular area.

- *Inconsistent* – the knowledge users *think they know but do not.* PIACH would point out errors to highlight such inconsistencies.

5.3 A Classification for the Help to Provide to Users

Interesting models and theories of instruction and instructional design exist [Reigeluth, 1983]. David Merrill's [Merrill, 1983] *Component Display Theory* describes structures which educators use to organize their efforts. He separates domains of content into facts, concepts, procedures, and principles which can be known well enough to remember and apply correctly in appropriate situations. Such a taxonomy centers the process of learning on a domain and its interrelationships. This is extremely useful for creating a lattice of knowledge relationships which characterize the domain.

Edwina Rissland [Rissland, 1978] created a simple taxonomy of help examples. It highlights the importance of providing different kinds of examples appropriate to the expertise level of the user.

Rissland's help taxonomy consists of four levels of help:

- *Starter* help is used at a novice level. Only simplified basic information is provided. Novices depend on the

literal cues in a problem situation [Glaser, 1985]. The information given them, then, must be carefully designed so as not to mislead them.

- *Reference* help is more complete to familiarize users with standard usage.

- *Model* help is a complete description of what something is and how to use it.

- *Expert* help is machine-level description such as one might find in reference manuals.

This taxonomy of help examples shown to a user can be refined to segment each *level* of help into types of help a student may need. For PIACH, this taxonomy is extended to distinguish and include examples of correct usage, legal syntax and descriptive text telling how and when to use something:

- *Example* help is an actual demonstration of an exemplary solution or solutions. Despite efforts to teach design through concepts and theory, the only effective teaching tool for design is commonly agreed to be the providing of examples [Vertelney *et al.*, 1991]. Moreover, procedural and syntactic knowledge are often most easily conveyed through examples.

- *Description* help is an explanation of the utility and use of a solution type. This information can range from philosophical background to an explanation of the use of a specific language part suggested for the solution to a user's problem.

- *Syntax* help is a template showing the structure of a legal solution. For users to apply a specific statement in varying situations, they must internalize a model of its utility; syntax is the essence of a concise definition.

The appendices show examples, descriptions and syntax for starter help, reference help and model help for Lisp and the UNIX operating system command language.

5.4 Tracking User Proficiency as the User is Working

Various approaches to analyzing knowledge about users have been tried and have been found to be problematic. Systems such as Don Gentner's [Gentner, 1986] used mathematical proofs of programming correctness to decide what a user was doing. The problem with this approach was that the computation required for these proofs grew exponentially with the amount of user work under analysis [Hopcroft and Ullman,

1979] which limited its utility for large bodies of work.

Zissos and Witten's EMACS critic system, ANACHIES, used cluster analysis to make determinations of user capabilities [Zissos and Witten, 1985]. This, too, was a computationally intensive procedure which grew exponentially with the size of the language.

Care has to be taken to create an architecture capable of recording user activity in a representation which it can use to reason about how to provide help and react as the user is typing.

Careful use of representational, reasoning and learning techniques make this real time response possible. Several strategies can be used in the representation to limit knowledge search and access problems. Relationships in the knowledge representation may be recorded and stored as links as soon as they are known. Search would thereby be diminished by the pre-defined lattice created by these links. By limiting the depth of relationship links, search difficulty caused by complex links is decreased. As many user model characteristics as possible would be recorded as scalars, so as to limit representation growth and reasoning difficulty.

Several strategies would be used to make the reasoning

system efficient. The rule system would use an "if/then forward-chaining approach" and avoid the less determinate, computationally extravagant "backward-chaining means/ends analysis" [Barr and Feigenbaum, 1984]. The reasoning can be broken into rule sets which would be used on specific knowledge and on specific parts of the reasoning process. These segmentations of reasoning problems decrease complexity when searching through the rules and searching in the knowledge.

Finally, it is important to choose learning techniques which are feasible for real time computation. In order to learn in real time, the system should limit itself to opportunistic and simple hill-climbing learning. Below is a classification of learning taken from *Machine Learning* [Michalski *et al.*, 1983]. This classification is annotated to show low computational cost learning techniques which could be used in a PIACH framework to organize the ways the system can change its behavior:

- *Learning from examples* is the practice of using specific solutions already achieved in more complex situations. This technique can be used in the PIACH interaction style to collect user provided syntax and examples,

and to offer help for user-defined variables and statements. In the interaction style, syntax may be collected by recording user definitions. Examples can be collected from user work.

- *Learning by analogy* is gathering knowledge in one situation for use in another similar situation. This kind of learning can be used by PIACH to determine when to explain things in terms of skill domain parts the user already knows. The PIACH architecture would also include a lattice of skill domain parts in which it could access information for this purpose.

- *Learning from instruction* is utilizing a user interface or special language to introduce knowledge into the system. A courseware designer uses this technique to modify a CAI system without programming. Expert systems and most state-of-the-art AI education systems rely exclusively on developer modification to change response behavior. Modifications and improvements for the Lisp Tutor [Corbett and Anderson, 1989], and the Lisp Critic [Fischer *et al.*, 1985] are made in this way. PIACH is designed to allow a researcher to add facts and rules that improve the adaptive user model system without writ-

ing Lisp code. Observations of students using the system give the researcher ideas of how to change the way the system treats a user in different situations. These ideas are put into additions and changes in presentation text or presentation rules.

- *"Learning by programming"* is simply the practice of having a developer add knowledge to a system by actually writing code. Before rule systems existed, this was the only way of improving an AI application. Any system can be extended by programming to add function or change domain.

The technology described above is used in the following chapter to present the PIACH architecture, which enables real time adaptive help in interactive computer environments.

6 An Architecture for PIACH

This chapter introduces the Proactive Interactive Adaptive Computer Help (PIACH) structure that enables adaptive coaching. An implementation which demonstrates the architecture is described in Appendix A.

The PIACH scenario can be modeled with four interacting parts or objects: a window interface, a reasoning system, an Adaptive User Model (AUM) which relies on coaching knowledge and domain knowledge, and a parser (Figure 7).

To use a computer language effectively, a student needs to understand its syntax and semantics. So, it is reasonable to use a language definition as part of the structure of the AUM and use the language definition as a way to classify user progress and to guide instruction. This would include the domain knowledge used to compose *statements* and the *token* types used. This definition by itself, however, does not include all knowledge needed to understand and use a computer language, because a user must also understand fundamental programming concepts and relationships. PIACH should represent the *concepts* underlying the language, and the *basis sets* necessary to accomplish defined tasks (see Sec-

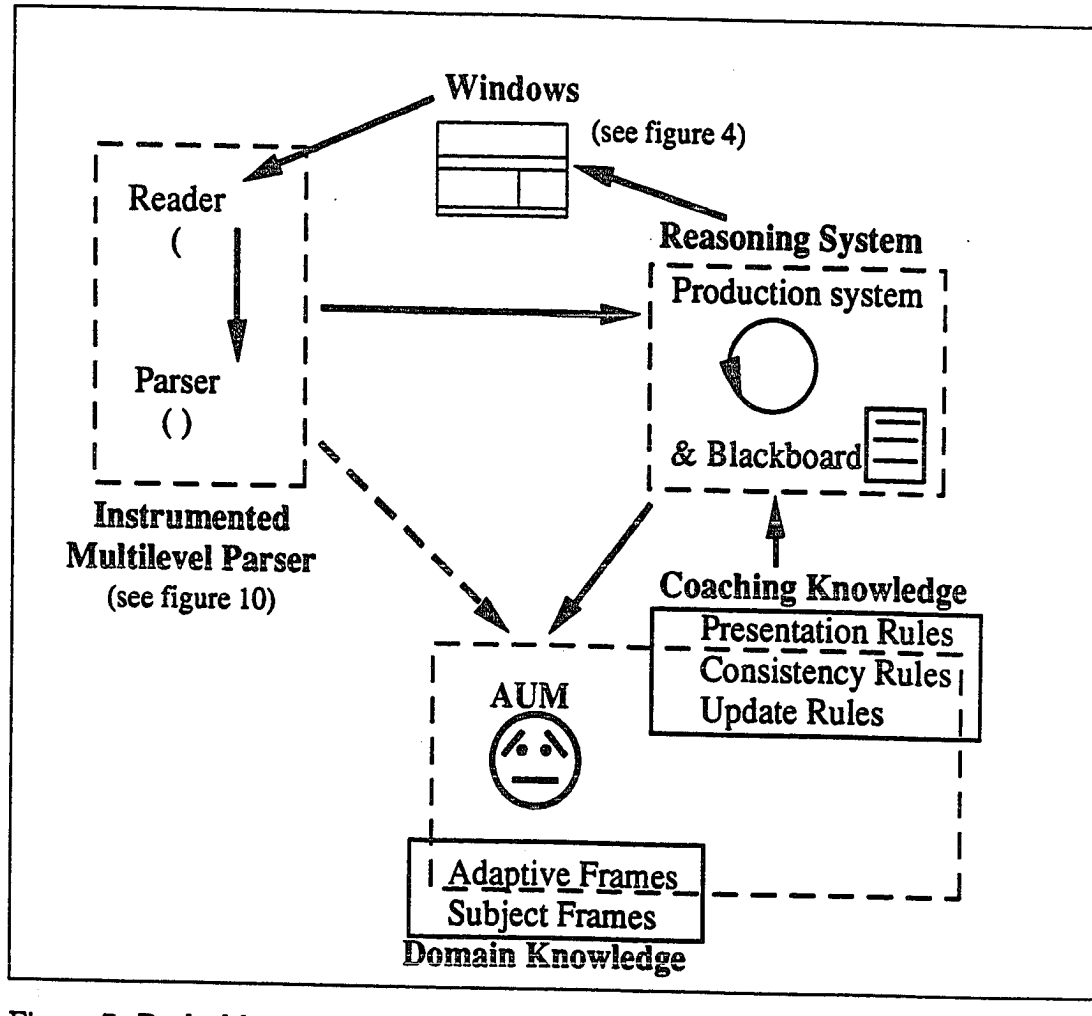

Figure 7: Dashed lines in the figure represent logical relationships, solid lines represent physical relationships. The PIACH architecture is composed of interacting parts or objects. The window interface manages text editing, output formatting and menus. The reasoning system creates and uses the AUM to display domain knowledge help and to modify domain knowledge. Coaching knowledge controls these reasoning activities. A multilevel parser notes a user's work context and dispatches information to the reasoning system.

tion 6.3.1). Each *statement*, *token*, *concept* and *basis set* may be referred to as a *learnable unit*, that is, the smallest quantity of information represented as a discrete entity to a user. Each of these *learnable units* is represented in frames with named slots for useful attributes, one frame for each *learnable unit.*

The adaptive automated help framework must represent examples of each of these *learnable units* and a model of the status of the student in terms of the particular student's understanding and ability to use each one.

A user model frame is recorded for new user-defined *learnable units* as they are created, allowing PIACH to give help for these as well. A skill domain like Lisp, for which a user is being helped, is represented in the system by these syntactic and conceptual parts. Rules draw on knowledge in frames as they update user help and frame knowledge. A simplified blackboard mechanism allows the knowledge module to propose and veto help text before it is presented. The presentation rules build a list of help items to present. Veto rules and help presentation space constraints eliminate all but the most appropriate items of help.

The AUM relies on the Production System to make deci-

sions based on the user model it has built and to decide how to advise the user. The architecture relies on AI technology both for building the AUM and for guiding instruction. The guiding knowledge is embodied in domain knowledge facts and coaching knowledge rules.

Domain Knowledge is represented in the help system parser grammar and in subject and adaptive frames (see Section 6.3.1 below). *Subject Frames* contain knowledge about the skill domain a user is trying to learn (e.g. Lisp). These frames are associated with each *learnable unit. Adaptive Frames* hold usage data and user examples for each function. They are collected as the user works and comprise the AUM knowledge structure.

Coaching Knowledge is contained in rules that create and control the adaptive frames and the help presentation blackboard. *Update Rules* control the recording of user experience for the AUM. *Consistency Rules* contain knowledge about how to build the AUM. These two rule sets work to update the AUM. *Presentation Rules* embody knowledge for using the AUM.

The parts or "objects" guided by expert systems knowledge comprise the PIACH adaptive automated help frame-

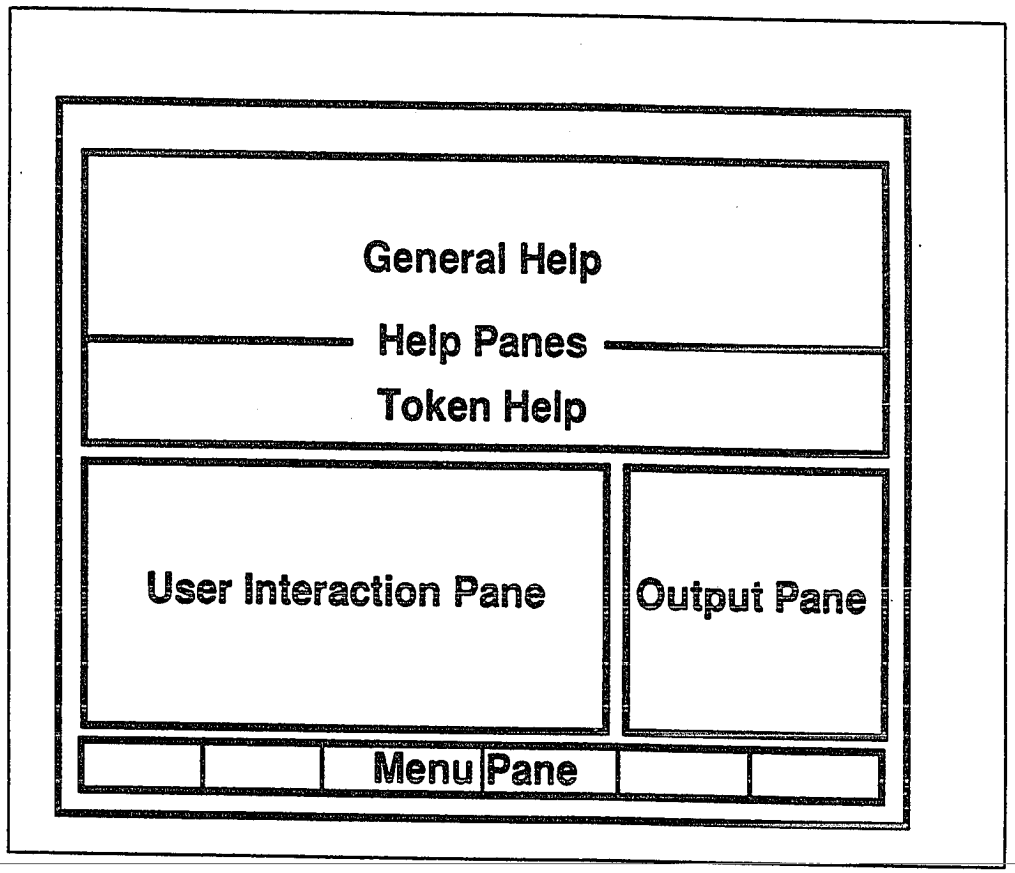

Figure 8: The Window Interface separates user input from help and system responses. A menu at the bottom allows a user to request help directly.

work. These parts and the way they work together are described in the following sections.

6.1 Window Interface

The **Window Interface** manages the screen real estate. It provides separate panes for help text, user input, computer output, and for a menu by which the user can request help (see Figure 8). It dispatches input key and mouse events

to the other modules and presents computer response and advisory help text.

Text-based interactive environments generally type computer output, help, and error messages to a single user console window. The user also types into this window. Confusion often arises concerning which text the computer typed and which the user typed. The combination of such different streams of information into one communication channel requires that the user remember which text on the screen was written by the user, and which was written by the computer.

The window interface design envisioned for PIACH would physically separate user input from computer output and advisory help. This segmentation would insure that computer help and advice do not physically interfere with user input. One way to do this is to vertically separate the token help pane and the general help pane from the user interaction pane and the computer output pane. The user interaction pane would not lock the keyboard when an error intervenes; instead, the character that caused the error (reported on other panes) would be highlighted. Using character highlighting to replace keyboard locking and separate panes to preclude typing on the user interaction pane provide visual aids per-

mitting the user to focus more attention on the problem to be solved and less on the computer mechanics.

More specifically:

- The **Token Help Pane** should be positioned as closely as possible to the user interaction pane to allow a user to see it easily while typing. This pane would be designed to give focused help concerning the specific characters a user is typing. For example, when a novice is typing a token (e.g., a number, symbol, defined variable, etc.), the token help pane would display help concerning that token. When the computer can reasonably predict the next token required, such a pane would provide advice concerning it. The pane would show the token name and, as described in Section 6.3.2, present various levels of description, example and syntax help adapted to the particular user. This local information would not be as useful to users at intermediate and expert levels of proficiency. The adaptive user model would choose when to eliminate this kind of help.

- The **General Help Pane** would be positioned farther from the user's immediate view than the token help pane. This pane would present all teaching knowledge

not presented for token help. Various kinds of teaching text concerning concepts, functions and user work would compete to be presented here. This help window could shrink as a user improves.

- The **Output Pane** would be placed next to the user interaction pane, so as to be noticed and available but not intruding on the user's work-space. This placement would allow a user to easily compare computer output with input text, without their interfering with each other on the screen. By contrast, on a standard "Lisp listener" console, user text competes on the screen with system error and output text. This output pane would provide the same output and error feedback that a standard system gives.

- The **User Interaction Pane** is a text editor window. This pane would allow users to enter and edit work just as they do without a help system. Errors could be highlighted in sequence to allow users to correct them in an organized way. As users improve and the need for other help windows diminishes, this window could take up more screen real estate.

- **The Menu Pane** would be a fixed menu area with spatially separated words that call attention to additional system support. This menu pane would allow the user to request explicit interaction by using a mouse, rather than by the usual method of typing requests in a console pane and possibly obscuring current work already there. While the help panes ordinarily would automatically provide computer generated assistance to the user, the menu would permit users who recognize the need for help to initiate a request for it. The help generated by the request would appear on the help panes. By selecting menu items with a mouse, the user could ask the computer to give help, show undefined variables or functions and so on. This pane would give the user an explicit medium for interaction with the coaching system. The user would also use the menu for such routine functions as saving and reading files, logging in or logging out.

  Pressing a mouse button in the other window panes could be arranged to provide appropriate so-called "pop up" menus useful for the context of the window.

6.2 Reasoning System

The reasoning system controls all aspects of user monitoring and assistance in PIACH. It is made up of a production system which interprets rules and a simplified blackboard which resolves presentation goal conflicts. Together they operate on the Adaptive User Model (AUM) by referring to domain knowledge and using coaching knowledge to make decisions, as described in the following sections.

In the two decades since MICRO-PLANNER demonstrated the utility of using a rule interpreter, or production system, to achieve reasoning tasks represented in rules [Hewitt, 1972; Sussman *et al.*, 1970], many AI architectures have included them [Davis and Shortliffe, 1977; Barr and Feigenbaum, 1984]. For a time it seemed that AI and rule systems were synonymous. The components of a rule system are, in fact, basic to AI representation and search [Winston, 1977; Barr and Feigenbaum, 1984]. In order to demonstrate reasoning and learning in real time, PIACH limits itself to a forward-chaining rule system. This means that rules are searched through in order and fired when they apply. By breaking the system into small rule sets and not using backward chaining (a goal-oriented rule search), PIACH avoids

both indeterminate and long searches. This is necessary to allow real-time advising.

Knowledge for building a user model and for coaching can be separated into sets of rules, each of which operates in specific situations. The search time in a rule system can be significantly decreased by breaking reasoning rules into groups or rule sets that are scanned for specific reasoning needs. For example, with this kind of segmentation, if a token is used incorrectly, only rules that provide help for incorrectly used tokens need be consulted. Section 6.3.2 will describe rule sets upon which the model depends for reasoning.

Rules are made up of an antecedent and a consequent. The antecedent part must be true for a consequent part to fire. Both parts are Lisp s-expressions.

Rules may be defined with the following simple Lisp syntax:

```
(DEFINE-RULE
(rule-name rule-set-name)(user-model-parameters)
  IF  s-expressions
  THEN  s-expressions )
```

The order of rules in a rule set determines the order in which they will be run. The rule's antecedent, consequent

and position in the rule set determine the reasoning behavior.

A blackboard allows statement-proposals and statement-vetoes to interact in reasoning decisions. Blackboard architectures were first introduced in Hearsay, a speech activated chess playing program [Erman and Lesser, 1975]. The Hearsay blackboard was an innovative distributed decision making paradigm for running the system. Different levels of speech recognition each had different parts of a blackboard. Knowledge sources could post proposals and look at proposals on the blackboard. Through this blackboard communication process, multiple knowledge sources collaborated to interpret speech related to chess moves. This architecture has had a continued and marked influence on the artificial intelligence community.

A blackboard would be used in PIACH to arbitrate adaptive help presentation. Rules in the presentation rule set are knowledge sources that propose and veto various kinds of help to choose the appropriate text to be presented to a user. The order in which proposals are posted on the blackboard determines their priority. Vetoes might take proposals off the blackboard. A knowledge source decides to present between one and three help text items on the help window. The three

highest priority proposals on the blackboard represent text which would then be presented to the user.

6.3 System Knowledge and the Adaptive User Model (AUM)

An Adaptive User Model (AUM) is a formal description of a user relative to a domain that tracks changes in the user's knowledge in that domain. The PIACH scenario uses an explicit user model. Frames, facts and rules represent the user and the skill domain the user is learning. The AUM is a set of user model frames [Minsky, 1976] for syntactic and conceptual parts of the domain being coached. While the user is working on a task, these frames record aspects of the user's successes and failures. The AUM for PIACH would be composed of this representation of the user and an associated reasoning system for creating and accessing knowledge frames. The defined network of relationships between skill domain parts, what the user is doing, and the state of the user model is the basis for selecting user help.

The reasoning system uses this network of domain knowledge and coaching knowledge in the form of rules together with the AUM. Reasoning and planning about how information interacts, the way the system updates the AUM, and

even the system's adaptation algorithms reside in rules in PIACH. By changing these rules, a researcher could tailor help for different skills and pedagogical ideas.

Each skill domain part has a help knowledge frame. These frames can include descriptions, syntax and example help at the four levels of help proposed by the taxonomy described in Section 5.3 .

6.3.1 Domain Knowledge

1. **Adaptive Frames.** The AUM frames for each *learnable unit* have the following user model characteristics or slots:

   - **User Examples:** Examples are recorded of user errors and user corrections of those errors. When a user makes a mistake, the system records it. When the user is able to correct the mistake, the system stores that "fix" with the example. If the user later makes a syntactically isomorphic mistake, the system displays the familiar earlier example.
   - **Usage Data:** The following information is also recorded:
     - **Experience** (how often a particular *learnable*

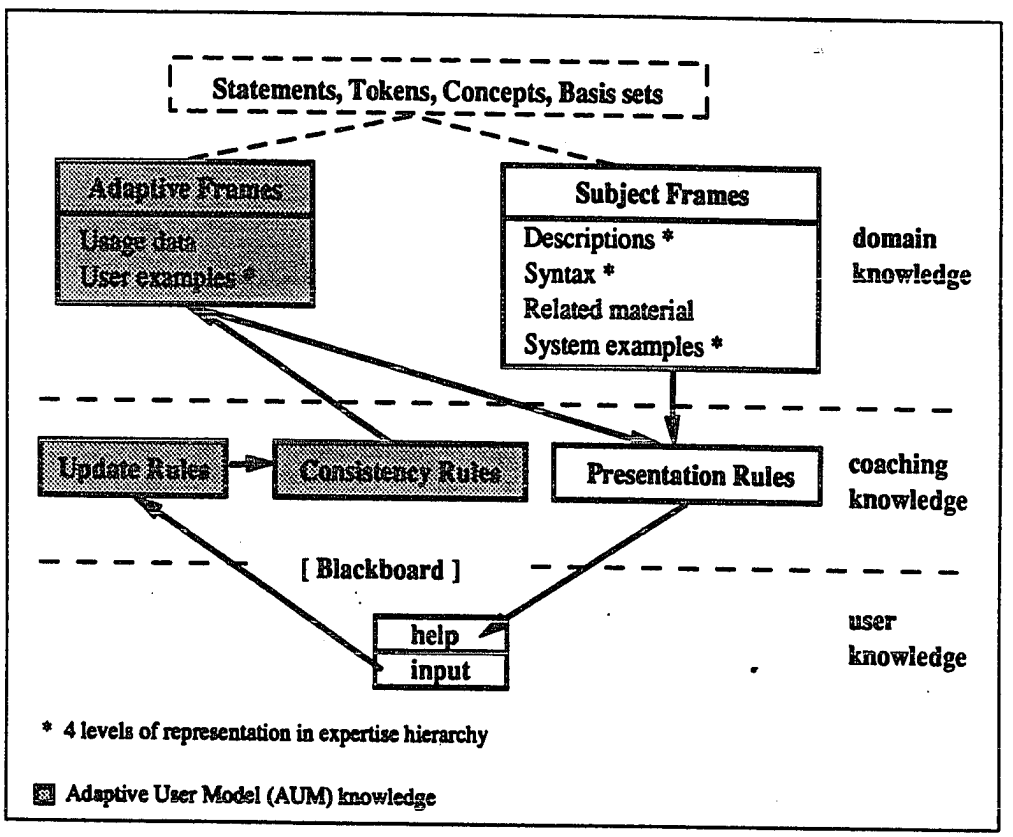

Figure 9: The knowledge and reasoning structure in an adaptive coaching environment. For each learnable unit, AUM and subject frames are built and controlled by model building and help presentation rules.

unit has been used by this user).

- **Latency** (how long since the user has used this *learnable unit*).
- **Slope** (how fast the user is learning or forgetting something).
- **Goodness** (a measure of the user's overall performance with respect to this *learnable unit*).

A demonstration of rules that use these slots is contained in Section 6.3.2.

2. **Subject Frames.** Subject frames (Figure 9) are made up of a subject definition along with various kinds of related material and description text, syntax text and system-defined examples (by contrast with the user-created examples) for the four levels of help.

   The reasoning system described above (see Section 6.2) and the multilevel parser described below (see Section 6.4) rely on the subject frames and adaptive frames to run the scenario. The knowledge exists in slots within the frames.

   Subject frame styles are defined for each type of *learnable unit* (statements, tokens, concepts and basis sets).

Each *learnable unit* has its own frame. Formally, the *learnable units* are represented as follows:

- **Language Statements: S**

  Statements are learnable units which are defined in a syntax facts table described more fully in Section 6.4. This table is extended by user defined functions. A simplified definition for PLUS, for example, is `([ PLUS * N ])` . In this notation described in more detail below (see Table 4 and Table 5), the star, "*", indicates that what follows can occur zero or any other number of times. Open and closed brackets, "[" and "]", represent parentheses and the `N` represents a number type argument.

  Statement knowledge is a tuple **S {L,SH,D,SE,R,O}** where:

  (a) **Language Syntax: L** is the statement's formal definition which PIACH uses to evaluate user work (see Section 6.4.3).

  (b) **Syntax Help: SH** frame slots contain formal abstract help definitions of a *learnable unit.* For a specific level in the help taxonomy (starter, model, reference or expert), the syntax describes

the *learnable unit* in more detail. Concepts are introduced by examples; the syntax assists a user in internalizing the concepts (see Appendix C.3).

(c) **Description Help:** D frame slots contain help text for each level of the help taxonomy. These slots contain text explaining what a specific *learnable unit* is useful for, information describing when it can be used, and an explanation of what it does (see Appendix C.3).

(d) **System Example:** SE frame slots contain helpful examples typifying a *learnable unit* for a specific level in the help taxonomy. The user examples contained in the adaptive frames described above supplement these system examples (see Appendix C.3).

(e) **Required Knowledge:** R frame slots a hold the set of skill domain parts with which a user must be familiar to use a particular *learnable unit* (e.g., using the `CONS` function requires a user to understand the evaluation, s-expression and atom concepts). This set defines a network of related material. Required knowledge makes

it possible for a reasoning system to form the strategies needed to create teaching goals in a coaching environment. The required knowledge set details the necessary prerequisites for understanding a particular *learnable unit.* If a user is having trouble with a particular *learnable unit,* PIACH displays related things with which the user is already familiar or which could be considered as alternatives. If a user is doing well, this knowledge allows PIACH to see how to encourage the user to learn new *learnable units* which relate to ones already known .

Required knowledge is a tuple **R** {***F**,***C**,***T**} where:

- **Function:** **F** is a statement name.
- **Concept:** **C** is a concept name (described more fully below).
- **Language Token:** **T** is a token type of the domain language (described more fully below).

(f) **Other related knowledge:** **O** is a set of *learnable units* which are pertinent to another *learn-*

able unit. This set defines a network of relationships which helps the framework utilize the concepts that tie the domain together. This network is crucial to the coaching that will expand a user's breadth and, when necessary, search for alternative teaching approaches.

Other related knowledge is a tuple **O {*F, *C, *T}** **F, C** and **T** are defined above in "required knowledge".

- **Language Tokens: T**

  Tokens are *learnable units* which are keywords and acceptable variable types for a skill domain (e.g., "(" , "CONS"). They are defined in a table with associated token methods described in Section 6.4.1.

  Token Knowledge is a tuple **T {SH, D, SE, O}** **SH, D SE** and **O** are defined above in "language statements".

- **Concepts: C**

  Concepts are *learnable units* which are semantic ideas not codified by syntactic parts (e.g., evaluation, iteration, stored variable, etc.).

  A **Concept** is a tuple **C {*F, *C, *T}** **F, C** and

T are defined above in "required knowledge".

- **Basis-sets: B**

  Basis sets are *learnable units* composed of groups of other *learnable units* comprising minimal sets of knowledge necessary to understand a topic. This term is borrowed from mathematics where it defines a similar concept.

  An arithmetic basis set for example, would require a user to know about `PLUS`, `DIFFERENCE` and the List and Number concepts. The elements of a basis set are skill domain parts, all of which must be known to do a task in a particular topic area (e.g., the basis set for "simple-lists" includes `CONS`, `CAR` and, `CDR` and the atom and s-expression concepts). Generally basis sets will be a subset of a required knowledge set; for example required knowledge for `List` includes the Eval concept as well as `CONS`, `CAR` and, `CDR`. Basis sets may be elements in **O** or **R** sets of *learnable units.* These allow the system to reason about basic knowledge a user may be missing when trying to use a *learnable unit.*

  A **Basis-set** is a tuple **B** {***F**, ***C**, ***T**} **F**, **C** and

T are defined above in "required knowledge".

The subject frames described in this section create a domain representation in PIACH. This representation consists of syntax, descriptions, system-examples, and related materials for each *learnable unit* in the domain. The language syntax definitions, L, define knowledge with which PIACH can record correctness of user work. Required knowledge, related material and basis sets, included as R, O, and B, define relationships between parts of the domain, much like the links in a hypertext system [Conklin, 1986]. This network, R, O and B, can in fact be browsed like hypertext. More importantly, these are the basis for PIACH reasoning about relationships in the subject domain. Rules like "explore exploration" and "out of practice" (see Appendix C.2) described in the next section use these to orient and teach users.

These subject frames are augmented by the AUM to give a rich representation from which coaching knowledge makes decisions.

6.3.2 Coaching Knowledge

Coaching knowledge is embodied in rule sets which suggest information to place on help windows.

Rule sets for creating and maintaining the AUM consist of update rules and consistency rules. These rule sets change the AUM frames each time the parser signals a change in parse state. In this way, user model frames are changed each time a function is closed, a token is typed, a token is found to be undefined, etc..

Presentation rules consult the parser, the AUM and the blackboard to make decisions about what to present. Detailed analysis of two of the more interesting and complex of the presentation rules is provided below.

1. **Update Rules.** A simple update rule set consists of rules with the following mnemonic names:

   - Note-Success (activated by a correct usage of a *learnable unit*; improves user rating on AUM frame slots for a *learnable unit* and related material).
   - Note-Failure (activated for an incorrect usage of a *learnable unit*; decreases user rating on AUM frame slots for a *learnable unit* and related material).
   - Was-Bad-but-Getting-Better (activated when a user has a success with a *learnable unit* which has been problematic; increases user ratings).

- Was-Good-but-Getting-Worse (activated when a user begins making mistakes for a *learnable unit* which has previously been rated well; decreases ratings slowly).
- Best-and-Getting-Better (activated when a user continues to use a *learnable unit* correctly at more sophisticated ratings; bumps up Best).
- Worst-and-Getting-Worse (activated when a user continues making mistakes in use of a *learnable unit* that has been being used poorly; bumps down Worst).

2. **Consistency Rules.** A consistency rule set would work with update rules to create and maintain a user model. A simple consistency rule set has the following mnemonic names:

   - Note-Used (activated each time a *learnable unit* gets used).
   - Maintain-Best (works with Best-and-Getting-Better to bump up Best).
   - Maintain-Worst (works with Worst-and-Getting-Worst to bump down Worst).
   - Bound-Goodness-and-Best (activated to record user's

"personal Best").

- Bound-Goodness-and-Worst (activated to record user's "personal Worst").

3. **Presentation Rules.** Presentation rules determine the help that will be provided to the user, posting and removing the various possibilities on the blackboard. Specific presentation rules "argue" for their position and the blackboard records the result which is presented to the user. Separate presentation rules exist for statements, tokens and concepts. Their particular order determines which text will have priority for use as help.

   For the purpose of creating the PIACH framework (including the evaluation of PIACH in Chapter 8), a model rule set for presentation of statements contains the following rules mnemonically named:

   - Losing-Ground (provides the user with the most basic help).
   - Out-of-Practice (reminds the user of information that was previously understood).
   - Encourage Exploration (suggests useful information not presently being used).

- Veto Overly Sophisticated Help (protects the user from help beyond an appropriate level).
- Veto Extra Help (protects the user from too much help).

To give a feeling for how these rules work, an examination of a few of them follows.

A simple "Losing Ground" rule provides a user with examples:

```
IF
    learnable unit used has a low Goodness score
    and a low learning Slope,
THEN
    PUSH a User-Example onto the blackboard, and
    PUSH a System-Example onto the blackboard.
```

This rule implements the following concept: if the Slope and Goodness measure are both low, then the person is doing poorly. In such a situation, the rule proposes that both a prior user example, if available, and a system example of correct use of the statement be placed on the blackboard for the confused user.

Besides pushing things onto the blackboard, the system might use other rules, such as "Veto Extra Help", to take inappropriate information off the blackboard.

```
IF
   user expertise for this learnable unit is NOT
   better than the Best it has been
THEN
   PUSH Veto Extra Help onto the blackboard.
```

This rule implements the following concept: if a user's expertise is not at its highest point so far, tell the blackboard not to provide overly verbose help.

The defined network of relationships is used in rules such as "Encourage Exploration" to expose a user to new information.

```
IF
   learnable unit has a high Goodness score, a
   non-negative Slope, and has been used many Times
THEN
   PUSH previously unused Related and Required
   knowledge for the learnable unit
```

```
onto the blackboard.
```

This rule implements a tutoring concept: if a user is doing well with a learnable unit, expose them to more material in that area of knowledge.

Rules for an implementation created to demonstrate the validity of the model can be found in Appendix C.2.

6.4 Instrumented Multilevel Parser

The domain a user is trying to learn (e.g., Lisp) has a syntax – the set of things a user can type that are correct and interpretable. Like the standard UNIX facility, Yet Another Compiler Compiler (YACC) and LEX [Kernighan and Pike, 1984a], the PIACH architecture includes a general purpose parser which uses state machines to classify character, and token types to drive lexical analysis. A formal language definition drives actual parsing strategy. Unlike other parsers, the PIACH parser is *instrumented* to run rules and add knowledge to a user model after each keystroke.

The multilevel parser (see Figure 10) structurally separates different kinds of data about the user's interaction with the system. It is made up of a character classifying table, a token parse table, a token attribute grammar parser and a

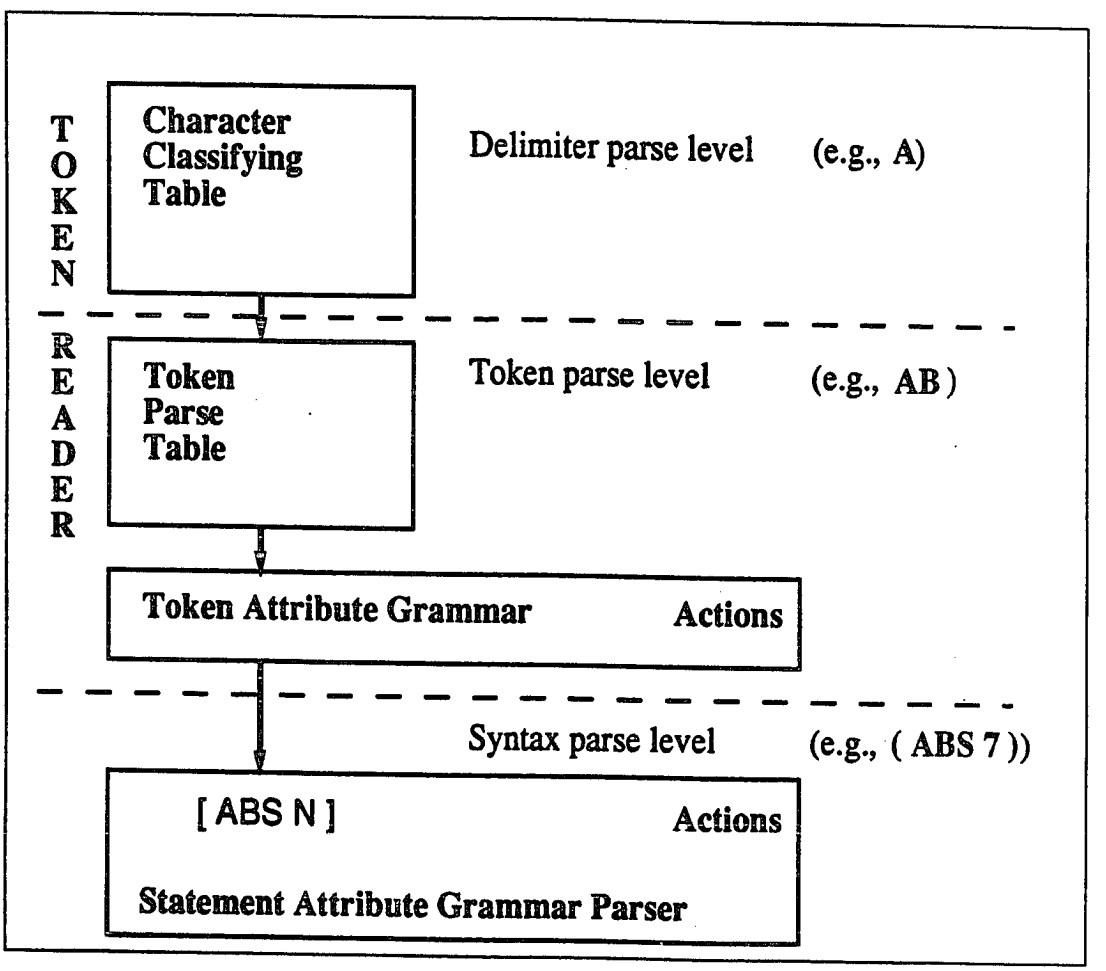

Figure 10: The structure of a multilevel parser.

statement attribute grammar parser. The parser structure, function and PIACH syntax are described below.

6.4.1 Parser Structure

The Character Classifying Table

The character classifying table lists all the characters and their functions (see Section C.1). It is an efficient mechanism for classifying each character's impact on a user's work. Character types can readily signal delimiters and user mode changes with one computer array reference instruction. Use of this technique for the two "bottom-most" analysis levels is part of the overall strategy of real time response required to provide an automated adaptive coaching interaction style.

The Token Parse Table

A token or word in a user input language has meaning by nature of its kind or "type"; it might be a number, keyword, variable name, etc.. The token parse table (see Tables 1, 2, 3) notices token type changes and dispatches tokens to the token reader. It is the next level of user input analysis. As with the character classifying table, the token parse table uses a lookup technique, which permits most user context changes to be recognized without resource intensive reasoning.

Current State Character Read	pls	min	sym	qte	st	cmt	num	shp	hlp
er	hlp	hlp	hlp	hlp	hlp	hlp	hlp	hlp	hlp
sp	hlp	hlp	sym	sym	st	cmt	hlp	hlp	hlp
num	num	num	sym	sym	st	cmt	num	hlp	min
chr	sym	sym	sym	sym	st	cmt	Sym	hlp	min
opn	pls	pls	pls	pls	st	cmt	pls	hlp	hlp
cl	min	min	min	hlp	st	cmt	min	hlp	min
qte	hlp	qte	qte	qte	st	cmt	qte	shp	hlp
st	st	st	st	st	min	cmt	st	hlp	hlp
cmt	cmt	cmt	cmt	cmt	st	cmt	cmt	hlp	hlp
ecm	pls	min	min	hlp	st	min	min	hlp	hlp
blnk	pls	min	min	min	st	cmt	min	hlp	hlp
shp	shp	shp	sym	sym	st	cmt	hlp	hlp	hlp

Table 1: Model token parse table sufficient to break Lisp input into delimiters, error states and tokens.

er	cntrl characters etc.	Characters not used in the domain.
sp		Special characters.
num	0 - 9 and .	The decimal numbers.
chr	a - z and A - Z	The standard roman characters.
opn	(	Open parenthesis symbol.
cl	)	Close parenthesis symbol.
qte	'	Single quote is the Lisp QUOTE macro symbol.
st	"	Double quote is the string delimiter symbol.
cmt	;	Semicolon is the Lisp comment delimiter symbol.
ecm	lf cr	Characters that end Lisp comments.
blnk		Blank space character.
shp	#	"Pound" or sharp character.

Table 2: Symbols in a token parse table break up input into possible delimiters and specific token types.

pls	Is the start of form parse state.
min	Is end of parse context state.
sym	Indicates a symbol is being parsed.
qte	Indicates a quoted object is being parsed.
st	Indicates a string is being parsed.
cmt	Indicates a comment is being parsed.
num	Indicates a number is being parsed.
shp	Indicates a macro is being parsed.
hlp	Indicates an illegal object is being parsed.

Table 3: States in token parse table. These states model an executable parse of tokens. Language parsing is driven by these states.

The Token Attribute Grammar Parser

Recognition of a token could have side effects. The token attribute grammar parser is tied to the token parse table to handle token level interpretation of user typing. Each token has an associated "Object method" which analyzes the impact of a token's completion. The methods call for coaching help, update the user model, and change the way PIACH views the domain and the user.

The Statement Attribute Grammar Parser

Syntax is the defined order in which tokens can legally follow each other. The statement attribute grammar parser handles the lexicon and builds a structure describing the syntactic unit the user is typing. User input is filtered through this parser for lexical analysis. Static semantics refers to the meaning attainable from a program without running it. Syntactic and static semantic advisory responses are triggered by this parser. The statement parser sends information about the user to the AUM. As explained below, a simple description language can be used to create statement templates. The parser steps through these templates accepting user input. The parse state pushes onto and pops off a stack as

expressions are evaluated.

6.4.2 Parser Function

The mechanics of, and the relationships required between, these elements of the parser will now be described. The token parse can be modelled by a finite state automaton. The PIACH interpreter is parsing, not to interpret the domain language, but rather, to build a user model and to note pedagogical opportunities.

Table 1 shows details of transitions that can determine Lisp token delimiters for PIACH. As characters are accepted by this token parse table, they are added to the partially constructed token. The token parse table takes the current parser "state" (the column) and the current character (the row) as input to determine the reader state. For example, if the reader were in string state, `st`, and received an illegal character, `er`, the reader would change to the help, `hlp`, state.

The character classifying table, the token parse table and the token attribute grammar parser collectively comprise the token reader. The accepted characters and the state of the parse table drive the token reader.

A function for each token type checks token side effects of the parse state. When a new Lisp variable is read, for exam-

ple, such a function would add it to the user's environment as necessary.

When a token is accepted, the AUM is updated. A statement is composed of legal tokens in a syntactically legal sequence defined by language parse templates. The statement attribute grammar parser is controlled by language parse templates. Each time a token is accepted by the token reader, the statement parser takes a step through the currently active template and predicts what the user might need to do.

The acceptance of a token and progress through a template cause the rule system to select presentation help. The template state and past input give the AUM knowledge of user goals that are often adequate to predict expected needs (as described in Section 6.3).

The statement parser uses a formal language to describe syntactic parsable expressions in templates. If a statement call is made, a new context is started, for example

```
(SETQ a (CONS
```

makes the `CONS` statement parse template active, pushing the `SETQ` parse template onto the pending parse stack. The statement parser steps through these templates, pushing them on and popping them off the pending parse stack,

as new contexts are started or completed.

Expression side effects can be further detailed in an action function to be run after a parse is accepted so that PIACH can keep a record of the user's environment as well as the user's state. To implement side effects, a token or function can have an action function associated with it. When the adaptive automated help framework has completed recognizing the token or function, the action function will run. The SETQ function, for example, has an action function which adds new variables to the known variables list.

A more complete example can demonstrate the statement parser in action: writing an s-expression to sum three with the product of five and four. Starting in the beginning state, when the user types

```
(
```

the reader puts the statement parser in the `pls` state. The token rule set now consults the adaptive frames to decide whether help concerning a function name should be displayed for this user, and if so, what kind of help is needed. As the user types

```
PLUS
```

the statement parser makes the PLUS parse template the current template. The function rule set and blackboard now use the AUM to decide how much and what kind of help to present. The token rule set fires to decide what immediate help to present, possibly indicating that a number is called for in the PLUS parse template. If the token rule set demonstrates user need, number help is given to the user as three is typed in. While the product is being entered, the parse stack has to remember the PLUS parse state. The TIMES parse template is now put on the parser stack. The function help rule set is fired again to reason as to what help to present for the TIMES function. When the product is ended with a

)

the sum parse template comes into force. When the sum parse template is closed, the beginning state would be in force. A rule set now gives top level help if appropriate.

Each transition change above causes an action function, if it exists, to run when a token or function is recognized.

The multi-level parser rule system and the AUM work together as a PIACH architecture for adaptive help.

A	atom
S	defined symbol
N	number
L	list
F	function
X	any of the above types
Q	check only parenthesis level
FS	function specification

Table 4: Language parse model: token types. These allow modelling of Lisp's major token types. Such a parser table is designed to analyze user proficiency; a language parser designed to implement a language might have more types.

{	open a syntactic parse unit
}	close a syntactic parse unit
[	(, open a clause
]	), close a clause
*	next syntax part can occur 0 or more times
?	next syntax part can occur 0 or 1 time
V	at least one component of the next clause must occur at least once
@	consider the following characters a symbol

Table 5: Language parse model syntax delimiters. The parse modelling language itself has immutable token type control symbols to allow designers to describe a language to be coached.

6.4.3 PIACH Syntax

Languages can be formally defined. Specifically, they are defined by a grammar and alphabet [Hopcroft and Ullman, 1979]. PIACH uses a formal language representation of a subject domain to assess user work and progress. The domain language is defined for the statement attribute grammar in a formal, context-sensitive syntax notation. This notation is shown with the Lisp system key symbols in Table 4.

Logical conjunction in a template is indicated by juxtapo-

sition. A legal statement or sentence, S, consists of a string of symbols from the alphabet (described in Table 4) that satisfies an expression in a legal syntax table, like the one shown in Appendix C.1.

In the formal language definition, key symbols, control symbols and delimiters are surrounded by a set of parentheses:

```
{S := ( *{A} )}
```

The language can be described as being made up of the alphabet:

```
A := A,S,N,L,F,X,Q,FS,{,},[,],V,?,*,@XXXX
```

where XXXX is any string of characters. Delimiters should always come in pairs:

```
{}, [].
```

In the Lisp language, a new S is signaled by an open parenthesis, represented by a [, so all s-expressions end with a closed parenthesis, represented by a]. (The UNIX command language ends commands with a carriage return, and so does not require this].)

The PIACH architecture uses an attribute grammar parser. Each token type and each completed parse sends an `:action`

message when recognized. Each key symbol in the notation has a method (function) associated with it which can cause an action during the parse.

Functions that the system parses are described in this notation. The simple example

```
[ ABS N ]
```

defines the absolute value function as requiring a parameter of type number. A slightly more complicated syntax such as

```
[ SETQ * { A X } ]
```

requires 0 or more atom-anything pairs for a legal "sentence".

Appendix C.1 shows a COACH Lisp definition; appendix C.6 shows a COACH UNIX definition.

6.5 Conclusion

This chapter has described the PIACH architecture. Several representations work together to create help for the user: the subject frames (definitions of the domain), the adaptive frames (recording of a user relative to a domain) and the presentation rule sets (which embody a model of teaching) and the multi-level parser (syntax domain definition).

The chapter has further shown how formal representations are used for domain knowledge, teaching knowledge, adaptive strategy and the way the adaptive frames are used to create a PIACH framework. The next chapter discusses additional reasons for using these multiple interacting representations.

7 A PIACH Shell

COgnitive Adaptive Computer Help (COACH) is a proof by demonstration of PIACH (see Appendix A). Demonstrations of possibility are important, but further progress in a field requires tools to make experiments feasible. As well as being a demonstration, COACH was designed to be a tool for understanding PIACH. The structure is organized to allow a courseware designer to change the skill domain information, the presentation approach or the adaptive strategy with minimal effort. It has been used to show that PIACH can work for open systems (see Chapter 5) and with different domains, and can support experimentation with user modelling and help presentation strategies.

Using PIACH in Open Systems

Open systems are systems which are too big to be analyzed or which grow with use (see Chapter 5). The Lisp COACH demonstration showed that PIACH can be used for open systems.

COACH creates user models and provides help for any number of system functions and new user functions. The system was tested as a PIACH for a twenty-five thousand func-

tion Lisp programming environment. Since help text could not be provided for all of the constantly changing Genera Lisp, a mechanism for adding help for functions as they get used was provided instead. Multi-level help was provided for a basic set of functions and was automatically augmented to include any other functions actually used or added by a user.

COACH queries the Genera Lisp environment for a syntax description which it uses to start a user model for a previously unknown function. When a user defines a new function, COACH records its syntax to start a user model for this function. As newly added functions get used, examples of correct and incorrect usage are compiled for use as example help text.

Using PIACH For Different Domains

The PIACH framework coaches a user with a courseware designer's definition of a domain language. A UNIX version was created to demonstrate that PIACH can be a help system for different domains.

Matt Schoenblum, a talented seventeen-year-old high school student without programming experience, was able to demonstrate this capability by adapting the COACH system to teach the UNIX operating system's shell command language [Kernighan and Pike, 1984b] in a ten-week internship. Schoen-

blum learned enough UNIX to be comfortable using it to edit documents, send mail, transfer data, print things, etc., to accomplish his work. He interviewed UNIX users to identify twenty key UNIX commands and wrote multi-level help text for these commands. He then defined delimiter and token types needed for the system to parse UNIX commands. Finally, he wrote syntax definitions for all identified commands. The PIACH framework enabled such an accomplishment by only requiring data, rather than reprogramming, to create a help system for a new domain.

Two functions were provided to handle the new tokens which were used in UNIX but not in Lisp: the carriage return delimiter and the "anything" (or *) token. Changes to the parse table – altering the end-of-statement delimiter from ")" to carriage return and eliminating the ";" for comments – were provided as well. At the time, the COACH system only ran on Symbolics computers. A command caller which interfaced to a UNIX workstation over a Telnet [Kernighan and Pike, 1984b] connection was proposed but has not yet been tested.

This UNIX help system was experimented with by several people, and improved through iterative experimentation. No

formal study has yet been performed with it.

Experimentation With User Modelling

The PIACH framework supports experimentation with user modelling. The user model in PIACH is created and managed by rules (see Section 6.3). Additional slots can be added to the frames by adding instance variables to the adaptive object.

Experimentation With Help Presentation Strategies

The PIACH framework also supports experimentation with help presentation strategies. All help presentation is managed by rules. These rule sets have allowed continued testing and changing of the PIACH coaching strategies in the COACH implementation (see Chapter 6 above). Several students have experimented with the rule system to learn about adaptive strategies (Matt Kamerman, Kevin Goroway, Frank Linton, and Chris Frye).

These experiences demonstrated that the PIACH framework can be used as a shell for developing Proactive Interactive Adaptive Computer Help (PIACH) systems. The COACH implementation gives proof by demonstration that PIACH works in open systems, for different domains, and

supports experimentation with adaptation and help strategy. The next chapter describes experiments which showed PIACH can improve student performance.

8 Evaluation of the PIACH Adaptive Coaching Style

Two user studies have been performed to evaluate PIACH, one in 1988 and one in 1990. The first reports PIACH user perception differences. The second statistically demonstrates these differences and performance improvements as well. In this five session Lisp course, the system was found to improve both performance and perceived usability when compared to a version which offered only non-adaptive user-requested help. This is the first demonstration of an adaptive interface showing performance differences for users.

Enhanced interface features available to both groups may have improved productivity as well (e.g., the pointing device, on-line selectable help, separate input and output windows, real-time error detection, etc.).

8.1 1988 Pilot PIACH Study

A pilot study was conducted in 1988 to evaluate the value of automated adaptive help in the COACH system and to flesh out issues for the full scale 1990 study. Six programmers who had no knowledge of Lisp were recruited from research staff, programmers and co-op students at IBM T. J. Wat-

son Research Center. The three day course consisted of a classroom lecture each evening followed by a work period. The students worked through exercise sets, and responded to interview questions and quizzes. Each evening involved a new exercise set. One group performed work using COACH, the other group used a standard interpreted Lisp reader on a IBM PC-RT computer.

Many differences between the actions and reactions of the two groups were noticeable. Students appeared much more energetic and productive when they were using COACH. They used the on-screen help and they put their fingers on the screen. While the COACH students tended to experiment within the system, the other group tended to write ideas on paper. When, on the last night, the groups switched places, the behavior also switched.

However, technical difficulties and the small number of participants make formal analysis of the 1988 study uninteresting.

8.2 1990 Study; Demonstrating PIACH Usability Improvements

Major improvements to the pilot study were included in the 1990 study:

- The lecture format of the 1988 study was changed to a self-paced format in the 1990 study because it had appeared that the lectures made the students feel pressured. They seemed to believe that the difficulties they were having were caused by the lecture, when in fact, the course was designed to be difficult.

- The 1988 students appeared anxious and self-recriminating when they could not finish the entire exercise set provided for that evening. Thus, for the 1990 study, the three exercises from the 1988 study were combined into one unbroken problem set. This relieved some of the unnecessary performance pressure.

- In the 1988 study, the group using the adaptive automated help seemed to be enjoying themselves, while the other group did not, so a daily comment sheet was added in 1990 to the course to record the way students felt.

- Recorded audio interviews were added for the same reason.

- Improvements in the COACH implementation reliability made the mechanics less daunting.

- A version of COACH without an AUM was arranged for the control group. This allowed the study to concentrate on the value of an AUM, the central PIACH technology, rather than other ergonomic advantages of PIACH.

The 1990 study tested the hypothesis that an adaptive coaching paradigm can improve user productivity. Normally COACH adapts to its user and automatically offers help at an appropriate level of understanding for that user. A method was devised to focus the user study on the comparison of automatic adaptive help with user requested help. A control version of COACH was created which included all interface aids, but excluded the AUM, which had the effect of eliminating the automated adaptive help. It still separated user actions on the window panes from system actions when reporting errors and displaying user requested help (Figure 8). The study compared user experiences with this control version and experiences with the automated adaptive version.

8.2.1 Method

Nineteen employees of IBM T. J. Watson Research Center were recruited. They varied from summer interns to professional programmers. While all of these "students" had prior programming experience, none had previous experience with Lisp.

The students were recruited with an electronic poster. The poster solicited people who knew how to program but had no exposure to the Lisp programming language, and who wanted to participate in a short class/study teaching Lisp. Incentives to participate were sandwich dinners, exposure to the experimental system, and the promise of learning a new language.

The students were separated into an early session meeting from 5:00 p.m. to 6:00 p.m., and a late session meeting from 6:00 p.m. to 7:00 p.m., each day for five days. Attempts were made to assign students to whichever session best fit their schedules. Students were assigned at random to use the manual help or to use the adaptive help. By the time the course was underway, eight students were using the manual help system and eleven students were using the automatic adaptive help system.

Courseware created to support the user study include a course introduction, a Lisp tutorial, and an EMACS editor reference card. Materials used to evaluate the students consisted of a test given before the course began (pre-test), daily comment sheets and a test given at the conclusion of the course (post-test). In addition, audio interviews and student exercise solutions were used as sources of data. These are described in more detail below.

Pre-Test

Before the course began, the students were tested for their knowledge of Lisp and programming concepts in general (see Appendix B.1). The written pre-test was administered to them to collect background information and to insure that they had no working knowledge of Lisp. The test included questions to evaluate previous programming experience, to establish which programming languages the students knew, and to measure awareness of common programming concepts. Lisp-specific questions were asked to eliminate any students who had prior Lisp experience.

Precourse Materials

The COACH system runs on the Symbolics and on IBM PC-RT computers. In this study, students worked with Symbolics workstations with a screen layout as shown in Figure 8 in Chapter 6 above. The students were isolated from one another and not permitted to converse with each other. Some had separate offices; one group of three sat facing three different walls in a large office. All users in this "communal room" were members of the non-adaptive group.

Students were instructed in basic operations they would need to use, such as where the rubout key was on the keyboard and how to use the mouse. They were all given the same set of course materials. The materials consisted of a brief Lisp tutorial, a quick reference sheet for the EMACS editor, and an exercise set (see Appendix B.1).

Students were encouraged to use the computer help whenever possible. The tutorial was stapled to the exercises facing backwards to force them to turn over the tutorial to see the exercises.

The students were told they would only receive help from the experimenters in the case of machine problems, not in

learning Lisp.

They were instructed to read as little of the tutorial as they felt necessary to familiarize themselves with Lisp, to work on the problem sets in a self-paced manner, and to refer to the help windows often.

Lisp Tutorial

A tutorial presentation of the major Lisp constructs was provided on paper. It was written so as to not to solve the exercise problems, yet complete enough to aid the students with learning Lisp.

A motivational introduction listed advantages of Lisp as an application development language. The format of the tutorial was similar to a textbook; concepts were explained in an order so as to build on each other. Once a concept was explained it was followed by simple examples.

The brief nine-page tutorial outlined below can be found in Appendix B.1. Its breadth, including the range of topics a full Lisp course would cover, might be daunting to beginning students. They could have read all of this material, but were not required to. If they read it all, they would certainly have been introduced to many more challenging topics than could normally be mastered in five hours.

The topics of the tutorial were:

- *What is Lisp, and Why Use it?* This section described the value of an interpreted environment with dynamic variables and an extensible language.
- *Read-Eval-Print Loop.* This section extended the description of Lisp with a basic explanation of how Lisp evaluates s-expressions to give "effect" results and "side effect" changes to the Lisp environment.
- *Functions.* Here the idea of a statement, or function operator, was explained. Simple functions were described.
- *Lists.* To work with variables, a user must understand the basic Lisp data structure, which is a list. A simple explanation was included.
- *Conditionals.* The `COND` function was explained, with a small example to show how conditional execution could be accomplished in Lisp.
- **`MAP` & `LAMBDA`.** Complex macros which are typically used in Lisp for repetitive tasks were explained. Students could have used those to solve the most complex problems in the exercise set.

- *Defining Functions.* The mechanism for extending the Lisp language with DEFUN was explained, which would be used by students to complete the exercise set's data base exercises.

- *Repeated Computation; Iteration.* An alternative to the MAP family of macros, iteration, was demonstrated in order to give students another alternative for accomplishing their exercises.

- *Repeated Computation; Recursion.* The notoriously confusing method of calling a function from within that function to repeat operations, recursion, was demonstrated.

- *Data Structures; Property Lists.* The more sophisticated property list data structure was introduced.

- *Data Structures;* DEFSTRUCT. Finally, DEFSTRUCT, the sophisticated and complex macro which builds data structure slots with accessor functions, was introduced.

Exercise Sets

The exercise sets contained problems covering basic arithmetic operations, list operations, conditional execution, and

a small data base project (see Appendix B.1). To avoid a ceiling effect (all or many users completing all exercises), the exercise sets were intentionally longer than what a student could complete in the time available. Solutions to the first nine exercises consisted of expressions composed of built-in Lisp functions. Correct solutions indicated completion of exercises. When the students finished these single-answer questions at their own pace, they began the database project. The students were not told which Lisp functions they should use, nor how they should construct the database.

Arithmetic problems introduced the concept of evaluation order, forcing the students to understand that function names come first in Lisp s-expressions. For example, one student, while trying to accomplish

```
(times (plus 5 5) (plus 2 2))
```

was observed trying

```
((plus 5 5) times (plus 2 2))
```

but was able to understand the problem with the aid of COACH. The system instantly recognized that an error was being made and popped up an attention-grabbing error message in the immediate help window. When this error message

was noticed, the student was able to use the COACH on-line help to figure out the problem in the model of evaluation used to construct the statement. The student was able to understand the order of evaluation problem and fix the operator/argument order in the solution.

The list operations required for solutions to the exercises included CAR, CDR and, CONS, the concept of nested lists, and QUOTE. The students were asked to create lists of varying degrees of difficulty. The simplest was a single level list (1 2 3), and the most difficult involved a multiple level list that required an understanding of quoting.

A conditional statement was required to solve one of the exercises. The task required the student to cause the computer to print yes if a certain element was contained in a list. Students could have used COND and iteration or recursion to solve this problem. An easier approach using the MEMBER function could simplify the solution.

Unlike earlier problems, students had to write their own functions in solutions to the database project exercises. One might expect students to use the simplest data structure, a list, or more rarely, property lists, the first time they need accessor functions. The tutorial covered these data structures,

and the sophisticated DEFSTRUCT macro as well. Surprisingly, most students chose to use DEFSTRUCT instead of lists or property lists, all of which were covered in the Lisp tutorial. The reason most students gave for this choice of using DEFSTRUCT was, "It does everything for you." This indicates that they understood the value of a macro that writes functions the student would normally have to write.

The database problem was worded carefully to segment the solution into six small user written functions. For example, the students were asked to write a function to add a person to the database, and to write a function to retrieve a person's phone number from the database. This allowed a direct measure of productivity by the number of functions written.

Analysis of student solutions was used to compare productivity of the two groups. In addition, quality of user solutions was evaluated. Students' code was examined for appropriateness and sophistication of Lisp functions used, use of variables and condition checking as well as overall style.

Comment Sheets

During each of the five one-hour sessions, the students were given a comment sheet to record impressions (see Appendix B.1).

They were instructed to write as much or as little as they chose. The following questions were asked on the comment sheet:

1. How often do you look at the help screen while solving a problem?
2. How helpful is the help screen?
3. How helpful is the COACH window system, as compared to a line-based interpreted environment?
4. Observations about COACH? (Answers from this question were evaluated for perceived value or rating of the COACH environment.)
5. Observations about Lisp? (Answers from this question were evaluated for perceived utility of the Lisp programming language.)
6. What problems are you having?
7. What problem are you working on?
8. What is your motivation to learn Lisp? (Answers to this problem were given on a scale of one to ten.)

To compare the answers of the two groups, all the written answers were analyzed and coded as varying from zero to

five. Two readers evaluated each answer and assigned it a value without knowing which group it came from. These numerical values were used to evaluate the likelihood that the two groups had the same experience with the system (see Table 6 and Figure 11).

Interviews

Near the end of the course, six randomly chosen students from each group were interviewed for a few minutes on audio tape while they were working. The important question asked was:

- What do you find most helpful while solving a problem: the help screen, the selectable menus, or the tutorial?

Post-Test

At the end of the user study, the students were given a post test (see Appendix B.1). To measure the amount of Lisp learned by each student, questions covered the same material as the pre-test. In addition, questions about specific feelings toward COACH and Lisp were posed. Although similar to the comment sheets, these questions were worded differently to reveal more about the students' personal views.

8.2.2 Experimental Data and Analysis

The following sections give detailed descriptions and analyses of data taken in the study. The data recorded came from the pre-test, comment sheets from each day, saved exercise solutions from each student, verbal interviews of students during their sessions, and the post-test taken after the course was completed.

Pre-Test

The pre-test showed that all students selected were experienced programmers, but had no prior Lisp experience. Answers to the Lisp-specific questions showed that, except for the simplest cases, the students were unable to answer Lisp questions by guessing.

Saved Exercise Solutions

The COACH system internally stores information about each user. This internal representation is the student's adaptive user model (AUM). At the completion of users' work sessions, COACH creates two files. The first contains the user's work (Lisp code), and the second contains the user model (usage data for determining the level of the user's expertise for specific learnable units). Unfortunately, user model files were

not kept for manual help students.

The students' saved work and their saved user models showed that all students had finished the eight introductory exercises, those exercises which did not require defining functions. This was followed by a database project requiring students to write functions.

The comment sheets indicated that all the students had begun writing functions for the database project by the last session. Examining their work showed surprising differences in the percentage of the ten database project functions the students in the two groups had actually completed. The users of the adaptive system wrote an average of 2.5 functions, as compared to 0.5 for the users of the nonadaptive system. No user of the nonadaptive system wrote more than two functions. This is the most significant result of the study: on the average, *the users of the adaptive system defined five times as many of the functions required in the data base project.* In addition, the style and quality of functions written by the adaptive system users were much better than that of the control group.

One user of the adaptive system wrote a function which demonstrated astonishing progress for five hours of experi-

ence. This function, included below, demonstrates an understanding of parameters, scoping and formatting, as well as boundary checking, DEFUN, and lists.

```
(DEFUN add-person (name phone lang)
  (COND
    ((MEMBER name USERS))
    ((EQUAL USERS nil)
      (SETQ USERS (LIST
        (LIST name phone lang))))
    (T (SETQ USERS (CONS
        (LIST name phone lang) USERS)))))
```

Comment sheets

At the end of each day, the students were asked to fill out a comment sheet. The data from the comment sheets are the students' ratings of various aspects of the course. The values vary from zero to five, zero being the worst and five being the best. The statistical analysis was done using a two-tailed p-test, to determine the probability (p) that the mean rating of the users of the adaptive system was different than the mean of the control group. This was done using Welch's Method [Brownlee, 1984] (see Table 6). The strongest result

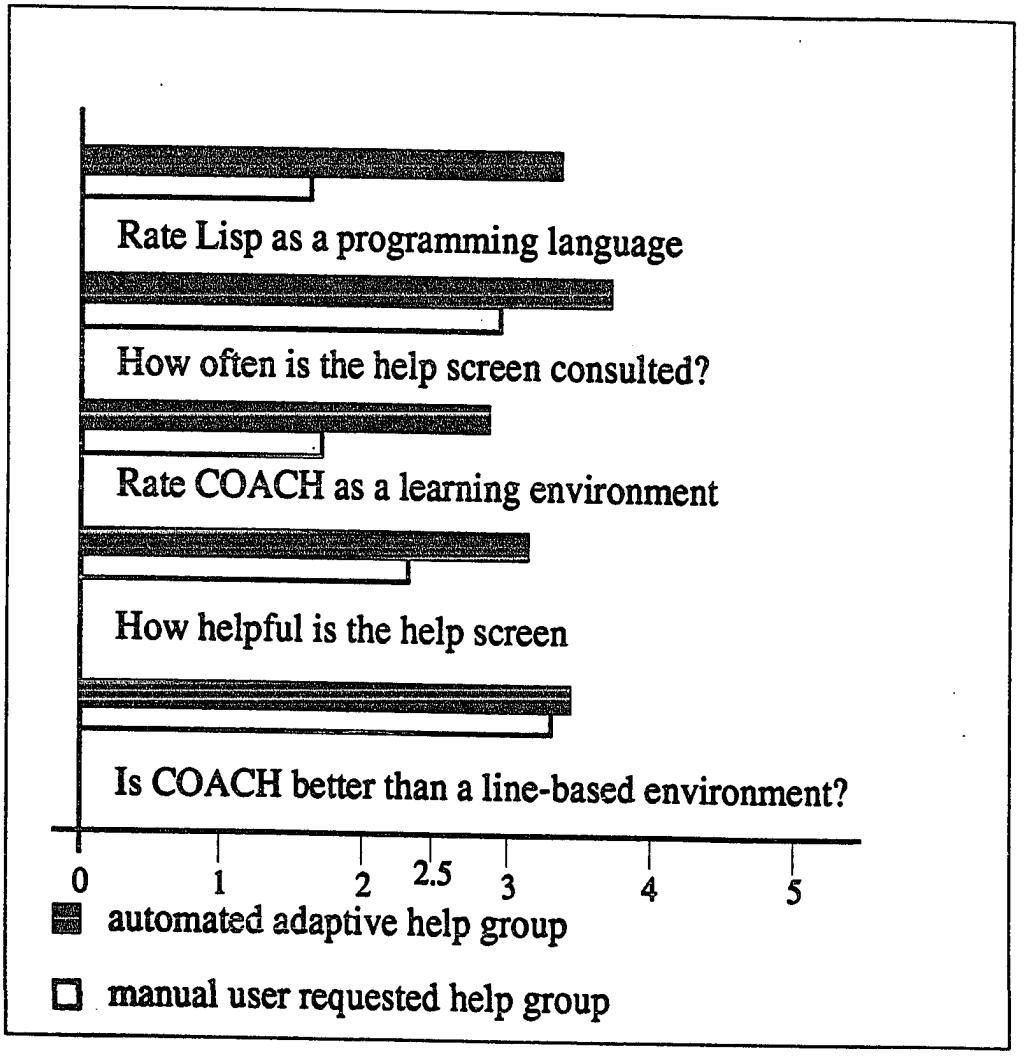

Figure 11: Graphical depiction of comment sheet data from Table 6.

concerned users' ratings of Lisp as a programming language. The users of the adaptive system indicated that they had a higher regard for the Lisp language; the means of the answers for the two groups have a .01 probability of being the same as each other, giving a 99 percent probability that the adaptive help group had a higher regard for Lisp than the control group ($p = 0.01$). The question, "How often do you look at the help screen while solving a problem?" showed that users of the adaptive system used the help screen more often ($p = 0.04$). The results from the question, "How helpful is the COACH window system, as compared to a line-based interpreted environment?" showed that both groups thought the COACH environment was an improvement over a standard interpreted environment. The mean rating for the question "How helpful is the help screen?" seemed to be higher for the users of the adaptive system; however, with the study sample size it did not prove to be significant ($p = 0.11$). This is probably because it is a hard question to answer. The question might have yielded better data if it had asked the students to compare the help screen to the tutorial or some other form of help.

Interview Tapes

Students were interviewed on audio tape during their work sessions with standardized questions. Six randomly chosen individuals from each group participated in these interviews. The data collected from the interview tapes show differences in the ways the two groups utilized help while solving a problem (see Table 7 and Figure 12). All users reported the usefulness of menus to ask for help. The manual user requested help group received help messages for syntax errors, the kind of help for which an interpreted Lisp environment is known. They reported that this computer presented help was not particularly useful. All users of the automated adaptive help reported finding it useful. While only one member of the manual group reported making use of the tutorial packet, all interviewed members of the adaptive group reported the tutorial useful. While one member of the manual group reported relying on trial and error to solve problems, none of the adaptive group reported using this technique.

In Figure 12, it is notable that students in the adaptive system group used all the different types of help available to them, while students in the nonadaptive group did not.

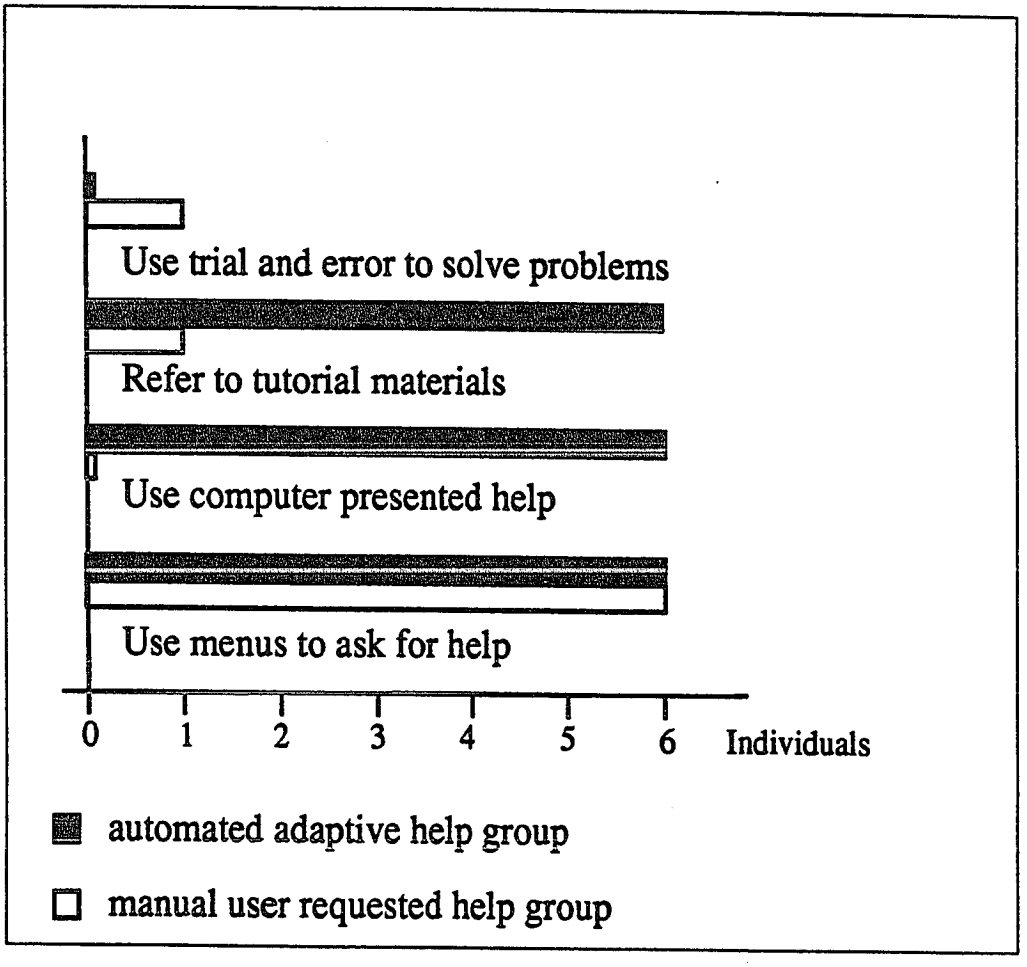

Figure 12: Number of users reporting use of each Method to solve problems. Data recorded from the six people interviewed from each of the two groups from Table 7.

Post-tests

A post-test was given to the students at the end of the course. The post-test asked the students how comfortable they felt with the Lisp language. The data collected from the post-tests show the difference in comfort levels between the two groups. Out of the six post-tests completed by users of the nonadaptive group, two students felt somewhat comfortable and four were uncomfortable. Of the nine post-tests completed by users of the adaptive system, three students were comfortable, five were somewhat comfortable, and one was uncomfortable (see Figure 13).

8.2.3 Discussion

The data demonstrate differences in self-assessment and performance between users of an adaptive automated help system, as compared to users with manual help.

The terse nature of the tutorial purposely masked the quantity of information with which the users were familiarizing themselves. The amount of knowledge to which the students were exposed was close to what students might be expected to master in a full semester Lisp course.

Although the students were sometimes frustrated, they

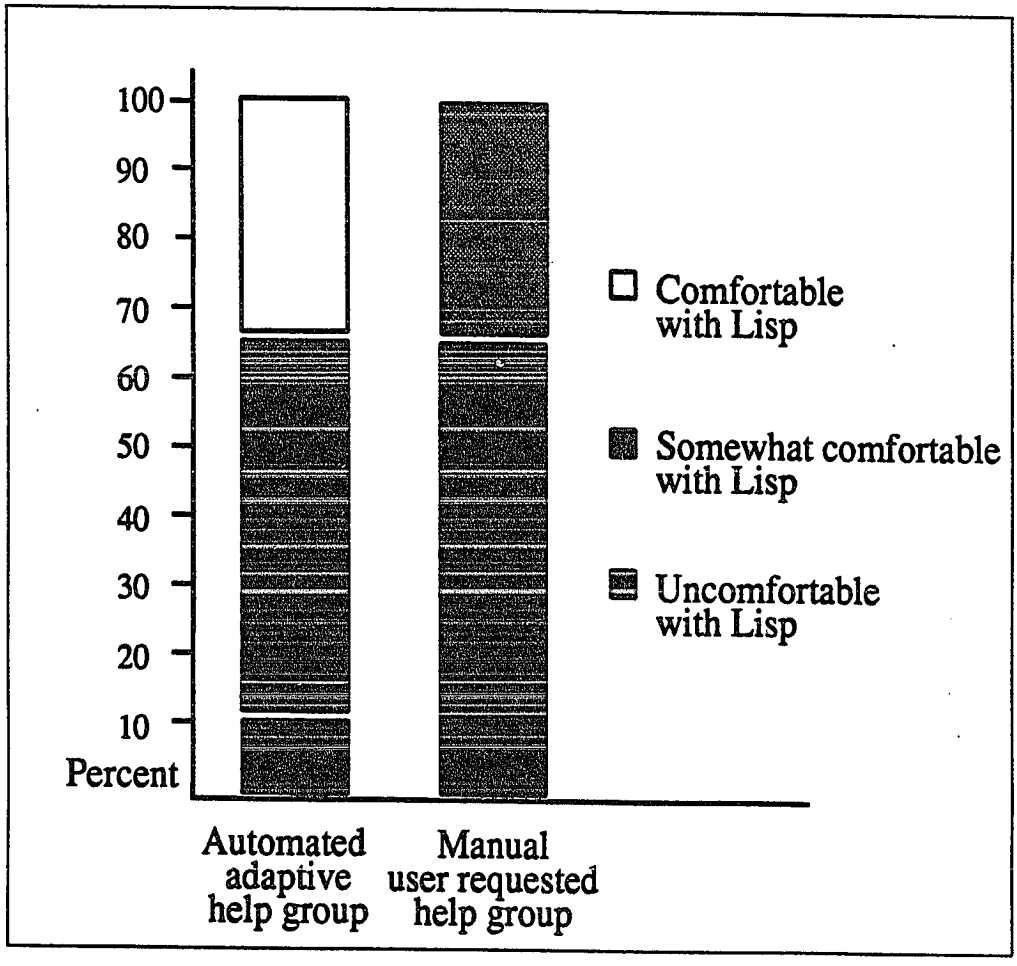

Figure 13: Comfort levels reported by students at completion of course. Percentages are calculated for nine students from the adaptive automated help group and six students from the manual help group who returned the post-course questionnaire.

learned a lot of Lisp. The goal of requiring them to use the help system to solve their problems was achieved. Even though large performance differences were found between the groups, the nonadaptive group still performed extremely well. When compared to the amount of work a novice might accomplish in a typical learning environment, it is clear that the COACH environment was a significant aid. Even the nonadaptive users described their interface as an improvement over the usual tools that are available (Figure 11).

The pressure of learning so much Lisp in such a short time without any human teacher help was a challenge. Although all students completed the study, one of the manual help group students required extensive persuasion to continue after the third day. A comparison of the current study to the 1988 pilot COACH study showed that the pressure of the amount of material to be learned was decreased by the self-paced presentation.

8.3 Future Work

This study has shown that an adaptive automated help system can increase user performance. Many important questions remain unanswered (see Chapter 9).

To simplfy the user study, the most experimental rule was eliminated from the COACH adaptive user model. A small number of adaptive rules were used to change the quantity and quality of help given to a user for each function, token, and concept. It is important to examine the various kinds of adaptation and knowledge bases in such an adaptive user interface. The value of each individual rule should be studied.

A PIACH also records user information concerning required and related knowledge. As described in Section 6.3.2 rules can use this information to describe alternative solutions or point to related *learnable units* when a user is struggling. For simplicity, the "encourage exploration" presentation rule was deactivated for the study. Additional rules could interject syllabi on which to tutor a user in such a situation (see Chapter 9). It would be interesting to study the value of such facilities, which may distract students from their task.

This study tested the effectiveness and importance of an adaptive system with Lisp illiterate users. The ways in which COACH will be helpful to experts is probably quite different from the ways it will be helpful to novice programmers. Experts will benefit from COACH's "Level 3" exam-

ples and "Level 4" complete syntactic descriptions of functions. COACH leaves experts alone when they are working on something with which they are experienced. These experienced Lisp programmers will benefit from the fact that COACH keeps track of context dependent situations, scope, and undefined variables, while exposing the user to the relationship between functions and concepts. Novices, on the other hand, find the changing adaptive help for functions and tokens quite useful for learning the syntax of simple functions and token types. While studies showing the different benefits for different users would be straightforward, they were beyond the scope of this experiment.

8.4 Conclusions

In this study, significant differences were found between students who used automated adaptive COACH help and students who had only manual COACH help.

While the responses to the comment sheet question concerning motivation during work sessions did not show a difference between the two groups, other indicators did. One might expect the group with less computer support to make greater use of the paper tutorial; however, the converse was

true. Both groups had the same access to the paper tutorial and on-line help. While the group with manual COACH help only valued the user-requested help, the automated adaptive help group utilized all available materials, the Lisp tutorial and user-requested COACH help as well as automated coach help (see Figure 12). Students from the adaptive group reported feeling more comfortable with Lisp and also completed many more of the exercises than the control group (see Figure 13).

The automated adaptive help system succeeds in improving productivity and raising motivation to use available materials.

This chapter has described efforts to evaluate the PIACH framework. The study raises many interesting questions for future research. The following chapter discusses these in more detail.

Question (Answers on a scale of 0 – 5)	Manual Mean	Adaptive Mean	p-value
Rate Lisp as a programming language.	1.90	3.36	0.01
How often is the help screen consulted?	3.16	3.91	0.04
Rate COACH as a learning environment.	1.97	2.97	0.05
How helpful is help screen?	2.44	3.20	0.11
Is COACH better than a line-based environment?	3.31	3.49	0.70

Table 6: Data collected from the comment sheets shows that the students using the adaptive version of COACH liked Lisp more than the other group, consulted the help screen more often, and rated COACH higher as a learning environment. Although the students using the adaptive COACH tended to find the help screen more helpful than the other group, the difference was not significant. Notice that both groups rated COACH as better than a standard line-based environment. In the above table, p-value is the probability that the means in two samples are the same. This data is shown graphically in Figure 11.

Learning Materials Used	Manual	Adaptive
Asks COACH for Help	6	6
Uses COACH presented Help	0	6
Refers to Tutorial	1	6
Uses Trial and Error	1	0

Table 7: Students using different methods to solve problems. Of six students interviewed in the manual group and six students interviewed in the adaptive group, the adaptive help group found more of the support materials useful. This data is shown graphically in Figure 12.

9 Future Research Goals

The work presented in this dissertation raises many questions concerning pedagogical approaches, adaptive interfaces and tools for exploring educational scenarios. The PIACH framework and the COACH implementation can be useful for future research concerning these questions. The following is a list of productive directions for continuation of the PIACH work:

1. Further experiments could establish the validity of particular **instructional techniques** in specific situations.

2. Specific **adaptive mechanisms** should be more fully studied to establish their **impact** on the user.

3. Currently PIACH gives advice addressing user problems it identifies. The use of PIACH to actually perform the solutions to these problems as an active **agent** could **save** users from rote work.

4. PIACH could be expanded to work with **graphical interfaces** to coach in the use of menus, drawing and other non-textual interactions.

5. Integrating **multi-media** help such as video graphics or audio into the system would be valuable.

6. **Tutorial curricula** could be added to PIACH to make the framework useful for teaching a syllabus.

As well as being a research platform, the COACH PIACH is already intrinsically useful as a coaching interface. Development work could make the technology more usable and available:

1. The adaptive help technology could easily be integrated with other program development tools in a **standard work environment.**

2. The COACH system could be **ported** to be made to run with popular computer operating systems.

The COACH system is more than a proof-by-demonstration of the PIACH framework. It is a shell which allows researchers to fill in the details of when, how, where and why adaptation improves help applications.

The rest of the chapter discusses future directions for PIACH research in more detail.

9.1 Future Research

1. Evaluating instructional techniques.

   Various basic instructional techniques can be embodied in a computer coaching system. In a useful exploration of the goals and techniques of teaching, Alan Collins and Albert L. Stevens [Collins and Stevens, 1983] put forward a list of ten such techniques demonstrated in computer teaching systems:

   (a) Selecting positive and negative exemplars

   (b) Varying cases systematically

   (c) Selecting counter-examples

   (d) Generating hypothetical cases

   (e) Forming hypotheses

   (f) Testing hypotheses

   (g) Considering alternative predictions

   (h) Entrapping students

   (i) Tracing consequences to a contradiction

   (j) Questioning authority

   A fruitful area of research would be the formal evaluation of these instructional techniques. It remains to

be proven which are most effective, which can be used together, and, most vital, which are appropriate for a particular situation. The PIACH system is in a unique position to allow teaching techniques to be tested objectively. They would be embodied in coaching rules.

2. How do **adaptive mechanisms** impact the user?

   The adaptive strategies present in the COACH system used in the 1988 and 1990 studies were arrived at by informal experimentation. Guinea pig users worked with the system to test various adaptive and presentation strategies. A researcher can change rules to alter the system's coaching actions and strategies (see Chapter 7). Formal studies to determine which strategies are best could be set up to address issues in education, cognitive psychology and cognitive science. Many questions could be easily tested. For example, is it better, when users are first exposed to a performance help level, to show them syntax and description, or would it be better to simply focus on an example? Presently the system waits to show a user related knowledge until the user has shown experience with the learnable unit. The current hypothesis is that too much information might over-

whelm a beginner. A different hypothesis might state that the novice should instead be provided as much information as possible when just getting started. Exploring such hypotheses in more detail would give insight to better understand student cognitive models. PIACH is designed to explore such issues.

3. The use of **agents** in an adaptive teaching interface.

   Should the computer tell a user how to do something, or should the computer do it for them? The most unobtrusive use of the adaptive paradigm is the advisory one; the framework tells a user how to do things. In an agent paradigm, on the other hand, when the computer knows something needs to be input by a user (e.g., an open parenthesis), the computer takes control and types the solution. In such a paradigm, when the computer identifies some way of simplifing what a user needs to do, the computer does it. The computer, then, has built a new instruction to come to the aid of the user. The computer is building a private, helpful set of tools for the user, a language the user and the computer both know. To the extent that the things the computer tries to do to the user's program are what the user really needs and

wants, and to the extent that either the user can easily ask for it or the computer can recognize the need for it, the agent is helpful.

PIACH does not currently include agents. The hypothesis is that if users do not have to perform things themselves, they will not learn them. Another reason that agents were not included in the coaching paradigm was based on the hypothesis that the private interface that agents provide could be difficult for a teacher or colleague to understand when the computer failed to be of help.

4. Creating a PIACH to work for **graphical interactions.**

The framework was created for a textual domain, but it could be adapted to give help with graphical interfaces. Brad Myers' [Myers, 1986] seminal automated interface construction system, PERIDOT, addressed issues concerning graphical agents. PERIDOT assumed certain goals of a user wanting to build a graphical interface (e.g., words positioned above each other should be left-justified). It used mixed initiative to test its hypotheses in designing a user interface (e.g., if the user did not like the justification, the user could change it). The kind of

user interface technique a user had previously applied was used to predict what to do in a particular situation.

PERIDOT demonstrated the possibility of a graphical coaching system. While a graphical PIACH could give help in textual terms, the idea of giving graphical advice is even more interesting. PERIDOT used an advisor mouse to show what the system was doing, or to demonstrate a graphical action that a user could perform. This advisor mouse was an icon that looked like the mouse animal, which moved around on the screen and highlighted or acted on spatial things. Such a visual mouse could be used in a graphical PIACH to draw, highlight things with color, or blink in helpful ways. Apple's System 7 operating system used cartoon-like balloons to present help text for a graphical item on the screen [Danuloff, 1991].

Extending a graphical coaching system like PERIDOT to provide adaptive help would require adding the following elements from PIACH:

- a more sophisticated AUM to record the abilities of a user.

- an experience hierarchy to model help needs of a user.
- a mechanism for changing help style to accommodate users with different expertise levels and abilities.
- an arbitration mechanism for choosing how to communicate with users based on their individual abilities.
- a mechanism for adding user examples and new graphical statements to make the system extensible.
- a mechanism for relating concepts within the domain for reasoning about pedagogical strategies.
- a language for creating graphical help systems. Just as a formal language description of a domain allows PIACH to reason about user work (see Section 6.4), so a formal graphical language could describe an interface's graphical commands, thereby allowing a PIACH to reason about graphical user work in a general way. The VIEW implementation of Selker's "Elements of Visual Language" [Selker and Koved, 1988; Selker and Schoenblum, 1990] demonstrates such a formal language description capable of inter-

preting real graphical interfaces.

5. Integrating **multi-media** help.

   Video segments could replace text for explanations in PIACH. Hardware and demonstrations integrating video and computers are becoming popular. An early work by Steve Gano, "Movie Manual", integrated text, menus, and video to demonstrate automotive repairs, such as changing the oil [Gano, 1982]. This technology could be harnessed to augment the textual help format used in PIACH. The Anchored Instruction researchers at Vanderbilt University have proposed using COACH to build a multi-media help system for teaching children.

6. Integrating **tutorial curricula** with PIACH.

   This thesis demonstrates an AUM-based teaching aid based on the assumption of a goal-directed user. While most of peoples' lives are spent trying to achieve their own goals, we all go through a period of schooling – a time when others define our goals. The coaching scenario supports a user's goals, leaving any "syllabus" or goal definition up to the user or a human teacher.

The framework could easily support more directed teaching materials, as demonstrated in systems like the Lisp Tutor [Reiser *et al.*, 1985]. The network of relationships between learnable units in PIACH could be made to work with a check list, (an overall basis-set, see Section 6.3), to go through teaching materials in a sequence. Specific syllabus teaching materials with assignments and problems could be represented in domain knowledge frames (see Section 6.3.1). Adding a rule "what-to-teach-next" to the PIACH rule set (see Section 6.4), could interface between the syllabus and the AUM to select appropriate teaching materials.

Such a system would have curricular goals, a syllabus, and the ability to evaluate a user relative to these goals. In addition to giving programmed lessons as other tutors do, such a system would be able to follow and help users in their programming, even when they were not doing exactly what the curriculum wanted them to do.

9.2 Future System Development

As well as being a research system, COACH can be used in real work. Below, two efforts are outlined which would make

the scenario more useful and available to users.

1. Integrating PIACH into **standard work environments.**

   Users of the Hemlock EMACS-like editor can already integrate PIACH into their Lisp programming work. Certain integration projects could make the framework more widely available:

   - User interface environment aids.

     Rules could be added to PIACH which provide already well known agents. As demonstrated in Do What I Mean (DWIM) [Teitelman and Massinter, 1981], the system could correct spelling errors and correct variable naming or function naming. Agents could also search for conflicts in other functions, similar function definitions, fix data type uses, etc.. Agents could also create solutions for equations and allow users to work with alternative representations, as demonstrated in the Mathematica [Wolfram, 1988] mathematical problem solving and visualization system.

   - Efficiency improvements.

Improved data structures could help the system. The help text, rules and user interface grammars could be moved into improved speed-of-access data structures for increased performance on large user languages.

Improved algorithms could also help the system. The system could cache text that is relevant to the particular user's AUM, leaving unneeded help information in files on disk. An improved scoping algorithm could allow the COACH system to reconsider smaller pieces of users' work when users make changes.

2. **Porting COACH** for use on different computers.

COACH has been ported to work on the IBM PC-RT running Mach, as well as the Symbolics computer family [Moon, 1987](see Appendix A). Automated tools have been built for converting the COACH Flavors source code into CLOS. This allows COACH to run equivalently with one set of source code under Flavors [Symbolics, 1986] or CLOS on Common Lisp. A window-frame has been built to create a COACH interface with the Hemlock text editor under CLX on the X11-windows

system [Scheifler, 1987]. Since the Mach Lisp tools can be used on non-Mach machines, the system should be able to run on Common Lisp implementations that support the CLX interface to Xwindows without modification.

Porting a COACH system to a personal computer platform is also underway.

Appendices

A Implementation Status

COACH,which demonstrated the viability of the PIACH framework, was written in Lisp Flavors. Versions of it run on Symbolics Genera-6 through Genera-8 [Moon, 1987]. A conversion procedure allowed the Symbolics version to run identically on Mach Lisp with the Common Lisp Object-oriented System (CLOS) on an IBM PC-RT. COACH is able to present appropriate help at the speed at which the user is typing. The COACH system has been shown to function as a shell for help systems for various text-based interfaces (e.g., Lisp, UNIX and parts of the General Markup Language (GML) text formatting language).

COACH's object-oriented modular construction is faithful to the architecture presented in Chapter 6. One major programming object "flavor"is used for each of the parts of PIACH (see Figure 7). This makes it reasonable to judge the ideas in PIACH by the actual demonstrated technological achievements in the COACH system.

The object-oriented design demonstrated its value in several ways. It localized structurally distinct parts of the ar-

chitecture. This simplifed the use and interaction of multiple representations in the system. It made changes to the program easier. This design also aided porting COACH to another operating system.

The multilevel parser was implemented by COACH-READER and COACH-PARSER. The user model contains the USER object frames. The interpreter to which COACH sends user code (e.g., the Genera Lisp interpreter) is also treated as though it were an object. The domain representation is contained in the production system PS frames. The reasoning is done by the PS object using coaching and domain knowledge. Finally, the WINDOW-FRAME object implements all user interface operations; it implements the user input text editor, user menus, and windows to present user results, output and coaching help.

The object-oriented design of COACH allows it to be modified easily by making changes to the PS and USER objects. To demonstrate the validity of COACH as a shell for testing adaptive teaching, several students modified and experimented with adaptive and help presentation rules. Some of these students did not have previous programming expertise. Input and output screen layouts were changed without

additional programming, simply by changing tables.

Porting COACH to the IBM PC-RT under the Mach operating system and the Xwindows system was achieved in a few months. A system which would convert Symbolics Flavors code into Common Lisp was written. A new `WINDOW-FRAME` had to be written for the Xwindows system. All other COACH source code was identical in the two implementations.

On a Symbolics 3640 computer with eight megabytes of memory, the system was able to keep up with a user's typing for reasonably sized Lisp statements. Performance problems occur when a user leaves mistakes unfixed. A few errors can still be tolerated before performance degrades noticeably. In those cases, the system re-parses all statements scoped by the user's errors each time a user types a character.

COACH was originally demonstrated to support Lisp programming. To show its usefulness as a PIACH shell (see Chapter 7), courseware was created to support use of the UNIX command language. The COACH Common Lisp help system has courseware for approximately 40 important Common Lisp functions (see Appendix C.1). It automatically constructs help for all other functions by accessing system argument list data and accumulating examples from users.

The UNIX help system has courseware for 20 basic UNIX commands (see Appendix C.6). Its output has yet to be connected to a UNIX shell. Pilot courseware packages have been explored for the KRep knowledge representation system [Mays *et al.*, 1988], and the GML tagged text markup language [Perry, 1984].

The COACH implementation demonstrates that an adaptive user model can currently be constructed in real time in a workstation-sized computer. It demonstrates that such an implementation can be useful across domains and for experimentation with adaptive approaches for teaching. Chapter 9 described directions and opportunities for the implementation.

B User Study

B.1 Materials

Precourse Assessment Exam

Precourse Assessment Name _____

1) Please describe briefly your current professional responsibilities.

2) What programming languages do you know?

3) Are you now, or have you ever been a professional programmer?

4) Please describe briefly the programming accomplishment of which you are most proud.

Table 8: First page of quiz given preceeding the course to calibrate the subjects and validate their lack of Lisp experience.

The following questions are designed to assess your knowledge of programming concepts, so that we may determine the effect of the instruction and experience in this study. **Knowledge of these concepts is neither expected nor required. If you are unfamiliar with them, leave the space blank.**

1) What is a computer program?

2) What is a variable?

3) What is a loop?

4) What is a conditional statement?

5) What is recursion?

Table 9: Second page of quiz given preceeding the course to calibrate the subjects and validate their lack of Lisp experience.

The following questions are designed to assess your current knowledge of the Lisp programming language, so that we may determine the effect of the instruction and experience in this study. **Knowledge of these concepts is neither expected nor required. If you are unfamiliar with them, leave the space blank.**

How would Lisp evaluate the following:

1) `(setq language 'Lisp)`

2) `(cdr '(100 200 300))`

3) `(list 'begin 'end)`

4) `(equal 'a a)`

5) `(member 'b '(a b c))`

6)
```
(cond ((greaterp 1 3) (print 'first))
      ((lessp 1 3) (print 'second)))
```

7) a. `(defun product (x) (times (x x)))`
   b. `(product 3)`

8) `(cons 'head '(shoulders))`

Table 10: Third page of quiz given preceeding the course to calibrate the subjects and validate their lack of Lisp experience.

Last Evening Exam

Post-test Name _____

How would Lisp evaluate the following:

1) `(setq dog-name 'fido)`

2) `(car '(a b c))`

3) `(list 2 'end)`

4) `(member 'b '(a b c))`

5)
```
(cond (NIL (print 'yes))
      (T (print 'no)))
```

6) a. `(defun my-plus (x) (+ (x 5)))`
   b. `(my-plus 3)`

7) `(cons 'a '(b))`

8) Write a single Lisp expression to compute: $25^2 + 14$

9) Make a list containing the names Washington, Adams, Jefferson, and give it the name PRESIDENTS.

10) How many paremeters can the SETQ function take?

11) How many parameters can the PLUS function take?

12) What concept, in Lisp, do you feel was the hardest to grasp?

13) Do you feel comfortable enough with Lisp that you feel you could solve simple problems in different ways?

14) How do you feel about the Lisp language in general?

15) What was the last problem (from the problem sets) that you were able to begin work on?

Table 11: The exam taken after the course was completed to evaluate total progress.

Nightly Comment Sheet

Comment Sheet – Name _____ Date _____

We will give you this handy comment sheet each day.
Please answer these questions as simply or verbosely as you like.

How often do you look at the Help screen while solving a problem?

How helpful is the Help screen?

How helpful is the COACH window system, as compared to a line based interpreted enviroment?

Observations about COACH?

Observations about LISP?

What problems are you having?

What are you working on?

Motivation to learn lisp: Circle one

Un-motivated	1	2	3	4	5	6	7	8	9	10	Motivated.

Table 12: This comment sheet was provided to accumulate specific observations as students had them while using the help systems.

Problem Set

Problem Set Name _____

1) Write a Lisp expression to compute:
`(3 + 17) * (19 - 4)`

2) Write a Lisp expression that combines CAR and CDR to return x, when applied to the list:
`(a b x d)`

3) Write a Lisp expression that combines CAR and CDR to return x, when applied to the list:
`(a (b (x d)))`

4) Construct the following LIST by using only atoms and CONS.
`(January February March)`

5) Construct the following LIST by using only atoms and CONS.
`(January (February (March (April))))`

6) Create a variable YEAR and give it the value of the names of all the months of the year.

7) Make a list containing the names Washington, Adams, Jefferson, and give it the name PRESIDENTS.

8) Write a Lisp expression to test if Jones is in the list of Presidents.

9) Create a variable called FIRST and give it the value 10. Create a variable called SECOND and give it the value 10. Write an expression that compares FIRST to SECOND, checks to see if they are equal, and prints the word EQUAL if they are.

Table 13: First page of exercises stapled back to back with tutorial as a set of student goals.

10) Construct a data base. Each entry in the data base will refer to one person and will be composed of three parts:

1. The person's name
2. The person's phone number
3. A list of the programming languages the person knows

A sample entry may look like this:
`((SUSAN WILLIAMS) 3398 (C BASIC COBOL LISP))`

Write the following functions:

add-person Add a person to the data base

get-phone Given a persons name, retrieve their phone number

get-name If the name is in the data base, it returns the name.

get-language Returns the list of languages

change-phone-number Changes a person's phone number

add-language Adds a language to the language list

Using these existing functions, complete the following exercises.

12) Write a function that returns a list of the names of people who know the language `FORTRAN`.

13) Write a function that returns a list of names and phone numbers in alphabetical order.

14) Write functions that return the average number of languages known by programmers in the data base.

15) Write functions that return a non-repetitive list of languages known by any programmer.

Table 14: Second page of exercises stapled back to back with tutorial as a set of student goals.

Students' Lisp Tutorial

Lisp Course Manual
Ted Selker and Kevin M. Goroway

What Is Lisp, And Why Use It?

Lisp advocates choose Lisp to write programs which create languages, programs, and complex data structures, especially for Artifical Intelligence. Lisp is chosen for its special features that aid program development; it has advantages for prototyping any program.

Lisp is normally used as an *interpreted* language by program developers; the computer runs instructions as they are encountered or typed. This allows a user to try out pieces of a program without the normal compile, link, load cycle. A user can interactively see how programs work as they are written.

Lisp environments have *dynamic storage allocation*; when a user program needs new variables the user need not do anything, except use it (the user does not have to specifically allocate memory).

Lisp is an *applicative* language; functions are applied to data. Among other things, this makes it easy to work on complex data structures and allows programmers to create their own languages. Lisp is *extensible*; new user functions become part of the language as they are defined.

Read-Eval-Print Loop

Everything typed into a Lisp listener is evaluated. The computer interprets typed-in text as statements or *s-expressions*. These statements can be functions or atoms. Atoms, such as 7 or Red, are things that are *defined* . Atoms return their defined value. The atom T always evaluates to *true*. Only the *null set* and the atom NIL return a *false* value.

Functions

Functions are defined as a "(" followed by a function name and any input *argument parameters*, followed by a ")". Functions return an output (i.e., its *effect*; such as a number). The Lisp function PLUS, adds numbers:

```
(PLUS 2 4)
6
```

PLUS is the name of the function, and 2 and 4 are called *arguments*. Functions normally evaluate their arguments. This can be demonstrated by looking at the following bit of Lisp code.

```
(PLUS 2 (TIMES 3 5))
17
```

The number 17 is returned (3 times 5 is 15, and 2 plus 15 is 17).

On the other hand, if you want to use something literally in a function call (i.e., 007 or Red), the QUOTE function will keep its arguments from getting evaluated.

```
(QUOTE red)
RED
```

Normally, Lisp would look for the value of red, but since it has been quoted, this step is skipped.

In order to make it easier to keep things from being evaluated the QUOTE function can be invoked with a single forward quote (').

```
'red
RED
```

Notice that there are no parenthesis surrounding the 'red as these are not needed when using '. Some functions in Lisp have *side effects* which change the Lisp environment. For example:

```
(SETQ color 'red)
RED
```

sets the value of the *atom* color to be "red". Then, when one asks for the value of the atom:

```
color
RED
```

Lisp responds with "red", as you might expect.

Lists

Function calls (i.e., (plus 2 4)), can be viewed as lists. Lisp is designed to operate on lists. A list can also be considered a data structure, in that it can be used to represent groups of data. One way to make a list is to use the function

```
(LIST 1 2 3)
(1 2 3)
```

which returns a list containing the elements 1, 2, and 3, denoted by parenthesis surrounding the elements. Lists can be looked at or accessed with the built-in functions *car*, which returns the first element in a list, and *cdr*, which returns the list without the first element.

```
(CAR (LIST 1 2 3))
1
(CDR (LIST 1 2 3))
(2 3)
```

Cons is used to create lists.

```
(CONS 0 (LIST 1 2 3))
(0 1 2 3)
```

Lists can be searched with the use of *member*.

```
(MEMBER 1 (LIST 1 2 3))
(2 3)
(MEMBER 8 (LIST 1 2 3))
NIL
(MEMBER 3 (LIST 1 2 3))
(3)
```

MEMBER returns NIL if the element searched for is not found in the list, otherwise, it returns a list beginning with the element searched for, and containing all elements after that element. of the list.

QUOTE can be used to simplify the use of LIST also:

```
(LIST 1 2 3)
(1 2 3)
```

can also be written:

```
'(1 2 3)
(1 2 3)
```

CONDitionals

COND can be used to make decisions. The basic structure of a conditional statement is:

```
if (condition) then (perform action)
```

in Lisp:

```
(COND (condition1 action1)
      (condition2 action2)
      .
      .
      .)
```

would test the conditions until one is found to be true, and then perform the action associated with that condition. For example:

```
(COND (T 5))
5
```

returns the value 5 since the *atom* T evaluates to true.

```
(COND ((EQUAL 2 2) 'yes))
yes
```

returns "yes" since 2 is, in fact, equal to 2.

Maps & Lambdas

Data structures hold data in meaningful groups, and to utilize this data we must look through the data structures, often applying functions to each bit of data. The MAPCAR function can be used to operate on lists:

```
(MAPCAR #'ABS '(1 -2 3 -4))
(1 2 3 4)
```

Notice that this MAPCAR applied the function ABS (absolute value) to each member of the list (1 -2 3 -4) and returned the list consisting of the answers. The #' preceding the ABS tells Lisp to evaluate it.

To group several functions together in Lisp you can make another function (or *lambda list*); for MAPCAR, if you want to apply several functions to each element of a list you surround them with the special function LAMBDA.

```
(MAPCAR #'(LAMBDA (x)
            (ABS
               (TIMES 2 x)))
          (1 -2 3 -4))
(2 4 6 8)
```

This LAMBDA function creates a new function that multiplies its argument by 2, and returns the absolute value of that answer.

Defining Functions

To name a LAMBDA function for later use you could:

```
(SETQ function-name '(LAMBDA ...))
```

but this is inconvientient. Lisp provides the DEFUN macro which does just that. DEFUN can be used to create named functions, and extend the Lisp language as follows:

```
(DEFUN abs-times-two (x)
   (ABS
      (TIMES 2 x)))
ABS-TIMES-TWO
```

which, when called,

```
(ABS-TIMES-TWO -6)
12
```

does just what we intended it to do.

Repeated Computation

Lisp supports repeated computation very well; both iteration and recursion are available.

Iteration

Iteration can be done by using the DO macro, which allows multiple variables to change values during the execution of the loop. The DO macro allows for a halting condition and return value to be specified.

```
(DO ((x 5 (1- x)))
    ((LESSP x 1) T)
    (PRINT (TIMES x x)))
25
16
9
4
1
T
```

This construct first sets x to be 5. It then checks if $x < 1$. Since it is not, the PRINT function is executed. On each of the following executions of the loop x becomes $x - 1$, until the halting condition ($x < 1$) is met. When the halting condition is true, the DO construct returns T, since we instructed it to do so. This return value could have been anything evaluable.

Recursion

Recursion is also supported in Lisp. Any iterative problem involves doing parts of the problem in sucession. Recursion is the practice of calling a function with successively smaller sub problems to solve the problem.

This DEFUN defines the previous DO in a recursive manner.

```
(DEFUN squares (x)
   (COND ((LESSP x 1) T)
         (T (PRINT (TIMES x x))
            (squares (1- x)))))
SQUARES
```

The result would be the same, when invoked:

```
(squares 5)
25
16
9
4
1
T
```

Data Structures

The most common data structure in Lisp is a list, but Lisp has facilities for other data structures.

Property Lists

Property lists provide symbols with modifiable named components. A property name can be associated with a Lisp symbol. This can be used to ease the task of searching through lists. Property lists can be used as follows:
Before the color of dog has been set, GET returns NIL.

```
(GET 'dog 'color)
NIL
```

After the color has been set, using SETF,

```
(SETF (GET 'dog 'color) 'brown)
BROWN
```

GET returns the proper value, BROWN.

```
(GET 'dog 'color)
BROWN
```

The DEFSTRUCT Macro

Another facility is the DEFSTRUCT macro. DEFSTRUCT allows one to create named record structures with named components. This can be used to create new data types, and every structure of the new type has components with specified names. Lisp automatically creates functions to access the components of the new data structure when it is created. An example of setting up a new data structure is shown here:

```
(DEFSTRUCT dog
   color
   name
   owner)
DOG
```

A dog can then be created by:

```
(SETQ dog1 (MAKE-DOG color 'brown
                     name  'fred
                     owner 'Cheryl))
DOG1
```

and information about a dog can be retrieved and set using:

```
(COLOR dog1)
```

```
BROWN
(SETF (COLOR dog1) 'black)
BLACK
(COLOR dog1)
BLACK
```

Table 15: This tutorial was given to all students to give them a standardized way of learning the material.

Emacs Quick Reference

Single letters in **boldface** type have mnemonic meanings.

C- means to hold down Control while hitting the following key.

M- means to hold down Meta while hitting the following key.

Moving Cursor

Move Over	Backward	Forward
character	C-b	C-f
word	M-b	M-f
line (previous/next)	C-p	C-n
line (begin/end)	C-a	C-e

Deleting Text

Delete	Previous	Next
character	RUBOUT	C-d
word	M-RUBOUT	M-d
line (to end of)	M-C-k	C-k

Regions

Action	Command
Delete Region	C-w
Restore (Yank) last thing deleted	C-y
Restore (Yank) last restore with previous delete	M-y
Set mark here	C-Space

A *region* is defined as starting at the last point the *mark* was set, and ending at the current cursor location.

Table 16: An Emacs reference page given to each participant in the study.

Introduction To Study

The user interface you will be using is part of a research project on help systems.
You will be using Symbolics computers:

- The Symbolics keyboard uses rub out (on the left) for backspacing.
- The control and meta keys are next to the space bar.
- Return is the end of line indicator.

You will be typing Lisp expressions into a "lisp listener".
Output from your program will be shown in the output window to the right of the "lisp listener" window.
You can get help on various things in Lisp by pressing a mouse button. When you press this button you might find sub-menus which you want to press.
Careful attention should be paid to the computer screen, as important help often appears there.
There is online "Editor Help" if you are having trouble with the "emacs-like" editor interface, and there is also an included quick reference quide.
You will not receive any help from the people running this course, except if there is a problem with your computer. This is a study in Computer Aided Instruction, and another person's help would taint the results. You are in no way expected to finish all of the problems that you are given; in fact, we have intentionally given you too much work, so that you will not finish. We will be measuring your learning by examining how many problems you answer, and the correctness and techniques used in obtaining your answers.
DO NOT FEEL INTIMIDATED! We are in no way measuring your intelligence, or learning ability. We may not have given you all of the necessary information for all of the problems, as you are expected to figure them out by trial and error, and by using the computer as a tool. If you do get stuck, try to understand what the computer is telling you, or ask for more help, or, as a last resort, consult the brief tutorial on Lisp.
DON'T GIVE UP. OUR EXPERIENCE HAS SHOWN THAT YOU WILL LEARN MUCH MORE THAN YOU WOULD IN A NORMAL CLASS ROOM SETTING! HAVE FUN.
Before you begin using `COACH`, make sure you can answer the following question correctly: When you ask `COACH` for help on the `LISP` function `SETQ`, what is the result of the first example? You can find this information by following these steps:

1. Click the left mouse button once, and look at the menu that pops up.
2. Move the mouse to the `Available-Functions` item, and click the button again.
3. When the next sub-menu pops up, find and click on the `SETQ` item.
4. What appears now is help on the `SETQ` function. The first example returns `ENGINE`.

When you can find that answer, you should be able to find your way around `COACH` well enough to use it.

B.2 Results Data

Student Data

Pretest

Little differences in LISP knoweldge was found between students before the course.

Exercises

From user model:

Student	# of functions defined
Manual group	
AT	0+
DZ	1
SS	0
RG	1
WJ	0
RV	0+
BR	2
W	0
AUM group	
LC	1
JD	3
JC	0
EG	1
BP	1
JP	6
UB	5+
JM	2
SC	6
LU	0
RB	3

Table 17: An average of .5 functions were defined by students in the manual non adaptive, control group, and 2.5 by students of the automated adaptive group for data base project.

Comment sheets

Question	1				2				3				4			
Day	T	W	Th	F	T	W	Th	F	T	W	Th	F	T	W	Th	F
AL	2	4	4	-	2	3	5	-	N	4	4	-	N	2	4	-
DZ	N	5	4	5	1	2	2	1	N	2	3	N	N	N	1	0
SS	3	3	-	2	2	2	-	2	3	5	-	4	N	3	-	3
RG	1	0	0	0	2	1	0	0	N	N	N	2	N	2	2	1
WJ	5	-	4	4	3	-	4	3	5	-	5	5	2	-	N	2
RV	5	5	5	4	5	5	4	4	5	5	4	4	5	4	N	3
BR	2	2	N	2	1	N	N	N	1	N	0	0	N	1	N	N
W	2	N	N	-	2	N	2	-	2	2	0	-	1	2	1	-

Question	5				6				7				8			
Day	T	W	Th	F	T	W	Th	F	T	W	Th	F	T	W	Th	F
AL	4	5	5	-	m	c	f	-	3	8	d	-	-	-	9	hg
DZ	1	N	1	N	m	m	v	o	0	0	d	d	-	-	8	8
SS	N	N	-	1	l	e	-	f	3	8	-	d	-	-	-	8
RG	1	N	N	N	l	N	1	1	3	N	d	h	-	-	4	2
— WJ	1	-	N	2	N	-	p	N	N	-	d	d	-	-	10	10
RV	3	3	2	N	l	xx	x	m	6	d	d	d	-	-	8	8
BR	3	2	N	2	l	l	f	N	N	N	d	f	-	-	3	4
W	3	N	N	-	p	l	v	-	5	d	d	-	-	-	8	-

Table 18: Data From Nonadaptive Users Comment Sheets. See legend keys below.

Question	1				2				3				4			
Day	T	W	Th	F	T	W	Th	F	T	W	Th	F	T	W	Th	F
LC	4	5	N	N	3	N	N	N	N	N	N	N	N	N	N	N
JD	N	3	N	-	N	3	N	-	N	1	N	-	2	4	N	-
JC	4	5	-	-	4	3	-	-	5	5	-	-	5	4	-	-
EG	-	5	4	-	-	3	N	-	-	4	5	-	-	4	2	-
BP	4	N	N	-	5	N	3	-	N	2	N	-	N	2	N	-
JP	3	2	-	-	2	2	-	-	N	N	-	-	1	2	-	-
UB	N	-	3	N	4	-	4	3	5	-	N	4	4	-	4	n
JM	-	4	5	N	-	N	2	3	-	2	2	4	-	N	N	2
SC	3	5	5	4	3	4	2	1	4	4	N	2	N	3	2	2
LU	3	4	4	4	5	5	3	N	4	4	5	4	4	3	2	3
RB	4	-	4	-	3	-	4	-	3	-	N	-	3	-	5	-

Question	5				6				7				8			
Day	T	W	Th	F	T	W	Th	F	T	W	Th	F	T	W	Th	F
LC	N	N	N	N	N	N	N	N	N	N	N	N	-	-	6	0
JD	N	3	N	-	h	N	N	-	N	3	d	-	-	-	10	0
JC	5	5	-	-	m	mx	-	-	2	N	-	-	-	-	-	-
EG	-	2	4	-	-	e	p	-	-	2	d	-	-	-	-	7
BP	N	N	N	-	l	q	N	-	3	5	6	-	-	-	-	6
JP	N	N	-	-	m	p	-	-	3	d	-	-	-	-	-	7
UB	4	-	5	5	N	-	N	m	N	-	N	d	-	-	7	6
JM	-	2	N	1	-	m	l	e	-	5	d	d	-	-	-	-
SC	N	2	4	2	l	l	h	h	3	5	d	d	-	-	8	7
LU	1	4	3	2	N	m	m	v	7	d	d	v	-	-	-	6
RB	5	-	N	-	N	-	N	-	3	-	d	-	-	-	-	8

Table 19: Data From Automatic Adaptive Users Comment Sheets. See legend keys bellow.

N no answer

- no comment sheet

0	can't say
1	never bad
2	not often not terrible
3	sometimes ok
4	often good
5	all the time great

Table 20: Key for questions 1,2,3, 1 - 5

1.	hate
3.	interesting
4.	like
5	love
p	paper user

Table 21: Key for questions 4,5 1 -5

1	unhappy
l	lists
m	machine problem
C	cons
D	Defun
F	Defstruct
v	variable (data base)
e	evaluation
q	quote
h	help system
x	editor
p	parenthesis

Table 22: Key for question 6.

AUM Group, Types of problems:
Instances: 4-l 7-m 0-l 0-f 2-p 1-v 1-x 2-e 0-c 3-h 1-q

=21 non machine: 14 lisp related: 10

People: 3-l 5-m 0-l. 0-f 2-p 1-v 1-x2-e 0-c 2-h 1-q

=17 non machine: 12 lisp related: 9

Nonadaptive control group, Types of problems:
Instances: 7-l 4-m 2-l 2-f 2-p 3-v 3-x 1-e 1-c 0-h 0-q

=26 non machine 22 lisp related 19

People: 6-l 3-m 1-ll 2-f 2-p 2-v 1-x1-e 1-c 0-h 0-q

=19 non machine 16 lisp related 15

Interviews

Non–Learners

AT:
I used the on-screen menus more for help.
DZ:
Q: When you're trying something do you find the info on the screen to be helpful?
A:No.
SS:
Q: Do you find the stuff on the screen to be helpful?
A:I use available functions. I'm confused on making a data base and a function.
RG:
Q: Do you get help from screen or from tutorial?
A: At this point I get help from trial and error the help screens are wrong and the tutorial is very abstract and it doesn't say too much.
J:
Q: What's your biggest question?
A: What is the role of the parenthesis?
Q: What is the simplest thing you could do with cons?
A: 1,2,3 adding things in front of a list.
? Do you find yourself relying on the tutorial or menu?
A: I go up to information a lot but I can't call up several things at once. How do you access lists?
B:
Q: Do you look for info in the tutorial or menu or guessing?
A: I look at menu, it's easier. I wish they would have an improvement for help on my part, comment - on the language itself it's very precise, one that's all.
BR:
A: I could really use more information off the property list but there is nothing that would help.
Q: Did you think of just using lists for your data base?
A: Right now I'm trying to learn what the various options are.
Q: When you're learning the options are you finding the help on the screen or tutorial?
A: A lot of time I go to the help available functions, list concept properties.
Q: what's the most interesting thing you've learned?
A: That everything's a function, there are no statements. Feeling though like you're withholding useful information.
Q: What's the most frustrating thing?
A: Parenthesis very unforgiving also.
SS:

Q: DO you look at the help above you while you're typing?
A: Yeah,I use the available functions. I'm liking the syntax help less and less because it has less info. The tutorial is more helpful than the menu

Learners

LC:
I'm trying to develop a program.
Q: Have you figured out what functions to use from tutorials or from menus?
A: Look at syntax first before I type and menu.
JD:
I'm working on the problem of data structures.
Q: Where do you get information?
A: Tutorial first then available functions.
Q: Do you pay attention to the help that comes up?
A: Yeah, I look at it.
Q: Do you look up things before you type them to get them right the first time or try them first?
A: I don't care if I get it right the first time, sometimes I try it first but I like to look at a sample.
Q: What is the most interesting thing you've noticed about Lisp?
A:You feel like it's unstructured but I can't figure out the language.
Q: What's the most frustrating thing?
A:Not having an overview ahead of time and I don't have a book with a list of available functions so I can build on the samples.
JC:
Q: Where do you get info from?
A: I look at help screen more than tutorial.
? Do you feel like you're learning anything?
A: a little?
EG:
Q: What are you working on right now?
A: Constructing a data base.
Q: So you have successfully made a function?
A: No, don't think so.
Q: You made a data structure of some sort?
A: I made a data structure, person, and I was going to see what it looks like when I have nothing in it and then put a couple of things into it and see what it prints out then I have some idea how to do the function and to extract the function from it.
Q: So you were using DEFSTRUCT in other words?
A: Right.
Q: Can you think of another way you could have done that?
A: Yeah, I could've done it in the same way as DEFSTRUCT which just builds a list in a fixed way and then use cdr and car and cons to extract things from

the list and I could've written a function that uses those same pieces to attack the list or attach things to it, DEFSTRUCT seems to do all those things for you.
Q: Do you use the tutorial, menu or help window for info?
A: Stuff on the window that comes up when you're half way through a function is more helpful than the help you look for on the function because the idiosyncrasy of the help is that it changes as it thinks you're doing better and the window doesn't change, but they're not at the same level. the balance between this and the written examples is very helpful.
Q: Do you find yourself exploring by asking for help from the menus?
A: Oh yeah,a lot of time poking around but I only looked at about a dozen of the total menus.
BP:
Q: This isn't doing anything for you?
A: Well it's not exciting me if that's what you're asking.
Q: (from user)a function is effectively an atom by the time you've written down a pair of parenthesis?
A:The function's name is an atom(the entire function is a list)
Q: Do you get info from menus, tutorial or help windows?
A: I use tutorial to see syntax and I tend not to look up when I'm typing so most of the messages go by without me looking at them. The menu is more useful.
GP:
Q: What are you working on?
A: I'm working on a data base
Q: Do you use the help from the screen or tutorial ?
A: First I look under the available functions but that's usually not very helpful then I look at the tutorial. I usually wait until I get an error to look at the syntax above.
UB: Q: Do you look at the help above you.
A: Sometimes, not really.
Q: Do you ask questions from the menu?
A:Yes,I use it mostly I get help from the line below

C COACH Definitions

C.1 COACH Lisp Definition

The following code and tables defines Lisp syntax for the purpose of creating the COACH AUM and identifying appropriate situations where help is needed.

```
(defun IVtokens () ;(send *coach-parser* :set-tokens
  '(FS SpecName SpecFunc FUNC L LISTFUNC Q Quotable QList
    F A N NUMFUNC NUMBER ST S X [ ]  &OPTIONAL &KEY
   &ALLOW-OTHER-KEYS &REST &AUX))

;;=============================================================================;
;                                  Machine syntax
;key symbols:
; A - Atom, S - Defined symbol, N - number, L - List, F - Form, X - anything
; Q -  check only parenthesis level            FS - Function Spec
;
;syntax delimiters: { -      open a clause,    } -       close a clause,
;              [ - '('  open a clause,    ] -  ')' close a clause,
;
;vary syntax: * - next syntax part can occur 0 or more times,
;             ? - next syntax part can occur 0 or 1    time
;             V - At least one part of the next clause must occur at least once
;;;              @name is an internal syntax
;;;  Logicals:   Conjunction in a template is indicated by juxtaposition
;;;              Disjunction is indicated by V {} syntax
;
;;=============================================================================;;
(defun  IVfunction-templates   ()

" initializes the syntax of user target language functions which coach
can parse. The language of the functions is described in the above comment"
;;
;; side effects for a template can be attached by making coach-parser methods:
;; :templatename-token-actions,
;; :templatename-function-actions
;; :Init-templatename-Effects
 (LIST
;(send *coach-parser* :set-function-templates (list

 '(ABS    N ] )
 '(CAR    L ] )                '(CDR   L ] )
 '(COND  * [ * X ] ] )         '(CONS  X X ] )
 '(CASE   X * [ A * X ] ] )
```

```
 '(CATCH  A * X ] )

  ;; Destructuring is a list of quotables.
   ;;&Body and dotted lambda syntax not checked.
  '(DEFMACRO A [ ? { &Whole A A }
        ? { * A * [ * Q ] }
             ? { &Optional * { ? A ? [ A ? X ] ? [ A X A ] }}
             ? { &Rest A }
 ? { &Key * { ? A ? [ A ? X ] ? [ A X A ]
               ? [ [ A A ] ? X ] ? [ [ A A ] X A ]
         ? &Allow-Other-Keys } }
               ? { &Aux * { ? A ? [ A ? X ] } }
              ] * X ] )

   ;; no checking for Option conflicts.
'(DEFSTRUCT V { ? A
           ? [ A * { ? [ :Conc-Name A ] ? [ :Conc-Name [ ] ]
              ? [ :Conc-Name ]
      ? [ :Constructor A ? QList ] ? [ :Constructor [ ] ]
      ? [ :Constructor ]
      ? [ :Copier A ] ? [ :Copier [ ] ] ? [ :Copier ]
      ? [ :Predicate A ] ? [ :Predicate [ ] ]
      ? [ :Predicate ]
      ? [ :Include A ]
      ? [ :Print-Function FS ]
      ? [ :Type Vector ] ? [ :Type [ Vector A ] ]
      ? [ :Type Vector LIST ]
      ? [ :Named ]
      ? [ :Initial-Offset NUMBER ] } ] }

  * { * A * [ A ? Q ] * [ A Q ? { :Type A } ? { :Read-Only A } ]
                      * [ A Q ? { :Read-Only A } ? { :Type A } ]
     * [ A Q ? { :Type A } ? { :Read-Only [ ] } ]
                      * [ A Q ? { :Type A } ? :Read-Only ] }
] )

  '(DEFUN      A [ * A
               ? { &Optional * { ? A ? [ A ? X ] ? [ A X A ] } }
               ? { &Rest A }
 ? { &Key * { ? A ? [ A ? X ] ? [ A X A ]
               ? [ [ A A ] ? X ] ? [ [ A A ] X A ]
         ? &Allow-Other-Keys } }
               ? { &Aux * { ? A ? [ A ? X ] } }
               ] * X ] )
```

```
 '(DEFVAR     A X ] )

 '(DO         [ * [ A ? X ? X ] ] [ X * X ] * X  ] )
 '(DO*        [ * [ A ? X ? X ] ] [ X * X ] * X  ] )

 '(FUNCTION  FS ] )
 '(FUNCTIONM FS   ) ;invoked by #'

 '(LAMBDA     [ * A
                ? { &Optional * { ? A ? [ A ? X ] ? [ A X A ] }}
                ? { &Rest A }
? { &Key * { ? A ? [ A ? X ] ? [ A X A ]
               ? [ [ A A ] ? X ] ? [ [ A A ] X A ]
        ? &Allow-Other-Keys } }
                ? { &Aux * { ? A ? [ A ? X ] } }
              ] * X ] )

'(LET        [ * { * A * [ A ? X ] } ] * X ] )
'(LET*       [ * { * A * [ A ? X ] } ] * X ] )

'(LIST       * X ] )        '(QUOTE   * Q ] )     '(QUOTEM Q )
'(MAPC       X * L ] )      '(MAPCAR  X * L ] )

'(MAPLIST    X L ] )        '(MAPL    X L ] )

 '(MEMBER    X L ] )         '(RETURN * X ])
 '(PRINT     X ? X     ])   '(SET    X X ] )
 '(SETF      * { X  X } ] ) '(SETQ   * { A X } ] )
 '(times     * N  ] )       '(*      * N ] )
 '(PLUS      * N ] )        '(+      * N ] )
 '(quotient  N * N ] )      '(//     N * N ] )
 '(DIFFERENCE N * N ] )     '(-      N * N ] )
 '(equal     N N ] )        '(=      N N ] )
 '(greaterp  N * N ] )))
```

Character Parse Table

The following state table controls the character level parse for for the Lisp COACH.

```
(defun IVstate-array ()
  ; (SEND *Coach-Parser* :Set-State-Array
  (CL:make-array 108 :Initial-Contents

;Form+ delim- Symbol Quote string comment  number  help (state,char)
;                  Old State     (will be Last-State)
; :pls  :min   :sym  :qte   :st   :cmt  :num  :Shp  :hlp
                                                        ;"Type" of CHR
'(:hlp  :hlp   :hlp  :hlp   :hlp  :hlp  :hlp  :hlp  :hlp   ;er  unused char
  :hlp  :hlp   :sym  :sym   :st   :cmt  :hlp  :hlp  :hlp   ;sp  macro char
  :num  :num   :sym  :Sym   :st   :cmt  :num  :hlp  :min   ;num number
  :sym  :sym   :sym  :Sym   :st   :cmt  :Sym  :hlp  :min   ;chr character
  :Pls  :pls   :pls  :Pls   :st   :cmt  :pls  :hlp  :hlp   ;opn open paren
  :min  :min   :min  :hlp   :st   :cmt  :min  :hlp  :min   ;cl  close paren
  :Hlp  :qte   :qte  :qte   :st   :cmt  :qte  :Shp  :hlp   ;qte '
  :st   :st    :st   :st    :min  :cmt  :st   :hlp  :hlp   ;st  "
  :cmt  :cmt   :cmt  :cmt   :st   :cmt  :cmt  :hlp  :hlp   ;cmt ;
  :pls  :min   :min  :hlp   :st   :min  :min  :hlp  :hlp   ;ecm "end comment"
  :pls  :min   :min  :MIN   :st   :cmt  :min  :hlp  :hlp   ;blnk blank
  :Shp  :Shp   :sym  :sym   :st   :cmt  :hlp  :hlp  :hlp)));Shp  #
```

Required Knowledge

The following sets of lists defines the required knowledge slots for Lisp functions in the Lisp COACH.

```
;;;=============================================================================;;;
;;;    Required knowledge
;;;=============================================================================;;;

;;;=====                                                                  =====;;;
(DeFun Initialize-Required-Knowledge ()
  (DoList
    (tuple
      '((Atom Quotation Eval )
(CASE S-expression Predicates )
(COND S-expression Predicates )
(CONS S-expression Atom )
(CATCH S-expression Predicates )
(DefMacro Lambda-List S-expression Atom EVAL)
(DEFSTRUCT Lambda-List SETQ LET)
(DEFUN Lambda-List S-expression Atom)
(DO S-expression Atom)
(LAMBDA Lambda-List S-expression Atom)
        (Lambda-List Atom Eval)
(LET S-expression Atom)
(LIST CONS)
(MAP CONS LIST)
(MAPC MAP MAPCAR)
(MAPCAR MAP CONS LIST)
(MAPCON MAP MAPCAR)
(MAPL MAP MAPCAR)
(MEMBER LIST)
(NIL T EVAL )
(SETF DEFSTRUCT LIST SETQ CAR CDR MEMBER)
(SETQ S-expression Atom)
(T EVAL       Atom)
(+ Atom      Lists)
(- Atom      Lists)
(= predicates numbers)
(< numbers predicates)))
    (APPLY 'Add-Required-Knowledge tuple)))

;;;=============================================================================;;;
```

Sample Function Actions

The following represents a sample of the kind of code that can be written to control a Lisp action in the Lisp COACH.

```
;;========                                                          ========;;
;; Function actions are the side effects for COACHED functions
;; Side Effect Methods must be named according to the following format so
;; that they can be found by (:Invoke-Action Coach-Parser):
;;   Function Actions MUST BE NAMED :<function-name>-Function-Action
;;     Unless they are in a package not used by AUM, in which case they
;;     must be named :<Package-Long-Name>-<function-name>-Function-Action
;;   Token Actions    MUST BE NAMED :<Function-name>-Token-Action
;;     Unless they are in a package not used by AUM, in which case they
;;     must be named :<Package-Long-Name>-<function-name>-Token-Action
;;
;; Note:  if the package name must be included, it is likely that it will
;;   have a different name in each implementation, so be prepared to
;;   use #+ and #- syntax for the method names.
;;;=============================================================================;;;

(DefMethod (:SetQ-Function-Action Coach-Parser) ()

  (MapCar
    #'(Lambda (sym) (SEND Self :Add-Symbol sym))
    (SEND Self :Pop-Closure)))

;;========                                                          ========;;

(DefMethod (:SetQ-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)

       (0 (SEND Self :Push-Closure))
       (3 (SEND Self :Add-Symbol))))

;;=============================================================================;;

(DefMethod (:Let-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)

       (0   (PUSH NIL Symbol-Buffer))

       ((5 8)
            (PUSH  (String-UpCase Token?) (FIRST Symbol-Buffer)))
```

```
        (13  (SEND Self :Push-Closure)

             (MapCar
               #'(Lambda (sym) (SEND *Coach-Parser* :Add-Symbol sym))
               (FIRST Symbol-Buffer))

             (POP Symbol-Buffer))))

;;=============================================================================;;

(DefMethod (:Let*-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)

        (0   NIL)

        (1   (SEND Self :Push-Closure))

        ((5 8)  (SEND Self :Add-Symbol))))

;;=============================================================================;;

(DefMethod (:Do*-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)

        (0   NIL)

        (1   (SEND Self :Push-Closure))

        (4   (SEND Self :Add-Symbol))))

;;=============================================================================;;

;side effects for do

(DefMethod (:Do-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)

        (0   (PUSH NIL Symbol-Buffer))

        (4   (SEND Self :Push-Closure)
             (SEND Self :Add-Symbol)
```

```
            (PUSH (String-UpCase Token?) (FIRST Symbol-Buffer)))

       (9   (SEND Self :Pop-Closure))

       (10  (SEND Self :Push-Closure)

            (MapCar
              #'(Lambda (sym) (SEND Self :Add-Symbol sym))
              (FIRST Symbol-Buffer))

            (POP Symbol-Buffer))))

;;============================================================================;;

(DefMethod (:Lambda-Token-Action Coach-Parser) ()

  (CL:CASE (SEND Self :Func-Pos)
    (0   NIL)
    (1   (SEND Self :Push-Closure))
    ((3 10 13 19 21 28 36 39 45 47 53 62 65 77 80)
     (SEND Self :Add-Symbol))))
```

C.2 COACH Statement Strategy Rules

The following Rules can be used to control the Lisp COACH.

```
; Active values on Update Form
; goodess, latency, slope, times, best, worst, history
; Modify Function-Print-Help, Present-Form, Update-Form
; Write Resolve

;HELP TEXT:
;syntax, arglist, description, example, user-example, super-concept
;
;example: Shows sample input and sample output.
;  Examples of what to do and what to expect.  A
;  floundering user can imitate the example and use form correctly.

;user-example:  Shows the most recent successful use of form.
;  A user-example may remind a user of the correct approach to using form,
;  provides an example of what to do, and is a record of what the
;  user was working on the last time form was properly used.
;
;syntax: Form, arguments, and arguement definitions.  Defines in English
;  how to use form.
;  Syntax provides a template which, if filled correctly ensures correct
;  usage of form.  Mnemonic names & type requirements for template positions
;  assist in selecting proper arguements.

;arglist:  Formal definition of function syntax.
;  Arglist provides a template defining syntaxes accepted by form. Since
;  arglist is read from the form's definition, syntax  definitions
;  are formal and may be difficult to understand.

;description:  Description of form's usage.
;  Description tells what form is good for.

;super-concept:  Description of form's role in Lisp programming.
;  Helps a user get oriented on when and how to use form.

;Also useful:
;  Basis Set: The functional family into which form, as a tool, falls.

;Required Concepts: Concepts which must be mastered for mastery of form.
;

;USER FACTORS:
;  Times, latency, slope, net successes (goodness), best, worst
```

```
;  Above for conceptual siblings and basis set.
;
;Times: Number of attempts which have been made at using form.
;   Low times indicates that the user may be unfamiliar with the syntax.
;   High times indicates that this form is important to the user, so user
;   may be willing to read more text to get it right, may be interested
;   in pointers to related functions. A high times and a low goodness may
;   indicate abiding problems with required concepts.
;Latency:The time since the most recent attempt at using form
;   BOUNDARY CONDITION:  The first use has an undefined Latency.
;   A high latency indicates that user hasn't seen help onform for a
;   long time.  Unresolved problems will tend to manifest.
;   keep Coach offset time (+ universal time) = effective time
;
;   note: Text has a time dependent history. It's irrelevant how
;      often text has  been seen if it hasn't been seen for a long time.
;Slope:The ratio of correct usages to attempts over the last n trials.
;  BOUNDARY CONDITIONS:  On the first use slope is undefined.
;  Before n trials have been completed slope has insufficient data and
;  shouldn't be relied upon.
;  Slope shows how well the user is currently approaching the problem of
;  using form.  Slope gives an indication of how much a success or failure
;  should influence the estimate of goodness.  A good slope shows that the
;  user has an effective approach, and more advanced information
;  on the ways in which form can be used should be given.  Bad slope shows
;  the user has been unable to arrive at a good approach to using form,
;  that information regarding the "why's and wherefore's" and practical
;  examples of correct usage should be provided.
;Goodness:Total of successful attempts minus failures, modified by the
;  effects of slope Goodness estimates current proficiency.
;
;Best:  The highest Goodness achieved.
;   If high, indicates that at some point the user had the right idea.
;Worst:  The lowest Goodness achieved.
;  If high, the user has trouble with at least one of the underlying
;  concepts.



;;============================== Update Form ==========================;;
;; Slope and Latency maintainers come before Goodness which comes before;;
;; Best and Worst Maintainers which come before bounds maintainers      ;;
;;=====================================================================;;

(Define-Rule (:Note-Success Update-Form) (form-used)
```

```
  :IF (SEND *Coach-Parser* :Good-Token?)
  :THEN (SEND form-used :Insert 1))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Note-Failure Update-Form) (form-used)
  :IF (NOT (SEND *Coach-Parser* :Good-Token?))
  :THEN (SEND form-used :Insert 0))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Good-but-Neglected Update-Form) (form-used)
  :IF(AND(SEND form-used :High :Latency)(SEND *Coach-Parser* :Good-Token?))
  :THEN (SEND form-used :Set-Goodness (+ 1 (SEND form-used :Goodness))))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Bad-but-Neglected Update-Form) (form-used)
  :IF (AND (SEND form-used :High :Latency)
   (NOT (SEND *Coach-Parser* :Good-Token?)))
  :THEN (SEND form-used :Set-Goodness (- (SEND form-used :Goodness) 1)))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Was-Bad-but-Getting-Better Update-Form) (form-used)
  :IF (AND (NOT (SEND form-used :High :Latency))
   (SEND *Coach-Parser* :Good-Token?)
   (< (SEND form-used :Goodness) (SEND form-used :Best)))
  :THEN
    ;; Makes an S-Curve from current Goodness to Best, increasing by at
    ;; least 1.
    (LET ((goodness (SEND form-used :Goodness)))
      (SEND form-used :Set-Goodness
    ;; Truncate to Lower instability and reduce memory overhead
    (+ goodness (MAX 1 (TRUNCATE
 (* (SEND form-used :Slope)
    (- (SEND form-used :Best) goodness)))))))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Was-Good-but-Getting-Worse Update-Form) (form-used)
  :IF (AND (NOT (SEND form-used :High :Latency))
   (NOT (SEND *Coach-Parser* :Good-Token?))
   (> (SEND form-used :Goodness) (SEND form-used :Worst)))
  :THEN
    ;; Makes an S-Curve from current Goodness to Worst, decreasing
    ;; by at least 1.
    (LET ((goodness (SEND form-used :Goodness)))
      (SEND form-used :Set-Goodness
    ;; Truncate to Lower instability and reduce memory overhead
```

```
    (+ goodness
       (MIN -1 (TRUNCATE
 (* (SEND form-used :Slope)
    (- goodness (SEND form-used :Worst)))))))))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Best-and-Getting-Better Update-Form) (form-used)
  :IF (AND (NOT (SEND form-used :High :Latency))
   (SEND *Coach-Parser* :Good-Token?)
   (>= (SEND form-used :Goodness) (SEND form-used :Best)))
  :THEN
    ;; Increase Goodness proportionate with Slope, always increasing
    ;; by at least 1.
    (LET ((goodness (SEND form-used :Goodness)))
      (SEND form-used :Set-Goodness
    ;; Truncate to Lower instability and reduce memory overhead
    (+ goodness (MAX 1 (TRUNCATE (* (SEND form-used :Slope)
    (SEND form-used :Best)))))))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Worst-and-Getting-Worse Update-Form) (form-used)
  :IF (AND (NOT (SEND form-used :High :Latency))
   (NOT (SEND *Coach-Parser* :Good-Token?))
   (<= (SEND form-used :Goodness) (SEND form-used :Worst)))
  :THEN
    ;; Decrease Goodness proportionate with Slope, always decreasing
    ;; by at least 1.
    (LET ((goodness (SEND form-used :Goodness)))
      (SEND form-used :Set-Goodness
    ;; Truncate to Lower instability and reduce memory overhead
    (- goodness (MAX 1 (TRUNCATE (* (SEND form-used :Slope)
    (SEND form-used :Worst)))))))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Maintain-Best Update-Form) (form-used)
  :IF (> (SEND form-used :Goodness) (SEND form-used :Best))
  :THEN (SEND form-used :Set-Best (SEND form-used :Goodness)))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Maintain-Worst Update-Form) (form-used)
  :IF (< (SEND form-used :Goodness) (SEND form-used :Worst))
  :THEN (SEND form-used :Set-Worst (SEND form-used :Goodness)))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Bound-Goodness-and-Best Update-Form) (form-used)
```

```
  :IF (> (SEND form-used :Goodness) (SEND *User* :Max-Goodness))
  :THEN
    (LET ((limit (SEND *User* :Max-Goodness)))
      (SEND form-used :Set-Goodness limit)
      (SEND form-used :Set-Best limit)))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Bound-Goodness-and-Worst Update-Form) (form-used)
  :IF (< (SEND form-used :Goodness) (SEND *User* :Min-Goodness))
  :THEN
    (LET ((limit (SEND *User* :Min-Goodness)))
      (SEND form-used :Set-Goodness limit)
      (SEND form-used :Set-Worst limit)))

;;;========================== Present Form ==========================;;;

(Define-Rule (:Veto-Arglist Present-Form) (form-vetos)
  :IF (NOT (SEND 'Lambda-List :Good :Best))
  :THEN (PUSH :ArgList form-vetos))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Veto-Syntax Present-Form) (form-vetos)
  :IF (NOT (SEND 'Lisp :Good :Best))
  :THEN (PUSH :Syntax form-vetos))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Veto-User-Example Present-Form) (form-used form-vetos)
  :IF (SEND form-used :High :Latency)
  :THEN (PUSH :User-Example form-vetos))

;;;-------------------------------------------------------------------;;;
(Define-Rule(:Encourage-Exploration Present-Form)(form-used form-proposals)
  :IF (AND (SEND form-used :Good :Slope)
   (SEND form-used :Good :Goodness)
   (SEND form-used :High :Times))
  :THEN (PUSH (Get-Form-Advice form 'other-related-knowledge))
        (PUSH (Get-Form-Advice form 'required-knowledge))
        (PUSH :Basis form-proposals))

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Out-of-Practice Present-Form) (form-used form-proposals)
  :IF (OR (NOT (SEND form-used :High :Times))
  (SEND form-used :High :Latency))
  :THEN (PUSH :Super-Concept form-proposals)
        (PUSH :Description form-proposals))
```

```
;;;-------------------------------------------------------------------;;;
(Define-Rule (:OK Present-Form) (form-used form-proposals)
  :IF (AND (SEND form-used :Good :Slope) (SEND form-used :Good :Goodness))
  :THEN (PUSH :Syntax form-proposals)
        (PUSH :ArgList form-proposals)

;;;-------------------------------------------------------------------;;;
(Define-Rule (:Losing-Ground Present-Form) (form-used form-proposals)
  :IF (NOT (SEND form-used :Good :Slope))
  :THEN (PUSH :User-Example form-proposals)
        (PUSH :Example form-proposals))

;;;===================================================================;;;
; Rule to get concepts behind required knowledge to show disoriented user ;
; It is commented out because it has not been useful so far               ;

;(Define-Rule Disoriented-User (Present-Bad-Form)
; :IF
;  (AND (<  (- (Get-Numeric-Property (SEND *User* :Form-used) 'Times)
;              (Get-Numeric-Property (SEND *User* :Form-used) 'Latency))
;           8)
;       (NOT (SEND form-used :Good :Slope))
;  :THEN
;  (LET* ((form (SEND *User* :Form-used))
;  (advice (Get-Form-Advice form 'required-knowledge)))
;    (WHEN advice
;      (PUSH Essential Concepts: (SEND *User* :Form-Help-Board))
;      (PUSH
; (MapCan #'(LAMBDA (concept)  (LIST (FORMAT NIL "~A:~&~A~&" concept
;   (GET concept 'concept-level1))))
; advice)
; (SEND *User* :Form-Help-Board)))))
```

C.3 Sample Lisp Help Text

The following example shows the Lisp COACH help text format.

```
(DO Description-Level1
| DO loops through a set of expressions until it is told to stop.

;;===                                                                    ===;;
(DO Syntax-Level1
|  The general form of the DO function:
      (DO  ()
           (exit-condition termination-expression1... )
            body-expression1...  )
 The exit condition is the only required part of the DO function.
 () must be in the first position.|)
;;===                                                                    ===;;

(DO Example-Level1
| > (SETQ X 1) ------------------------------------------->  1
  > (DO ()
         ( ( EQUAL X 10) )
         ( SETQ X (PLUS X 1) ) )--------------------------->  NIL  |)

;;===                                                                    ===;;

( DO Description-Level2
| DO is a very flexible loop function allowing internal variables to be
initialized the same  as LET. It can return an evaluated expression.|)
;;===                                                                    ===;;
( DO Syntax-Level2
| (DO   (   (variable1   initial_value1   update_expression1)
           (variable2   initial_value2   update_expression2)  ...)
           (exit_condition   exit_action1   exit_action2   ...)
            body_expression1 body_expression2 ...)|)
;;===                                                                    ===;;
( DO Example-Level2
|> (SETQ train '(engine boxcar caboose)) ----------> (ENGINE BOXCAR CABOOSE)
 > (DO (reversed)
             ((null train) (print reversed))
       (SETQ reversed (CONS (CAR train) reversed))
       (SETQ train (CDR train ) ) ) --------------> (CABOOSE BOXCAR ENGINE)|)
;;===                                                                    ===;;
( DO Description-Level3
| DO allows internal variables to be reset by an expression each time
  through the loop. The return function exits the DO wherever it is placed.|)
;;===                                                                    ===;;
```

```
( DO Syntax-Level3
| (DO ( (variable initialvalue increment value) )
         (end-test expression-to-evaluate-at-end)
          body-expression1 body-expression2 ...)    |)
;;===                                                                    ===;;
( DO Example-Level3

| > (DO ((i 1 (plus i 1)))              ;change internal variable each iteration
       (NIL)                            ;If no end test, NIL IS REQUIRED
       (COND ( (greaterp i 5) (RETURN 'end) ) )----------------> END |)
```

Do Syntax level 4 is the PIACH definition of DO:

```
(DO          [ * [ A ? X ? X ] ] [ X * X ] * X  ] )
```

C.4 Lisp Token Help Text

The following token help text is used in the Lisp COACH and shows the form it is written in.

```
;(token-facts-Init)

(DEFUN Token-facts-Init ()
 (MAPCAR #'(LAMBDA (FACT)
  (Send *USER* :put-Attribute (first fact)
                        (STRING (Third fact)) 'token (Second fact)))
'(
(N Description-Level1  |NUMBER     Any combination of the numbers 0 - 9|)
(N Syntax-Level1       NIL)
(N Example-Level1      |             123|)
;;===                                                              ===;;
(N Description-Level2 |NUMBER      Any combination of the numbers 0 - 9 |)
(N Syntax-Level2       |              Can have a decimal point or a sign|)
(N Example-Level2      |             -123.5|)
;;===                                                              ===;;
(N Description-Level3
        |NUMBER  are atoms, can be used as symbols, can have properties.|)
(N Syntax-Level3      |        #/O for octal, #/X for hex, #/D for decimal|)
(N Example-level3     |       #/X10.2|)


;=================================================================;;

(NUMFUNC Description-Level |NUMBER  Any combination of the numbers 0 - 9 |)
(NUMFUNC Syntax-Level1     NIL)
(NUMFUNC Example-Level1    |             123|)
;;===                                                              ===;;
(NUMFUNC Description-Level2 |NUMBER   Can have a decimal point or a sign |)
(NUMFUNC Syntax-Level2      NIL)
(NUMFUNC Example-Level2     |             -123.5|)
;=================================================================;;
(NUMFUNC Description-Level3
           |NUMBER numbers can be used as symbols, can have properties.|)
(NUMFUNC Syntax-Level3
           |can be preceded by #/O - octal, #/X - hex, #/D - decimal|)
(NUMFUNC Example-level3 |      #/X10.2|)
;;===                                                              ===;;
(NUMBER Description-Level
                  |NUMBER  Any combination of the numbers 0 - 9 and + -|)
(NUMBER Syntax-Level1      NIL)
(NUMBER Example-Level1     |             123|)
```

```
;;===                                                                    ===;;
(NUMBER Description-Level2 |NUMBER    Can have a decimal point or a sign |)
(NUMBER Syntax-Level2      NIL)
(NUMBER Example-Level2     |              -123.5|)
;;===                                                                    ===;;
(NUMBER Description-Level3
                    |NUMBER can be used as symbols, can have properties.|)
(NUMBER Syntax-Level3 | #/O for octal, #/X for hexidecimal, #/D for decimal|)
(NUMBER Example-level3 |  #/X10.2|)

;=======================================================================;;

(A Syntax-Level1       |              Any string of characters| )
(A Description-Level1 |ATOM      A symbol or a number|)
(A Example-Level1      |              X            last_name          100|)
;;===                                                                    ===;;
(A Syntax-Level2       |              Any string of characters| )
(A Description-Level2 |ATOM      A symbol or a number|)
(A Example-Level2      |              X            last_name          100|)
;;===                                                                    ===;;
(A Syntax-Level3       |        Any string of characters| )
(A Description-Level3 |ATOM A symbol which hasn't necessarily been defined yet|)
(A Example-Level3      NIL)

;=======================================================================;;

(S Syntax-Level1       |       A string of characters that has been defined|)
(S Description-Level1 |SYMBOL A defined Lisp variable  or constant  |)
(S Example-Level1      |       After evaluating (SETQ A 10), A is a symbol|)
;
;;===                                                                    ===;;
(S Syntax-Level2       |       A system or user defined atom |)
(S Example-Level2      |       After evaluating  (SETQ A 10), A is a symbol |)
(S Description-Level2 |SYMBOL A defined Lisp variable or constant|)
;;===                                                                    ===;;
(S Syntax-Level3       |       A system or user defined atom  |)
(S Description-Level3
        |SYMBOL Symbols are defined by Lambda Lists, Let, SETQ & DefVar|)
(S Example-level3      NIL)

;=======================================================================;;

;(LISTFUNC      (SEND Self :ListFunc?)) ;= a function which returns lists

;=======================================================================;;
```

```
(L Syntax-Level1       |          '(listelement1 listelement2...) |)
(L Description-Level1
                |LIST Sequence of elements inside matching parentheses|)
(L Example-Level1      |          '(a b c)|)
;;===                                                                ===;;
(L Syntax-Level2
 |'(listelement1 listelement2...)or( LIST listelement1 listelement2...)|)
(L Description-Level2
    |LIST  Variables of the type list, or forms that evaluate to a list|)
(L Example-Level2
   |     '(first second third)   -or-  (SETQ x '(first second third) ) |)
;;===                                                                ===;;
(L Syntax-Level3       NIL)
(L Description-Level3 |LIST |)
(L Example-level3      |       |)


;=====================================================================;;

(LISTFUNC Syntax-Level1        |        '(listelement1 listelement2...) |)
(LISTFUNC Description-Level1
               |LIST  Sequence of elements inside matching parentheses|)
(LISTFUNC Example-Level1       |       '(a b c)|)
;;===                                                                ===;;
(LISTFUNC Syntax-Level2
          '(listelmnt1 listelmnt2...)or( LIST listelmnt1 listelmnt2...)||)
(LISTFUNC Description-Level2
            |LIST Variables of type list,or expressions evaluate to list|)
(LISTFUNC Example-Level2
         '(engine boxcar caboose) or(SETQ train '(engine boxcar caboose))|)
;;===                                                                ===;;
(LISTFUNC Syntax-Level3       NIL)
(LISTFUNC Description-Level3 | |)
(LISTFUNC Example-level3       |)

;=====================================================================;;

(QUOTABLE Syntax-Level1       |        Any string of characters |)
(QUOTABLE Description-Level1
                     |LITERAL a word that is not necessarily yet defined|)
(QUOTABLE Example-Level1      |        train  |)
;;===                                                                ===;;
(QUOTABLE Syntax-Level2       |          Alphanumeric characters |)
(QUOTABLE Description-Level2 |LITERAL    unevaluated token |)
```

```
(QUOTABLE Example-Level2     |          *&== |)
;;===                                                                ===;;
(QUOTABLE Syntax-Level3      |Any non token delimiter or macro characters|)
(QUOTABLE Description-Level3 |LITERAL    |)
(QUOTABLE Example-level3     |            '% |)

;=================================================================;;

(Q Syntax-Level1      |        Any string of characters |)
(Q Description-Level1 |LITERAL  a word that is not necessarily yet defined |)
(Q Example-Level1     |        train  |)
;;===                                                               ===;;
(Q Syntax-Level2      |         Alphanumeric characters |)
(Q Description-Level2 |LITERAL  unevaluated token |)
(Q Example-Level2     |         *&== |)
;;===                                                               ===;;
(Q Syntax-Level3      |        Any non token delimiter or macro characters|)
(Q Description-Level3 |LITERAL    |)
(Q Example-level3     |       '% |)

;=================================================================;;

(QLIST Syntax-Level1      |          Any string of characters |)
(QLIST Description-Level1 |LITERAL a word that is not necessarily yet defined |)
(QLIST Example-Level1     |             '(coach) |)
;;===                                                               ===;;
(QLIST Syntax-Level2      |            Alphanumeric characters |)
(QLIST Description-Level2 |LITERAL    unevaluated token |)
(QLIST Example-Level2     |            '(LISP programming && coach)  |)
;;===                                                               ===;;
(QLIST Syntax-Level3      |    Any non token delimiter or macro characters|)
(QLIST Description-Level3 |LITERAL    |)
(QLIST Example-level3     |            '( && 6 ( %%5)) |)

;=================================================================;;

(F Syntax-Level1   | (function-name function-argument) or a defined symbol|)
(F Example-Level1  | (PRINT 'coach)      or    9     or    T   |)
(F Description-Level1 |FORM  Something the LISP system can evaluate    |)
;;===                                                               ===;;
(F Syntax-Level2   | (function-name function-arguments) or defined symbol|)
(F Example-Level2  | (let ((h 1)) (print h)) |)
(F Description-Level2 |FORM  anything that LISP can evaluate |)
;;===                                                               ===;;
(F Syntax-Level3     | (function-name function-argument) or defined symbol|)
```

```
(F Example-level3     |       '(9 ,naughty) |)
(F Description-Level3  |FORM  any Interned Structure |)

;;=================================================================;;

([ Syntax-Level1          |              ( |)
([ Example-Level1         NIL)
([ Description-Level1     |              Open parenthesis |)
;;===                                                          ===;;
([ Syntax-Level2          |              ( |)
([ Example-Level2         NIL)
([ Description-Level2     |              Invokes functions, begins a list |)
;;===                                                          ===;;
([ Syntax-Level3          |              ( |)
([ Example-level3         NIL)
([ Description-Level3     |              Invokes functions, begins a list|)

;;=================================================================;;

(] Syntax-Level1          |               ) |)
(] Example-Level1         NIL)
(] Description-Level1     |               A close parenthesis  |)
;;===                                                          ===;;
(] Syntax-Level2          |              ) |)
(] Example-Level2         NIL)
(] Description-Level2     |              Closes functions, ends a list |)
;;===                                                          ===;;
(] Syntax-Level3          |              ) |)
(] Example-level3         NIL)
(] Description-Level3     |              Closes functions, ends a list |)

;;=================================================================;;

(ST Syntax-Level1         |               "any characters"|)
(ST Example-Level1        |              "COACH"|)
(ST Description-Level1
  |STRING     Strings are any characters between double quote marks.|)
;;===                                                          ===;;
(ST Syntax-Level2
  |              "any characters" or a symbol of the type string.|)
(ST Example-Level2        |              "(chars(" -or-  (setq a "hi")  |)
(ST Description-Level2
  |STRING     Strings are implicitly defined with the " delimiters.|)
;;===                                                          ===;;
(ST Syntax-Level3
```

```
  |              can use  \" to insert a double quote.|)
(ST Example-level3       |         "our position has been \"finalized\".|)
(ST Description-Level3   |STRING    art or artfat arrays of characters |)

;;=================================================================;;

(FUNC Description-Level1  |FUNCTION NAME Name of program|)
(FUNC Syntax-Level1      NIL)
(FUNC Example-Level1     |                          (PLUS|)
;;===                                                          ===;;
(FUNC Syntax-Level2      NIL)
(FUNC Example-Level2     |                       (COND|)
(FUNC Description-Level2 |FUNCTION NAME A defined  function name|)
;;===                                                          ===;;
(FUNC Syntax-Level3      NIL)
(FUNC Example-level3     |                            (LET |)
(FUNC Description-Level3 |FUNCTION NAME A defined macro or function name|)


;;=================================================================;;

(&OPTIONAL Syntax-Level1    |             followed by  variable names |)
(&OPTIONAL Example-Level1   NIL)
(&OPTIONAL Description-Level1 |&OPTIONAL  allows optional parameters |)
;;===                                                          ===;;
(&OPTIONAL Syntax-Level2    |followed by initable variable definitions |)
(&OPTIONAL Example-Level2   |  &OPTIONAL (a 10) |)
(&OPTIONAL Description-Level2
                    | &OPTIONAL variables can be given default values |)
;;===                                                          ===;;
(&OPTIONAL Syntax-Level3
                   | variable-descriptions; var or (var default-value)|)
(&OPTIONAL Example-level3     | &OPTIONAL (a (Plus 6 5))(b 2)c |)
(&OPTIONAL Description-Level3
                  | &OPTIONAL variables default nil, evaluated in order|)

;;=================================================================;;

(&KEY Syntax-Level1       |           followed by user  keyword(s) |)
(&KEY Example-Level1      |           &KEY cat |)
(&KEY Description-Level1 |&KEY allows user to define parameters by name |)
;;===                                                          ===;;
(&KEY Syntax-Level2       |    followed by user keyword(s) and a value |)
(&KEY Example-Level2      |    &KEY (cat 'tabby)dog |)
(&KEY Description-Level2 |&KEY arbitrarily ordered parameters default to nil|)
```

```
;;===                                                                ===;;
(&KEY Syntax-Level3       |   keywords can have values and default value |)
(&KEY Example-level3      |          &KEY (cat 'tabby)dog |)
(&KEY Description-Level3
                    |&KEY Allows parameters to be passed in arbitrary order|)

;=====================================================================;;

(X Description-Level1  |EVALUABLE  anything LISP understands|)
(X Syntax-Level1       NIL)
(X Example-Level1      |                 9        or       (plus 8 9)|)
;;===                                                                ===;;
(X Description-Level2 |EVALUABLE LISP  evaluable|)
(X Syntax-Level2       | A function, number, defined symbol, a string, etc.|)
(X Example-Level2      NIL)
;;===                                                                ===;;
(X Description-Level3 |EVALUABLE  |)
(X Syntax-Level3
            |A function or macro, number, defined symbol, a string, etc.|)
(X Example-level3  NIL)
;(X Example-level3 |  #'(LAMBDA ...)|)
)))
```

C.5 Lisp Concept Help

The following text demonstrates concepts available in the Lisp COACH and the way their help text is formatted.

```
;;==                                                                    ==;;
(ATOM Concept-Level1

Numbers, ie. 3, and symbols, ie. teds-thing are Atoms.

(QUOTE A) -- makes an atom ---> A )
(ATOM Concept-Level2

ATOMs are the basic indivisible data unit of LISP.  Anything that
is not represented as a cons.  Atoms like 3 or 99.5 are  NUMBERS.
Atoms such as FIRST, B27 or + are called SYMBOLS.)

(ATOM Concept-Level3

Two specializations of atoms are Numbers and named variables:
1. symbols [which have values that they return when they are called],
2. forms [which perform operations when they are called],
3. objects [things which have  properties])

;;==                                                                    ==;;
(BINDING Concept-Level1

associating a variable with a value

(SETQ A 9) ---- binds a to 9 ----> A)
(BINDING Concept-Level2

Binding is the process of preparing memory space for parameter values.

Parameters are bound when procedures are called.)

(BINDING Concept-Level3

Binding is the process of preparing memory space for parameter values.

Parameters are bound when procedures are called.)

;;==                                                                    ==;;
(COMMENT Concept-Level1
```

```
Comments start with a semicolon ;

;this is a comment.)


(COMMENT Concept-Level2

Lisp Comments are signaled by a Semicolon.

Semicolons and anything that follows on the line are ignored by Lisp.)

(COMMENT Concept-Level3

As well as ; comments, Functions can have Documentation lines which
follow the parameter list as a double quoted string i.e. "document string".

These documentation strings are available to other programs.)

;;==                                                                    ==;;
(CONS-CELL Concept-Level1

The elements of a list are called cons cells.)

(CONS-CELL Concept-Level2

Lists are made up of cons cells, extending a list or making new ones makes
new cons cells.)

(CONS-CELL Concept-Level3

Cons cells are dynamically allocated data cells with pointers.
Efficiency of Lisp code is dimished directly proportional to how many
cons cells are created.  For every cons cell that you create you must
collect it later through a garbage collection procedure.)

;;==                                                                    ==;;
(CONDITIONALS Concept-Level1

Conditionals are used for decision making.)

(CONDITIONALS Concept-Level2

Conditionals; COND and IF are used for deciding what to
execute in an if-then-elseif manner.)
```

```
(CONDITIONALS Concept-Level3

Conditional are used to execute code if and only if
a ceratin condition is met.)

;;==                                                              ==;;
(DATA-TYPES Concept-Level1

Examples of Data Types are numbers, symbols, and arrays.

9 ---- has a number data type)


(DATA-TYPES Concept-Level2

Different data types are needed for different purposes.
3 (number) is a different data type than 3.14 (floating point).)


(DATA-TYPES Concept-Level3

New data types can be created through the use of
Defstruct.)

;;==                                                              ==;;
(DYNAMIC-SCOPING Concept-Level1

Dynamic Scoping is the process of creating new variables and functions
(or gettting rid of them) as Lisp needs to (without recompiling).

(setq A 9) ----------------> 9

(let ((A 8))
 ) ----- in common lisp a statically scoped enviroment --> 8

 A  ---- outside the let a is still ----------------------> 9)

(DYNAMIC-SCOPING Concept-Level2

COMPILED LISP IS STATICALLY SCOPED.
Dynamic Scoping refers to practice a variable's value being dependent on
the order of things which happened.
Value of a variable is dependent on which function was the last to bind it.
This is in contrast to lexical scoping in
Algol or Pascal where one can tell how a variable will be bound by looking
```

at the function.)

(DYNAMIC-SCOPING Concept-Level3

Dynamic Scoping refers to practice a variable's value being dependent on the order of things which happened. The value of a variable at any given time is dependent solely on which function was the last to bind it. COMPILED LISP IS STATICALLY SCOPED.

;;==
==;;
(ENVIRONMENT Concept-Level1

The current things that are defined

(setq A 9) ---Defines A in the ENVIRONMENT ---> 9)

(ENVIRONMENT Concept-Level2

The current depth you are working on.)

(ENVIRONMENT Concept-Level3

Lisp enters a new environment each time it reads an open parenthesis, it leaves that environment when it reads the mathcing parenthesis.)

;;==
==;;
(EVALUATION Concept-Level1

The expression (plus 1 2) ---- evaluates to ----> 3.

The Quote function is used to keep things from being evaluated.

(EVALUATION Concept-Level2

Evaluation is the process of determining the value of expressions.
The Eval function can be used to explicitly evaluate data as program.
The Quote function is used to keep things from being evaluated.

The expression (plus 1 2) ---- evaluates to ----> 3.)

(EVALUATION Concept-Level3

Evaluation is the process of computing the VALUE of an S-EXPRESSION.
The expression (plus 1 2) is evaluated as 3.
The Eval function is often used to evaluate code in a function.

```
The Quote function is used to keep things from being evaluated.)

;;==
(FORM Concept-Level1                                                    ==;;

A FORM is a list that is meant to be evaluated.

(SETQ A 9) is a FORM)

(FORM Concept-Level2

A FORM is an EXPRESSION that is meant to be EVALUATED.
If it is a LIST, then the first ELEMENT
is generally a PROCEDURE that is to be used in the evaluation process.)

(FORM Concept-Level3

A FORM is an EXPRESSION that is meant to be EVALUATED.
If it is a LIST, then the first ELEMENT
is generally a PROCEDURE that is to be used in the evaluation process.)

;;==
(FUNCTIONS Concept-Level1                                               ==;;

PLUS is a function which returns a number

FUNCTIONS are are lists that can be evaluated.)

(FUNCTIONS Concept-Level2

FUNCTIONS are forms which return a value.)

;;==
(INTERPRETER Concept-Level1                                             ==;;

The LISP INTERPRETER  reads what you have typed and evaluates it.)

(INTERPRETER Concept-Level2

The LISP INTERPRETER is a PROGRAM that accepts LISP inputs and
EVALUATES them by translating them into machine readable
statements and executing them.)

;;==
(ITERATION Concept-Level1                                               ==;;
```

```
Itereration is doing something over and over again.
DO is a function which implements iteration.)


(ITERATION Concept-Level2

Iteration is the use of constructs for repetition.)

;;==                                                                    ==;;
(LAMBDA Concept-Level1

A LAMBDA is the way that LISP implements FUNCTIONS.

>((defun add4 (x) (plus 4 x) ------------------> ADD4
>(LAMBDA (x) (add 4 x) )

(LAMBDA Concept-Level2

LAMBDA LISTS allow a user to create temporary functions.)

;;==                                                                    ==;;
(Lambda-List Concept-Level1

A LAMBDA is the way that LISP implements FUNCTIONS.)

(Lambda-List Concept-Level2

LAMBDA LISTS allow a user to create temporary functions.)

;;==                                                                    ==;;
(LISTS Concept-Level1

A LIST is data inside of parenthesis.

> (LIST 'engine) ------------------------->((ENGINE))

(LISTS Concept-Level2

A LIST is a sequence of zero or more objects inside matching parenthesis.)

;;==                                                                    ==;;
(PARENTHESES Concept-Level1

PARENTHESES ( and ) are used to set off function calls and surround lists)
```

```
(PARENTHESES Concept-Level2

Parentheses demark Lisp objects,
ie.(plus 1 2)  (red white blue) (January 12 1948).
When LISP EVALUATES a list, it assumes that the first object is a FUNCTION
and that the remaining objects are ARGUMENTS for that function.

If a list is to be used literally and is not intended to be evaluated,
it must be preceded by a ' or use the QUOTE function.
In the expression (SETQ colors '(red white blue)), red does not
refer to a function to be performed, but to a literal value red.

(PARENTHESES Concept-Level3

Parenthesis are the syntax elements of LISP.)

;;==                                                                  ==;;
(NIL Concept-Level1

NIL is a SYMBOL to which LISP attaches the meanings false or empty.

Everything other than NIL is treated as true by lisp functions.

Functions which answer questions t or nil are Predicates.)


(NIL Concept-Level2

NIL is a SYMBOL to which LISP attaches the meanings false or empty.
Everything other than NIL is treated as true.

A list of zero elements ie. () is called the empty list.
The empty list can also be written as the symbol NIL.
PREDICATES functions answer questions with the symbols T or NIL
indicating true or false.)

(NIL Concept-Level3

NIL is a SYMBOL to which LISP attaches the meanings false or empty.

PREDICATES functions answer questions with the symbols T or NIL.
convention NIL is the only way to say NO in LISP, everything other than
NIL is treated as true(?).)
;;==                                                                  ==;;
```

```
(NUMBERS Concept-Level1

A NUMBER is any sequence of the digits 0 - 9.

numbers EVALUATE to themselves, ie. 99 evaluates to 99. )

(NUMBERS Concept-Level2

A NUMBER is any sequence of the digits 0 - 9.
Numbers may be preceeded by a + or - sign, and may have a decimal point.)

(NUMBERS Concept-Level3

A NUMBER is any sequence of the digits 0 - 9.
Numbers  may be preceeded by a + or - sign, and may have a decimal point.
#/0 imakes octal numbers, #X makes hexadecimal numbers.)
;;==                                                                    ==;;
(PREDICATES Concept-Level1

Predicates are functions which return true or False.  (ATOM 8) --> T)

(PREDICATES Concept-Level2

A predicate is a function that returns either true or false.
In LISP false is represented by NIL and true is represented by any value
other than NIL. (atom 'a) --> T  Predicates are used to alter the  flow of
control so that certain functions are performed only if specified
conditions exist. )

;;==                                                                    ==;;
(PROCEDURE Concept-Level1

A Proceedure list of things to do to solve a problem
FUNCTIONS are called procedures in PASCAL.)

(PROCEDURE Concept-Level2

A Proceedure is the thing in list that specifies how something is done.
Procedures may be defined by the user using the DEFUN command,
or may be PRIMITIVES, such as PLUS or CAR that are defined by the system.)

;;==                                                                    ==;;
(PRIMITIVE Concept-Level1

Built-in functions that the user doesn't have to define.)
```

```
;;==                                                              ==;;
(PROGRAM Concept-Level1

A program is a set of PROCEDURES that are designed to work together
to accomplish a specific task.

The DEFUN function is used to define programs.)
```

```
;;==                                                              ==;;
(PROPERTIES Concept-Level1

Are user definable slots attached to symbols.

(setf (get 'dog 'name) 'fido)
(get 'dog 'name)
FIDO)

PROPERTIES Concept-Level2

Are user definable slots attached to symbols.
SETF and GET are used to nake and access them.

(setf (get 'dog 'name) 'fido)
(get 'dog 'name)
FIDO)


(PROPERTIES Concept-Level3

Properties are a form of data structure.

Are user definable slots attached to symbols.
SETF and GET are used to nake and access them.

(setf (get 'dog 'name) 'fido)
(get 'dog 'name)
FIDO)
;;==                                                              ==;;
(RECURSION Concept-Level1

A function is said to be recursive, if it refers to itself in its
definition.)

(RECURSION Concept-Level2
```

```
A function is said to be recursive, if it refers to itself in its
definition.  Definition must include a check for an appropriate
termination condition.  Each time the function is called the task
must be simpler or it will never terminate. )

(RECURSION Concept-Level3

 LISP saves and restores the parameter bindings each
time it enters and leaves any function. )

;;==                                                              ==;;
(S-EXPRESSION Concept-Level1

A SYMBOLIC EXPRESSION is anything evaluable.
The term is often abbreviated S-EXPRESSION. )

;;==                                                              ==;;
(EXPRESSION Concept-Level1

A SYMBOLIC EXPRESSION is anything evaluable.
The term is often abbreviated S-EXPRESSION.)

;;==                                                              ==;;
(SYMBOLS  Concept-Level1

A SYMBOL is any sequence of letters, digits and permissible special
characters that is not a number.)

(SYMBOLS  Concept-Level2

A SYMBOL is any sequence of letters, digits and permissible special
characters that is not a number.)

(SYMBOLS  Concept-Level3

A SYMBOL is any sequence of letters, digits and permissible special
characters that is not a number.  Symbols are also called literal
ATOMS.  They serve the role of both FUNCTION and VARIABLE names, as well
as VALUES.  Symbols evaluate to the last value BOUND to them.)
```

C.6 COACH UNIX Definition

The following UNIX command language definition was controls a demonstration UNIX COACH.

```
(defun IVtokens () ;
(send *coach-parser* :set-tokens
      (append
(send *coach-parser* :tokens)
'(a c d f g i l q r s t u A C L F R n b p o + - = w x k )))

;;=============================================================================;;
;                              Machine syntax
;key symbols:
; A - Atom, S - Defined symbol, N - number, L - List, F - Form, X - anything
; Q -  check only parenthesis level              FS - Function Spec
;
;syntax delimiters: { - open a clause,    } -       close a clause,
;              [ - '(' open a clause,    ] -  ')' close a clause,
;
;vary syntax:      * -  next syntax part can occur 0 or more times,
;           ? -     next syntax part can occur 0 or 1    time
;                 V -  At least one component of the next clause must occur
;                      at least once
;;;;              @name is an internal syntax
;;;;  Logicals:   Conjunction in a template is indicated by juxtaposition
;;;;              Disjunction is not yet implemented
;;;;              Negation is not yet implemented
;;=============================================================================;;



;(unix-function-templates)
(defun unix-function-templates   ()

  (send *coach-parser* :set-function-templates
    (append
      (send *coach-parser* :function-templates)
      (list
 '(more ? {-c} ? {+ N} ? {- N} A  ) ;***
'(grep ? {- V {s n v i c } } ? {f {A} } ? {e {A} } '' A '' A ] )
'(ftp A ] )
'(alias A A ] )
'(chmod V { u g o a } V { + - = } V { r w x } ? A ] )
'(man ? { -k } A ] )
'(who ? { - V { u l q } } ] )
```

```
'(date ] )
'(ps ? { - V { l x a } } ] )
'(finger ? { - V { l  q  f } ? { g A } A ] )
'(mkdir V A ] )
'(login A ] )
'(ls  ?  { - V { a c d f g i l q r s t u 1 A C L F R } } A ] )
'(cd ?  A ] )
'(rm ? { - V { f r i } } A ] )
'(cat ? { - V { n s v b } A } ] )
'(cp ? { - V { i p } A ? A ] )
'(passwd ] )
'(history ] )
'(! ] )
'(diff A A ] )
'(jobs ] )
'(% N ] )
 '(fg ] )
'(bg ] )
'(awk A A ] )
'(find V {/A} A ]
) ) ) )

;;=============================================================================;;
  (defun IVstate-array ()  ;(SEND *Coach-Parser* :Set-State-Array (IVstate-array)
    (CL:make-array 108 :Initial-Contents

    ;Form+  delim- Symbol Quote string comment number help  (state,char)
     ;                Old State    (will be Last-State)
;   :pls   :min   :sym    :qte  :st  :cmt  :num  :Shp  :hlp
;unix                                                      ;"Type" of CHR
  '(:hlp  :hlp  :hlp   :hlp   :hlp :hlp  :hlp  :hlp  :hlp ;er  unused char
    :hlp  :pls  :sym   :sym   :st  :cmt  :hlp  :hlp  :hlp ;sp  macro char
    :num  :num  :sym   :Sym   :st  :cmt  :num  :hlp  :min ;num number
    :sym  :pls  :sym   :Sym   :st  :cmt  :Sym  :hlp  :min ;chr character
    :Pls  :pls  :pls   :Pls   :st  :cmt  :pls  :hlp  :hlp ;opn open paren
    :min  :min  :min   :hlp   :st  :cmt  :min  :hlp  :min ;cl  close paren
    :Hlp  :qte  :qte   :qte   :st  :cmt  :qte  :Shp  :hlp ;qte '
    :st   :st   :st    :st    :min :cmt  :st   :hlp  :hlp ;st  "
    :cmt  :cmt  :cmt   :cmt   :st  :cmt  :cmt  :hlp  :hlp ;cmt ;
    :pls  :min  :min   :hlp   :st  :min  :min  :hlp  :hlp ;ecm "end comment"
    :pls  :min  :min   :MIN   :st  :cmt  :min  :hlp  :hlp ;blnk blank
    :Shp  :Shp  :sym   :sym   :st  :cmt  :hlp  :hlp  :hlp))) ;Shp   #
```

C.7 Sample Unix Help Text

The following is a sample of the way the UNIX COACH help text is formatted in the system.

```
(CD Description-level1

"        cd changes the current directory/.  ")

;;===                                                                    ===;;

(CD Syntax-level1

"        cd directory ")
;;===                                                                    ===;;

(CD Example-level1

"        $ pwd ==========>>    //u//schoen//unix

        $ cd sam

        $ pwd  ==========>>    //u//schoen//unix//sam

        ---------------------------'------------------
        The working directory was changed to 'sam'  ")
;;===                                                                    ===;;

(CD Description-level2
"     cd changes the current working directory/.

     cd /./. changes your directory one step up in the levels of directories/.
     pwd prints the current directory/. ")

;;===                                                                    ===;;

(CD Syntax-level2
"        cd directory          cd /./.   ")

;;===                                                                    ===;;

(CD Example-level2
"        $ pwd     =========>>    //u//schoen//unix

        $ cd /./.
```

```
        $ pwd      =========>>    //u//schoen
        -------------------------'--------------------
        cd /./.  took you up one level in the hierarchy/. ")
;;===                                                                    ===;;
(CD Description-level3

"       cd changes the current working directory/.

You can use "cd directory" to go down one level in the hierarchy/.
You can use "cd /./ " to go up one level in the hierarchy/.
To jump more than one level you must specify the pathname "//pathname"  ")
;;===                                                                    ===;;
(CD Syntax-level3

"       cd directory          cd /./.          cd //pathname /./.    ")
;;===                                                                    ===;;
(CD Example-level3

"       $ pwd      ============>>    //u//schoen//unix

       $ cd //lib//bin

       $ pwd      ============>>   //lib//bin
       -------------------------'-----------------------------
          cd //lib//bin  took you to the directory //lib//bin in one step/. ")
 . . . *
```

C.8 Unix Token Help Text

The following is the token help definition for the UNIX COACH.

```
;;; = = = = = = = = = = Token Facts Initialization = = = = = = = = = = ;;;
;(unix-token-facts-Init)

;;;; Tokens-     a c d f g i l q r s t u A C L F R n b p o + - = w x k ))
(DEFUN unix-Token-facts-Init ()
   (MAPCAR

#'(LAMBDA (FACT) (PutProp (FIRST fact) (STRING (THIRD fact)) (SECOND fact)))

'(

(- Description-Level1      |-          -|)
(- Syntax-Level1           NIL)
(- Example-Level1           |            -|)
;;===                                                          ===;;
(u Description-Level1      |u          u|)
(u Syntax-Level1           NIL)
(u Example-Level1           |            u|)
;;===                                                          ===;;
(a Description-Level1      |a         n a|)
(a Syntax-Level1           NIL)
(a Example-Level1           |            a|)
;;===                                                          ===;;
(o Description-Level1      |o         n o|)
(o Syntax-Level1           NIL)
(o Example-Level1           |            o|)
;;===                                                          ===;;
(g Description-Level1      |g          g|)
(g Syntax-Level1           NIL)
(g Example-Level1           |            g|)
;;===                                                          ===;;
(x Description-Level1      |x         n x|)
(x Syntax-Level1           NIL)
(x Example-Level1           |            x|)
;;===                                                          ===;;
(c Description-Level1      |c          c|)
(c Syntax-Level1           NIL)
(c Example-Level1           |            c|)
;;===                                                          ===;;
(d Description-Level1      |d          d|)
(d Syntax-Level1           NIL)
(d Example-Level1           |            d|)
```

```
;;===                                                                    ===;;
(f Description-Level1      |f          n f|)
(f Syntax-Level1           NIL)
(f Example-Level1           |             f|)
;;===                                                                    ===;;
(i Description-Level1      |i          n i|)
(i Syntax-Level1           NIL)
(i Example-Level1           |             i|)
;;===                                                                    ===;;
(l Description-Level1      |l          n l|)
(l Syntax-Level1           NIL)
(l Example-Level1           |             l|)
;;===                                                                    ===;;
(q Description-Level1      |q           q|)
(q Syntax-Level1           NIL)
(q Example-Level1           |             q|)
;;===                                                                    ===;;
(r Description-Level1      |r          n r|)
(r Syntax-Level1           NIL)
(r Example-Level1           |             r|)
;;===                                                                    ===;;
(s Description-Level1      |s          n s|)
(s Syntax-Level1           NIL)
(s Example-Level1           |             s|)
;;===                                                                    ===;;
(t Description-Level1      |s          n s|)
(t Syntax-Level1           NIL)
(t Example-Level1           |             s|)
;;===                                                                    ===;;
(A Description-Level1      |A          n A|)
(A Syntax-Level1           NIL)
(A Example-Level1           |             A|)
;;===                                                                    ===;;
(C Description-Level1      |C          n C|)
(C Syntax-Level1           NIL)
(C Example-Level1           |             C|)
;;===                                                                    ===;;
(L Description-Level1      |L          n L|)
(L Syntax-Level1           NIL)
(L Example-Level1           |             L|)
;;===                                                                    ===;;
(F Description-Level1      |F          n F|)
(F Syntax-Level1           NIL)
(F Example-Level1           |             F|)
;;===                                                                    ===;;
```

```
(R Description-Level1      |R          n R|)
(R Syntax-Level1           NIL)
(R Example-Level1           |             R|)
;;===                                                          ===;;
(n Description-Level1      |n          n n|)
(n Syntax-Level1           NIL)
(n Example-Level1           |             n|)
;;===                                                          ===;;
(b Description-Level1      |b           b|)
(b Syntax-Level1           NIL)
(b Example-Level1           |             b|)
;;===                                                          ===;;
(p Description-Level1       |p          p|)
(p Syntax-Level1           NIL)
(p Example-Level1           |             p|)
;;===                                                          ===;;
(+ Description-Level1       |+          +|)
(+ Syntax-Level1           NIL)
(+ Example-Level1           |             +|)
;;===                                                          ===;;
(= Description-Level1       |=         n =|)
(= Syntax-Level1           NIL)
(= Example-Level1           |             =|)
;;===                                                          ===;;
(w Description-Level1       |w          w|)
(w Syntax-Level1           NIL)
(w Example-Level1           |             w|)
;;===                                                          ===;;
(k Description-Level1      |k           k|)
(k Syntax-Level1           NIL)
(k Example-Level1           |             k|)
;;===                                                          ===;;

)))
```

References

[Alexander and Jagannathan, 1986] S. M. Alexander and V. Jagannathan. Advisory system for control chart selection. *Computer And Industrial Engineering*, 10(3), 1986.

[Aronson and Briggs, 1983] Dennis Aronson and Leslie Briggs. *Instructional-Design Theories and Models: An Overview of Their Current Status*, chapter Contributions of Gagne' and Briggs to a Prescriptive Model of Instruction, pages 75–163. Lawrence Erlbaum Associates, New York, 1983.

[Barr and Feigenbaum, 1984] Avron Barr and Edward A. Feigenbaum, editors. *The Handbook of Artificial Intelligence.* William Kaufmann, Los Altos CA, 1984.

[Bereiter and Scardamalia, 1984] C. Bereiter and M. Scardamalia. *Learning and comprehension*, chapter Information Processing Demand of Text Composition, pages 407–421. Erlbaum, Hillsdale, NJ, 1984.

[Bonar and Soloway, 1985] Jeff Bonar and Elliot Soloway. Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1:133–161, 1985.

[Borenstein, 1985] Nathaniel S. Borenstein. *The Design and Evaluation of On-line Help Systems.* PhD thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1985.

[Brehm and Brehm, 1981] S. Brehm and J. W. Brehm. *Psychological Reactance: A Theory of Freedom and Control.* Accademic Press, New York, 1981.

[Brownlee, 1984] K. A. Brownlee. *Statistical Theory and Method in Science and Engineering.* Krieger, Malabar, FL, 1984.

[Burton and Brown, 1982] Richard Burton and John Seely Brown. *Intelligent Tutoring Systems*, chapter 2. Addison-Wesley, New York, 1982.

[Burton, 1978] Richard Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2, 1978.

[Burton, 1982] Richard Burton. *Intelligent Tutoring Systems*, chapter 4. Academic Press, New York, 1982.

[Campbell, 1989] Robert L. Campbell. Developmental levels and scenarios for smalltalk programming. Technical Report RC15305, IBM T. J. Watson Research Center, Yorktown Heights, NY, December 1989.

[Campbell, 1990] Robert L. Campbell. Online assistance: Conceptual issues. Technical Report RC15407, IBM T. J. Watson Research Center, Yorktown Heights, NY, December 1990.

[Carbonell, 1970] Jaime G. Carbonell. An artificial intelligence approach to computer assisted instruction. *IEEE Transactions on Man-Machine Systems*, MMS-11(4), 1970.

[Carbonell, 1979] Jaime G. Carbonell. Computer models of human personality traits. Technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh PA, 1979.

[Carroll and Aaronson, 1988] John M. Carroll and Amy Aaronson. Learning by doing with simulated intelligent help. *CACM*, 31(9), 1988.

[Clancy, 1986] William Clancy. From guidon to neomycin and heracles in twenty short lessons: Orn final report 1979–1985. *The AI Magazine*, pages 40–60, August 1986.

[Collins and Stevens, 1983] Alan Collins and A. L. Stevens. *Instructional-Design Theories and Models: An Overview of Their Current Status*, chapter A cognitive Theory of

Inquiry Teaching, pages 247–229. Lawrence Erlbaum Associates, New York, 1983.

[Conklin, 1986] Jeff Conklin. A survey of hypertext. Technical Report STD-356-86, MCC, Austin TX, October 1986.

[Corbett and Anderson, 1989] Albert T. Corbett and John R. Anderson. Feedback timing and student control in the lisp intelligent tutoring system. In *Proceedings of The International Conference on Artificial Intelligence*, pages 64–72. IOS, Amsterdam, 1989.

[Danuloff, 1991] Craig Danuloff. *The System 7 Book.* Ventana Press, Chapel Hill NC, 1991.

[Davis and Shortliffe, 1977] Randal Davis and Edward Shortliffe. Production rules as a representation for a knowledge- based consultation system. In *CHI '87 Proceedings*, volume 8, pages 15–45, 1977.

[Ellis and Sibley, 1966] Tom O. Ellis and William L. Sibley. The grail project. *Spring Joint Computer Conference, Boston, MA*, 1966. Verbal and film presentation.

[Erman and Lesser, 1975] L. D. Erman and Victor Lesser. A multi-level organization for problem solving using many diverse, cooperating sources of knowlege. In *IJCAI*, volume 4, pages 483–490, 1975.

[Feldman, 1980] David Henry Feldman, editor. *Beyond Universals In Child and Adult Development.* Ablex, Norwood NJ, 1980.

[Fikes and Keeler, 1985] Richard Fikes and Keeler. The role of frame based representations in reasoning. *CACM*, 28(9):904–920, September 1985.

[Fischer *et al.*, 1985] Gerard Fischer, A. Lemke, and T. Schwab. Knowledge-based help systems. In *CHI Proceedings*, 1985.

[Gano, 1982] Steve Gano. Movie manual. Technical report, MIT Media Lab, Cambridge MA, 1982.

[Genesereth, 1982] Michael Genesereth. *Intelligent Tutoring Systems*, chapter The Genetic Graph. Addison-Wesley, New York, 1982.

[Gentner, 1986] Don R. Gentner. A tutor based on active schemas. *Computational Intelligence*, 2, 1986.

[Glaser, 1985] Robert Glaser. Thoughts on expertise. Technical Report AD-A157 394, Learning Research and Development Center, Pittsburgh, PA, May 1985.

[Grise, 1986] Richard F. Grise, Jr. ANGEL: A pleasant user-interface for an interactive computing environment. Master's thesis, Cybernetic Systems Department, San Jose University, San Jose, CA, 1986.

[Heidegger, 1977] Martin Heidegger. *The Question Concerning Technology.* Harper & Row, New York, 1977.

[Hewitt, 1972] Carl Hewitt. Description and theoretical analysis (using schemata) of planner, a language for proving theorems and manipulating models in a robot. Technical Report TR-258, MIT A.I. Laboratory, Cambridge MA, 1972.

[Hewitt, 1985] Carl Hewitt. The challenge of open systems. *Byte Magazine*, pages 223–342, April 1985.

[Hopcroft and Ullman, 1979] John E. Hopcroft and Jeffery D. Ullman. *Introduction to Automa Theory, Languages, and Computation.* Addison-Wesley, New York, 1979.

[Houghton, 1984] R. C. Houghton. On-line help systems: A conspectus. *CACM*, 27(2), 1984.

[Kernighan and Pike, 1984a] Brian W. Kernighan and Bob Pike, editors. *The UNIX Programming Enviroment*, pages 240–255. Prentice-Hall, New York, 1984.

[Kernighan and Pike, 1984b] Brian W. Kernighan and Bob Pike, editors. *The UNIX Programming Enviroment.* Prentice-Hall, New York, 1984.

[Lawrence, 1984] K. Lawrence. Artificial intelligence in the man/machine interface. *Data Processing*, 1:231–236, 1984.

[Lieberman and Hewitt, 1980] Henry Lieberman and Carl Hewitt. A session with tinker. Technical Report 577, MIT AI Laboratory, Cambridge MA, September 1980.

[Lieberman, 1985] Henry Lieberman. There's more to menu systems than meets the screen. In *Proceedings of the ACM/SIGGRAPH Conference*, volume 24, 1985.

[Mackinlay, 1986] Jock Mackinlay. *Automatic Design of Graphical Presentations.* PhD thesis, Computer Science Department, Stanford University, Stanford CA, 1986.

[Malone and Lepper, 1987] Tom W. Malone and M. R. Lepper. chapter Making Learning Fun: A Taxonomy of Intrinsic Motivation For Learning. Lawrence Erlbaum Associates, New York, 1987.

[Manna, 1972] Z. Manna. *Mathematical Theory of Computation*, chapter 5–3. McGraw-Hill, NY, 1972.

[Mastaglio, 1989] Tom Mastaglio. Tutors coaches and critics. Technical report, Computer Science Department, University of Colorodo, Boulder CO, 1989.

[Mays *et al.*, 1988] Eric Mays, Chidanand Apte, James Griesmer, and John Kastner. Experience with K-Rep: An object-centered knowledge representation. In *The Fourth Conference on Artificial Intelligence Applications, Proceedings.* IEEE Computer Society Press, 1988.

[Merrill, 1983] M. David Merrill. *Instructional-Design Theories and Models: An Overview of Their Current Status*, chapter Component Display Theory, pages 279–334. Lawrence Erlbaum Associates, New York, 1983.

[Michalski *et al.*, 1983] R. S. Michalski, J. G. Carbonell, and T. M. Mitchell. *Machine Learning: An Artificial Intelligence Approach.* Tioga Publishing Company, Palo Alto, CA, 1983.

[Minsky, 1976] Marvin Minsky. Frames. Technical report, AI Laboratory, MIT, Cambridge MA, 1976.

[Moon, 1987] David Moon. *Users Guide To Symbolics Computers*, 1987.

[Morris and Rouse, 1986] N. M. Morris and William B. Rouse. Adaptive aiding for human-computer control: Experimental studies of. Technical Report AAMRL-TR-86-005, Armstrong Medical Research Laboratory, Wright-Patterson Air Force Base, OH, 1986.

[Myers, 1986] Brad A. Myers. Visual programming, programming by example, and program visualization: A taxonomy. In *CHI '86 Proceedings*, pages 59–66, 1986.

[Perry, 1984] Susan F. Perry. *Document Composition Facility: General Markup Language Starter Set Reference; Release 3.* IBM, 1984.

[Pirolli, 1986] Pete Pirolli. A cognitive model and computer tutor for programming recursion. *Human-Computer Interaction*, 2, 1986.

[Reigeluth, 1983] Charles M. Reigeluth, editor. *Instructional-Design Theories and Models: An Overview of Their Current Status.* Lawrence Erlbaum Associates, New York, 1983.

[Reiser *et al.*, 1985] B. J. Reiser, John R. Anderson, and R. G. Farrell. Dynamic student modeling in an intelligent tutor for lisp programming. In *IJCAI*, volume 1, 1985.

[Reisner, 1986] Philis Reisner. Human computer interaction: What is it and what research is needed? Technical Report RJ5308, IBM Almaden Research Center, Almaden CA, 1986.

[Revesman, 1983] Mark E. Revesman. *Validation and Application of A Model of Human Decision Making For.* PhD thesis, Industrial Engineering and Operations Research, Virginia Polytechnic, VA, 1983.

[Rich, 1983] Elaine Rich. Users are individuals: Individualizing user models. *Int. Man-Machine Studies*, 18:199–214, 1983.

[Rissland, 1978] Edwina Rissland. Understanding understanding mathematics. *Cognitive Science*, 2:361–383, 1978.

[Scheifler, 1987] R. W. Scheifler. *Xwindow System Protocol, Version 11.* MIT, 1987.

[Schofield *et al.*, 1990] J. Schofield, D. Evans-Rhodes, and B. Huber. Artificial intelligence in the classroom: The impact of a computer-based tutor on teachers and students. *Social Science Computer Review*, 1990.

[Selfridge, 1985] Oliver Selfridge. Personal communication, 1985.

[Selker and Koved, 1988] Ted Selker and Larry Koved. Elements of visual language. *1988 IEEE Workshop On Visual Languages*, October 1988.

[Selker and Schoenblum, 1990] Ted Selker and Mathew Schoenblum. Visual elements workspace (view). Technical report, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1990.

[Selker, 1989] Ted Selker. Cognitive adaptive computer help (coach). In *Proceedings of The International Conference on Artificial Intelligence*, pages 25–34. IOS, Amsterdam, 1989.

[Selker, 1991] Ted Selker. Cognitive adaptive computer help. Technical Report, videotape, 1991.

[Sleeman and Brown, 1982] Derrik Sleeman and John Seely Brown, editors. *Intelligent Tutoring Systems.* Academic Press, New York, 1982.

[Snelbecker, 1983] Glenn E. Snelbecker. *Instructional-Design Theories and Models: An Overview of Their Current Status*, chapter Is Instructional Theory Alive and Well?, pages 437–472. Lawrence Erlbaum Associates, New York, 1983.

[Suppes, 1967] Patrick Suppes. Some theoretical models for mathematics learning. *Journal of Research and Development in Education*, pages 4–22, 1967.

[Sussman *et al.*, 1970] Gerry Sussman, Terry Winograd, and Eugene Charniak. Micro-planner reference manual. Technical Report AI Memo 203, AI Laboratory, MIT, Cambridge MA, 1970.

[Symbolics, 1986] Inc. Symbolics. *Symbolics Document Set.* Symbolics, Inc., 1986.

[Teitelman and Massinter, 1981] William Teitelman and Larry Massinter. The interlisp programming environment. *Computer*, 14(4):25–34, 1981.

[Vertelney *et al.*, 1991] Laurie Vertelney, Michael Arent, and Henry Lieberman. *The Art of Human-Computer Interface Design*, chapter Two Disciplines in Search of an Interface: Reflections on a Design Problem. Addison-Wesley, New York, 1991.

[Waters, 1982] Richard C. Waters. The programmer's apprentice: Knowledge based program editing. *IEEE Transactions on software Engineering*, SE-8(1):1–12, 1982.

[Weiss, 1987] L. Weiss. Conceptual model of an intelligent help system. Technical Report DDC/LW-15, ESPREE, May 1987.

[Whiteside and Wixon, 1986] John Whiteside and Don Wixon. Improving human-computer interaction: A quest for cognitive science. Technical report, Digital Equipment Corporation, Maynard MA, 1986.

[Winograd and Flores, 1986] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Ablex, Norwood NJ, 1986.

[Winston, 1977] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, New York, 1977.

[Wolfram, 1988] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, New York, 1988.

[Zellermayer *et al.*, 1991] Michael Zellermayer, Gavriel Salomon, Tamar Globerson, and Hanna Givon. Enhancing writing-related metacognitions through a computerized writing partner. *American Education Research Journal*, pages 373–391, Summer 1991.

[Zissos and Witten, 1985] Adrian Y. Zissos and Ian H. Witten. User modeling for a computer coach: A case study. *Int. J. Man-Machine Studies*, (23), 1985.