# A USER MODELLING APPROACH FOR COMPUTER-BASED CRITIQUING

by

THOMAS WALTER MASTAGLIO

B.S., U.S. Military Academy, 1969

M.S., University of Colorado, 1978

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1990

This thesis for the degree of Doctor of Philosophy by

Thomas Walter Mastaglio

has been approved for the

Department of Computer Science

by

[illegible]

Gerhard Fischer

[illegible]

Clayton Lewis

Date 12 July 1990

Mastaglio, Thomas Walter (Ph.D., Computer Science)

A User Modelling Approach for computer-based Critiquing

Thesis directed by Professor Gerhard Fischer

Theoretical studies and implementations of computer-based critiquing systems indicate that it is desirable to enhance that approach to better support human-computer collaborative effort. A user model will enable these systems to individualize explanations of their advice to provide better support for cooperative problem solving and enhance user learning. User modelling research in advice-giving dialog and intelligent educational systems was studied together with theoretical analyses of the limitations of human-computer interaction, and empirical observations of human-to-human collaborative effort. A framework for a user modelling component for a critiquing system was developed and implemented in a critic for LISP programs. The user models developed by the system were compared to self-assessment questionnaires completed by subjects learning the LISP language. The analyses indicated a favorable correlation and potential improvements to the framework. The user model is based on the conceptual domain model required for explanations; its semantic structure allows the system to implicitly enrich the user model contents. The significance of this ~~work~~ is a framework for a user modelling component that can be used for a more general class of cooperative knowledge-based systems. Additionally, using the structure of the conceptual domain model as the basis for the indirect implicit inference techniques is unique. The theoretical foundations for the work, the framework developed, and an analysis of the implementation are presented.

To the women in my life. To my mother, Mildred, who helped me to develop the personal dedication and self confidence to get to this point in my education. To my daughter, Mandye, who, unwittingly, provided me an example of what dedication and sense of purpose are all about, one that motivated me during times of discouragement and frustration. And to my wife, Diane, who was there for me during those times of frustration, and all too frequently had to endure their result. I love you all.

## ACKNOWLEDGEMENTS

It is obvious from the list of referenced publications that the foundations for much of this dissertation rest on the theoretical ideas and concepts developed by my advisor and committee chairman, Gerhard Fischer. My attempts to extend those ideas to specify how one should model users of cooperative knowledge-based systems resulted from his patient tutoring and encouragement. The support from the members of my thesis committee: Barbara Fox, Walter Kintsch, Clayton Lewis, Jim Martin, and Gary Nutt helped to clarify my understanding and is much appreciated. Bob Kass, who served as an outside reader, provided me invaluable feedback. This thesis could not have been completed without the joint study of the critiquing paradigm accomplished together with Andreas Lemke and Anders Morch. Although our discussions were sometimes heated, they were always helpful to me. My good friend and office mate, Brent Reeves, was both a major contributor to the LISP-CRITIC system, and served willingly as my personal ''critic'' during the last several years of this research. Others who played a role in either refining LISP-CRITIC or working on the theoretical framework include Hal Eden, Patrick Lynn, and John Rieman. Andreas Girgensohn was always willing to provide much appreciated assistance during system building work; his expertise is an appreciated source of advice. The always helpful feedback from the entire Human-Computer Communications Research Group, although at times a source of frustration, challenged my thinking and helped me direct my efforts. I would also like to publicly acknowledge how fortunate I am that the United States Army provided the me the time and financial support to complete the requirements for a Doctor of Philosophy degree; I feel honored to have been selected for such a program. Finally, without the support of my wife and loving family this entire effort would not have been possible.

CONTENTS

## FIGURES

## TABLES

# CHAPTER I

# INTRODUCTION AND CONTEXT FOR THE RESEARCH

## 1.1. Introduction and Overview

This thesis discusses a user modelling approach to support cooperative problem solving. The problem investigated in this project is how to represent, acquire and provide access to individual user models to support computer critics. Critics are knowledge-based computer systems that use the critiquing approach to support their users in their work. The critiquing approach theoretically enhances the work produced by their users (these are called performance critics) and supports their learning (called educational critics) [Fischer et al. 90]. Future systems that support users in their working environments need the capability to accomplish both objectives. A user model will be an important component of such a system; it will assist the system give knowledge-based advice and, when it is appropriate, explain that advice. A framework for a user model to accomplish this was developed, implemented, and evaluated during the course of this research. Other user modelling research was studied as were approaches for generating explanations of domain expertise. Proven techniques from other user modelling research were incorporated, where possible, into the user model. An understanding of what is required to generate explanations guided the development of the architecture for the user modelling component. The user modelling component is as an extension of an existing critic for program enhancement (LISP-CRITIC).

The approach to user modelling combines methodologies developed by other research with innovative acquisition techniques. This work is unique in that it investigates enhancing the critiquing paradigm with the capability to individualize the explanations of advice given by the computer critic. The major contributions in this project are a framework for a user modelling component for critics that is also of potential use in other applications, and a set of techniques for indirect implicit acquisition of the user model. These techniques use the semantic structure of the conceptual domain model, the same model required for explanation-giving. The framework can provide support to both a broader range of applications, and to systems that use different interaction paradigms (such as tutoring or advising). The user model captures the expertise of individual users at the conceptual level of the application domain. The model is a fine grained representation of users' knowledge and therefore its contents could support other types of human-computer interaction. Cooperative problem solving systems are a general class of knowledge-based systems that will help future users of computer technology to both accomplish their vocational tasks and to enhance their understanding of the application domain. The approach to user modelling described here is general enough to serve the general class of cooperative problem solving systems.

The thesis is organized into three sections. Part 1 consists of Chapters 1 through 4; it describes the author's understanding of the theoretical foundations and analyzes related work. The four areas discussed are: using computers to support cooperative problem solving and learning, the paradigm of critiquing, an analysis of related user modelling research, and the implementation environment, LISP-CRITIC. Part 2 consists of Chapters 5 through 7; it covers the instantiation of the user modelling approach in system design and implementation. A requisite domain model to support this work, a framework for an explanation component that makes use of the user model, and the user modelling component that was developed, are described. Part 3, Chapters 8 through 10, analyzes the effectiveness of the implementation and the contributions of the work. Possible directions for continuing the research are identified, and the thesis concludes with a summary.

When people use a knowledge-based system they expect that it will help them to produce a better product.[1] As a by-product of this process, users improve their understanding of the application domain for that product. Therefore, to understand what it means for any system to support both doing and learning, it was necessary to examine two paradigms:

- cooperative problem solving systems, and
- user-centered computer learning environments.

The ability to explain its actions is a necessary characteristic for a system that attempts cooperative interaction and also supports learning. To achieve this it is necessary for these collaborative systems to employ user models to individualize those explanations.

Creating computer systems that facilitate cooperation between a human and a computer requires more than just developing powerful interaction technologies. We need an approach to computer support of problem solving that includes knowledge-based techniques, a computer-user dialog based on the idea of natural communications, and support for system adaptivity. The types of systems that achieve this will be collaborative symbiotic human-computer working environments that support user learning. A conceptual framework for systems that use knowledge-based techniques to aid users in accomplishing their tasks is provided by the cooperative problem solving paradigm.

[1]Product is used here in a general sense; it includes both specific objects generated by the work, such as a design, and abstract results, such as a decision.

## 1.2. Cooperative Problem Solving

There are methods and technologies in the field of Artificial Intelligence that can help improve the productivity of computer users. A paradigm for designing systems that goes beyond current autonomous expert systems to address human needs and potential is that of *Cooperative Problem Solving Systems* [Fischer 90]. These use knowledge-based techniques to work in symbiotic consonance with the user. The systems are cooperative in that they operate in a similar manner to the way a helpful person acts, and they attempt to assist their users as best they can. The relationship between the human and the computer is symbiotic in that there is mutual benefit; the resulting product of their collaboration is better than either could produce by themselves.

In cooperative problem solving, the user and computer-based system work on the same problem using a collaborative interaction style. Systems that can support cooperative problem solving will have to fit Illich's [Illich 73] notion of convivial tools [Fischer 83]. Cooperative systems allow the combining of human skills and computing power to accomplish a task which could not be done by either the human or the computer alone; or in those cases where it could be, the quality of the result or the speed with which a solution is obtained is significantly improved when the two agents work together. The idea of symbiotic cooperation between the user and the computer has also been embraced by researchers in the related field of decision support systems [Mili, Manheim 88; Manheim, Srivastava, Vlahos, Hsu, Jones 90; Hefley 90]. To achieve a cooperative system requires some degree of adaptivity to individual users; a system can adapt successfully when it knows something about the individual; and user modelling provides to a system the capability to acquire and use that type of knowledge.

The objective of an interaction between a human and a computer generally falls into one of two classes: problem solving or information retrieval.[2] Design might be considered as a third major class; the view here is that design is a subset of problem solving because the design is generated to address problematic needs of either users themselves or those of a client for whom they are working. Information retrieval is often one part of problem solving. The computer's role is to aid users in arriving at a solution or to help locate information that solves or helps to solve their problems.

Autonomous expert systems have a different approach — they develop problem solutions independent of user input, except for the problem specification or task designation. In many problem solving and information retrieval situations, articulation of the task is difficult and people are unsure of their objective or exact problem. They start with a general view of what they expect to achieve and refine their own understanding and problem specification as part of the solution process. To put cooperative problem solving systems into the context of current artificial intelligence technology we need to consider system designs that go beyond current notions of expert systems and to understand when it is appropriate to use these alternative approaches.

Feigenbaum and McCorduck state in their book on expert systems: "Most knowledge-based systems are intended to assist human endeavor and are almost never intended to be autonomous agents" [Feigenbaum, McCorduck 83, p. 115]. This view, unfortunately, is not held throughout the field. Those expert systems often cited as the major success stories of the past 10 to 15 years, for

[2]Amusement is also a significant application for computers, but we focus on applying technology in the workplace to accomplish a specific goal or task, rather than using it to entertain.

example R1, MYCIN, Dipmeter Advisor, have been designed as domain experts that are capable of solving a certain set of problems. These are problems that require either the heuristic knowledge of a human expert with considerable experience or problems that require excessive computation to yield to timely solutions using standard algorithmic approaches.

The major difference between classical expert systems (such as R1 or MYCIN) and cooperative problem solving systems is that the users of cooperative problem solving systems are active agents. They are actively engaged in reasoning about the problem and generating the solution rather than participating as mere providers of information to the system. Conversely, traditional expert systems ask users for information about the problem situation and then return a solution; from an operational perspective they appear as a ''black box.'' Cooperative problem solving system are designed so that the user and the system share in the problem solving and in the decision making. Because human-computer communication is central, cooperative problem solving systems require better interaction facilities than those offered by traditional expert systems.

When knowledge-based systems support decision at higher levels of societal and organizational responsibility they should not usurp the user's responsibilities for a decision. For example, a commander of a military operation should have access to the expertise captured in a knowledge-based system that knows strategy, combat resources, and heuristics about how to apply them to accomplish the mission. However, this situation involves significant danger to human lives and we would not want to turn control completely over to an expert system to run the battle. Similar scenarios exists in natural disaster emergency planning, and the operating of complex, potentially dangerous equipment (e.g., nuclear power stations). Autonomous expert system approaches must be replaced by a more general knowledge-based system paradigm, such as cooperative problem solving systems.

Communications between a human and a computer is a fundamental design problem for cooperative systems. Specifically, to facilitate interaction between a human and a computer it is necessary to exploit the asymmetry of the two communication partners. Each agent or partner contributes what they can do best. People are better than computers at applying common sense, defining goals, and decomposing problems. Computers should be used as an external memory, to do consistency checking, to hide but not lose irrelevant information, to capture and summarize problem solution steps, and to help visualize concepts. In this thesis the user model serves a system in which the user is a programmer who understands the problem and develops the code to solve that problem. The computer does not understand the problem and could not write a program (automatic program generation) to solve it even if it did. But the system knows more efficient ways to implement program code. The system will be described in detail in Chapter 4 but in the context here it is important to see that the system is designed to exploit each agent's expertise by sharing responsibility for producing a good solution. The programmer produces code that algorithmically solves the problem and the system reviews that code suggesting to the programmer ways to improve it.

Communicating means an agent, human or computer, has to know or must assume something about its partner. One approach is for systems to capture implicit assumptions about all of the users in their design (a default generic user model). Ideal systems might adapt everything they do to each individual. A more reasonable middle ground, is to have systems tailor their side of an interaction based on what they are able to infer about their human partner by applying user modelling methods. Understanding what is required to accomplish user modelling requires an examination of the human-computer communication process.

When two agents are engaged in cooperative effort, a process of natural communication takes place. Natural communication is more than natural language; it is the ability to engage in a dialog that makes use of diectic techniques, indexicals, graphic representation, and references to previous conversation. In human dyads more goes on than an exchange of information, if one partner serves as an advisor or critic they are expected to understand what the other is trying to do and guide them correctly. Cooperative computer systems need techniques for helping them attempt similar efforts. There is significant research into techniques for goal and plan recognition on the part of computers and also considerable skepticism whether computers will ever be able to accomplish this [Suchman 87; Dreyfus, Dreyfus 86]. This does not restrict systems from using knowledge of goals or plans that can be obtained by querying the users to help structure interactions with them. The user model component plays a role in that it must be able to represent users' goals and provide that information to the system aid in the communication process. Obtaining that information indirectly may eventually be part of acquiring the user model, but it is going to require effective goal/plan recognition techniques that are the subject of another direction of research in artificial intelligence.

It is important to think in terms of natural communications so that techniques in addition to natural language are integrated into system designs. Systems that use natural language generation and recognition techniques are frequently too brittle [Winograd, Flores 86]. They experience breakdowns in interacting with users when the unexpected occurs, especially in situations not anticipated by the system designer; techniques are needed to get past the breakdown. Another reason to think about the entire spectrum of communication techniques is that techniques for natural language generation and interpretation have matured to the point where they are generally useful in systems such as LISP-CRITIC. As is discussed in Chapter 4, the communication medium for LISP-CRITIC is a set of available techniques provided in a powerful workstation environment (menus, command languages, and hypertext). While waiting for natural language capabilities to mature to the point of general utility, it is necessary to build systems that address real needs using available technology, and to use them as a context for related research, such as user modelling.

Related to natural language limitations is the idea of situated action [Suchman 87]. Research by Suchman using the situated action perspective highlights some inevitable shortcomings of human-computer interaction paradigms. The limited bandwidth across which the human and computer can communicate preclude the machine from having access to both the quantity and quality of information available to a human. This observation should motivate efforts that investigate how to improve the capabilities of computers. User modelling is one technique that can help the computer to improve on its ability to aid a user in the situated context. The degree to which it will help is an open research issue that can only be investigated after suitable user modelling techniques are developed and implemented.

Computers can understand task domain knowledge; this knowledge can be used as the basis for advice, and as a source for guidance about how to better communicate that advice and explain it. The distinction between advising users (telling them what to do) and explaining that advice (telling them the reason for it) is not made in most research. A notable exception is the work on the EUROHELP project [Kelleher 88]. In this thesis the distinction is significant because the subject system gives both suggestions (or advice) and we want to endow it with the ability to explain those suggestions to the user. It is the latter process that makes use of the content of the user model.

A model of the individual user can be an important component of any system, such a model can aid in the natural communication process, assist in managing breakdowns, and help make systems more acceptable to their users. Modelling another agent occurs on both sides of a cooperative dyad. Users develop models of the systems with which they interact and computers need to be designed so they can develop models of human users. The work presented here focuses on the latter class of models.

A user modelling component will not solve all human-computer interaction problems but it is essential to investigate its potential impact. We must first develop system techniques for representing and building these models. But even with good user models, cooperative problem solving systems will not always be successful in their initial attempt to provide advice or explanation — what is sometimes called a ''one-shot'' approach to the interaction. Even people do not always ''get it right'' the first time; a great deal of the effort in any communications involves repairing breakdowns between the two partners. Techniques to achieve something similar are needed in interactive systems. When users do not understand the system's advice, critique, or explanation, followup techniques are required [Moore 89].

Human experts model their communications partner in order to provide the appropriate level of assistance and explanation. This motivates the requirement for cooperative computer systems to be designed to attempt something similar. Reeves conducted an empirical study of collaborative problem solving efforts where sales clerks in a large, well-stocked hardware store assist patrons in solving their problems [Reeves 90]. When interviewed, these expert agents related that part of the process involved modelling the client so that the advice given and any explanations requested could be specifically designed for that individual.

Modelling their customers played a crucial role in identifying, for these sales agents, the level at which to share their understanding.

Communication between cooperating agents can be viewed in terms of two roles, that of speaker and listener. The speaker presents information and the listener interprets it. The listener's role is usually more difficult because the listener has to understand the problem based on the speaker's description. Knowledge-based systems that communicate with a user have to be designed to accommodate both roles. This is especially true in cooperative problem solving systems where users play an active role in both the problem solving and the decision making processes. In this research the interface confined users to communicating with the system using only available technologies; a natural language context was not assumed. This restriction on users is necessary in order to accommodate the system's role as a listener. The system serves in the speaker role when it gives suggestions and when it explains suggestions. The user model developed here was primarily designed to accommodate the system in the speaker role. Ultimately, user models will have to help the system fulfill both roles.

In addition to helping users to solve their problems, systems should also help them learn about the task domain. Systems that serve knowledge workers, such as designers, authors, and programmers, must accomplish both objectives. In normal use it is difficult to distinguish a situation or episode oriented strictly on problem solving; learning and doing frequently intermix during human-computer interactions. Support for user learning is also a goal of a cooperative knowledge-based system, so the theory behind computer learning environments was also examined.

## 1.3. Learning Environments

Computer-based learning environment enable users to improve their proficiency in a domain by providing for the knowledge communication process [Wenger 87]. One approach to designing such learning environments is that of building instructionally focused systems, such as intelligent tutors [Sleeman, Brown 82]. In many situations it is desirable to provide a learning opportunity within the context of a user's work and with the user in control of the interaction. Here, paradigms that are more general than intelligent tutoring systems are needed. There is interest and research that addresses the problems in developing learning environments in several disciplines to include cognitive science [Burton, Brown, Fischer 84; Fox 88], human-computer interaction [Fischer 88a], and computer-based training technology [Duchastel 88; Mastaglio 90a].

The opportunity to develop computer-supported learning environments has motivated the reexamination of existing paradigms of education and the formulation of new ones. Many of these paradigms connect learning with experience, ascribing to a philosophical view and emphasizing techniques that make this connection the basis for system design [Psotka, Massey, Mutter 88a]. Because this is an emerging discipline the learning approaches have various names, such as: collaborative learning, reactive learning, situated learning, learning on demand, and incremental learning. There is some overlap between the paradigms; what is significant is that they share a common ideal. Their goal is to provide opportunities to learn skills by practicing them in a realistic work setting — the workplace or system in which the skills are to be used, or a simulation of that environment. The approach of augmenting a work environment is well suited to situations where the computer system is that environment (for example a CAD/CAM system). Computers are good for simulating in circumstances where training on the actual equipment (for example a power plant or aircraft) is prohibitively expensive, or there are safety considerations. In order to individualize learning opportunities, systems need to know what knowledge their users have about the domain; idiosyncratic models of individual domain expertise are needed. To understand what it means to have such a model, the theories behind learning approaches for interactive environments was examined.

## 1.3.1. Foundations for Learning Environments

There is a need to find efficient and practical ways to improve education using computer technology. Studies by Bloom and colleagues demonstrated that if we can individualize instruction, significant increases in student performance can be reasonably expected [Bloom 84]. The computer can provide an efficient methodology for individualized learning situations and researchers concerned with education and training want to use computer-based training to meet educational needs in a variety of contexts [Seidel, Weddle 87].

A shortcoming of early computer learning systems was their fundamental design philosophy; it was based on conventional ideas about programmed instruction used for self-paced learning material. An alternative approach using artificial intelligence techniques was first proposed by Carbonell [Carbonell 70]. During the ensuing 20 [illegible] since that work, research efforts have resulted in the development of sev[illegible] paradigms for computer-based instruction using artificial intelligence [Wenger 87]. The computer can provide a suitable context for learning both procedural and declarative knowledge and as reasonably priced hardware support for graphical displays becomes a reality, these types of environments will become more feasible. The theoretical limitation to effectiveness is not the availability of material or simulated scenarios but incorporating, into the system, a didactic agent to guide learners. For learning to occur in a computational context in which users are involved in some action, the system must provide feedback to its users in the form of advice, critiques, and explanations; higher quality of the feedback results in an improved learning process.

One way to provide a system the ability to provide high quality feedback is using knowledge-based components. Three processes in the learning environments require the use of knowledge:

1. providing information or instruction with an expected outcome that users' knowledge improves,
2. determining the state of users' present knowledge (user or student modelling), and
3. motivating the user to learn.

To execute the first process, a computer agent has to know both the domain and strategies for guiding users in the direction of learning. Methods to accomplish the second process are the subject of this research. Cooperative problem solving systems assume that the third process is inherent in the situation; they assume that users are motivated because they have chosen to use the system in the first place.

After investigating the literature, I found there are three distinguishable paradigms for computer-based education based on artificial intelligence techniques — three different classes of intelligent learning environments [Mastaglio 90a]:

- Intelligent tutoring systems present instructional material in a manner similar to a classroom teacher. [Sleeman, Brown 82; Psotka, Massey, Mutter 88b; Polson, Richardson 88].
- The coaching approach does not use teaching techniques in the strictest sense of the lecturing classroom teacher but its metaphorical basis is the human coach who places students into a suitable context, then

observes how they perform and provides advice on how to improve [Burton, Brown 82].

- The paradigm that serves as the foundational interaction methodology in this work is the computer-based critic [Fischer, Lemke, Mastaglio, Morch 90]. It serves as an intelligent agent, able to evaluate user's work on call like a human mentor or colleague. The critic is a domain expert ready to evaluate users' actions and provide suggestions whenever asked.

In this research, critiquing had been emphasized over tutoring because it holds promise as a more general approach. Education and training should not be viewed as only isolated activities that occur during set periods of a lifetime where the focus is on the acquisition of skills and knowledge. Instead, the broader perspective taken is that they are life long activities needed to maintain proficiency and accommodate changes in domain theory or technology. A computer-based critic can help users improve their skills within their working context, but besides giving suggestions they need to be able explain their expertise, not just in a canonical form but in a manner that is tailored to each user's current state of knowledge. For critics to fully support the learning process they will require explanation and user modelling capabilities.

Critiquing is not the only approach to designing a learning environment but it is effective in the application domain (programming) for this research. The user model developed should be able to serve not just critiquing but the general class of learning environments. To achieve that goal some of the theoretical foundations for learning environments were studied. The notions of situated learning [Brown, Burton, Kleer 82], and learning on demand [Fischer 88a] provided general guidance and understanding.

### 1.3.2. Learning on Demand

Learning on demand supports learning in the context of a user's work, allowing people to improve their knowledge whenever a need arises [Fischer 87a]. It is based on an optimistic view that people want to know how to do their jobs better and are willing to engage in learning activities. If computer-based working environments, (e.g., design environments [Lemke 89]) are to provide a complete and appealing context for working, then they need to support learning on demand. Learning environments can be designed as extensions of existing computer-based systems, ones that already support some types of work. Some examples of these are design, programming, and authoring.

The need to support learning on demand requires architectures that differ from instructionally oriented systems. Learning on demand requires that the user retain primary control of the learning situation. A continuum of approaches to designing learning environments is shown in Figure 1-1; the dimension for the continuum is control of the interaction. At the left extreme, primary control is the responsibility of the system; at the right, the responsibility of the user. In the center, the system and the user share responsibility for control. Exploratory learning, in the spirit of LOGO [Papert 80], is the type of system found at end of the continuum where the user has total control of the interaction. Exploratory learning environments do not require domain knowledge, but they must be carefully designed to provide opportunities for interesting exploration while at the same time protecting users from fatal errors. Traditional computer-aided instruction (CAI) is shown at the other end of the continuum. These systems are algorithmically controlled by the computer program and allow minimal, if any, student control; traditional CAI systems also do not have knowledge in the sense of artificial intelligence.

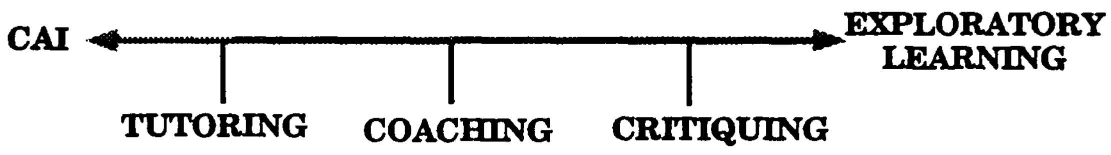


**Figure 1-1:** A Continuum of Approaches to Learning Environments

In the middle of the spectrum are the three knowledge-based paradigms for learning environments that were previously enumerated. Within the class of intelligent tutoring systems various degrees of control may be given to the student but, in general, problem selection, monitoring of student actions, and intervention fall under the auspices of the intelligent tutor. Coaching, as used in the WEST system [Burton, Brown 82], allows users to practice skills in a computer-supported context with a computer-based intelligent mentor ''watching over their shoulder.'' The coach intervenes with suggestions and instruction when appropriate.

Research efforts in both exploratory learning [Miller 79] and coaching systems [Brown, Burton, Kleer 82] recognized the need for a paradigm which is not as intrusive as a coach but extends the power of exploratory environments by providing the user contextual, on call, intelligent assistance in the application domain. Computer critiquing is a paradigm which meets that requirement, it allows users more control of the interaction as well as responsibility for selecting the problem.

A system that provides for learning on demand cannot help but situate that learning in the user's context. Because it is in this context that users will request support for learning; for the system to coerce them into a special learning microworld defeats the objective behind learning on demand. Realizing the situatedness of the working and learning context makes it important, in this research, to understand the theoretical studies of situated action and consider their impact on the goals of the user modelling effort.

## 1.3.3. Situated Action

Suchman made an in-depth study of how people interact with machines [Suchman 87] and argues that research approaches which attempt to explicitly represent or infer user plans are inadequate. First we need to explore the relationship of knowledge to action, keeping in mind that a machine's resources for interpreting the user's behavior are significantly poorer than those of a human.

A cautious attitude toward what to expect from a machine is important because during face-to-face communication between people there are resources that help them detect and remedy trouble when it develops, for example, facial expression or tone of voice, among others. The same range of resources are lacking, for the most part, in human-machine interaction because of the impoverished communication channel. A pessimistic perspective would discount attempts to make machines act intelligently. A more optimistic view is embraced here, the philosophy that it is important to investigate whatever possibilities to support the system do exist, while keeping the limitations of situated action in mind. Research on natural language as a communication medium assumes understanding and expression capabilities on the part of the computer. The orientation in this project was very different, the work looked at those communications capabilities which are available with current technology to determine how they could best be used to improve what the systems knows about the user, and also what knowledge about the domain is required by the system to make best use of these capabilities. One such idea considers the content of the man-machine interaction dialog as a source of information that can aid in acquiring models of users.

An environment that provides users the opportunity to learn in the context of the task domain or by using a simulation of that context, requires that the system react so as to increase their understanding of the situated action [Brown, Burton 86]. Critiquing can facilitate situated learning because it provides learning opportunities in the work context. The critique exposes situations for improving the product or the actions of users; these are learning opportunities as well. If users do not understand the critique then there is an opportunity for an explanation in context. They learn a set of conditions under which their knowledge can be applied and as a result improve their understanding. This learning scenario is a task-driven environment, providing a unique, situated context for learning to take place. This context is the central notion in situated learning; in critiquing systems it comes about naturally. Critiquing combined with explanation approaches can clarify understanding and help to restructure users' knowledge [Psotka, Massey, Mutter 88a]. Learning has been traditionally supported with instruction, but a more likely situation, one similar to the manner in which human-to-human interaction occurs, is to support it with an explanation capability [Wenger 87]. Explanation of the system's critique is supported by the user modelling system that was developed in this research.

## 1.4. Summary

This chapter examined the context for this research in terms of paradigms for cooperative problem solving and computer systems that support learning. The two cannot be easily separated on either a theoretical level or in systems designed to support users in their work. Therefore, both become considerations in how systems should be designed and in determining the role played by their user modelling component. To achieve true cooperativity, systems must adapt to their users; while support for learning requires that they provide the users feedback and explanations; explanations are best when tailored to the individual. This means idiosyncratic models of users are required. The exact content of those models is determined by the needs of the systems that will use the models. This chapter has explained the theoretical concepts for designing cooperative systems that support task accomplishment, and the motivation for having systems that support learning; both of these provide motivating goals for this dissertation research. The critiquing paradigm, the specific system design framework within which the user modelling work was completed, will be discussed next.

# CHAPTER II

# CRITIQUING

Critiquing, as a technique for building systems, is of research interest in both artificial intelligence and human-computer interaction. It has been a major topic of investigation for the Human-computer Communications Group at the University of Colorado during the past several years [Fischer, Mastaglio 89; Fischer et al. 90; Fischer, Mastaglio 90; Fischer, Lemke, Mastaglio, Morch 90; Mastaglio 89]. Critiquing is of interest because it is a way to use knowledge-based system techniques in situations where autonomous expert systems are inappropriate. In studying the approach we found that in order for a critic to meet the goals of cooperative problem solving and accommodate user learning it needs to be able to explain system knowledge in an individualized manner. This finding was the primary motivation for choosing to investigate user modelling in this thesis. That research required a clear understanding of the paradigm, so an important collaborative effort was to characterize critiquing; this chapter is an overview of that effort. The term critiquing is intended to mean the paradigm or technique, while we refer to systems that use critiquing as "critics".

Critics can support users in both problem solving and learning, they play an essential role in extending applications-oriented design kits to design environments, and are an alternative to traditional expert system. The approach has been used successfully in diverse application domains, to both aid in a cooperative problem solving process and to provide support for learning. Ideally a single system will achieve both of these goals; such a system will need the capability to adapt explanations of its advice to individual users.

## 2.1. Foundations for Critiquing

Powerful computer hardware makes it possible to use computers in an increasing range of application areas. As technical complexity increases, the associated cognitive costs to master computers grow dramatically and limit our ability to make full use of computer systems. Systems that offer rich functionality to their users need to be designed to be both useful and usable. It is a way to meet the goals developed in Chapter 1, providing support for learning and for cooperative problem solving. Critiquing also plays an important role in the concept of Design Environments; other work in our group has investigated and reported on that line of research [Lemke 89].

Cooperative Problem Solving. Critics, by their nature, operate in a somewhat cooperative manner; they can be further enhanced to more fully achieve the objective of having a cooperative problem solving system. They identify proposed solutions or strategies that could be done using an alternative approach. For users to accept critics as a useful feature of their working environment they need to provide explanations and, where appropriate, suggest alternative solutions.

Some shortcomings of traditional expert systems were pointed out in Chapter 1; another one is that these systems are inadequate when it is difficult to capture all requisite domain knowledge. Because expert systems often leave the human out of the process, they require comprehensive knowledge that covers all aspects of the tasks; all "intelligent" decisions are made by the computer. Some domains are not sufficiently well understood, and to create a complete set of principles that capture them is not possible. Some domains require considerable effort in order to acquire all relevant knowledge. Critics are suited to these situations because they need not be complete domain experts. Critics can still offer the user helpful guidance even when their expertise is limited to only some aspects of the problem domain.

The traditional expert system approach is also inappropriate when the problem is ill-defined. This is because the problem cannot be precisely specified before a tentative solution is attempted. In contrast, critics are able to function with only a partial task understanding. Even when the system contains only general knowledge about the problem domain, it can provide helpful support because there exist general principles that apply.

**Support for Learning.** The computational power of high functionality computer systems can provide qualitatively new learning environments; future learning technologies will be multi-faceted and support a portion of the spectrum of approaches that was shown in Figure 1-1. Some versions of intelligent tutoring systems developed in research laboratories allow the student to exercise greater control of the interaction. LISP TUTOR was reimplemented in a mode that permitted students to decide when the system could assess their work [Anderson, Conrad, Corbett 89]. Student performance on post tests were equivalent for the immediate feedback (tutor-controlled) and the demand feedback (user-controlled) versions. Students actually took longer to solve problems when feedback was under their control rather than the systems, however, the quality of the learning experience is not degraded. This has clear implications for systems designed to support learning in situations where it is necessary for users to provide the problem specification, such as in LISP-CRITIC. As previously mentioned, the need to provide knowledge-based assistance in exploratory learning environments is also recognized. There is a recognizable trend for designs of learning systems to move toward the middle of that continuum — closer to the critiquing paradigm.

Critics in passive help systems may not require users to formulate a specific query, but because they assist only when called, they allow users to retain control, providing advice only when the products or actions are recognized as significantly inferior. By integrating working and learning, critics offer unique opportunities for the user:

- to understand purpose of or use for the knowledge they are learning,
- to learn by actively applying knowledge rather than by passive exposure to it, and
- to learn one condition under which that knowledge can be applied.

A strength of critiquing is that learning occurs as a *natural byproduct* during the problem solving process.

## 2.2. The Critiquing Approach

Human-to-human critiquing is used in many problem solving contexts: design, authoring, student work groups, and collaborative research. People working together in these and similar areas naturally use critiquing as an interaction style. Critiquing is a way to present a reasoned opinion about a product or action (see Figure 2-1). The product could be a computer program, a kitchen design, a medical treatment plan; an action could be a sequence of keystrokes that corrects a mistake in a word processor document or a sequence of operating system commands. An agent (human or machine) that is capable of critiquing in this sense can be called a critic. Critics can be implemented on computers as a set of rules or specialists for the different issues that may be associated with a product; sometimes critics are the term used for each individual system component that reasons about a single issue. In this project we call the entire system a critic; part of its structure is a composite rule set.

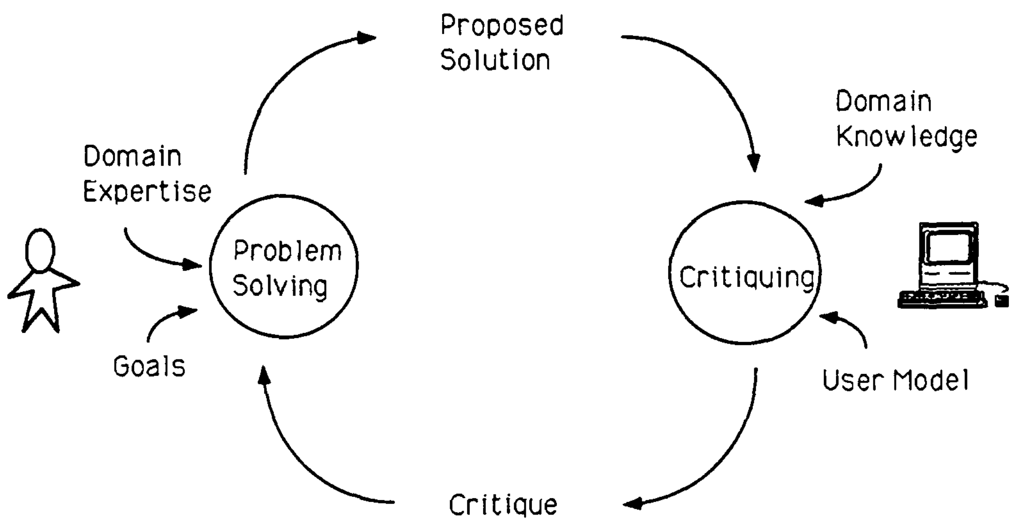


**Figure 2-1:** The Critiquing Approach

This figure shows that a critiquing system has two agents, a computer and a user, working in cooperation. Both agents contribute what they know about the domain to help solve some problem. The human's primary role is to generate and modify solutions, while the computer's role is to analyze those solutions, producing a critique for the human to apply during the next iteration of this process.

Critics do not directly solve users' problems, but they recognize deficiencies in a product and communicate those deficiencies to the users. Critics point out errors and suboptimal conditions that might otherwise remain undetected; frequently they suggest how to improve the product. Users apply this information to fix the problems, seek additional advice or trigger requests for explanations.

It is probably instructive to clarify the distinction between critics and constraints. A significant aspect of critiquing is that users remains in control and are free to accept or reject advice from the critic. Constraints are often "hard coded" into the working environment of systems or enforced on the user by some system process (e.g., a file name extension in MS/DOS cannot be more than 3 characters); they are narrowly focused criteria that must be adhered to in order for something to function properly. Critiquing primarily focuses on improving the functionality of a product that is already usable. It is possible to incorporate hard constraints into the critiquing agent and have the system inform users when they trigger that the product must be changed to comply with the constraint, or that the system already modified it to comply. The majority of the research on critiquing has used critic expertise that is based on what might be called soft constraints or design guidelines.

Advisors [Carroll, McKendree 87] perform a similar function, but they are the source of the primary solution. Users describe their problem, and the computer advisor proposes a solution. In contrast to critics, advisors do not require users to generate either partial or complete solutions to the problem. Advising as an interaction approach is best suited to situations where one-time advice is needed. User models are not as significant in these one-shot affairs, and ones that are used emphasize modelling users' goals rather than their domain expertise. An important research issue is to determine the commonalities that exists between user models in advisory systems and user models for critics.

To clarify any conflicts in terminology, note that the term "critic" was also used in the work on planning systems. In that context they describe internal demons that check for consistency during plan generation. For example, critics in the HACKER system [Sussman 75] discover errors in blocks-world programs.

When the critics discover a problem, they notify the planner, which modifies the plan accordingly. The NOAH system [Sacerdoti 75] contains critics that recognize planning problems and help to modify general plans into more specific ones. Critics in planners interact with the internal components of the planning system; critics in the sense of this paper interact with and critique the work of human users.

## 2.3. The Critiquing Process

The canonical process underlying critiquing is comprised of the sub-processes shown in Figure 2-2. Not all of these processes are present in every critiquing system; in fact, several of these processes are only conceptual and represent emerging research directions.

**Goal Acquisition.** Critiquing a product requires that the system either infer some limited understanding of the product's intended purpose or be designed to support standard user goals. Problem knowledge can be either domain knowledge or goal knowledge. If a critic just has domain knowledge without understanding the user's goals, it can only reason about characteristics that pertain, in a general sense, to all products in that domain. Such is the case for LISP-CRITIC; it analyzes programs for syntactic correctness. For a more extensive evaluation of a product, an understanding of the user's specific goals and situation is desirable. A critic may be able to acquire an understanding of the user's goal in several ways:

- Standard goals are built into the system, in LISP-CRITIC these goals are to produce code that is either more readable or more efficient.
- Goals can be recognized by observing users work and the evolving products. Findings from research on plan recognition in artificial intelligence [Schmidt, Sridharan, Goodson 78] would support this method.

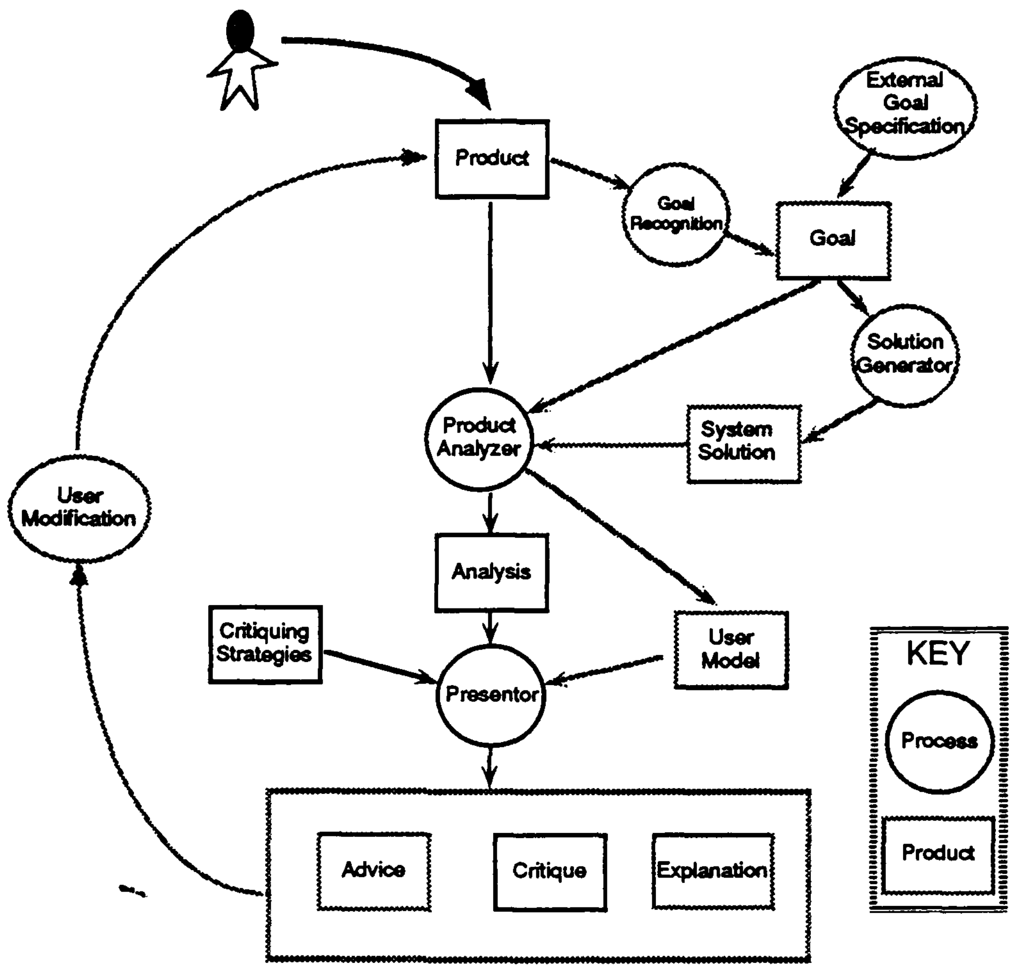


Figure 2-2: The Critiquing Process

Users initiate the critiquing process by presenting a product to the critic. To evaluate the product the critic use a goal specification if one is available. To help analyze the product some critics generate a solution and compare it to the user's, others analyze the user's work directly. A presenter formulates a critique using the product analysis; it provides advice and explanations. Critiquing strategies and a user modelling may be used to aid the presenter. From this output, the user modifies the product and the cycle can repeat. The essential processes and components for a system to be considered a critic are outlined in black. The objects in the figure with grey outlines are optional and in several cases represent research directions.

- A critic may have access to an explicit representation of the problem to be solved, one that encapsulates a particular goal. A simple technique is to limit the possible goals and ask to users to select one from that set.

**Product Analysis.** The two general approaches to critiquing are : *differential* and *analytical*. In differential critiquing, the system generates a solution and compares it to the user's solution. Analytical critiquing checks the product with respect to predefined features and effects. They identify suboptimal features using techniques such as pattern matching [Fischer 87b], finite state machines [Fischer, Lemke, Schwab 85], and expectation-based parsers [Finin 83]. Critics which use the analytical approaches do not require a complete understanding of the product.

**Critiquing Strategies.** Critiquing strategies and the user model can aid the presentation component. The critiquing strategies determine what aspects of a design to critique, and when and how to interrupt users' work. Strategies differ depending on whether the predominant use for the system is helping users to solve problems or for an educational application.

The manner in which critics are integrated into a work environment should be chosen so that users welcome them and find them cooperative. Like recommendations from colleagues or co-workers, messages from a critic can be perceived as helpful or hindering, and as aiding or interfering with the accomplishment of their goals. When selecting a critiquing strategy two factors to consider are intrusiveness and emotional impact on the user.

- Intrusiveness is users' perception of how much the critiquing is interfering with their work. Critics have to trade-off interfering too much with failing to provide sufficient help. Factors to consider include how frequently feedback occurs, the complexity of the tasks, and the sophistication of the user. Critics should intervene when it is critical, but interventions should not occur so frequently that users are bothered and become frustrated.
- Emotional impact refers to how users react toward the computer as an intelligent assistant. Computer critiquing may be more tolerable than critiquing from humans because it can is handled privately between users and the system. When dealing with a machine, users do need not to face the negative aspects of shortcomings in their work being exposed to other people who might form a negative opinion.

The prime objective of *educational critics* is to support learning; and for *performance critics*, to improve the product. Each type of system has different requirements for selecting appropriate strategies. A performance critic should help users create high-quality products in the least amount of time while conserving resources. Learning, although not the primary concern of performance systems, occurs as a by-product of the user and critic interaction. Educational critics try to maximize the information or skills that users acquire and retain for future use. Most performance critics evaluate the product as a whole and determine if it can be changed to achieve a higher quality result. Some critics selectively critique based on a policy specified by the user. Educational critics need more complex intervention strategies to maximize information retention and users' motivation. For example, an educational critic may forego the opportunity to critique if it occurs too soon after a previous critiquing episode. Continuous critiquing without giving users a chance to explore their own ideas can become intrusive and impact motivation.

Existing critics operate primarily in the *negative* mode by pointing out suboptimal aspects of the user's product or solution. A *positive* critic should recognize and point what is good about a user's solution. For performance critics, a positive approach can help users recognize the good aspects of their work. For educational critics, positive critiquing can reinforce desired behavior and thereby aid learning.

Intervention strategies determine when and how a critic intercedes. *Active critics* control intervention because they can critique a product or action at a time of their choosing. They are active agents continuously monitoring user actions. *Passive critics* are explicitly invoked whenever users want an evaluation. Most passive critics are able to evaluate partial products but not individual user actions.

**Adaptation Capability.** To avoid repeating the same type of advice and to accommodate different users with different preferences and skills, a critiquing system needs an adaptation capability. A critic that persistently critiques users using positions with which they disagree is unacceptable, especially when the critique is intrusive. A critic that constantly repeats an explanation that the user already knows is, similarly, unacceptable.

There are two aspects to an adaptation capability: critics can be adaptable or adaptive. Systems are adaptable if a user can change their behavior or knowledge: more recent research has called these systems ''end user modifiable'' [Fischer, Girgensohn 90]. On the other hand, an adaptive system is one that automatically changes its behavior based on observed or inferred information. An adaptation capability can be implemented by disabling or enabling the firing of particular critic rules, by allowing the user to modify or add rules, or by making the critiquing strategy depend on an explicit, dynamic individual user model.

**Explanation.** Explanations are desirable and necessary in most knowledge-based systems [Swartout 81; Teach, Shortliffe 84]. Critics need to be able to explain their rationale so users can assess the critique and decide how to deal with the advice. Knowing why a product was critiqued helps users to learn underlying principles and avoid similar problems in the future. In a critiquing system, explanations can focus on the specific differences between the system's and the user's solutions, the rationale underlying the critique, or on violations of general guidelines.

**Advisory Capability.** Critics detect suboptimal aspects of a user's work; this is the triggering condition for a critiquing episode. When an episode stops here, the user is required to generate and implement any changes to the current product. One improvement on the process is for the critic to suggest alternatives; these we call *solution-generating* critics. Another is to provide the critic the ability to explain or direct users toward information that increases their understanding. User models play a role in facilitating this part of extending the paradigm; it is a subject that will be discussed in detail in the remainder of this thesis.

## 2.4. Survey of Critiquing Systems

This section provides an overview of critiquing systems that play an important role in the development of the paradigm or illustrate an interesting aspect of the theory. In addition to LISP-CRITIC, the Human Computer Communications

Group has developed JANUS [Fischer, McCall, Morch 89a] and FRAMER [Lemke 90] to deepen our understanding of the critiquing paradigm. LISP-CRITIC will be discussed in detail in Chapter 3. Not all systems developed by other researchers are described by their authors using the terminology presented here, but they do fit into the critiquing framework. Because the range of systems covers diverse application domains, a claim can be made that critiquing has general application as a central approach to building knowledge-based systems. During the process of developing the user modelling framework an attempt was made to retain this perspective of domain generality.

Critiquing is attractive because of its generality across a wide range of domains, such as medicine; electronic circuit design; and support for education, writing, programming, and text editing. This section briefly surveys the critiquing systems in these domains that were studied. Most of the systems discussed here were developed as research vehicles, but a few are successful commercial applications.

The WEST system pioneered many of the fundamental ideas behind the critiquing paradigm. It was an early effort to build a computer coach [Burton, Brown 82] that teaches arithmetic skill in a gaming environment (a game called "How the West was won"). The goal was to augment an informal learning activity with a computer coach, retaining the engagement and excitement of a student directed activity while providing context-sensitive advice on how students can improve.

Several important ideas were pioneered in WEST. It builds a bridge between open learning environments and tutoring in order to support what is called guided discovery learning. A model of each user prevents the coach from being too intrusive. The system uses diagnostic modeling strategies to infer problems from students' actions. WEST determines the causes of suboptimal behavior by comparing the solution of a built-in expert with the student's solution. In this manner, the student model is acquired by a process called differential student modelling. The system infers models of students in terms of the "issues" on which they are weak (mathematical procedures and game playing strategies). Intervention and tutoring strategies are explicitly represented in the system and make use of information contained in the model to enable the coach "to say the right thing at the right time" and provide coherence to that feedback.

**Medical applications.** Several researchers in the domain of medicine have embraced the critiquing approach. In general, these systems aid medical personnel in patient diagnosis and treatment. Clancey first proposed a critiquing approach to user-system interaction for expert medical consultation systems [Clancey 84]. Miller and colleagues at Yale Medical School did the most implementation work in this area, developing systems which assist medical personnel by analyzing plans for the prescription of medication, managing its administration, monitoring the use of a ventilator, and administration of anesthetics [Miller 86].

The most extensively developed system is ATTENDING [Miller 86]. It uses the differential critiquing approach, parsing the physician's plan starting with the top-level decisions and at each step trying to find alternatives that have lower or equal patient risks. The system works from the physician's solution to a system's solution to insure that it is as close to the physician's as possible, this makes the critique more helpful and easier to understand.

Differential critiquing is also used in one version of ONCOCIN, an expert system for cancer therapy [Langlotz, Shortliffe 83]. The developers' goal was to eliminate the need to override the system when justifying minor deviations from the therapy plan for the convenience of the patient.

ROUNDSMAN [Rennels 87; Rennels, Shortiliffe, Stockdale, Miller 89] is a critic in the domain of breast cancer treatment that bases its critique on studies from the medical literature. It is a passive critic with explicit goal specification. Text in the literature database serves as a domain knowledge set that is not interpretable by the system, but stored in a "canned" form; associated with each case description in the database are a set of case-factors that can be used for retrieval. ROUNDSMAN can automatically provide the case descriptions as a form of detailed explanation. Redundancy is a problem and no facilities are available for users to followup on the advice or textual descriptions found in the literature. The system is successful because there is a close mapping between the current case characteristics (e.g., tumor size, location, (patient age, etc) and recorded medical case studies. It could be viewed as critiquing system that uses case-based reasoning, except that the system does not really attempt to understand the cases, rather it knows how to match the symptoms of the patient undergoing diagnosis with those cases.

Circuit design. Several research and commercial systems use a critiquing approach for enhancing digital circuit designs. CRITTER [Kelly 85] is a design aid for digital circuits. It uses a schematic diagram and a set of specifications to evaluate the circuit using analysis techniques and knowledge about primitive components. The evaluation report includes information about how well the circuit will work.

A commercial system developed at NCR is the Design Advisor$^{TM}$ [Steele 88]. It is an expert system that provides advice on application-specific integrated circuit designs. The Design Advisor analyzes the performance, testability, manufacturability, and overall quality of CMOS semi-custom VLSI designs. Its knowledge is a hierarchy of design attributes compiled from a study of major problems in commercial VLSI designs. Critiquing is not interactive but done using a batch mode; designers submit proposed circuits and the system returns the analysis to them for any actual design modifications.

**Discovery learning.** A suite of three computer-based coaching systems for discovery learning, developed at LRDC, University of Pittsburgh, are based on critics. These systems each address a different domain: SMITHTOWN — microeconomics [Raghaven, Schultz, Glaser, Schauble 90], VOLTAVILLE — direct current electricity [Glaser, Raghaven, Schauble 88], and REFRACT — geometrical optics [Riemann, Raghaven, Glaser 88]. These discovery environments are designed to build scientific inquiry skills. Active critics judge the efficiency of the processes used to build scientific theory and inform users about errors that characteristically trap less successful students as well as guide them to effective strategies.

**Decision making.** The DecisionLab system developed at the European Computer Industry Research Center [Schiff, Kandler 88] applies the critiquing approach to guide users in managerial decision-making. DecisionLab provides constructive feedback on a user's management plan in a simulation game. The user gets critiqued whenever they attempt a non-optimal approach. This system integrates a critic and a simulation exercise.

Mili is investigating how to apply the critiquing approach to improve the performance of decision makers in the context of their actual work with a system called DECAD. It has not actually been built, but is designed to watch over the shoulder of the decision maker, interjecting advice or a critique when appropriate [Mili 88]. In the area of research into decision support systems, investigators place critiquing into a class of knowledge-based systems called "active and symbiotic decision support systems" [Mili, Manheim 88].

An operational symbiotic decision support system to support steel mill operations is being developed by Manheim and colleagues [Manheim, Srivastava, Vlahos, Hsu, Jones 90]. A manager develops a plan using a commercially available production planning and scheduling system which includes a mathematical model, heuristic, and optimization techniques, that plan is compared to a system developed plan using differential critiquing.

**Curriculum development.** The Alberta Research Council (Canada) and a company called Computer Based Training Systems developed and are marketing a knowledge-based system which provides assistance with curriculum and course development [Wipond, Jones 88]. An expert module monitors curriculum and course development, intervening when necessary or when assistance is requested. The expert monitor can suggest what to do next, where to find examples or how to get more help.

**Authoring.** Critiquing systems have been developed that help writers make their text more readable or help writers learn more efficient text editing strategies with which to produce that text. WANDAH [Friedman 87] is a system that assists authors in all phases of writing; it is commercially available for personal computers as HBJ Writer$^{TM}$. Text which need not be a completed document can be subjected to one of four sets of reviewing and revising aids that go over the written work; the system provides feedback on structural problems, and recommends revisions.

ACTIVIST is an active help system for a screen-oriented text editor that monitors users' activities. It recognizes sequences of actions that are intended to achieve one of the twenty different goals known to the system; some examples are deleting a word or moving the cursor to the end of the current line. ACTIVIST critiques the user after three suboptimal executions of a task type. After a certain number of correct executions, the system will no longer watch for that plan. It ceases to critique actions when a user ignores its suggestions for those actions. This system integrates a user model; that model plays a central role in informing the system when to intervene, when to discontinue looking for a plan, or when to ignore user actions. The user model represents plans or strategies that users may be following, ones can they execute optimally, and others they prefer not to change.

Software development. PROLOG EXPLAINING [Coombs, Alty 84] is designed to enhance a programmer's understanding of PROLOG, thereby helping the user to develop a better understanding of the language. Users are shown some PROLOG code and asked to construct an explanation of that code; the system critiques that explanation.

The GRACE system developed at the NYNEX Artificial Intelligence Laboratory is a multi-faceted learning environment for COBOL programming that integrates a critic, a tutor, and a hypertext information base. When the system is functioning as a critic, it can adopt a tutoring mode to give remedial problems; and conversely, when functioning as a tutor the student can decide to explore in the critiquing mode. The tutor is a production rule-based system modelled after the LISP TUTOR [Anderson, Reiser 85]. The tutor portion of the system contains a student model that is an overlay of the productions contained in the system. That model is not shared with the critic, nor does the critic attempt to tailor its interaction to the individual.

KATE [Fickas, Nagarajan 88] critiques software specifications (for automated library systems) that are represented in an extended Petri net notation. Its knowledge is represented as "cases" consisting of: a pattern describing a behavior in a specification, links to one or more goals, simulation scenarios, and canned text descriptions. The critic evaluates the specification with respect to goals or policy values given by the user.

**Mechanical design.** Feedback Mini-Lab [Forbus 84] was built as a follow-on to the original work on the STEAMER system. It is an environment in which simulated devices, such as steam plant controllers, can be assembled and operated. Students can assemble a device from the building blocks. Feedback Mini-Lab is designed to facilitate student understanding of control components. Mini-lab generates code specifications to produce the simulation for the device. After constructing their device, students can ask the system for a critique.

## 2.5. Limitations of Current Critics and Future Research Issues.

One features that is a strength of the critiquing approach is also a potential weakness. Supporting users in their own doing means that detailed assumptions about what a user might do cannot be built into the system. Our systems have a limited understanding of users' goals. This restricts the amount of assistance and goal-oriented analysis that critics can provide in comparison to systems such as PROUST [Johnson, Soloway 84], which have a deep understanding of a limited set of problems.

Most rule-based critics do not have an explicit representation of all the rationale for their knowledge. Therefore, to capture enough domain knowledge to provide explanations, these systems need more abstract representations of the application domain.

Critics should ideally have inspectable knowledge structures so that users can modify and augment them. This does not mean that users will have to possess detailed programming knowledge. As a minimum users should be able to deactivate (and reactivate) individual rules according to their needs and goals. With sufficient inference and user modeling capabilities, systems might be able to do dynamic adaptation.

Currently, most critics support only a "one-shot dialog" [Aaronson, Carroll 87]. They respond to actions taken by the user; in some cases they give suggestions and explanations but none have the ability to adapt those explanations to an individual user. Human critiquing is a more cooperative activity, during which an increased understanding of the problem develops. Research on how to incorporate more of the characteristics of human-to-human collaborative effort into these systems is needed. This happens to be one of three directions for research suggested for overcoming the limitations of human-machine interaction that were suggested in the analysis of situated action [Suchman 87].

## 2.6. Summary

Critiquing can be used as an approach to designing knowledge-based computer systems that support human work and learning. Critics are important steps towards the creation of more useful as well as more usable computer systems. Some of these systems will have elaborate problem understanding; more commonly, they will have limited yet helpful capabilities; such as modelling their individual users. Research on user modelling in other paradigms, such as tutoring and advisory systems, can establish ideas and techniques that might be of use in critics. A review of that user modelling research and theory will be the subject of the next chapter.

# CHAPTER III

# USER MODELLING

This chapter examines related research in user modelling and describes a general framework for the user modelling component of a system. Two related research areas have attempted to integrate idiosyncratic models of users. Research in Intelligent Computer Aided Instruction (ICAI) systems, most often referred to as Intelligent Tutoring Systems (or ITS), use models of their students to guide the instructional interaction [VanLehn 88]. Artificial intelligence techniques are the basis for modelling users of advice giving dialog systems [Kobsa, Wahlster 89]. In the work on user models in these areas I found some concepts that provide foundations for a user modelling framework to support cooperative problem solving, and some specific ideas that were adapted for a user modelling in critiquing . Those foundations will be discussed and the conceptual architecture for a user modelling component presented.

In the wider context of human-computer interaction the term ''user model'' is over-used; it has been applied to mean three different models:

1. the conceptual model a user forms of a system (more precisely a *user's model* [Norman 86]),
2. a models that represents the typical users of a system as a class and are used to aid in designing systems, and
3. models of a specific user inferred by the system, such as the ones investigated in this research.

Models in the first sense are conceptual models that provide part of a foundation for understanding the process of human-computer interaction. The second class of models above are psychological models developed by and for the analysis of human behavior when interacting with computers. They play an important role in guiding system development and research in the psychology of human-computer interaction; important examples are the GOMS model [Card, Moran, Newell 83] and cognitive complexity theory [Kieras, Polson 85]. Research in this area also compares these models to one another for given tasks [Moran 81; Young, Barnard, Simon, Whittington 89]. In the future there is the possibility that these two lines of research will converge to the point where psychological models can also serve as a basis for idiosyncratic representations of the individuals using a system, but neither research area has matured to a point where that is presently feasible. The distinctions are clarified here to insure there is no confusion concerning the interest of this research — it is user models in the sense of the third category.

An argument has been put forth that the lack of commercial systems with user modelling is evidence for a failure in the research, perhaps an argument for discontinuing it altogether [Williams 90]. My position is that this view is entirely too pessimistic and that the reasons we do not yet find the technology in general use are predominantly organizational and economic. Specifically, there are four possible explanations. First, the technology is not fully mature and additional research is needed, ergo the argument for pursuing this line of research. Second, the paradigms for the associated systems that use such models are neither completely understood themselves, nor fully developed to the point of being commercially viable — for ICAI that means computer-based instructional methodology, and for advisory dialog systems the ability to adequately generate natural language. Third, the computational environments that run these systems (most AI applications for that matter) are expensive and scaling the techniques to fit them to more common platforms is a significant area for research in itself. Fourth, the techniques are not generally understood by designers and builders of software to support commercial applications, a not uncommon phenomena in area of computer science and the reason we find suboptimal system design approaches in everything from text editors to commercial databases in the marketplace. The complete story is likely some combination of these reasons, and arguments based on personal conjectures of what will be successful backed primarily with observations about current commercial computational systems, should not dissuade us from pursuing additional understanding and new approaches to solving any problems in computer science, to include user modelling.

## 3.1. An Overview of User Modelling Research

One survey of user modelling definitions together with an effort to correlate that research in both human-computer interaction and intelligent tutoring resulted in a useful taxonomy based on who owns the model and its function [Murray 88]. It was still necessary to conduct my own study. There was a need to understand other research at level of their implementation methodologies in order that I could determine how the models work, and then decide if the techniques used have could be used in the user modelling component which we wanted to build. The most widely reported examples of working techniques for user modelling are those developed to support intelligent tutoring and dialog advisory systems. This section describes the analysis of those areas; awareness of that research provides both implementation ideas and has helped to determine the several requirements for a user model able to support cooperative problem solving.

### 3.1.1. Student Models in Intelligent CAI

User Models in intelligent tutoring systems, called student models, have been the subject of ongoing research for about a decade, there is significant literature surveying and discussing that work [Sleeman, Brown 82; Wenger 87; Polson, Richardson 88; Psotka, Massey, Mutter 88b; VanLehn 88]. Student models are derived from knowledge in the system such as rules, concepts, or strategies for learning a skill. The user's knowledge state is represented as a perturbation of that domain model — popular approaches are overlays to represent the portions of the knowledge base that a student knows, and a bug models that represent user misconceptions about the domain. Some ITS student models combine these two techniques into a comprehensive representation. Differential modelling is the term often used for these techniques [Wilkins, Clancey, Buchanan 88]. Several systems which are frequently cited as using successful approaches were studied in detail.

The WEST project was previously discussed in Chapter 2. It pioneered the differential modelling approach [Wenger 87]. Student behavior is modelled in terms of the *issues* they understand and correctly apply. Their behavior is compared to an expert's under the same conditions to determine their mastery of particular issues. The system finds an issue a student does not know, then selects an abstract explanation for that issue from prestored text. There are limitations to this approach, when compared to the conditions under which critiquing systems must function. The domain has a number of properties that are not characteristic of the domains in which critics are needed. The computer expert is able to play an optimal game because there is a best solution, and it can interpret all alternative student actions. In WEST it is possible to identify students' bugs, whereas in other domains one can only speak of "suboptimal" behavior. The set of issues, on which the methodology is based, is closed for the game, *How the West Was Won*, while it is frequently open-ended in other domains. The user's task goal is obvious; it is to win the game while obeying its rules, another simplifying assumption which does not apply to many other domains. The explanation strategy in WEST presumes that the advice given is self-explanatory because it contains a good illustrating example. Two ideas developed in WEST are of use in this research. One is the idea that students' actions in the ongoing dialog with the system contain information that can be used to analyze the state of their knowledge. Another is the notion that knowing this state provides a mechanism for guiding presentation of new knowledge by the computer coach.

The genetic graphs approach was first developed for the WUSOR-II computer coach as a way to overlay domain knowledge with a learner-oriented linkage of rules [Goldstein 82]. The rules are represented as nodes in a graph model. The domain for the WURSOR systems (three versions were developed in all) is an exploration adventure computer game called WUMPUS. In another project, the genetic graph approach was used as a basis for modelling procedural skills in two quite different domains, one mental, subtraction, and the other motor, ballet [Brech, Jones 88]. That research validated the generality of the approach and enhanced general understanding of the paradigm. The nodes in the genetic graph represent domain entities, such as skills, facts, rules, or concepts, all elements of expertise. The links between nodes capture the processes by which a student can learn those domain entities. A system component known as a *psychologist* interrogates the user model to determine what to teach next; it is also the entrusted with maintaining that model. Processes represented in the links, such as generalization or analogy, indicate methods by which students can learn a new piece of knowledge starting from one they have mastered. The system can determines a pedagogical approach because the student model is an overlay of the graph with marked nodes representing skills or knowledge that students possess. The links between the nodes provide paths to the target knowledge; they represent possible strategies for ''teaching'' that knowledge.

Genetic graphs are normative models that define in their link structure the manner in which knowledge in a specific domain can be acquired by a student; this is an inherent limitation. The graphs have to explicitly capture in the representation all possible ways for a user to learn a domain entity, requiring significant up-front analysis. To construct the graph a system designer has to determine the domain entities and, for each one, all methods by which a student could learn one entity when they already know another. This restriction is similar that of to traditional Computer-Aided Instruction which has to algorithmically pre-specify the possible paths through course material. Genetic graphs permit more flexibility in that users can traverse the graph during learning according to an arbitrary, rather than predetermined path, but the path must be one that has been captured and represented in the graph.

Clancey compiled survey of student models in ''AI-based instructional programs'' [Clancey 86] that contained a useful framework for research. He characterizes student mc ls as *qualitative* models in the sense that they predict how the modelled learner will solve selected problems, as opposed to representing the student with numeric measures of achievement. The system runs the model as a simulation of that student to predict and explain behavior. Inconsistencies between the prediction and actual student activities serve as a source of diagnosis to improve the student model: in some cases capturing new knowledge (new to the student model, that is) that the student possesses, in others identifying misconceptions or bugs, and in still others doing both. User models for cooperative problem solving systems will not (and cannot) be predictive because of the complexity of the domains and because the open-ended problem solving situations in which they operate, preclude the system from being able to generate a complete problem solution. If the system is not able to solve any problem in the domain, then it follows that it will not be possible to use such an approach in juxtaposition with a qualitative user model to predict user actions. One aspect of this study that fits with our analysis of what is required for modelling users of cooperative knowledge-based systems is Clancey's finding that existing instructional programs had to be enhanced by second-generation knowledge representation technique. As will be discussed in Chapter 5 a similar requirement for enhancing critiquing systems to more fully support cooperative problem solving and learning precipitated the development of a conceptual model for the domain of LISP.

The idea of student models based on the misconceptions or bugs (also called *mal-rules* in some research) that students holds about the domain was a theme in several ICAI research projects besides WEST [Brown, VanLehn 80; VanLehn 88]. Those results did not play a role in this work because the needs of our systems emphasize representing and using what users know about the domain rather than correcting deficiencies in that knowledge.

It is is significant that ICAI systems are able to solve any problem on which their users (the students) will work. Within their application domain they will restrict students to those problems. This allows them to use more detailed and specific model inferencing techniques than those available to systems serving in more open-ended problem solving situations. The requirement for our systems to have generality means that the techniques that are often used in student modelling are often not robust or general enough to support real world problem solving. The problem-space limitations that are imposed by tutoring systems are what make them effective at teaching within those restrictions, and also what enables them to compile accurate and complete models of students within the limits of their own domain understanding. An example is PROUST, which is able to infer possible programmer plans for solving the single problem it uses for all instructional episodes, computing average rainfall with a PASCAL program. For LISP-CRITIC to achieve a similar capability would require solving the plan recognition problem, a theme of significant research interest in its own right [Schank, Abelson 77; Schmidt, Sridharan, Goodson 78; London, Clancey 82; Carver, Lesser, McCue 84]. Related efforts in goal inferencing is important to dialog advisory systems; the other area where important results in user modelling have been achieved.

### 3.1.2. User Modelling in Computer Advisory Systems

User models for advice giving systems based on natural language dialog have approached the user modelling problem from a perspective of applying artificial intelligence and using linguistics theory. A popular approach is stereotyping; it was first proposed by Rich in the GRUNDY system [Rich 79]. Systems that use stereotypes need other acquisition methods to first provide some specific characteristics about a user. When the system obtains sufficient information about users, it categorizes them as fitting a prestored stereotype, and the stereotype then indirectly provides additional possible characteristics. One techniques, used in the work on GRUNDY, is to explicitly ask users for some of these characteristics. A user-generated description aids the system in selecting an appropriate stereotype.

Finin and Kass extended the stereotyping approach to provide implicit user model acquisition in a user modelling shell based on a hierarchy of prestored stereotypes [Kass, Finin 88a]. Their systems analyzes natural language communication between the user and the system using the implicature rules adapted from Grice's Maxims for cooperative communication [Kass 87a]. An example of such a rule is *If a user says P, the user modelling module can assume the user believes that P, in its entirety, was used in reasoning about the current goal or goals of the interaction.* These rules, in conjunction with the stereotypes, infer a model of the user's goals and beliefs. Chin's work in KNOME, a user modelling component for UNIX CONSULTANT, used a double stereotyping technique, one for grouping domain concepts and the other for classifying a user's expertise. The stereotyping approach is useful for *one-shot* advisory type systems that need some quick approximation of the user in order to quickly generate a piece of advice; it could be used as a way to initializing user models for critiquing systems, if a valid set of stereotypes is available.

Wahlster and Kobsa also use the content of a dialog to acquire a model of the user's beliefs, plans, and goals [Wahlster, Kobsa 88]. Their work attempts to emulate in a computer the mental modelling that occurs during human-to-human communication. Its focus is insuring the system serves the user in a cooperative manner, as opposed to system that might be considered adversarial (e.g., computer game-playing programs,) or that are at best ambivalent to the user (e.g., express-teller machines.) The user models in this research predict how a user will interpret an utterance the system is constructing for presentation. In this regard, the purpose of their models are related to Clancey's qualitative model framework for ICAI student models.

Some general characteristics of this class of systems are quite different than those of cooperative problem solving systems:

- The advice is given in a single episode and there is no notion of continuing dialogs over multiple problems and situations. The underlying assumption is that the system will never see a user again and if it does it will not attempt to recognize that fact or use previous information about them.

- The advice is generally atomic; it solves a given problem (e.g., investing some money, locating an apartment, finding the correct train to reach a destination, etc) with a single optimal recommendation.
- The system is an expert. It generally knows more about the advisory domain than the user; and an implicit assumption is that what the computer advisor recommends is accepted without question as being appropriate and optimal.
- The system is not concerned with supporting users' learning in the application domain. Its goal is to insure the advice is understood with an assumption that once users understand *what is being suggested* they willingly accept recommendation.

Cooperative problem solving requires that systems be prepared to deal with the same user repeatedly, and do so in domains where a complex product is being produced. Furthermore, it will be the case that both parties share responsibility for the result and each have some knowledge to contribute to the solution process. Cooperative problem solving systems will therefore need models that are dynamic, persistent, and idiosyncratic.

An important distinction in purposes for user models is important. User models can help a system to generate the appropriate suggestions (for our systems these are in the form of the critique, for advisory systems a recommended course of action), or in a general sense help explain some facet of the domain. Specifically, for critics and advisory systems, that is an explanation of the rationale behind the suggestion in terms of domain concepts. Ideally the same user model will serve both purposes. Research in advisory system has focused on the first situation — insuring the advice is appropriate to user goals and plans, while the research here focuses on the second purpose — explaining to the user the domain knowledge underlying a given critique.

To summarize, there are several significant differences between user modelling for critiquing and those that support advisory dialog systems. The notion of a product constructed through a collaborative effort between the system and the user is central to most critics. Advisory systems are designed as *all-knowing* experts which, once they infer sufficient information about the user, will select or generate proper advice. The user's role is passive while in critics both the system and the user are active in solving the problem at hand. Advisory systems predominantly exercise control of the human-computer interaction. They are less "system controlled" than intelligent tutoring systems, but overall responsibility for the interaction resides in the system. In critics, the system and the user share responsibility for solving the problem at hand and for guiding the interaction. Like in the student modelling work there are several techniques developed by research in this area that can potentially be integrated into a framework for models that support cooperative problem solving; they include: stereotyping approaches, the distinction between explicit and implicit acquisition techniques, and inference rules that use the content of the human computer dialog to enrich the user model contents. These together with key ideas from the student modelling work guided the articulation of some foundations for the approach followed in this dissertation work.

## 3.2. Foundations for User Models to Support Cooperative Problem Solving

There are three issues that need to be addressed for user modelling in cooperative problem solving system: how to represent the user model, how to acquire it, and how to access it. The first two areas proved to be the most difficult; access of the models is primarily determined by decision about the representation. The acquisition problem, viewed in the ITS literature as a problem of diagnosis, is the most challenging. To synthesize my review of the research literature concerned with user modelling, a topology was used to summarize the work. It categorizes specific ideas and projects into the areas of: the knowledge the user model represents, how it is acquired, and its primary purpose. Appendix A contains a table showing the systems discussed throughout this dissertation. Their characteristics in each category together with their application domains and the purpose of the systems themselves are listed.

In the area of acquisition techniques, I found it useful to categorize them based on the directedness of the inferencing method.

1. **Direct acquisition techniques** are those where a specific piece of information is obtained by *explicitly* questioning users or from *implicit* observations of them. Usually a single characteristic about a user is inferred.
2. **Indirect acquisition techniques** are shortcuts, such as stereotypes or classification schemes; they are always implicit.

In the literature, the more commonly used distinction for acquisition techniques, (described best in [Kass, Finin 87a]), is implicit versus explicit acquisition approaches, the orthogonality of these two categorizations is shown in Table 3-1.

**Table 3-1:** Two Orthogonal Classifications of Acquisition Techniques

| Categorizing User Model Acquisition Techniques | | |
|---|---|---|
| | Direct Techniques | Indirect Techniques |
| Implicit Acquisition | X | X |
| Explicit Acquisition | X | |

The user characteristics represented in the model make a claim about what users can do; what they know; and their goals, plans, prejudices or preferences. To support the first two types of information, the representation must be in terms of domain expertise. Users do not simply know or not know a skill or domain entity, so representing their knowledge using a binary value is inadequate. Research in some student models tackle this problem by attempting to rate the knowledge of each domain entity in the user model with a linear value. A linear coefficient used to represent the degree of proficiency would be ideal, but the difficulty is that to establish the validity of such coefficients requires extensive statistical analysis of the population of users. Prevailing approaches have used ad hoc methods for setting these values. That research usually is oriented on demonstrating how the acquisition process works rather than evaluating the validity of the models themselves. A simple approach is to represent each user according to a classification of domain expertise (e.g, expert, novice, beginner). This is what I call a ''classification method''; it is an approach which can be viewed as analogous, or even derived from, the stereotyping methodology. In this project, an alternative method for the system to categorize how well a user knows some piece of domain knowledge was needed.

## 3.2.1. Classifying the Users' Domain Knowledge

In [Fischer 88a] such a schema for classifying users' knowledge was presented, it is shown graphically in Figure 3-1. This schema provides a basis for the user modelling component developed in this research. It provides a conceptual model for the space of user knowledge in the application domain. In general, the domains in the figure represent the following:

**$D_1$:** The subset of concepts (and their associated commands) that users know and use without any problems.
**$D_2$:** The subset of concepts which they use only occasionally, users do not know details about them and are, possibly, unsure of their effects.
**$D_3$:** The mental models [Norman 82; Fischer 84] of the users, i.e., the set of concepts which they think exist.
**$D_4$:** This region represents the actual set of concepts in of a domain.

A specific interpretation of this model in terms of the domain our user model serves, LISP, will be offered in Chapter 7.

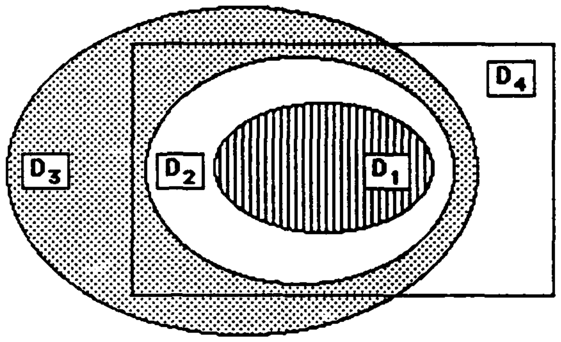


Figure 3-1: Levels of System Usage

Using this schema as a basis for the user model representation means that it is necessary to capture how well a user understands domain entities in accordance with these levels. The level at which users know a domain entity can, in turn, guide explanation giving; this will be shown in Chapter 6.

### 3.2.2. General Approaches to User Modelling

Human-computer interaction includes many different types of systems and interaction approaches. A common theory for how to design and apply idiosyncratic user models across different areas is desirable because it will allow sharing of research results and identification commonly usable features in integrated systems — system that use more than one approach to interaction.

Establishing the requirements for general user modelling can be pursued in two different ways. One strategy is framework-driven: it defines a common architecture that can be used by any system. The General User Modelling Facility (GUMS) [Kass, Finin 88a] provides such a framework. It is a top-down approach because the framework is conceptually predefined and can guide research as well as development efforts for specific systems, domains, or paradigms. Another attempt at developing a domain-independent modelling subsystem is the "User Modeling Front End" (UMFE) [Sleeman 84]. A common idea with this work is the specification of sets of inference rules based on diagnostic information about how user's knowledge propagates through a set of concepts. These generalized modelling approaches attempt to encapsulate a complete theory of user modelling that could be applied to any system.

A more system-driven approach is a bottom-up strategy, studying "successful" user modelling systems in different domains and paradigms, then reusing appropriate techniques and ideas in the user modelling component of a specific system. One example of this is the overlay modelling technique first proposed in WUSOR-II, it has become a standard ITS paradigm [Wenger 87]. Another is the use of bugs to perform student diagnosis and repair in systems such as BUGGY and DEBUGGY [Brown, VanLehn 80], and the Leeds Modelling System (LMS) [Sleeman 83]; then later applied to other domains such as programming [Gray, Corbet, VanLehn 88].

Common features of successful models can be used to drive tneoretical developments in the field; it is a case of the system implementation and testing work driving the development of a general approach or theory. This dissertation has primarily embraced this approach. A methodological first step toward developing a user modelling approach for critiquing systems is to build a system based on both what we understand to be the system's needs and integrating good ideas from other research. Theories need to be tested by developing systems, and system implementations need to be studied to refine the theory. The work here has concentrated first on selecting worthy techniques from other user modelling areas and integrating them into a proposed theoretical framework. That framework was enhanced during implementation of a user modelling component for a computer-based critic. Over the long term, that implementation should drive additional theoretical research to provide a theory of user modelling to support not just cooperative problem solving but a larger class of interactive systems that includes tutoring and advising, among others. The requirements placed upon a user model for systems that support of cooperative human-computer effort are discussed next.

### 3.2.3. Requirements for User Models in Cooperative Problem Solving Systems

Communication is at the heart of any cooperative effort. In order for a human and computer to collaborate effectively they must communicate about the product, perhaps the goal, and general information about the domain in the form of computer-produced explanations. Explanations in the systems we are investigating are currently uni-directional, from the computer to the user. In the future an application of machine learning research might be for users to also explain their knowledge to the computer as a way for the system to learn more about the domain. In either situation, dialogs between the system and user need to operate at a level commonly understood by both agents.

The user model needs to be accessible to other system components. Its contents will be used when presenting explanations, selecting items to analyze, and perhaps as a record of user preferences used to tailor the system. The ultimate objective is an integrated system which adapts to users, allows them to specify preferences, and is still somewhat consistent in the way it treats them.

The user models in cooperative problem solving systems will have to be more individualized than those provided in classification schemes or stereotyping approaches. Users of a complex system are not homogeneous and the system needs to treat each one differently. Having individual models alone is inadequate, their contents have to change as the individuals knowledge improves — users usually become more knowledgeable or proficient over time. A precept of cooperative problem solving is to provide an environment that serves users not once but on a recurring basis; this means the system adapts and changes as users change. Achieving system adaptivity requires a representation of each user that is:

- dynamic — it changes over time,
- persistent — it is retained between problem solving episodes and reused by the system, and
- idiosyncratic — it is unique for each individual.

Based on these requirements, several ideas from the analysis of related research on user and student modelling were identified for incorporation into a user modelling framework for cooperative problem solving systems:

1. Stereotyping (GRUNDY)
2. Explicit and implicit acquisition methods (GUMS)
3. Representing user knowledge as a perturbation of the domain (Genetic Graphs)
4. Using the dialog content as a basis for acquisition inferencing (GUMAC)
5. Acquisition methods based on the relationship between knowledge, the structure of the domain model (UMFE)

The architecture for that user modelling component will be covered next.

## 3.3. A User Model Architecture

A conceptual architecture for the user modelling component that is derived from the previously-discussed requirements was developed. That architecture is designed to serve the needs of the specific system and critiquing paradigm but with an eye toward retaining sufficient generality that it might serve as the user modelling component for any cooperative problem solving system. There are three major subcomponents of the architecture, the representation scheme, acquisition techniques, and access methods; they are shown in Figure 3-2.

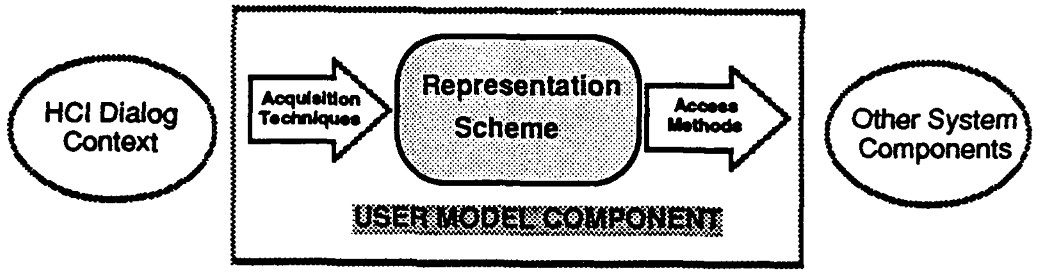


Figure 3-2: General Architecture for A User Modelling Component for CPSS

## 3.3.1. Representation

The representation scheme is central because it must support acquisition and access. It must also be general, efficient, and easy to expand or modify. An additional consideration is to make the schema understandable to a human or, at least, able to be presented by the system in a form that humans can read and modify. We would like for the modelling component to support either users themselves or a teacher in interpreting and editing individual models. User models are at best approximate representations of some cognitive aspects of an individual and we should allow for those situations where that individual or another human can improve on that approximation.

The two ideas from other research in user modelling that contribute to our representation scheme are: the use of a graph model for the domain, such as the genetic graph, and representing the user as an overlay of the domain model. The implementation we developed uses a more general approach than genetic graphs to represent the domain and a coloring of those graphs that is based on the schema for representing user knowledge.

### 3.3.2. Acquisition

Acquiring the user model is the most complex function in this architecture. It requires knowledge on the part of the system, knowledge about ways to infer the state of the user. Representing and accessing a user model could be achieved using common database techniques if acquisition was not such a complex problem.

The acquisition methodology will need to support various approaches for acquiring information about a user. The collection of acquisition techniques in the system can be conceptually viewed as a knowledge-based agent; an agent that is able to infer what a user knows from information provided by other system components which track the human-computer dialog; the agent also knows explicit questions to ask that help infer the model contents, or certain stereotypes, etc. Four categories of possible acquisition approaches were identified:

- *Explicit techniques* directly question the user for information that is entered into the user model. It is a suitable approach for obtaining an initial user model as it can be implemented as a simple up-front questionnaire or testing session when a user accesses a system for the first time. It is not as suitable during subsequent human-system interaction episodes because users are not willing to put up with such administrative requirements more than once. If a model so acquired is not changed to reflect changes to users' knowledge it will become continuously less valid and useful — the approximation of the user's knowledge state becomes progressively less *approximate*.

- *Implicit techniques* enrich the user model without interrupting the user. Two implicit techniques are of interest: stereotyping and the implicit implicature rules that operate on the human-computer dialog. Stereotypes are difficult to apply in many situations mainly because, as discussed earlier, it is hard to determine what stereotypes to use. Organizing those stereotypes into a hierarchy presents its own problems [Kass, Finin 87b]. Some implicature rules, as will be described in Chapter 7, can be modified so that they apply to the human-computer dialog present in most computer working environments rather than natural language situations alone. There are also implicit techniques that are indirect; they use the domain model structure to leverage the information provided by implicature rules.
- *Tutoring* methods acquire information from instructional episodes that can be added to or used to modify user models. These are episodes initiated either by user request or by the system for the express purpose of evaluating a user's knowledge. There are not any systems that attempt to do the latter but this appears to be a natural combination of ideas in tutoring and user model acquisition worthy of investigation. Information in the model that appears to be missing or in conflict triggers a tutoring episode in which the system poses a problem to the user, one designed or selected to evaluate user understanding of the knowledge in question. Given a comprehensive system, such as

GRACE [Dews 89; Atwood et al. 90] that combines both a critic and a tutor, if a user voluntarily requests some tutoring, whatever subjects are addressed during that tutoring can be used by the system in a similar fashion.

- *Statistical* user model acquisition methods could be included in the implicit category but they are of sufficient interest to warrant their own separate category. The acquisition technique is one of observing user actions, accumulating a history of those actions (usually in the form of a count), and triggering inference methods upon reaching predefined threshold levels in that statistical history. The thresholds trigger an inference about the user and precipitate an offer of critiquing type advice to the user. In the ACTIVIST system models based on statistical methods proved to be effective [Fischer, Lemke, Schwab 84]. Unfortunately, this approach has not been explored except in that research, and only conceptually in LISP-CRITIC.

### 3.3.3. Access

The access methods are the third part of the architecture. The model contents must be accessible and usable by other components, or perhaps human agents. Access methods provide information to other system components about what the user does or does not know about the domain. In the model developed in this research that access provides information to guide explanations, but the methods are generic in nature so that they could support the needs of tutoring, advisors and so forth. Access functions need to be general enough to support known requirements, and flexible to accommodate extensions to the system. Access methods are not conceptually or theoretically difficult but are most often determined by the *language* or methodology used to implement the representation.

## 3.4. Summary

A general framework for a user modelling component capable of supporting cooperative problem solving was developed in this chapter; it incorporates techniques from research on student modelling in intelligent computer-aided instruction and user modelling in advisory dialog systems. The initial strategy was to select a technique from one of these areas that could be modified to meet the needs of cooperative knowledge-based systems. However, such a direct application was not feasible because the theoretical analyses showed that there are sufficient differences in the needs of the different types of systems in terms of what they need to know about their users, and in the control of the interaction. Alternatively, an architectural framework for a user model was specified; one that is able to support cooperative human-computer effort, is based on a categorization of users' expertise, and is general in nature. That conceptual framework has been instantiated, in part, in a user modelling component that will be described in Chapter 7. The implementation context is LISP-CRITIC; the next Chapter provides an overview of how LISP-CRITIC has evolved over time, how it is currently configured, and how it presently operates.

# CHAPTER IV

## LISP-CRITIC

LISP-CRITIC was used as the development environment in which the user modelling framework was implemented. It is a knowledge-based system that is designed to support programmers in the context of their work. It does not have "automatic programming" capabilities but operates according to same principle of "intelligent assistance" that is fundamental in the PROGRAMMER'S APPRENTICE work [Rich, Waters 90].[3] In the terms of that research LISP-CRITIC belongs to the class of what are called "transformation system" [Rich, Waters 88].

Comparisons between LISP-CRITIC and the work on LISP Tutor are inevitable. As discussed in the context of learning environments in Chapter 1, the purposes for the two systems are actually quite diverse. LISP Tutor proposes to teach the LISP programming by leading students through a series of predetermined programming exercises known to the system in detail. LISP-CRITIC is oriented toward aiding programmers involved in real work by suggesting to them better ways to implement a specific piece of code they have written. In LISP-CRITIC, like in any critiquing system, learning will inevitably occur, but it would be best to incorporate capabilities into the system to make that learning as effective as possible.

[3]The long term vision for the PROGRAMMER'S APPRENTICE is that it "act as a software engineer's junior partner and critic (emphasis added)" [Rich, Waters 90, p. 1]. In our view, development of LISP-CRITIC provides significant understanding of what is involved in the critic portion of such a system.

LISP-CRITIC provides a suitable context for investigating both user modelling and the cooperative problem solving paradigm for several reasons:

- The rule knowledge base in LISP-CRITIC was previously developed and has been refined through several versions of the system, therefore this research did not have to contend with acquiring and testing the executable knowledge in the system.
- Critiquing is a paradigm that has been studied and is well understood, as discussed in Chapter 2; therefore, we could consider extending it, in the context of LISP-CRITIC, it to integrate user models and support cooperative human-computer work.
- The part of the process involved in giving a programmer advice (the initial critique or suggestion that we will see in the scenario) is stable and usable. This is due, in part, to the maturity of the rule-based knowledge.

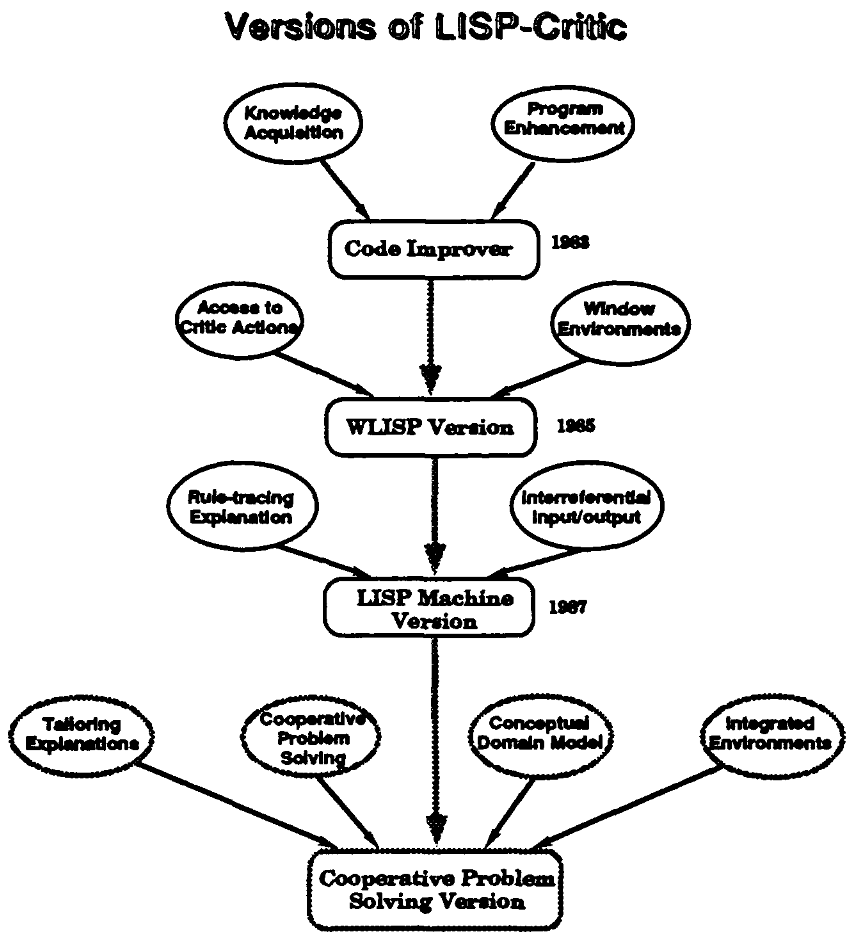
The system was not built from scratch for this project; it has been the focus of iterative development over several years. This chapter reviews the different versions of LISP-CRITIC and some specific research projects to enhance the system that in part motivated this work. It will then describe the current version in terms of its architecture and will use a scenario of a user interacting with the system to demonstrate specific points. Those portions of the current system central to this project, the domain model, explanation giving, and the user modelling component are described in more detail in Chapters 5, 6 and 7, respectively.

## 4.1. Lineage of LISP-CRITIC Versions and Research Issues Addressed

LISP-CRITIC has evolved from a knowledge-based code-enhancement tool to a programming-design environment. In that process, it has benefited from the integration of interactive capabilities, contextual critiquing and explanation capabilities; all help to evolve the system toward one which meets the theoretical notions of being a cooperative problem solving system. The system has existed as four distinct versions (see Figure 4-1,) each one using ideas and parts from the previous version, but improving on them, and integrating new ideas. Its system development history is similar to the series of mutation and selection steps found in genetic evolution. It exemplifies the Simon view of *evolutionary* software development [Simon 81].

Modifications made in producing each new version addressed new research issues; these are indicated with ovals in Figure 4-1. Each version generated an enhanced conceptual model of the system, and contained new or improved parts based on what was learned in developing, using, and evaluating the previous versions. The first three systems will be discussed briefly, followed by a description of the current version. None of the versions discussed here was intended to be a commercial product or even a full prototype for general use, rather they are more in the spirit of what Rich & Waters call "demonstration systems". They were developed as a context in which to investigate theoretical issues, hypothesize solutions, and implement the solutions to *demonstrate* how they work and gain additional insight that was used to refine the concepts.

**CODEIMPROVER.** The precursor to LISP-CRITIC was the CODEIMPROVER system [Boecker 84]. CODEIMPROVER is a knowledge-based program transformation system. Once invoked it operates independently, not allowing further user interaction. Input to CODE-IMPROVER is an executable FRANZLISP program and the output is a version of that program that either better supports human understanding, one that is more cognitively efficient, or a version



The versions of LISP-CRITIC are shown in the center of the above figure. Each version addressed new theoretical issues shown in the ovals.

**Figure 4-1:** Theoretical Issues Addressed in Versions of LISP-CRITIC

that makes better use of computing resources, one that is more machine efficient. The transformations used by the system are captured in a rule base that was developed using traditional knowledge acquisition approaches; these rules were elicited from expert programmers through interviews. An example of the sort of rules contained in that knowledge-base is shown in Figure 4-2. CODE IMPROVER operates in a batch mode in the UNIX operating-system environment, reading programs and then writing an improved version of them into user files.

**Replace a Copying Function with a Destructive Function**

```
(rule append/.1-new.cons.cells-to-nconc/.1...      ;;; the name of the rule
  (?foo:{append append1}                           ;;; the original code
    (restrict ?expr                                ;;; condition
      (cons-cell-generating-expr expr))            ;;; (only apply rule
                                                   ;;; if "?expr" generates
                                                   ;;; cons cells)
    ?b)
  ==>
  ((compute-it:                                    ;;; the replacement
    (cdr (assq (get-binding foo)
               '((append . nconc)
                 (append1 . nconc1)))))
   ?expr ?b)
  safe (machine))                                  ;;; rule category

  (append (explode word) chars)
  ==>
  (nconc (explode word) chars)
```

The rule "*append/.1-new.cons.cell-to-nconc*" replaces the function *APPEND*, which generates a copy of its argument data structure in memory, with the function *NCONC* which instead modifies the internal representation. The latter is preferred when users want to minimize memory use and the new data structure is not needed elsewhere in the program.

Figure 4-2: Example of a Rule in LISP-CRITIC

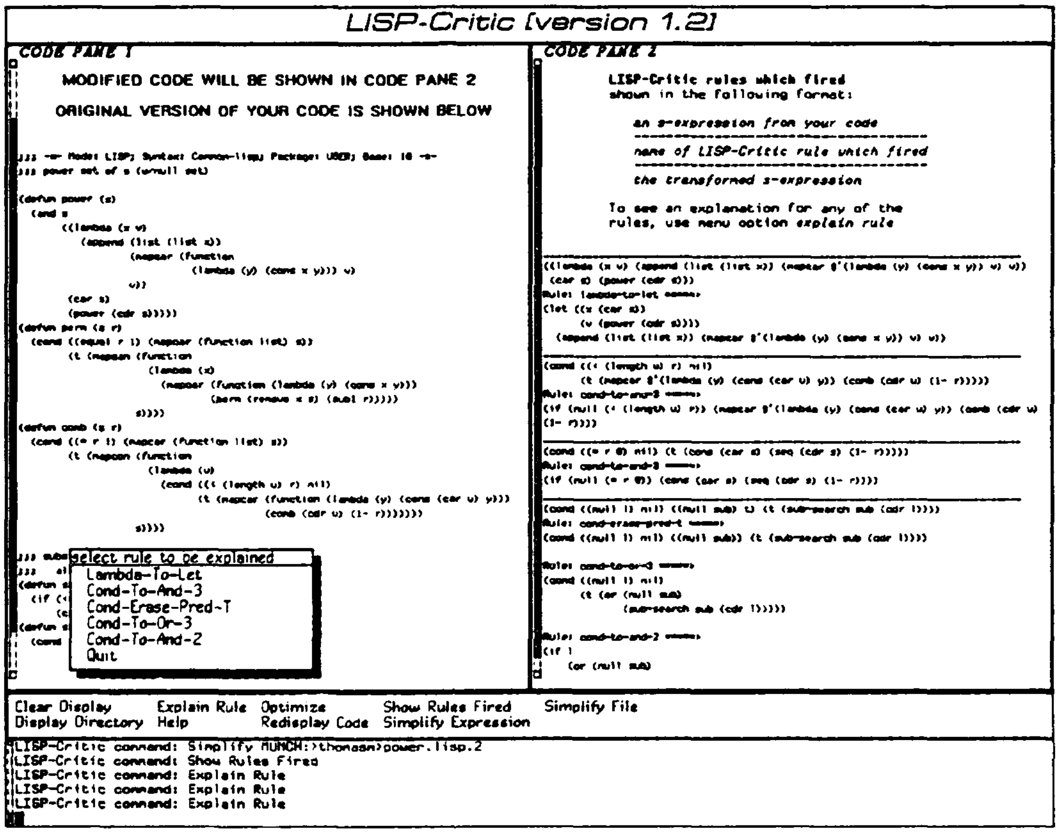
WLISP Version The first version actually called LISP-CRITIC [Fischer 87b] was designed to run in the WLISP windowing environment [Fabian, Lemke 85] on BITGRAPH terminals. It provides some rudimentary explanation capability of the critic's suggestions by showing what rules were fired. Users can choose the kind of suggestions in which they are interested. This version was designed to take advantage of advances in human-computer interaction techniques (such as windowing environments, menus, and the mouse) and to enhance learning.

**LISP Machine Version.** In order to bring LISP-CRITIC closer to supporting LISP programmers in their current working situation, it was integrated into a LISP Machine environment, the Symbolics 3600 Workstation. This version was a direct precursor to the work reported here. Integrating LISP-CRITIC with the other functionalities of the Symbolics Genera environment provided a better understanding of the capabilities and limitations of critiquing. When the system was able to make use of an environment that provides powerful interface capabilities, like those available on the Symbolics, this changed our view of what to expect from the system, and how to configure its architecture. Figure 4-3 shows that second version of LISP-CRITIC running as an activity in the Genera Environment. The knowledge base of LISP-CRITIC was updated to process COMMON LISP but the form of knowledge it contains and the way it applies that knowledge did not change from previous versions.

Several ideas were tested in this version that were designed to make the environment more interactive. Some of capabilities that were provided to users were:

- to view and compare the two versions of the program — their original code and the one generated by LISP-CRITIC (shown in *code pane 1* and *code pane 2*, respectively),
- to request explanation of the differences between the two versions,
- to invoke LISP-CRITIC on source code files in any local or remote directory, and
- to have use of the interreferential input/output features in the Genera environment.

Explanations were provided in the form of rule-traces, like in the MYCIN



```
MODIFIED CODE WILL BE SHOWN IN CODE PANE 2

ORIGINAL VERSION OF YOUR CODE IS SHOWN BELOW

;;; -*- Mode: LISP; Syntax: Common-lisp; Package: USER; Base: 10 -*-
;;; power set of s (w/null set)

(defun power (s)
  (and s
       ((lambda (x v)
          (append (list (list x))
                  (mapcar (function
                            (lambda (y) (cons x y))) v)
                  v))
        (car s)
        (power (cdr s)))))
(defun perm (s r)
  (cond ((equal r 1) (mapcar (function list) s))
        (t (mapcan (function
                     (lambda (x)
                       (mapcar (function (lambda (y) (cons x y)))
                               (perm (remove x s) (sub1 r)))))
                   s))))
(defun comb (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (mapcar (function (lambda (y) (cons (car u) y)))
                                        (comb (cdr u) (1- r)))))))
                   s))))
```

```
LISP-Critic rules which fired
shown in the following format:

   an s-expression from your code
   ------------------------------------
   name of LISP-Critic rule which fired
   ------------------------------------
   the transformed s-expression

To see an explanation for any of the
rules, use menu option explain rule

((lambda (x v) (append (list (list x)) (mapcar #'(lambda (y) (cons x y)) v) v))
 (car s) (power (cdr s)))
Rule: lambda-to-let ====>
(let ((x (car s))
      (v (power (cdr s))))
  (append (list (list x)) (mapcar #'(lambda (y) (cons x y)) v) v))

(cond ((< (length u) r) nil)
      (t (mapcar #'(lambda (y) (cons (car u) y)) (comb (cdr u) (1- r)))))
Rule: cond-to-and-3 ====>
(if (null (< (length u) r)) (mapcar #'(lambda (y) (cons (car u) y)) (comb (cdr u)
(1- r))))

(cond ((= r 0) nil) (t (cons (car s) (seq (cdr s) (1- r)))))
Rule: cond-to-and-3 ====>
(if (null (= r 0)) (cons (car s) (seq (cdr s) (1- r))))

(cond ((null l) nil) ((null sub) t) (t (sub-search sub (cdr l))))
Rule: cond-erase-pred-t ====>
(cond ((null l) nil) ((null sub)) (t (sub-search sub (cdr l))))

Rule: cond-to-or-3 ====>
(cond ((null l) nil)
      (t (or (null sub)
             (sub-search sub (cdr l)))))

Rule: cond-to-and-2 ====>
(if l
    (or (null sub)
```

```
LISP-Critic command: Simplify MUNCH:>thomasm>power.lisp.2
LISP-Critic command: Show Rules Fired
LISP-Critic command: Explain Rule
LISP-Critic command: Explain Rule
LISP-Critic command: Explain Rule
```

This is LISP-CRITIC's screen on the Symbolics 3600. Users can request a critique of a program code file using the menu options or can enter a LISP expression. They receive suggestions on to how to improve that code in CODE PANE 2, their own code is show in *code pane 1*. Rule tracing explanations of LISP-CRITIC'S suggestions are available. In the Figure, the user submitted a program for critiquing, has seen a trace of the rules and is about to select an explanation for one of those rules from a pop-up menu.

Figure 4-3: LISP-CRITIC Interface on the Symbolics Computer

[Buchanan, Shortliffe 84] and subsequent GUIDON [Clancey 87] research. If further clarification is required, the system presents a pre-stored textual description of a rule, a description that is general in nature, not specific to the suggested transformation. This generic explanation approach was one of the shortcomings in this version.

During development of this version, the system was evaluated by two different user groups. Intermediate users want to learn to produce better LISP code; for supporting this purpose, statistical data were gathered concerning the frequency of rules that fired in student programs. Another group of experienced users want to "straighten out" their code. Instead of refining a program by hand (which in principle they are capable of doing), they use LISP-CRITIC to cause them to reflect on the design decisions they made and the code produced to implement them. The critiquing approach is especially useful for improving code that is either under development or frequently modified. In the context of these development efforts, we investigated research issues in human-computer interaction, knowledge-based cooperative systems, and explanation giving.

## 4.2. Previous Research Projects to Enhance LISP-Critic

Two previous research efforts in the context of LISP-CRITIC provided ideas and motivation for some of this work. One effort investigated linking the knowledge contained in the rules with a representation of the user's knowledge using an increasingly-complex-microworld (ICM) model [Fischer 86; Fischer, Lemke, Nieper-Lemke 88]. Another developed on off-line statistical analysis component that analyzes the programmer's code.

Research surrounding the ICM approach developed a rich theoretical model which provides a domain structure to guide users learning LISP. The paradigm accepts the claim from work on learning environments that microworlds are a powerful techniques for achieving computer-based education [Papert 80]. It theorizes that one learns most efficiently when confined to a subset of the overall domain knowledge — a microworld. Once that microworld is mastered a learner can progress to the next more complex one and continue to learn by active exploration, critiquing, access to explanations, and so forth. The problem with this approach lies in defining the microworlds for a given domain. The idea is enticing, and a layered, onion-like model of the domain is conceptually neat. However, further investigations found that perhaps the microworlds were user, and not domain, specific [Fischer, Lemke, Nieper-Lemke 88]. As individuals, our knowledge about any one domain probably conforms better to a model that looks like a head of iceberg lettuce, we each learn a domain according to idiosyncratic microworlds rather than a canonical set of them.

The work in this dissertation first considered user modelling in LISP-CRITIC based on series of microworlds representing the domain. The fundamental difficulty with that approach is developing the underlying microworld structure for the domain — a domain model on which to base the user model. As will be explained, it turned out that a more straight-forward approach was a concept-based domain model.

The idea behind the statistical analyzer [Fischer 87b] is to process programs written by a user before they are transformed by the system. The analyzer collects data on structure and use of program constructs. Such information as average nesting depth for the functions a user defines, or the use of certain types of standard functions (e.g., mapping or loop constructs), could provide evidence about the expertise level of the user. The idea here is intuitively attractive and could be applied to a broad range of applications. What is required for the approach to be useful is a set of inference methods triggered by specific statistical data that can classify a user by expertise level, such as novice, intermediate, or expert, or into a stereotype. To determine these methods requires a significant data-collection and analysis effort on a large population of users, and the correlation of those results with an *a priori* classification that is based on an accepted measurement instrument, like a test or questionnaire. As a technique for acquiring information about users, statistical methods are important and form a category of acquisition methods in the framework presented in Chapter 3. For the time being, the emphasis has been placed on dialog analysis, and the indirect implicit acquisition methods; the statistical approaches were not further investigated.

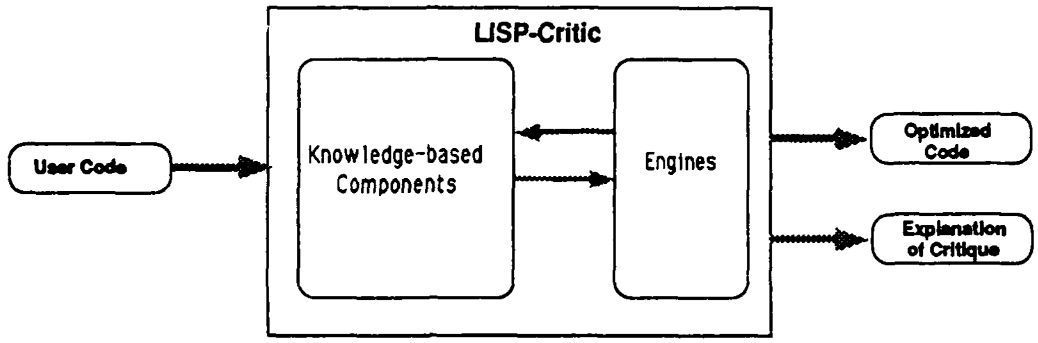
The increasingly complex microworld research indicated that there is a need for a model of the domain, one that can provide a foundation for a user model representation. The statistical analysis work indicated the possibility for evidence-based user model acquisition, that evidence being the contents of users' work. Evidence acquired about the knowledge of an individual programmer can then be used to assign them to a specific expertise classification. The problem here is similar to the difficulty with stereotypes: both ideas have merit as methodologies for acquiring user models, but depend on prior knowledge about the user population, knowledge that is not available without a significant analytical effort. The approaches investigated and implemented in this dissertation are, in some ways, simpler than either of these efforts; we found that a semantic network type conceptual domain model of LISP can support user model representation and some indirect acquisition techniques (see Chapter 7). Next is a description of the architecture of the current LISP-CRITIC which evolved from work on the previous versions. This version incorporates the domain and user models described in this dissertation.

## 4.3. Description of Current Version

The current LISP-CRITIC system allows interaction between the system and the user at the level of individual transformation rather than entire files of code; it is being enhanced to provide context-specific tailored explanations upon request, and to support some adaptability by users. The objective, to investigate user modelling, focuses on support for explanation-giving. Instead of transforming an entire LISP program, handing it back to the user, and trying to explain the differences, the design for this version is based on an assumption that to achieve a more collaborative style, users should be able to decide, on a transformation-by-transformation basis, whether or not they want each portion of code changed. Furthermore the system has to be able to change the code the user actually wants modified, while leaving the rest of the program intact; the resulting program must still compile and execute properly. Users need contextual access to explanations of any single suggestion. These goals led to the development of a version that enhances an existing, commonly used, program development environment, the Symbolics ZMACS editor. Users can access the critic at any time while they are editing LISP code in ZMACS (see Figure 4-7). The critic examines the code and makes one suggestion at a time; the programmer can accept the recommendation, reject it, or request an explanation. When a transformation is accepted the system changes the code in the editing buffer.

A general overview of the system architecture is shown in Figure 4-4. The user's code is analyzed at what is essentially the s-expression level. When an opportunity is found to improve that expression, the systems produces an improved (optimized) version and, when the user requests it, an explanation. Inside of LISP-CRITIC are a set of engines and a set of knowledge-based components that support this process. This architectural diagram does not capture the interaction between the user and the system that takes place; Figure 6-3 shows the interaction between the system and the user at the process level.

Figure 4-5 shows the internal components in greater detail, and the flow of information between them. Work on explanation-giving, as instantiated in the



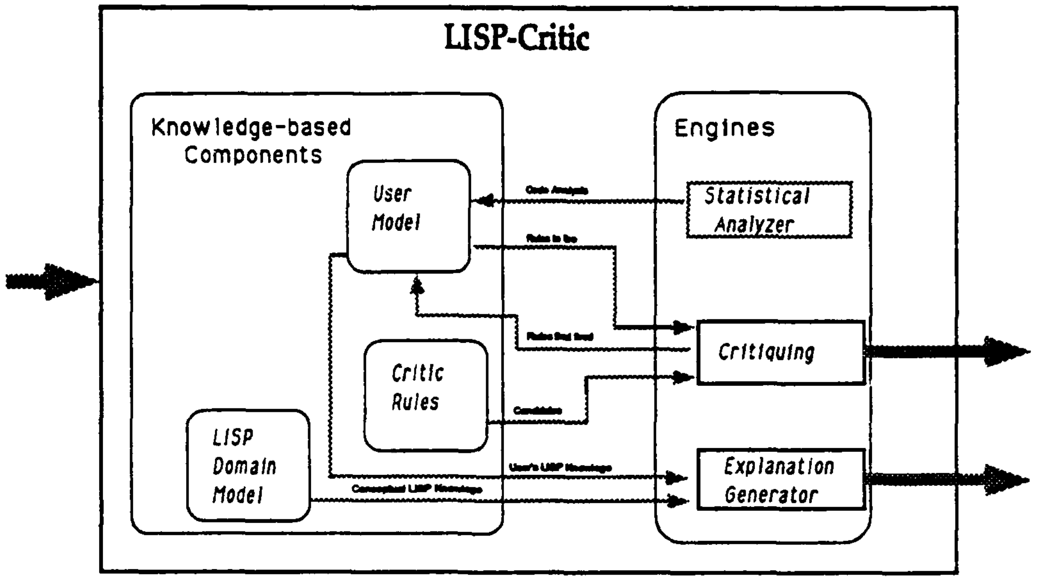
This figure shows the architectural components of LISP-CRITIC and the general flow of data.

Figure 4-4: The Architecture of LISP-CRITIC

explanation generator, is ongoing [Fischer, Mastaglio, Reeves, Rieman 90]. As discussed earlier, the statistical analyzer was developed previously but has not been integrated into the system. The critic rules and critiquing component are derivatives of work on the initial versions of the system, and have been adapted to the Symbolics environment. To provide a better understanding of how LISP-CRITIC operates, and the role played by each system component, an example interaction will be described. Portions of it will be used in other chapters.

## 4.4. Scenario

In this scenario a user is interacting with LISP-CRITIC. The internal actions taken to support the user's decision process and those performed by the user modelling component are not explained in detail here but are covered in Chapter 6 and Chapter 7, respectively. The LISP code in this scenario was written by an undergraduate Computer Science student enrolled in an introductory artificial in-



This figure shows the internal components of LISP-CRITIC and the information flow between them.

Figure 4-5: Internal Components of LISP-CRITIC

telligence course[4] and comes from the corpus of programs used in the evaluation of the the user modelling component described in Chapter 8. It is the program developed for the student's first assignment in LISP. An initial user model (its partial contents can be seen in Figure 7-2) was provided to the system. Theoretically, the contents of this initial model can come from a number of sources:

- Computer-based tutoring
- Explicit acquisition approaches, for instance the use of a questionnaire
- Testing of the user's knowledge level
- From a list of concepts taught during classroom instruction.

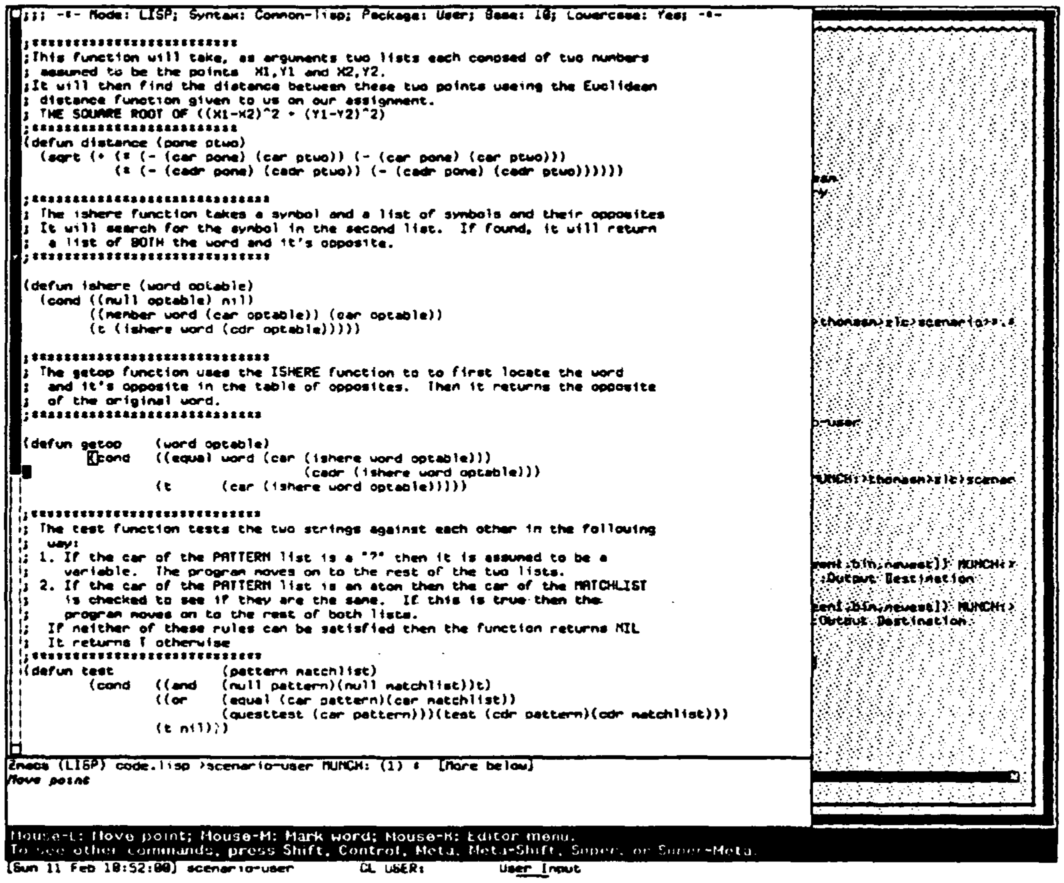
[4]This student, whose identity is not revealed, happens to be male; therefore, in this discussion *he* will be referred to using male pronouns.

LISP-CRITIC was not used previously by this student programmer, therefore the startup user model here is based on responses to a data collection questionnaire completed by him, augmented with information about concepts explained in class. The theoretical investigations of user modelling in this research concentrated on methods for enhancing an existing model using the context of the user-system dialog while assuming the existence of some sort of initial or start-up model of each user. The rationale for this assumption is the existence of several available techniques for providing the initial model (interactive questioning of users, stereotyping, classification categories, etc) that could be adapted for use in LISP-CRITIC. It was felt that rather than attempting to implement the entire range of methods that first build a model "from scratch" and then improve it over time, that the work should concentrate on the more difficult and less well understood problem of how to enhance that model over time (dynamically).

In the scenario the term *dialog* is used to mean the entire context of the human-system interaction. The dialog notion, as applied here, will be discussed more extensively later, for this scenario it should be understood to encompass that series of actions taken by either a user or a system which the other knows about.

### 4.4.1. First Dialog Episode

The initial screen image of the user working on his code in ZMACS is shown in Figure 4-6. He wrote and debugged a program using the ZMACS editor on a Symbolics LISP Machine. From the editor he invokes LISP-CRITIC using a HYPER-S key combination. LISP-CRITIC examines a single function definition (*defun*) at a time. That function definition is identified by the system as the one within which the user has positioned the cursor. For first scenario episode it is the *defun* for *getop*. The figure shows the entire buffer to emphasize that LISP-CRITIC recognizes the user's context (from the cursor), just as a knowledgeable human assistant might; the programmer does not have to scroll the window to a particular configuration or mark a section of the program to identify it to the critic.



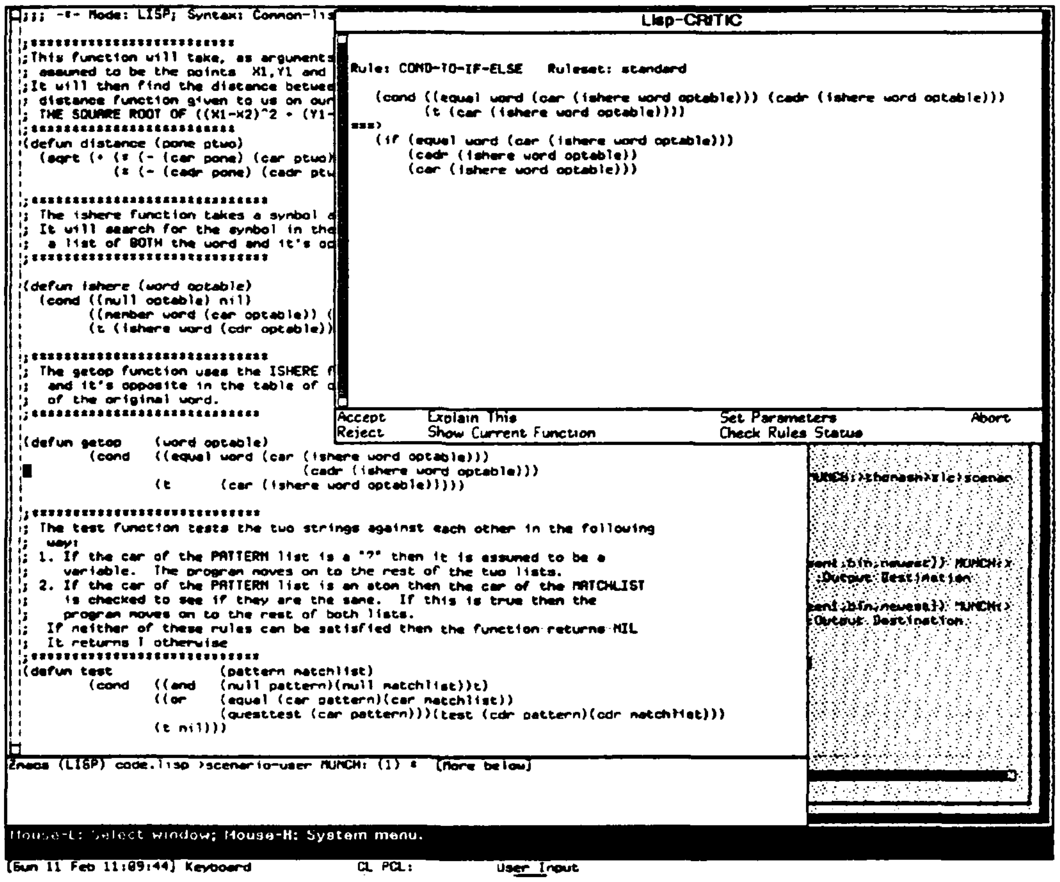
```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: User; Base: 10; Lowercase: Yes; -*-

;*************************
;This function will take, as arguments two lists each composed of two numbers
; assumed to be the points  X1,Y1 and X2,Y2.
;It will then find the distance between these two points useing the Euclidean
; distance function given to us on our assignment.
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-Y2)^2)
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)) (- (car pone) (car ptwo)))
           (* (- (cadr pone) (cadr ptwo)) (- (cadr pone) (cadr ptwo))))))

;*****************************
; The ishere function takes a symbol and a list of symbols and their opposites
; It will search for the symbol in the second list.  If found, it will return
;  a list of BOTH the word and it's opposite.
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (car optable))
        (t (ishere word (cdr optable)))))

;*****************************
; The getop function uses the ISHERE function to to first locate the word
;  and it's opposite in the table of opposites.  Then it returns the opposite
;  of the original word.
;****************************

(defun getop    (word optable)
       (cond    ((equal word (car (ishere word optable)))
                                  (cadr (ishere word optable)))
                (t      (car (ishere word optable)))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test             (pattern matchlist)
       (cond    ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))
```

```
Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More below]
Move point

Mouse-L: Move point; Mouse-M: Mark word; Mouse-R: Editor menu.
To see other commands, press Shift, Control, Meta, Meta-Shift, Super, or Super-Meta.
[Sun 11 Feb 10:52:00] scenario-user        CL USER:        User Input
```

User editing LISP code in the ZMACS buffer.

Figure 4-6: Scenario-User's LISP Program

LISP-CRITIC examines the user's code for possible ways to simplify it. In this case it finds that a *cond* special form could be replaced by an *if* special form and makes that suggestion, as shown in Figure 4-7. The user has the choices in the menu bar at the bottom of the LISP-CRITIC window, of interest here are the options to *accept*, or *reject* LISP-CRITIC's suggestion, or to ask the system to *explain this*. The user does not understand the suggestion, so he selects the *explain this* menu option. The system calls the explanation component which obtains from the domain model the concepts required to understand this rule. Then the explanation component calls the user modelling component to determine which aspects of that knowledge the user lacks.



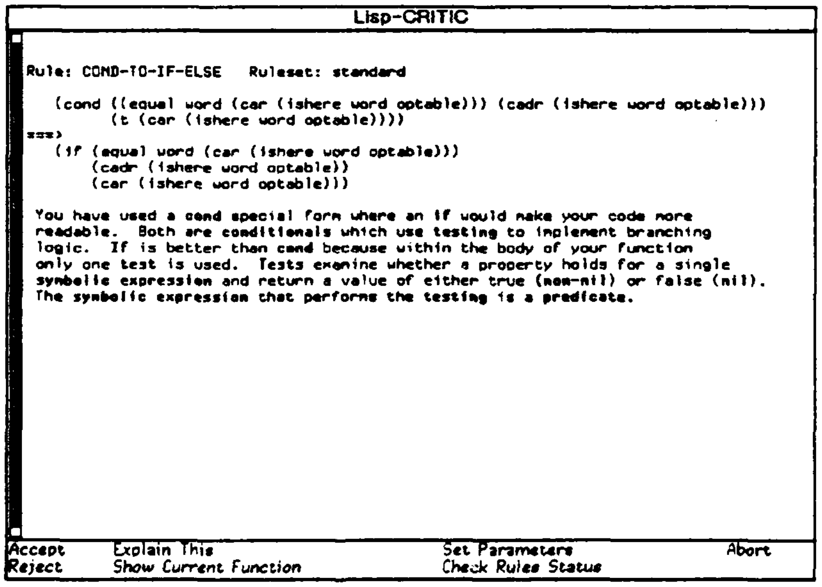
```
;;; -*- Mode: LISP; Syntax: Common-lis
;*************************
;This function will take, as arguments
; assumed to be the points  X1,Y1 and
;It will then find the distance betwee
; distance function given to us on our
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)
           (* (- (cadr pone) (cadr ptw

;*****************************
; The ishere function takes a symbol a
; It will search for the symbol in the
;  a list of BOTH the word and it's op
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (
        (t (ishere word (cdr optable))

;*****************************
; The getop function uses the ISHERE f
;  and it's opposite in the table of o
;  of the original word.
;****************************

(defun getop    (word optable)
        (cond   ((equal word (car (ishere word optable)))
                                (cadr (ishere word optable)))
                (t      (car (ishere word optable)))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test             (pattern matchlist)
        (cond   ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))
Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  (More below)
```

```
Lisp-CRITIC

Rule: COND-TO-IF-ELSE   Ruleset: standard

   (cond ((equal word (car (ishere word optable))) (cadr (ishere word optable)))
         (t (car (ishere word optable))))
===>
   (if (equal word (car (ishere word optable)))
       (cadr (ishere word optable))
       (car (ishere word optable)))
```

LISP-CRITIC is accessed and suggests how to improve the user's code.

Figure 4-7: Scenario-User Invokes LISP-CRITIC on Function *getop*

The domain model begins with the *cond-to-if-else* rule and, using the links between domain model entities, accumulates a *concept set* which consists of all prerequisites to understanding the rule. The user modelling component filters the concept set so the explanation component can focus on explaining only those concepts which users do not know. The final step in the explanation process is presentation of this information. The current implementation does not contain a fully developed presentation strategy so it uses a simple strategy of choosing to explain the first three concepts in the filtered list, in this case *predicates, conditionals, and tests* and displaying hypertext explanations for them. To create a more realistic scenario, mock explanations using these three concepts as a basis are displayed in Figure 4-8. These are more in line with what we would expect a fully competent explanation strategy to produce. In the present system a followup capability is provided for with hypertext. Clicking on any of the terms shown in bold causes an explanation associated with that object in the domain model or a description from the Symbolics Document Examiner's documentation to be displayed. The explanation in Figure 4-8 provides access to explanations of *s-expressions*, *tests*, *cond*, *if*, *nil*, and *non-nil*.

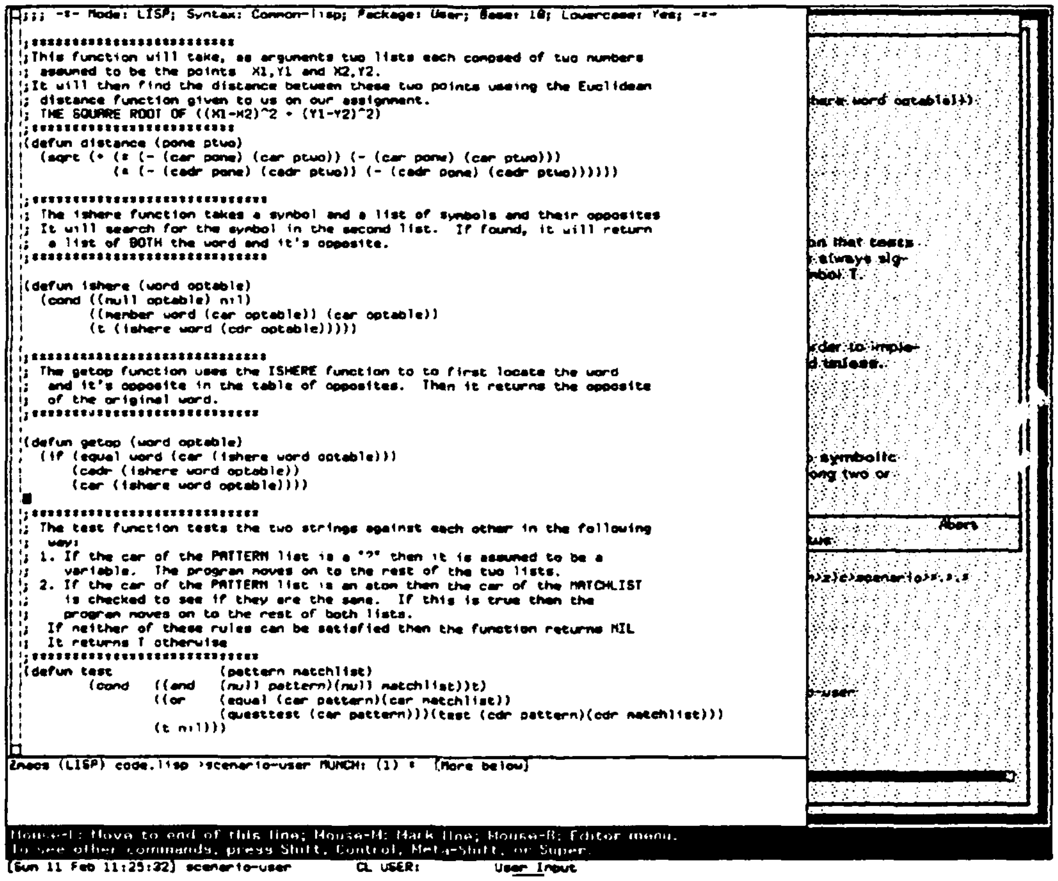


```
Rule: COND-TO-IF-ELSE   Ruleset: standard

   (cond ((equal word (car (ishere word optable))) (cadr (ishere word optable)))
         (t (car (ishere word optable))))
===>
   (if (equal word (car (ishere word optable)))
       (cadr (ishere word optable))
       (car (ishere word optable)))

 You have used a cond special form where an if would make your code more
 readable.  Both are conditionals which use testing to implement branching
 logic.  If is better than cond because within the body of your function
 only one test is used.  Tests examine whether a property holds for a single
 symbolic expression and return a value of either true (non-nil) or false (nil).
 The symbolic expression that performs the testing is a predicate.
```

LISP-CRITIC explains a suggestion based on the *cond-to-if* rule to include those prerequisite concepts the user does not know.

Figure 4-8: Explanation For *cond-to-if-else* Rule

The user accepts this suggestion, and LISP-CRITIC automatically rewrites the modified portion of the user's code in the editing buffer. In Figure 4-9 the body of function *getop* has been changed to reflect the *cond-to-if-else* transformation. In this function definition, LISP-CRITIC found only one transformation to suggest, therefore at this point the programmer is returned to his code editing buffer and LISP-CRITIC's window becomes inactive; it moves into the background, out of the programmer's view.



```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: User; Base: 10; Lowercase: Yes; -*-

;*************************
;This function will take, as arguments two lists each composed of two numbers
; assumed to be the points  X1,Y1 and X2,Y2.
;It will then find the distance between these two points useing the Euclidean
; distance function given to us on our assignment.
; THE SQUARE ROOT OF ((X1-X2)^2 + (Y1-Y2)^2)
;*************************
(defun distance (pone ptwo)
  (sqrt (+ (* (- (car pone) (car ptwo)) (- (car pone) (car ptwo)))
           (* (- (cadr pone) (cadr ptwo)) (- (cadr pone) (cadr ptwo))))))

;*****************************
; The ishere function takes a symbol and a list of symbols and their opposites
; It will search for the symbol in the second list.  If found, it will return
;  a list of BOTH the word and it's opposite.
;*****************************

(defun ishere (word optable)
  (cond ((null optable) nil)
        ((member word (car optable)) (car optable))
        (t (ishere word (cdr optable)))))

;*****************************
; The getop function uses the ISHERE function to to first locate the word
;  and it's opposite in the table of opposites.  Then it returns the opposite
;  of the original word.
;****************************

(defun getop (word optable)
  (if (equal word (car (ishere word optable)))
      (cadr (ishere word optable))
      (car (ishere word optable))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test                (pattern matchlist)
        (cond   ((and   (null pattern)(null matchlist))t)
                ((or    (equal (car pattern)(car matchlist))
                        (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
                (t nil)))

Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More below]

Mouse-L: Move to end of this line; Mouse-M: Mark line; Mouse-R: Editor menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Sun 11 Feb 11:25:32] scenario-user        CL USER:         User Input
```

After the user accepts LISP-CRITIC's suggestions, the system modifies the code in his buffer.

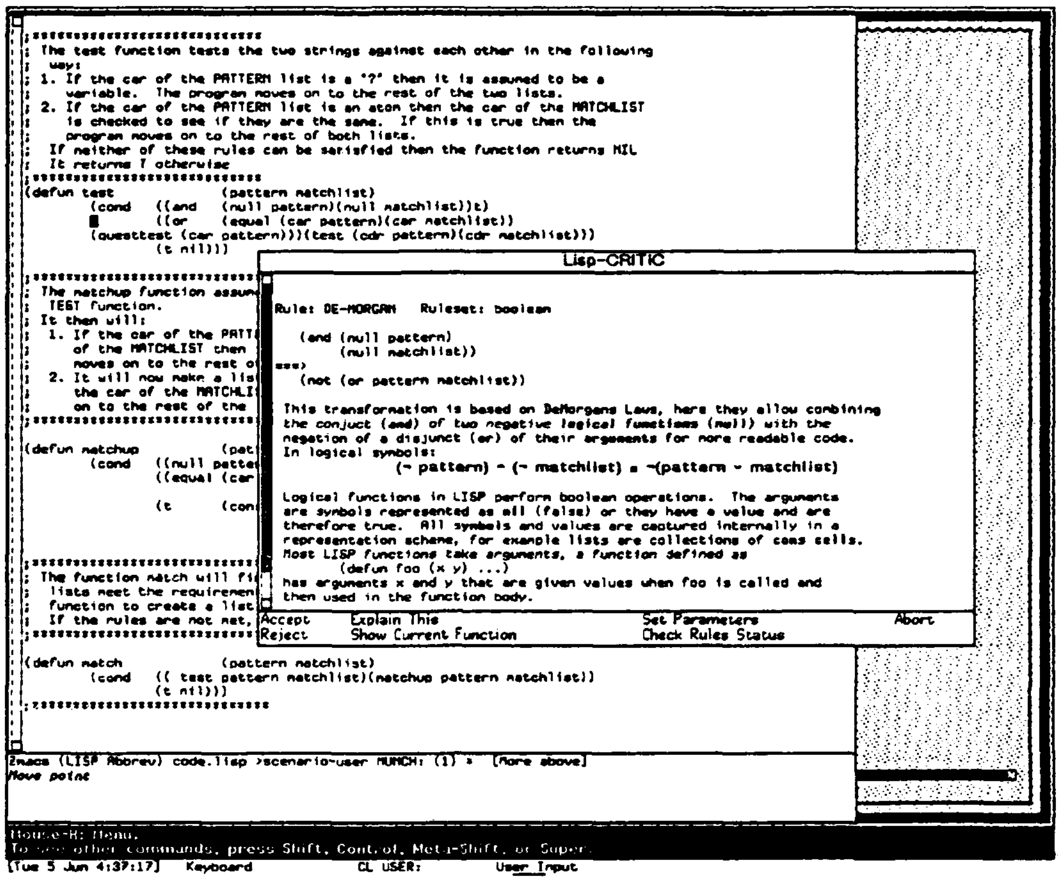
Figure 4-9: Modified ZMACS Buffer

The user's actions throughout this episode trigger changes in his user model. Those specific changes are described in detail together with an explanation of the user modelling process that precipitate them in Chapter 7. In general, any action taken by the user, or information received by him in the form of explanations, trigger those changes. The user's receipt of an explanation of the *cond-to-if* rule and concepts behind it, and the fact that he made a decision to accept the transformation trigger changes to his user model (that updated model is partially displayed in Figure 7-4). These direct changes, in turn, trigger indirect inferences.

### 4.4.2. Second Dialog Episode

The second scenario episode could occur immediately, or at a later time; the user model is saved between sessions and reused when the user subsequently accesses LISP-CRITIC. Our user now requests LISP-CRITIC to look over the code written for function *test* which causes a recommendation based on a rule *de-morgan* (the rule applies DeMorgan's law from logic to combine booleans) and again he requests an explanation. The explanation component once again consults the domain and user models to determine what needs to be explained to this particular user. Those top three concepts selected by the simplified explanation strategy are *logical functions*, *internal representation*, and *arguments*; these are integrated into the complete text of the mock explanation shown in Figure 4-10.

The user accepts the suggestion and the system shows the user a second suggested transformation for this piece of code; it is based on the *cond-erase-pred.t* rule, and the user asks the system to explain it; Figure 4-11 shows that explanation. Our scenario user also accepts this suggestion. A third rule, *cond-erase-t.nil*, triggers; it is very similar to the previous rule therefore, the user accepts LISP-CRITIC's recommendation without explanation. He is able to generate his own explanation because he knows a similar rule that was previously encountered, and is familiar with the underlying concepts. Throughout the dialog,



```
;*****************************
; The test function tests the two strings against each other in the following
;   way:
; 1. If the car of the PATTERN list is a '?' then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;*****************************
(defun test                (pattern matchlist)
       (cond   ((and   (null pattern)(null matchlist))t)
       █        ((or    (equal (car pattern)(car matchlist))
       (questtest (car pattern)))(test (cdr pattern)(cdr matchlist)))
               (t nil)))

;*****************************
; The matchup function assum
;  TEST function.
; It then will:
;  1. If the car of the PATT
;     of the MATCHLIST then
;     moves on to the rest o
;  2. It will now make a lis
;     the car of the MATCHLI
;     on to the rest of the
;*****************************

(defun matchup             (pat
       (cond   ((null patte
               ((equal (car

               (t      (con



;*****************************
; The function match will fi
;  lists meet the requiremen
;  function to create a list
;  If the rules are not met,
;*****************************

(defun match               (pattern matchlist)
       (cond   (( test pattern matchlist)(matchup pattern matchlist))
               (t nil)))
;*****************************
```

```
Lisp-CRITIC

Rule: DE-MORGAN   Ruleset: boolean

   (and (null pattern)
        (null matchlist))
==>
   (not (or pattern matchlist))

 This transformation is based on DeMorgans Laws, here they allow combining
 the conjuct (and) of two negative logical functions (null) with the
 negation of a disjunct (or) of their arguments for more readable code.
 In logical symbols:
              (¬ pattern) ∧ (¬ matchlist) ≡ ¬(pattern ∨ matchlist)

 Logical functions in LISP perform boolean operations.  The arguments
 are symbols represented as nil (false) or they have a value and are
 therefore true.  All symbols and values are captured internally in a
 representation scheme, for example lists are collections of cons cells.
 Most LISP functions take arguments, a function defined as
        (defun foo (x y) ...)
 has arguments x and y that are given values when foo is called and
 then used in the function body.
Accept      Explain This                          Set Parameters          Abort
Reject      Show Current Function                 Check Rules Status
```

```
Zmacs (LISP Abbrev) code.lisp >scenario-user MUNCH: (1) *  [More above]
Move point

Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 5 Jun 4:37:17]   Keyboard          CL USER:          User Input
```

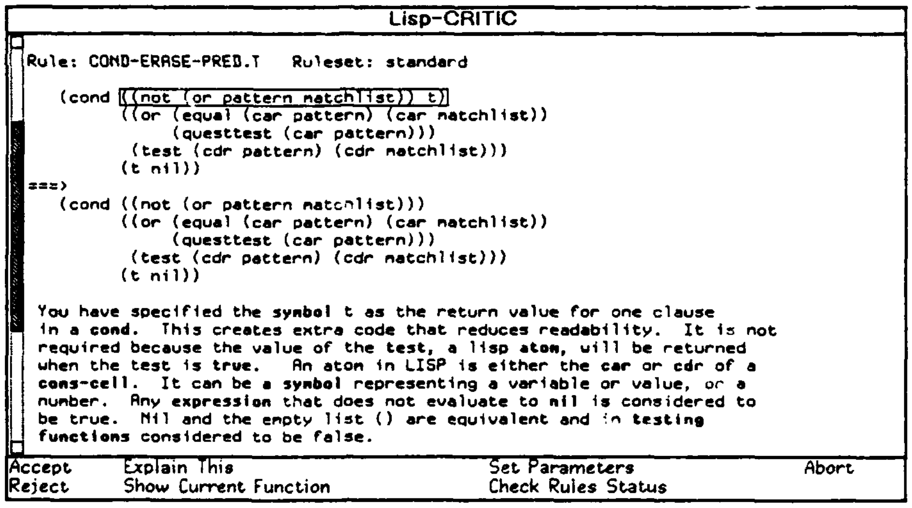
LISP-CRITIC is asked to examine another part of the user's program, a suggested transformation and its explanation is shown.

Figure 4-10: Explanation For *de-morgan* Rule

the user model is updated each time the user receives an explanation or makes a decision.

## 4.4.3. Third Dialog Episode

In a third dialog episode our programmer asks LISP-CRITIC to examine his definition for the function *match*. The systems recommends that an *if* be used in place of a *cond* (see Figure 4-12). This transformation is based on the same *cond-to-if-else* rule that fired in the first episode, and because the user already encountered this same rule and had it explained, he accepts the suggestion without requesting explanation. LISP-CRITIC changes the user's program code. The final



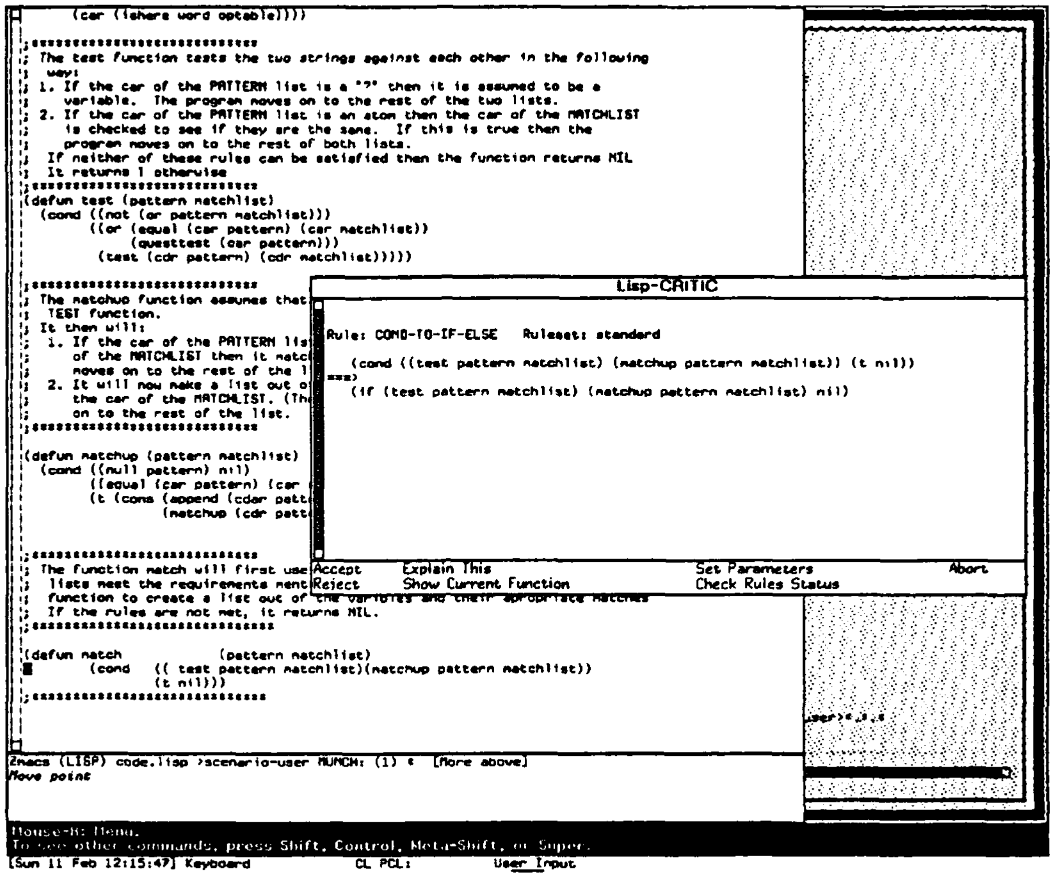
```
Rule: COND-ERASE-PRED.T   Ruleset: standard

   (cond ((not (or pattern matchlist)) t)
         ((or (equal (car pattern) (car matchlist))
              (questtest (car pattern)))
          (test (cdr pattern) (cdr matchlist)))
         (t nil))
===>
   (cond ((not (or pattern matchlist)))
         ((or (equal (car pattern) (car matchlist))
              (questtest (car pattern)))
          (test (cdr pattern) (cdr matchlist)))
         (t nil))

 You have specified the symbol t as the return value for one clause
 in a cond.  This creates extra code that reduces readability.  It is not
 required because the value of the test, a lisp atom, will be returned
 when the test is true.   An atom in LISP is either the car or cdr of a
 cons-cell.  It can be a symbol representing a variable or value, or a
 number.  Any expression that does not evaluate to nil is considered to
 be true.  Nil and the empty list () are equivalent and in testing
 functions considered to be false.
```

LISP-CRITIC recommends a second transformation in the *defun* for test, which the user asks to be explained.

Figure 4-11: Explanation For *cond-erase-pred.t* Rule

state of his editing buffer with is shown in Figure 4-13, the contents of his user model is partially shown in Figure 7-6 and its complete internal representation in Appendix B. That model has changed during the scenario and if another explanation of the *cond-to-if* rule had, in fact, been requested, the user modelling component can provide to the explanation component the fact that the rule was previously explained together with a new concept-set to use for presenting the explanation this time.

The development of the explanation component has not progressed past the conceptual and methodological stages. The explanations shown in this scenario are only intended to point out the relationship between the work on the domain and user modelling undertaken in this dissertation, and the requirements for explanation-giving. As presented, the explanations do not constitute a finished product and should be used primarily as a vehicle to understand how the system makes choices of what to present during its dialogs with the user. Additional work



```
        (car ((ishere word optable))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns 1 otherwise
;****************************
(defun test (pattern matchlist)
  (cond ((not (or pattern matchlist)))
        ((or (equal (car pattern) (car matchlist))
             (questtest (car pattern)))
         (test (cdr pattern) (cdr matchlist)))))

;****************************
; The matchup function assumes that
;  TEST function.
; It then will:
;  1. If the car of the PATTERN lis
;     of the MATCHLIST then it matc
;     moves on to the rest of the l
;  2. It will now make a list out o
;     the car of the MATCHLIST. (Th
;     on to the rest of the list.
;****************************

(defun matchup (pattern matchlist)
  (cond ((null pattern) nil)
        ((equal (car pattern) (car
        (t (cons (append (cdar patt
                 (matchup (cdr patt


;****************************
; The function match will first use
;  lists meet the requirements ment
;  function to create a list out of the varibles and their approprate matches
;  If the rules are not met, it returns NIL.
;*******************************

(defun match            (pattern matchlist)
        (cond   (( test pattern matchlist)(matchup pattern matchlist))
                (t nil)))
;*******************************
```

```
Lisp-CRITIC

Rule: COND-TO-IF-ELSE   Ruleset: standard

   (cond ((test pattern matchlist) (matchup pattern matchlist)) (t nil))
===>
   (if (test pattern matchlist) (matchup pattern matchlist) nil)

Accept    Explain This              Set Parameters       Abort
Reject    Show Current Function     Check Rules Status
```

```
user>*,*,*
Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More above]
Move point

Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Sun 11 Feb 12:15:47] Keyboard          CL PCL:          User Input
```

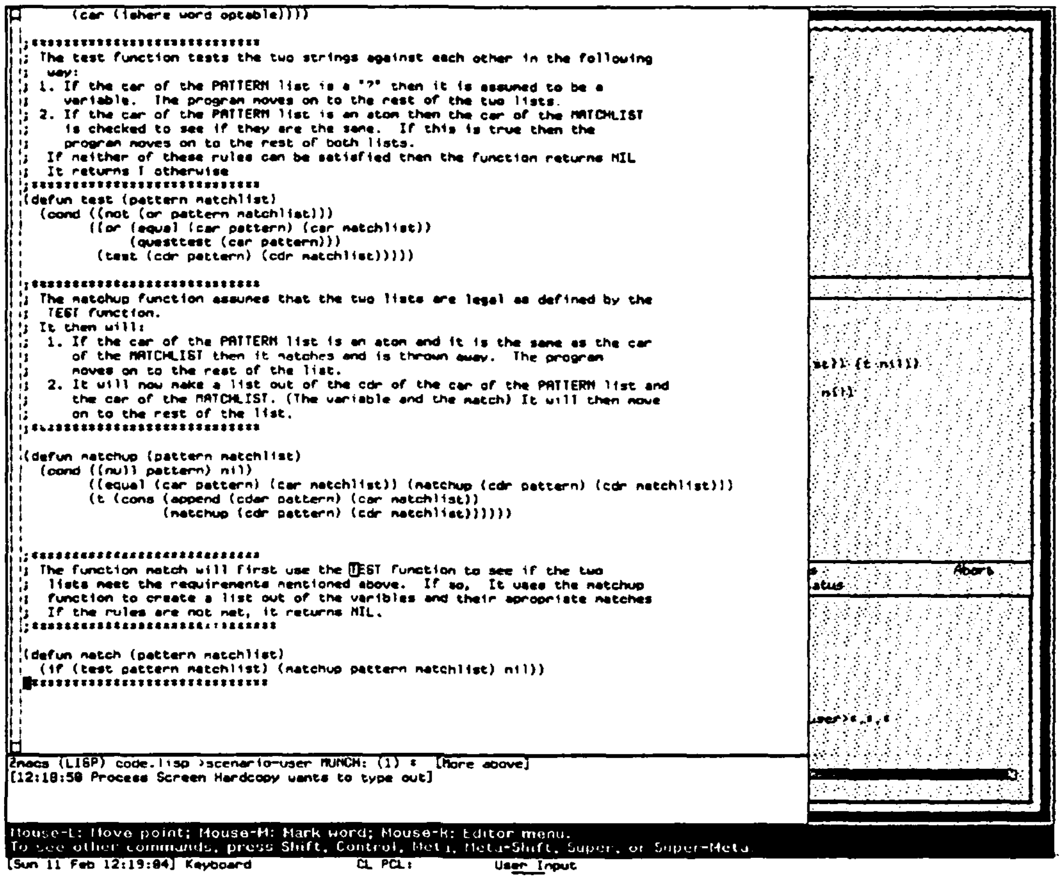
In the third dialog episode LISP-CRITIC examines code for the *defun match* in the user's program.

Figure 4-12: Scenario-User Invokes LISP-CRITIC on *defun match*

to implement the presentation strategies for explanation-giving is required. This is not a trivial problem; it is one of the three major issues in explanation identified in [Chandrasekaran, Tanner, Josephson 89]; the other two being the system's understanding or deep model of the domain and user modelling. Recent LISP-CRITIC work has concentrated on these latter two problems.

## 4.5. Summary

Developing a marketable system is not the ultimate goal in the LISP-CRITIC work, instead a prototyping process was followed; it is designed to help achieve a better understanding, at a conceptual level, of what is required from



```
       (car (ishere word optable))))

;****************************
; The test function tests the two strings against each other in the following
;  way:
; 1. If the car of the PATTERN list is a "?" then it is assumed to be a
;    variable.  The program moves on to the rest of the two lists.
; 2. If the car of the PATTERN list is an atom then the car of the MATCHLIST
;    is checked to see if they are the same.  If this is true then the
;    program moves on to the rest of both lists.
;  If neither of these rules can be satisfied then the function returns NIL
;  It returns T otherwise
;****************************
(defun test (pattern matchlist)
  (cond ((not (or pattern matchlist)))
        ((or (equal (car pattern) (car matchlist))
             (questtest (car pattern)))
         (test (cdr pattern) (cdr matchlist)))))

;****************************
; The matchup function assumes that the two lists are legal as defined by the
;  TEST function.
; It then will:
;  1. If the car of the PATTERN list is an atom and it is the same as the car
;     of the MATCHLIST then it matches and is thrown away.  The program
;     moves on to the rest of the list.
;  2. It will now make a list out of the cdr of the car of the PATTERN list and
;     the car of the MATCHLIST. (The variable and the match) It will then move
;     on to the rest of the list.
;****************************

(defun matchup (pattern matchlist)
  (cond ((null pattern) nil)
        ((equal (car pattern) (car matchlist)) (matchup (cdr pattern) (cdr matchlist)))
        (t (cons (append (cdar pattern) (car matchlist))
                 (matchup (cdr pattern) (cdr matchlist))))))


;****************************
; The function match will first use the TEST function to see if the two
;  lists meet the requirements mentioned above.  If so,  It uses the matchup
;  function to create a list out of the varibles and their apropriate matches
;  If the rules are not met, it returns NIL.
;****************************

(defun match (pattern matchlist)
  (if (test pattern matchlist) (matchup pattern matchlist) nil))
****************************

Zmacs (LISP) code.lisp >scenario-user MUNCH: (1) *  [More above]
[12:18:58 Process Screen Hardcopy wants to type out]

Mouse-L: Move point; Mouse-M: Mark word; Mouse-R: Editor menu.
To see other commands, press Shift, Control, Meta, Meta-Shift, Super, or Super-Meta.
[Sun 11 Feb 12:19:04] Keyboard              CL PCL:          User Input
```

Figure 4-13: Final State of Editing Buffer

a user model, and to inform the process of developing a general approach to accomplishing that. A critic was used as a context for investigating ideas and implementing some of that user modelling framework because the paradigm is well understood, and has been instantiated in at least one mature and well understood system, LISP-CRITIC.

The ideas for how to model users of critiquing systems have their foundation in theoretical notions about human-computer interaction and grew out of studying user modelling in other domains. Those ideas served to guide implementation of the user modelling component in LISP-CRITIC. It is one of the knowledge-based components of the system, the others are the critic rules and the conceptual domain model. A related component is the explanation generator; it is supported by the capabilities of the user modelling component.

In the past, LISP-CRITIC has been a platform for evaluating ideas about how computer systems should be designed. Integrating a user modelling component is a natural extension of that previous work; the story is actually more complex: LISP-CRITIC was not merely extended, but ported to a new computational environment and adapted to a new interaction style in support of this research. In the next section of this dissertation (Chapters 5, 6, and 7) the system components which were developed in and are directly related to this work, the domain model, the envisioned approach to explanation, and the user modelling component, are described in that sequence.

# CHAPTER V

# A DOMAIN MODEL FOR LISP

## 5.1. Introduction

This chapter describes the domain model developed to support explanation-giving and user modelling in LISP-CRITIC. The work refines and implements ideas developed in previous research [Fischer 88a]. In this chapter, I cover why there is a need for a domain model, how the domain was analyzed, the results which in turn determined the model's general form, and then the graphical notation used to conceptualize the model. Next I discuss the implementation of the domain model followed by the extensions and other potential uses for both the model and the methodology.

Developing the domain model was an enabling technology that was required in order for both the explanation and user modelling research to proceed. The development of that model and its ultimate form are described here because development did involve significant effort and knowing how the model was developed and then implemented in LISP-CRITIC will facilitate the reader's understanding when I describe its use in the explanation process in Chapter 6, and for user model representation and acquisition in Chapter 7.

The model represents the domain of LISP in terms of three entities: the concepts of the language, basic COMMON LISP functions, and the transformation rules in LISP-CRITIC. The latter are represented in the conceptual domain mode, even though they are captured in applicative form in the rule base, because a rule is the triggering condition for an explanation.

## 5.2. Requirements for a Domain Model

As previously discussed, in order to accomplish cooperative problem solving it is imperative that systems have the capability to explain their reasoning. In the case of critiquing, in general, and LISP-CRITIC in particular, this means explaining the reason a given transformation is being suggested — the rationale and concepts behind a rule.

A common theme of other research in explanation is that in order to provide an acceptable explanation capability, the system needs to represent knowledge of the subject domain at an abstract level [Clancey 87; Kass, Finin 88b; Paris 87; Wiener 80]; I say more about this in Chapter 6. For LISP-CRITIC, previous research determined the possibility of representing programming knowledge in terms of concepts, programmer goals and functions [Fischer 88a]. Such a representation can explain the improvements suggested by the rules and derive a model of the user. The implementation of the domain model described here supports those goals.

The rule base in LISP-CRITIC represents procedural knowledge in a compiled or applicable form that is appropriate for efficiently analyzing code and rapidly generating recommendations for how to improve segments of that code. However, knowledge in this form will only support rule tracing explanation approaches [Scott, Clancey, Davis, Shortliffe 84] and these were shown to be inadequate [Clancey 84]; systems need the ability to explain rules at the concept level so as to facilitate user understanding and support learning. To achieve that requires a more abstract domain model; a model that captures the abstractions representing the underlying domain at the level of its fundamental concepts.

A conceptual structuring of the domain should provide a way to link the applicative rule-base knowledge with explanation strategies and with the user model. In LISP-CRITIC, a rule, or set of rules, is the underlying causative mechanism behind a single piece of advice. To understand that advice well enough to decide whether or not to accept it, users needs to understand what concepts underlie that rule. A concept-based domain representation can be configured so that it provides that information; it can inform the system what concepts underlying that rule. In turn, the system needs to be able to determine which of these concepts are not part of the user's current knowledge so it can focus on explaining the unfamiliar concepts. The terms *understanding* and *knowing* are used synonymously; we do not try to make the theoretical distinctions between them that are important to some studies of cognition or philosophy.

## 5.3. Form of the LISP Domain Model

In order to support the explanation strategies and user modelling process in LISP-CRITIC, the information contained in the domain model consists first of the underlying concepts for LISP. To determine those concepts we reviewed the following commonly used LISP texts: [Steele 84], [Winston, Horn 81], and [Wilensky 84]. Forty-five commonly-referred-to concepts were identified in these texts. The list does not include more fundamental concepts that exist "in the world", such as the set of integers, but focuses on those concepts that are unique to LISP or programming languages in general. The terms used to identify these concepts are listed alphabetically in Figure 5-1. The term concept, as expressed in the Philosophy of Science literature, is an abstract notion; there is a distinction between concepts themselves and the terms that stand for them [Hempel 65]. The concepts shown in Figure 5-1 were designated using terms that seemed appropriate, while recognizing that in other research, and context, they may be described with other names. In comparing this analysis with an effort by Gray to capture the underlying entities of LISP in a hypertext database [Gray 88], we found sufficient overlap in terms and structure to provide confidence that the topology is valid and useful. His work could not be used directly because it was never completed.

ARGUMENTS
ASSOCIATION-LISTS
CAR-CDR-CONCATENATION
CONDITIONAL-EXITS
CONDITIONALS
CONS-CELL
DATA-TYPES
DESTRUCTIVE-FUNCTIONS
DOTTED-PAIR
EMBEDDED-FUNCTIONS
EVALUATION
EVALUATION-ORDER
FALSE/EMPTY-LIST/NIL
FUNCTION-DEFINITION
FUNCTIONS
IDENTITY-VS-EQUIVALENCE
INPUT-FUNCTIONS
INTERNAL-REPRESENTATION
ITERATION
LAMBDA-BINDING
LISP-ATOM
LIST-ITERATION
LISTS
LITERAL/QUOTE
LOGICAL-FUNCTIONS
MAPPING
MULTI-VALUE-RETURN
NUMERIC-ITERATION
OPTIONAL-PARAMETERS
OUTPUT-FUNCTIONS
PARALLEL/SEQUENTIAL-BINDING
PARAMETERS
PREDICATES
PROPERTY-LISTS
RECURSION
SCOPE
SIDE-EFFECTS
STRINGS
SYMBOLIC-EXPRESSION
TAIL
TESTS
TRUE/NON-NIL
VARIABLE-INITIALIZATION
VARIABLES

**Figure 5-1:** List of Domain Concepts

While selecting the set of concepts for inclusion in the domain model, two types of relationships between concepts were recognized, relationships that are useful for explanation-giving, and one that can be used in user model acquisition:

1. **The dependent-on relationship:** This indicates for a particular concept which other, more fundamental, concepts are prerequisites to understanding it.

2. **The related-concepts relationship:** This is a relationship between concepts that are similar, this information could be used by the explanation component in some presentation strategies.

We also selected 103 fundamental LISP functions to be represented in the domain model, primarily those that are found in the LISP-CRITIC rules. The term ''function'' is not entirely correct, this class of domain entity might more specifically be referred to using the term ''constructs'', as in [Steele 84].[5] However, ''Function'' is the term used here because the it was selected at the beginning of the domain analysis and continued to be used throughout the implementation. To understand a function also depends on understanding certain underlying concepts; therefore, functions are related to concepts via "dependent-on" relationships, like the one described above. Functions may also be similar to one another, for example, *cond* is similar to *if*: the model also captures this relationship.

Part of the analysis process was a grouping of the concepts and functions into logically related sets by several LISP programmers. We followed a methodology that has been successfully used to structure similar domains in other research [Doane, Pellegrino, Klatsky 89]. The 45 concepts were divided into five groups that represent an approximate consensus of the experts' categorizations. The groups seem to fit into a hierarchy when viewed across the "dependent-on" layer of relationship, but this attribute was not further investigated. These groupings are shown in Figure 5-2; the names assigned attempt to imply the commonality that exists between the concepts in that group. Similarly the 103 functions were classified into 14 categories. The rationale for the categorization exercise was to

[5]Still more precisely, the set actually consists of special forms and standard macros defined for COMMON LISP.

validate the domain entities that had been selected, and to integrate the knowledge of other domain experts into our specification for LISP. This exercise was also a way to reflect on and refine the concepts. The purposes for which these groupings might be used in LISP-CRITIC are not yet established, but that part of the process is discussed here to demonstrate the depth of the analysis and the generality of the modelling approach. The categorizations have been captured in the domain model for possible future use.

We also capture LISP-CRITIC rules in the domain structure because this is the level of application knowledge used by the system and for sake of having a complete single representation of the system knowledge. A rule has links to the functions that occur in the rule, and the LISP concepts that underlies it. When the system recommends a change to a program, the only thing it knows is that the same code conformed to a pattern expressed on the left hand side of that rule and that it could be rewritten according to the pattern on the right hand side. To model what is involved in understanding that rule, it was necessary to capture knowledge about the functions in the rule and any domain concepts that are behind it. Concepts and functions probably exist as part of a programmer's mental model of the domain [Gentner, Stevens 83], therefore, these parts of the domain model may be something close to a cognitive representation, possibly representing chunks. It is unlikely that users, with a few exceptions, retain a LISP-CRITIC rule as part of their mental model for the domain, even after they develop an understanding of it.

In summary, the domain model needs to capture three types of entities LISP concepts, LISP functions, and the LISP-CRITIC rules, together with the relationships between instances of them. Relationships are often one-to-many, but their topology, although somewhat hierarchical within certain relationships (like the dependent-on-concepts for all concepts in the model), is highly interconnected

**High-level Concepts**
Symbolic Expression
Functions
Evaluation
Evaluation Order
Tests
Conditionals
Arguments
Variables
Scope

**Intermediate Concepts**
Parameters
Logical Functions
Predicates
Recursion
Iteration
Side Effects
Function Definitions

**Implementation Concepts**
Strings
Data Types
Conditional Exists
Input Functions
Output Functions
Variable Initialization
Numeric Iteration
List Iteration

**LISP Externals**
Atom
Literal/Quote
Parallel/Sequential Binding
Optional Parameters
Mapping
Tail
Lists
Property Lists
Association Lists
Car-Cdr Concatenation
Multi-Value Return
Embedded Functions
Lambda Binding

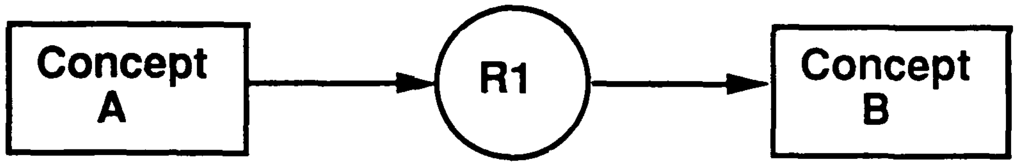
**LISP Internals**
Dotted Pair
Cons Cell
False/Empty List (nil)
Internal Representation
Destructive Functions
True(non-nil)
Identity vs Equivalence

Figure 5-2: Grouping of Concepts

and acyclic. Several paradigms, such as, frames and semantic networks were considered as possible representation schemes for the model.

## 5.4. Conceptual Graph Notation For Representing the Domain Model

An approach that provides the ability to visualize the entities and the relationships from the analysis above was conceptual graph notation; it also helped us to consider what are the needs of the explanation and user modelling approaches. Conceptual graph notation is part of the conceptual structures framework [Sowa 84]. There may be some confusion in the discussion because of overlapping meanings for terms in the theory with those chosen during the analysis of LISP. The underlying cognitive entity, according to conceptual-structure theory, is a percept; and the interpretation of a percept, a concept. Conceptual graphs model concepts and the relationships between them. For LISP there are three types of entities: LISP concepts, LISP functions, and, LISP-CRITIC rules; all instances of the theoretical notion concept. In the formal notation for conceptual graphs, concepts are shown as rectangles, and the relations between concepts, as circles; this is shown graphically in Figure 5-3.



Conceptual graph showing concept A is related to concept B by relation R1.

Figure 5-3: Conceptual Graph Notation

An example of how this representation allows visualization of the domain model for LISP is in Figure 5-4; it shows the LISP concept *Recursion* using this notation. In this example, *Recursion* is dependent upon the LISP concepts *Tests, Conditionals*, and *Functions*; it is related to the concept of *Iteration*. Recursion is not a concept underlying any LISP-CRITIC rule, but because it is one commonly used in most LISP texts, it has been captured in the domain model. It is shown here to demonstrate the generality of the approach. The conceptual domain model should be able to serve purposes more general than explanation-giving and user modelling in critics. Using recursion, a domain concept not required by our system, demonstrates generality, and should provide an intuition to the reader for how the conceptual graph model approach might be applied to serve other paradigms and applications.

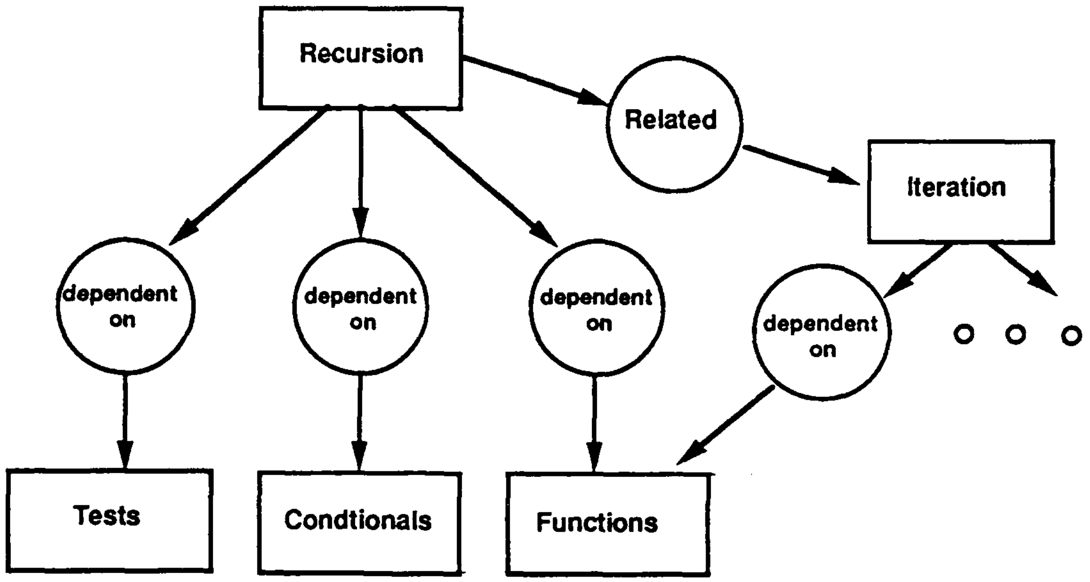


Figure 5-4: Example of LISP Concept Recursion in Conceptual Graph Notation

Conceptual graphs were a useful methodology for visualizing the domain model, but an implementation method was required. For reasons of portability, availability, and standardization, the domain model for LISP-CRITIC was implemented in the Common LISP Objects Systems (CLOS) extension to COMMON LISP.

## 5.5. Implementation of the Domain Model

In the domain model implementation, the concepts (rectangles) from Conceptual Graph notation were defined as classes, and relations between concepts (circles), captured in slot definitions. The class hierarchy for LISP consists of a super class, *lisp-object*, with three subclasses *lisp-concepts*, *lisp-functions* and *lisp-critic-rules*. There are slots in each object instance for name; dependent-on-concepts; related-concepts, related-functions, and related-rules; and the groupings shown in Figure 5-2. Group membership for *lisp-concepts* is represented in the *level* slot, such as *recursion* belonging to the category of *intermediate* shown in Figure 5-7, for *lisp-functions* in the *category* slot. The CLOS code that defines these objects is shown in Figure 5-5. The three types of entities inherent common slots for *name* and *dependent-on-concepts* from the fundamental class *lisp-object*.

The complete domain model is difficult to show graphically because it is highly interconnected. It can be considered to have three layers, one each for LISP concepts, functions, and LISP-CRITIC rules. Populating each layer are instances of the entity class for that level. Links are found between instances within a level, as well as between instances at different levels. For example the LISP-CRITIC rule *cond-to-if* is found in the LISP-CRITIC rule layer; it is linked both to similar rules (e.g. *cond-to-when*) within that layer, as well as to concepts (e.g., *conditionals*) in the concept layer, and of course to functions (e.g., *cond*) in the function layer.

To give the reader a flavor for the interconnectivity of the model, again consider the LISP concept *recursion*. Understanding *recursion* is dependent on the user understanding the concepts of *tests, conditionals,* and *functions. Recursion* is also related to the concept of *iterations*. The code to instantiate *Recursion* as an instance of a *lisp-concept* is shown in Figure 5-7; the conceptual graph representation for that concept in Figure 5-4. Most concepts, functions, and rules in the domain model have similar high degrees of connectivity.

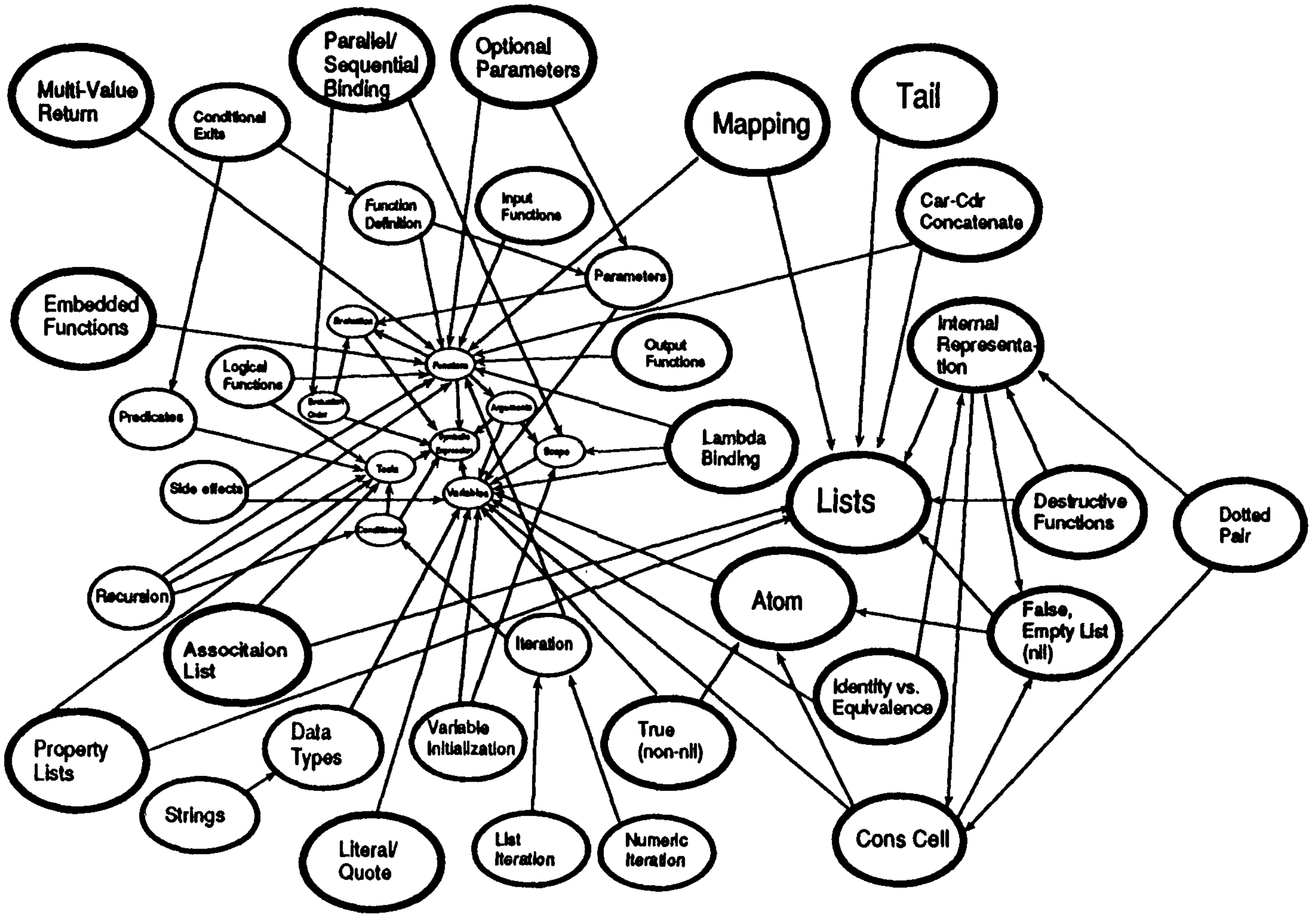
```
(defclass LISP-OBJECT()
     ;;; Generic Super Class for all LISP Objects
     ((name
     :accessor name
     :initarg :name)
      (dependent-on-concepts
        :accessor concepts-dependent-on
        :initform  nil
        :initarg  :dependent-on-concepts)))

(defclass LISP-CONCEPT (lisp-object)
     ((related-concepts
        :accessor related-concepts
        :initform nil
        :initarg :related-concepts)
      (level
        :accessor level
        :initform 'high-level
        :initarg :level)))

(defclass LISP-FUNCTION (lisp-object)
     ((pattern
        :accessor syntax-pattern
        :initarg :pattern)
      (related-functions
        :accessor related-functions
        :initform nil
        :initarg :related-functions)
      (category
        :accessor category
        :initform 'unclassified
        :initarg :category)))

(defclass LISP-CRITIC-RULE (lisp-object)
     ((functions-in-rule
        :accessor functions
        :initarg :functions-in-rule)
      (related-rules
        :accessor related-rules
        :initform nil
        :initarg :related-rules
        )))
```

Figure 5-5: CLOS Specification For LISP Domain Entities

It is difficult to display the entire domain structure in a single two dimensional graph. In Figure 5-6 we provide a feel for the complexity of the structure when viewed across a portion of a single strata or level of the domain. It shows graphically the *lisp-concept* layer together with the *dependent-on-concepts* links between the 45 concepts. For simplicity and readability sake, this figure does not use the conceptual graph notation; instead each oval represents a LISP concept, and links are all of the same type; they represent the *dependent-on-concept* relation. Different oval sizes represent each of the 5 categories shown in Figure 5-2.

The primary reason an abstract domain representation for LISP was investigated was the need to support explanation-giving and user modelling. In that regard the the model can support both of these processes in several ways.

The explanation component uses the domain model to determine what *concepts must* be explained to a user who does not understand a particular recommendation. Since all recommendations are generated from a rule firing, the user needs to understand the concepts a rule depends on, as well as the functions that are part of that rule. The system must explain to the user those concepts and functions the user does not already know. Furthermore, if the user does not understand the more fundamental concepts upon which the understanding of a given concept is dependent, the system may want to explain those as well. For our example concept, *recursion,* shown in Figures 5-4 and 5-7, the user must already know the concepts: *tests, conditionals and functions* or these must be addressed as part of the strategy for explaining *recursion.* The explanation system could also use the domain model to select an explanation strategy by using the *related-concepts* or *related-functions* relationship (slots). In the case of *recursion*, the domain model indicates that *iteration* is a related concept so one explanation strategy would be for the system to describe recursion as compared to iteration.



This graph shows the dependent-on-concepts relationships for the LISP concepts in the Domain Model

Figure 5-6: Concept Layer of Domain Model

```
(make-instance 'lisp-concept
                   :name 'recursion
                   :dependent-on-concepts
                        '(tests conditionals functions)
                   :related-concepts '(iteration)
                   :level 'intermediate)
```

Figure 5-7: CLOS Specification For Concept Recursion.

The user modelling component uses the domain model for two purposes. The model provides a representational basis for users' knowledge; the user model overlays the domain model to capture the LISP concepts and functions that a user already understands. The user model, as will be discussed further in Chapter 7, is an annotation or coloring of the conceptual graph for LISP. The model presently contains an implicit assumption that if users know two concepts then they also know about any relationships them. We have not yet considered whether this is something that should be explicitly represented and, if it should be, what modifications to the domain model representation might be required to accommodate it.

The user modelling component has a set of inference methods, again described further in Chapter 7, that build up individual models representing each user. Some of these methods use the structure of the domain model as a basis. For example, when the system determines that a user knows about recursion, it will annotate the user model with an assertion that the user understands *recursion*, and also infer that the prerequisites are known, these are defined in the *dependent-on-concepts* relation; they are concepts: *tests*, *conditionals*, and *functions*.

## 5.6. Extending the Approach

The approach to modelling LISP described here is not a unique conceptual structuring for it or similar domains. Researchers confronting this same problem have had to use similar formalisms and representation languages. Attempts to develop an explanation component for MYCIN were constrained until a representation for the domain other than the inference rules was used. Wallis and Shortliffe found it useful to describe the knowledge representation for their system in terms of a semantic network [Wallis, Shortliffe 84]. Kass used the NIKL representation language to model investment knowledge in his expert adviser so that it could explain advice in terms appropriate to the user's goals, beliefs, and prior domain knowledge [Kass 88]. A common theme in this research is that there is a need for a domain representation that is more abstract than rules. The conceptual model approach presented here meets the requirements in the domain of LISP for a critiquing system; it could possibly be used for a larger class of domains and applications as well.

The ideal knowledge acquisition approach is to capture deep domain concepts as *first* step in knowledge-based system development; an idea that is used in the explainable expert systems (EES) framework [Neches, Swartout, Moore 85]. However, it is far easier to capture procedural knowledge in rule form. The rule-based paradigm is consistent and constrains specification of knowledge; this assures the knowledge engineer that the system's actions or advice will agree with true human expertise. When we attempt to add explanation capabilities the rule-based paradigm breaks down and second order domain representations become necessary.

The specific concepts, functions, and their relationships included in this implementation may not be universally accepted. We found that experts frequently do not agree on what are the significant concepts underlying LISP, or how they relate to one another. The model implemented here was developed as an approximate consensus of what makes up the domain of LISP, it is required so that we can determine the effectiveness of the methods that generate explanations and model users.

## 5.7. Summary

This chapter described a domain modelling approach and implementation that captures LISP knowledge in a conceptual structure. The result was a graphical, concept-based domain model. The motivation for having the model is a need to link procedural knowledge already contained in LISP-CRITIC rules with explanation strategies and user models to determine how to accomplish the explanation process. The types of entities in the domain model are LISP concepts, LISP functions and LISP-CRITIC rules, represented as nodes, and interconnected via relationship links. Conceptual graphs provided an appropriate notation for visualizing and capturing the domain structure; CLOS was used as the implementation language. The approach is a suitable representation, able to support research on explanation-giving and user modelling. Next, we will describe a framework for explanation supported by this domain model.

# CHAPTER VI

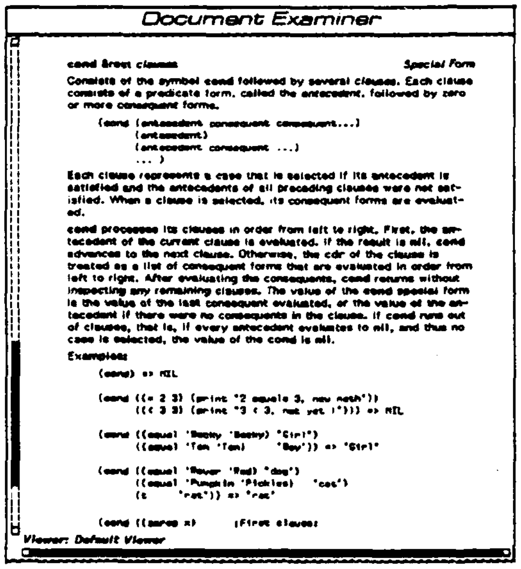
# THE FRAMEWORK FOR EXPLANATION

In the course of building cooperative knowledge-based systems one objective is take advantage of the different strengths of users and computer systems. The system provides a source of expert domain knowledge that is used to make suggestions to users; the system should also explain those suggestions. Current explanation systems frequently fail to satisfy users for a variety of reasons; explanations are too often based on the implicit assumption that the process of explaining is a one-shot affair, and that the system will be able to produce or retrieve a complete and *satisfying explanation provided it is endowed* with *artificial intelligence*. Our approach takes advantage of information and knowledge-based system technology already available to provide the user access to explanations at different levels of detail and complexity. Developmental efforts in this work focused on the concepts to be explained, rather than on selecting a complete prestored explanation appropriate for a given user. The domain and user models provide to the system the capability to determine that set of concepts.

Early research on how to explain expert knowledge in computers was done in the context of MYCIN. The approach taken was to provide a rationale by showing users an historical trace of the rules that fired in arriving at a diagnosis. Rule-tracing explanatory approaches, even when "syntactically sugared", to make them more readable, are difficult to follow. Readers of that literature quickly realize that anyone not familiar with medical terminology and concepts have great difficulty understanding them. This points up a general shortcoming of most explanation approaches, they too often use domain concepts there readers do not know. User models help systems to overcome this shortcoming. The failure of explanations in domains more closely related to our work on LISP-CRITIC are not difficult to locate. A standard example of unsatisfactory explanation is the UNIX *Man Page* command.

In Figure 6-1 we consider a more realistic example, one from the environment in which LISP-CRITIC is implemented, and from a system generally acknowledged as being better than many similar documentation systems. If LISP-CRITIC can only give suggestions, and not explain those suggestions, then users might attempt to achieve an understanding of the transformation by using other system resources, in this case the Symbolics on-line documentation. The first such transformation in the Chapter 4 scenario recommended replacing a *cond* with an *if*. If users consult the Document Examiner for information about these two LISP special forms, what they get are the descriptions displayed in Figure 6-1. The explanations shown are better than most, they contain examples; begin with one or two sentence minimal explanation, and step-wise expand on it; they use a hypertext display that allows followup and further exploration; and the description for *if* even refers to the LISP concept which it exemplifies, conditionals. As will be shown, the explanations still fail for a number of reasons. In general they are too long, attempting to cover everything, are not specific to the user, and, in this case, have to be viewed individually in sequence (they are only shown side-by-side in this figure to make the discussion easier to follow), this makes it difficult to compare the two and come up with the rationale for why they might be interchangeable in this situation.

In this chapter the theoretical understanding of explanation-giving in cooperative systems will be discussed, together with an overview of related research. It will also cover the purposes of explanations in these systems and the implications for the user model's role. Finally, an explanation framework for LISP-CRITIC will be described.



Document Examiner

**cond &rest *clauses*** *Special Form*

Consists of the symbol **cond** followed by several *clauses*. Each clause consists of a predicate form, called the *antecedent*, followed by zero or more *consequent* forms.

```
(cond (antecedent consequent consequent...)
      (antecedent)
      (antecedent consequent ...)
      ... )
```

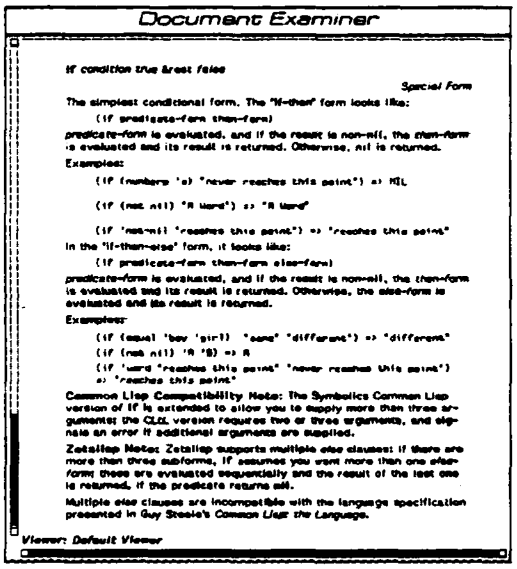
Each clause represents a case that is selected if its antecedent is satisfied and the antecedents of all preceding clauses were not satisfied. When a clause is selected, its consequent forms are evaluated.

**cond** processes its clauses in order from left to right. First, the antecedent of the current clause is evaluated. If the result is **nil**, **cond** advances to the next clause. Otherwise, the cdr of the clause is treated as a list of consequent forms that are evaluated in order from left to right. After evaluating the consequents, **cond** returns without inspecting any remaining clauses. The value of the **cond** special form is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If **cond** runs out of clauses, that is, if every antecedent evaluates to **nil**, and thus no case is selected, the value of the **cond** is **nil**.

Examples:

```
(cond) => NIL

(cond ((= 2 3) (print "2 equals 3, new math"))
      ((< 3 3) (print "3 < 3, not yet !"))) => NIL

(cond ((equal 'Bobby 'Bobby) "Girl")
      ((equal 'Tom 'Tom)     "Boy")) => "Girl"

(cond ((equal 'Rover 'Fred) "dog")
      ((equal 'Pumpkin 'Pickles)   "cat")
      (t     "rat")) => "rat"

(cond ((zerop x)      ;First clause:
```

*Viewer: Default Viewer*



Document Examiner

***if condition true &rest false***

*Special Form*

The simplest conditional form. The "if-then" form looks like:

```
(if predicate-form then-form)
```

*predicate-form* is evaluated, and if the result is non-**nil**, the *then-form* is evaluated and its result is returned. Otherwise, **nil** is returned.

Examples:

```
(if (numberp 'a) "never reaches this point") => NIL

(if (not nil) "A Word") => "A Word"

(if 'non-nil "reaches this point") => "reaches this point"
```

In the "if-then-else" form, it looks like:

```
(if predicate-form then-form else-form)
```

*predicate-form* is evaluated, and if the result is non-**nil**, the *then-form* is evaluated and its result is returned. Otherwise, the *else-form* is evaluated and its result is returned.

Examples:

```
(if (equal 'boy 'girl)  "same" "different") => "different"
(if (not nil) 'A 'B) => A
(if 'word "reaches this point" "never reaches this point")
=> "reaches this point"
```

**Common Lisp Compatibility Note:** The Symbolics Common Lisp version of **if** is extended to allow you to supply more than three arguments; the CLtL version requires two or three arguments, and signals an error if additional arguments are supplied.

**Zetalisp Note:** Zetalisp supports multiple *else* clauses: if there are more than three subforms, **if** assumes you want more than one *else-form*; these are evaluated sequentially and the result of the last one is returned, if the predicate returns **nil**.

Multiple else clauses are incompatible with the language specification presented in Guy Steele's *Common Lisp: the Language*.

*Viewer: Default Viewer*

These two screen displays show the Document Examiner descriptions for *if* and *cond* that are retrieved when a user searches for information on the functions in the *cond-to-if* rule. Both occupy the entire viewer and on a computer screen cannot be viewed together like they are here.

**Figure 6-1:** Explanations for *cond* and *if* from the Document Examiner

## 6.1. Theory

### 6.1.1. The Need for Explanations

In order for professionals, managers, and scientists to accept knowledge-based systems, it is essential to provide explanations of the knowledge. The need for good explanations was identified in a study of physicians' attitudes towards expert systems:

> *Explanation*. The system should be able to justify its advice in terms that are understandable and persuasive. In addition, it is preferable that a system adapt its explanation to the needs and characteristics of the user (e.g., demonstrated or assumed level of background knowledge in the domain). A system that gives dogmatic advice is likely to be rejected. [Teach, Shortliffe 84, p. 651]

Explanation in cognitive science can evoke two different meanings: the process of presenting information, and an internal cognitive process that develops a knowledge representation. Our work focused on the presentation process while attempting to keep both meanings in mind. The internal-process view claims that explanation is equivalent to understanding [Schank 86]. According to this perspective, humans achieve understanding using a process that involves generating their own explanations. For our work on presenting explanations this means that systems must provide the information that is needed to support the self-explanation process. If systems know exactly what information is required to insure understanding, can tailor that information to the individual, and then present it in an optimal form, users might adopt it as their own. Such a goal for computer-generated explanations (or even those produced by another human, for that matter) is too ambitious. Rather than attempting complete, ideal explanations which each user can understand and integrate into their mental models, computers must instead concentrate on providing users with the material required to produce their own explanations. This means *generating explanations with the computer* not merely displaying stored ones; explanations that use the domain concepts appropriate to a particular problem solving context, so as to provide users an opportunity to produce a self-explanation, and therefore achieve understanding.

## 6.1.2. Functions for Explanations

We are investigating how to design systems that serve users actively engaged in their own work — cooperative problem solving systems that provide a task-based environment in which users work toward goal accomplishment. Systems that support users' work are more than media used for describing their problems, and more than just tools to extract useful information from a database. They should be active agents that provide for problem-domain communications at the construction artifact level [Fischer, Lemke 88], can critique users work, and are able to explain their knowledge. We analyzed the reasons users seek explanations and determined that a common triggering condition is experiencing some sort of impasse. A similar idea motivates theory about what should happen during instructional episodes, there the emphasis is on determining how to communicate knowledge to overcome impasses and how students formulate new procedural knowledge [VanLehn 88]. We cataloged these as ''task-oriented impasses'' in order to develop a better understanding of where explanation fits in each situation. There are four categories of task-oriented impasses:

1. *Action impasses* occur when users do not know what to do next. Some action impasse questions are: What should I do next? Is *action* the right thing to do next? How do I do *action*? What did the system just do? What are the results of doing *action*? Can I do *action* now? These are the types of impasses help systems should be designed to address.

2. A *communication impasse* is a failure to understand a given object in

the environment. Representative questions are: What is *object*? Why is *object1* shown instead of *object2*? What is the rationale for suggesting *object* or *action*? This is the category of impasse the user experiences in the scenario when trying to decide whether to accept or reject a suggestion.

3. *Motivation impasses* fall into the realm of behavioral psychology; their basis is an anthropomorphic view of the computer system. Representative questions are: Why did the system do *action*? Why did the system just communicate with me? Why did the system just say *X*? Why should I do *action*?

4. *Curiosity impasses* are a bit different. The other categories consist of questions that arise when users encounter a problem. Curiosity impasses are not necessarily impasses, in a strict sense, but rather are diversions. They are circumstances in which users gather information that is interesting or helpful, but if it is missing, further work is not actually impaired. For consistency, these are also "impasses". Questions that illustrate curiosity impasses are: Is *object* a concept *X*? How do $object_1$ and $object_2$ differ? On occasion, LISP-CRITIC users also experience these; it is a case where they understand the suggestion but see it as an opportunity to improve upon their knowledge and therefore request explanations so as to engage in active exploration.

Assisting users during problem solving requires that explanations be designed to help them overcome impasses. Such explanations in cooperative problem solving systems can serve four functions. We adapted these functions for cooperative problem solving from ones that were found during investigations surrounding MYCIN in which they studied users of knowledge-based medical information systems [Wallis, Shortliffe 84].

1. Explanations allow users to examine the system's recommendations.
2. Users need explanations to relate recommendations to domain concepts — to understand ''what is suggested''.
3. The explanation should help users to see the rationale for recommendations — to understand "why this would be better".
4. Explanations are needed by users to learn the underlying domain concepts.

These functions are not mutually exclusive; single explanations in a cooperative problem solving system will have to accommodate multiple purposes.

### 6.1.3. Shortcomings of Current Approaches

Most attempts to provide explanations use prestored scripts in the form of canned text. Those types of descriptions have been criticized as difficult to understand, incomplete, and hard to navigate [Weiss 88]. Empirical studies of tutoring in both humans and computers determined that canned explanations are insufficient approaches [Fox 88]. Because critiquing and tutoring are closely related, many of the problems listed there apply to critiquing as well. ''Canned text'' is intended to meant pre-written text, stored in machine memory in a form that the system cannot interpret meaningfully (most likely as character strings.) The use of canned explanations is not inherently bad just because it is done by a computer. Empirical studies of human explanations found a similar strategy is often employed by people when explaining something ''for the sake of others'' [Schank 86]. The difference is that people understand their prestored explanations — they make sense to the explainer; they represent a form of mental model.

When it happens that the recipient does not understanding such an explanation, the nature of most human-to-human discourse allows them to query the explainer for clarification or elaboration.

Canned explanations captured in computer systems are inadequate when their content is poorly chosen or presented. There are five primary reasons for the failure of computer explanation approaches:

1. Explanations are too long; users get lost, bored, or confused; they do not bother reading the text just to find what they need. This is especially characteristic of many on-line help systems.
2. Too often, explanations attempt to tell the user everything they could possibly need to know rather than determining what is specifically required for the situation at hand, and for the individual requesting the explanation. This creates complexity and is frequently what makes them too long.
3. Users are not provided the capability to ask follow-up questions or enter into a dialog with the computer. The explanations are designed as if they could satisfy their reader with a single presentation.
4. The explanations do not provide examples to facilitate understanding textual descriptions. Even when available, examples may be inappropriate for the user's particular problem.
5. The explanation text is written from an author's perspective. It is based on that author's conceptual model of the domain, not the readers'.

The examples from the Document Examiner shown in Figure 6-1 exhibit some of these characteristics. They fail on the first account, being longer than most users would want in the *cond-to-if* transformation-situation. The system does not, of course, individualization the descriptions, therefore they also fail on the second account. Document Examiner does provide hypertext capabilities, therefore a limited form of follow-up is provided. Also, both documentation entries (as do quite a few in Document Examiner) contain multiple examples to facilitate understanding. On the last point, the explanation of *cond* is particularly poor, it appears to have been written by a LISP expert (hacker) from his or her individual perspective; the one for *if* is actually much better; its author attempted to direct it toward a less sophisticated programmer.

Some systems attempt to overcome several of these problems, but none addresses all shortcomings. Our strategy is to recognize the shortcomings while using an interactive approach based on available technology integrated with domain and user modelling capabilities. We consider the limitations of canned text but try to be realistic about current capabilities of computer systems. In developing explanation strategies there is too often an assumed environment which contains an intelligent computer able to generate natural language, predict users' needs, and enter into a followup dialog. Techniques are needed now that work within the constraints of available technology. Based on these limitations, an explanation framework was developed; it considers what is possible; a part of it was implemented to further our understanding and evaluate the role of the user model.

## 6.1.4. Basis for Minimalist Explanations

If a user model, such as the one described in Chapter 6, can provide a detailed representation of users' knowledge, then it will be possible to formulate and present an appropriate explanation. One method to achieve that is the minimalist approach [Fischer et al. 90]. The ideas for minimal explanations share the underlying theoretical foundations with minimal approaches to instruction

[Carroll, Carrithers 84]. Both use the principle that an optimal first approach is to provide users with the minimum amount of information required to accomplish their task. Theoretical bases for this approach are found in related work on discourse comprehension:

1. Short-term memory is a fundamental limiting factor in reading and understanding text [Dijk, Kintsch 83; Britton, Black 85]. The best explanations are those that contain no more information than absolutely necessary, since extra words increase the chances that essential facts will be lost from memory before the entire explanation is processed.
2. It is important to relate written text to the readers' existing knowledge [Kintsch 89; Fischer et al. 88].

Similar practical guidelines are also found in the theory of rhetoric. Flesch developed formulae to evaluate the readability of text [Flesch 49] which are frequently used to evaluate documentation and instruction. Computer explanation systems should comply with similar standards; using short sentences and known vocabulary are important criteria. Strunk and White's guide to good writing contains similar advice; they tell writers "Don't explain too much" when writing explanatory text [Strunk, White 57].

## 6.2. Related Work

Some research on explanations in knowledge-based systems assumes a natural language interaction, such as in the dialog advisory systems discussed in Chapter 2. Another approach also attempts to capture expertise during the knowledge acquisition phase of building an expert system; that approach uses a methodology which will later facilitate explaining that knowledge: the explainable expert system approach (EES) developed by Swartout [Swartout 83] is one example.

The fundamental claim behind EES is that explanation is simplified if the knowledge acquisition process occurs at the conceptual level and a system automatically generates the operational knowledge (i.e., rules). Then to explain a rule, the system can trace through that portion of the conceptual domain knowledge from which the rule was generated. That approach is appealing but has not been enthusiastically accepted as standard knowledge-engineering practice. LISP-CRITIC's rule base was developed using the traditional knowledge acquisition process of querying expert LISP programmers. For systems to explain something captured in procedural (rule-based) form requires reverse engineering of the applicative knowledge (in our case the transformation rules) in order to determine the concepts behind each rule. The process followed in developing and refining LISP-CRITIC, as opposed to the one proposed by EES, is more indicative of what will be the standard approach for providing knowledge-based systems with explanation capabilities, for the near future.

Moore extended the EES work, in a program-transformation system similar to LISP-CRITIC [Neches, Swartout, Moore 85]. Her specific research addressed a situation where users need to follow-up on explanations with clarification questions. Her ''reactive'' approach provides the user with an initial explanation, but accommodates the situation where it fails to satisfy the user; it provides increasingly informative fall-back explanations [Moore 87]. Her framework achieves a fall-back capability by monitoring and recording the dialog between the system and users, then using this dialog trace to identify and overcome difficulties. Moore still agrees with our goal [Moore 89], that a convivial system should make a good-faith effort to provide the right explanation the first time; it is when that fails that her reactive approach or something similar is needed. Providing the best possible initial explanation requires the system to understand the domain at a level that supports the generation process and to be capable of modelling its users. Her approach holds promise for future generations of knowledge-based systems but depends too heavily on natural language generation and dialog management. Until such capabilities are commonplace, other available techniques should be exploited. Whether a powerful access technique, such as hypermedia, or a dialog management approach to supporting fallback requirements is the better approach will only be determined when both have matured to the point where they can be subjected to comparative evaluations.

Several efforts to provide computer-based explanations generate strings of natural language. Some rely on a user model for tailored explanations [Kass 87b] while others generate the same explanation for any user [Danlos 87; Waterman etal. 86]. The natural language approach is complex and difficult, and the syntactic formats are limited.

Natural language approaches, such as Kass's reliance on Grice's rules for cooperative conversation and Moore's iterative fall-backs, use human-to-human discourse as their model for human-computer communication. This may be unreasonable, especially given the difficulties of reading large sections of text from a CRT screen [Hansen, Hass 88]. Knowledge-based system designers need to recognize the special capabilities and limitations of computers rather than trying to coerce the natural language paradigm into a screen- and keyboard-interaction style [Kennedy etal. 88; Fischer 88b]. Another crucial issue in explanation, whether between humans, or between a human and a computer, is not natural language, but using all of the available interaction facilities to insure that users are comfortable with the concepts presented during the explanation process; the essence of natural communications.

Paris [Paris 87; Paris 89] developed an approach to explanation based on an assumed user model. Her work provided initial motivation for our user modelling investigations [Mastaglio 90b]. She developed a theory that builds hybrid textual descriptions for devices using two strategies, a process trace and a constituency scheme.

- A process trace describes how an object works (her research was interested in explaining mechanical and electronic artifacts like the telephone.)
- A constituency scheme describes an object in terms of its component parts (like the receiver, transmitter, etc. of the telephone.)

A hybrid explanation for a device is actually a mixture of the two methods based on what users already know. Those *constituents* with which a user is familiar need only be indicated as component parts of the device being described, but others need to be explained in terms of how they operate (their process), or their own constituents, and so on. The process recursively executes, capturing those portions of the domain (objects or concepts) that an individual user needs explained to understand the device. A user model will indicate to her system which concepts and specific items in the knowledge base the user already knows. That information will, in turn, guide explanation-generation, combining the two strategies to insure that the explanations are presented at a level, and in terms of concepts, that users already understand. Her scheme can be used to generate an explanation for a LISP-CRITIC rule in terms of the LISP concepts and functions underlying the rules in the knowledge-base. LISP concepts are equivalent to the *underlying concepts* her model uses, and LISP functions are analogous to *specific items in the knowledge base*. This approach is a candidate strategy for use in the final steps of presenting an explanation to a user.

Requiring a knowledge-based system to have a concept level domain model is not a unique finding. Chandrasekaran and associates investigated the need for deep domain models in expert systems [Chandrasekaran, Tanner, Josephson 88; Chandrasekaran, Tanner, Josephson 89]. Their theoretical framework claims that explanation involves three issues: presenting the explanation, modelling the user, and endowing a system with "self-understanding". Their research focuses on the third issue. Their solution is similar to Swartout's in that they propose a "generic task methodology" approach to building expert systems. The paradigm focuses at the level of the task rather than that of abstract domain concepts; it makes basic explanation constructs available at a level of abstraction closer to the user's conceptual level, it is similar to the work on human problem-domain communications [Fischer, Lemke 88]. It also appeals to general domain knowledge in order to justify the system's problem-solving approaches.

In a perspective of what is happening to the user cognitively, one could consider explanations to be forms of knowledge retained in long term memory, and later reused to provide situational understanding. This is the basis for Anderson's work on learning by analogy [Anderson, Thompson 86], and has been investigated by Lewis as a substitution process [Lewis 89]. This theoretical view could be used by a system to chose a strategy for presenting explanations based on a user model containing a record of either the exact explanations previously received, or chunk-size domain entities (e.g., critic rules or lisp concepts) that were the focus of explanations; either of these could be used as a starting point for an analogical explanation. In the scenario, the user generates his own explanation for a rule (*cond-erase-t.nil*) using this process; this rule is similar to one previously explained (*cond-erase-pred.t*), and the user can forego requesting one from the system. The user model also needs to represent the user's possible goals: goals related to improving the immediate piece of code (e.g., make it easier for other programmers to read), or goals related to learning LISP (e.g., they want to become proficient users of the language). This work has not explored methods for inferring user goals or plans, but provides for goal representation in the user model. Goal and plan recognition is a significant research problem in itself.

Empirical observations of problem-solving interactions between salesmen and customers in a large hardware store, observed that explanation never took the *man page* approach. When explanations were required, the approach was one of minimizing the explanation, then following up on unclear concepts when necessary [Reeves 90]. This is interesting if you consider the fact that the particular store carries over 350,000 different items in over 33,000 square feet of retail space. If salesmen took the approach found in many computer systems, the explanations given would be extraordinarily long, to insure completeness, and complex, in order to account for relationships to other items in the store.

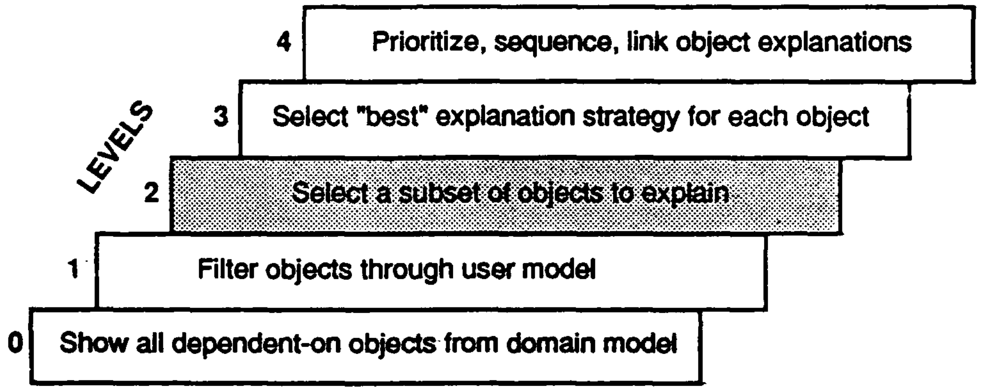
Argumentation is another approach to facilitating user understanding of the domain knowledge behind a critique or suggestion. Impressive results have been achieved using the argumentation approach in a critic for kitchen design [Fischer, McCall, Morch 89b]. Argumentation, as used in paradigms such as issue-based information systems (IBIS), provides a context for exposing the issues underlying a given suggestion. Argumentation approaches do not try to provide information at an appropriate level. Users of these systems retain primary responsibility for traversing the issue base. It is possible that they will find it difficult to locate exactly what they need, or to understand it, when the complexity level is not adjusted to their individual expertise.

## 6.3. An Explanation Framework to Support Critiquing

To support explanation in critics requires sufficient knowledge on the part of the system to describe what is going on and why. Operational knowledge in LISP-CRITIC is captured in transformation rules. For users to understand a transformation, they need to know the LISP functions in the rule and the concepts that makes it valid. This was informally observed during usability testing on the second version of LISP-CRITIC when we attempted to satisfy users with canned-text generic explanations of the rationale for each rule.

The domain model provides a conceptual basis for an explanation in terms of those functions and concepts that are prerequisite knowledge. Determining prerequisite knowledge is a recursive process because understanding those domain concepts that are prerequisites for the given concept requires, in turn, understanding their prerequisites and so on. To support such an approach, the deep structure in the domain model is queried to obtain a *concept-set* comprised of those prerequisites. A satisfactory explanation approach needs to still do something more, it must identify the concepts in that set that do not require explaining because the user already knows them. Furthermore it will reason about the best way to explain the remaining concepts.

We investigated ways to organize explanations for a system such as LISP-CRITIC and developed a framework that includes different levels for explanations (shown in Figure 6-2). The explanation levels capture necessary and sufficient conditions for adequate explanations. Each level incrementally enhances work done at a lower level, integrating additional knowledge about the user and the domain. A Level 0 explanation does not require knowledge about individual users. It uses the domain model to meet a necessary condition — knowing what to explain. The explanation component is provided the set of prerequisite concepts required to understand an object needing to be explained. Level 1 brings the user model into the process; here the prerequisite set of concepts is "filtered" through the user model to determine the subset appropriate for a given individual. In many cases, that filtered set is probably larger than we want to explain in a single episode. Therefore, at Level 2 the explanation component needs to know strategies that determine exactly which of that subset to explain and how.


LEVELS
4
Prioritize, sequence, link object explanations
3
Select "best" explanation strategy for each object
2
Select a subset of objects to explain
1
Filter objects through user model
0
Show all dependent-on objects from domain model

Five levels of explanation are identified. Level 0 insures all prerequisite knowledge for a given domain object is available to the explanation component. Level 1 builds on level 0 and so forth. The current LISP-CRITIC system provides simplified level 2 explanations. Level 3 and 4 will require presentation and natural language generation techniques.

Figure 6-2: Explanation Levels

A system operating at Level 2 passes a sufficiency test: it knows what concepts to explain to an *individual* user in a specific situation. However, it is still faced with the presentation problem; explanations need to be presented in a manner and style that will make them more readable. Achieving this level means the system will need to make use of additional domain knowledge or other information in the user model, in order to determine a "best" strategy for explaining a concept. For example, a system could make use of the related links in the domain model and the user model contents to determine candidate concepts or functions for use in a differential description [Fischer et al. 90]; one object can be described differentially in terms of another that the user already knows. Another example would be to apply user goals captured in the user model to determine the strategies that support those goals. Level 4 performs "syntactic sugaring." Here the individual explanations from Level 3 are ordered and appropriately linked, a non-trivial process that requires the system to have knowledge of discourse as well as natural language generation capabilities.

## 6.4. Role of the User Model in Explanations

The user model is discussed in the next chapter but, because its stated purpose is to support explanation generation it is important to consider, in the context here, what criteria for the user model are established by this explanation framework. This section will summarize, and review, the insights for the user modelling component that resulted from the analysis of the explanation process.

Cooperative systems must tailor their explanations to individuals in order to accommodate adequately the four functions previously listed. The system needs a basis for tailoring: this is the role of the user model. One simplified approach is to classify users by their expertise (e.g., novice, intermediate, expert); but, as reported earlier, this is not a valid representation for many domains and users. A finer grained representation that follows from Paris's work, represents user's knowledge in terms of the domain objects and concepts.

The user model needs a representation of the user's domain knowledge detailed enough to support each of the five "levels" of explanation shown in Figure 6-2. It has to be based, at least in part, on the conceptual model of the domain, so that it can filter the set of concepts that form the explanation basis.

The model needs to capture the user's goals in order to support Level 3 explanations. Programmers who use LISP-CRITIC have goals of either making their code easier for others to read (such as in the scenario) or making it execute more efficiently. A subsuming goal for both is learning to produce better code, the type of optimization goal merely determines the dimension along which they want to learn to improve their programs. Higher level goals, such as learning how to use new programming structures that can make programs better from the start, need to be acquired through explicit questioning. Problem-specific goals (such as writing a function to factorial)) are not within the scope of the current system. LISP-CRITIC does not know how to achieve these problem specific objectives; it is neither capable of automatic programming, nor does it have the knowledge, envisioned for the Design Apprentice portion of the the Programmer's Apprentice. Design Apprentice knows how to automatically select the low-level program cliches appropriate for achieving a specified objective [Rich, Waters 90].

Supporting Level 4 explanations is more difficult because they involve solving difficult issues on the research agenda for both explanation-giving and natural-language generation. We will not know all requirements for user models to support this level until that research matures. It is possible to conjecture some important capabilities, such as, knowing the education and reading comprehension level of users, because that knowledge could guide the generation of an appropriate explanation. To make the scenario in Chapter 4 more realistic, this type of higher level processing was manually applied to create the mock-up explanations. The system cannot presently generate anything that complex, it can only display short explanations for the selected concepts.

In its present form, the user model that was developed provides partial support for explanation-giving according to the presented framework. One supported approach is the minimal explanation strategy; it interrogates the user model to determine a minimal set of concepts to explain. Such strategies are possible because the model knows which concepts are familiar to the user.

## 6.5. LISP-CRITIC Explanation System

A conceptual overview of the explanation component shows how it conforms to the framework discussed above. One focus in this implementation is to use information already available to the system, and to present that information using ideas which provide the best support for users' needs.

There are several sources of information that is already available to the system and which can be used to satisfy some explanation requirements. This information is presented using techniques that were designed to provide users access to the information in four layers of increasing detail. These layers help to visualize how the system is designed and operates; they should not be confused with the conceptual levels shown in Figure 6-2.

1. A fundamental piece of information is the name of the rule that is the basis for a transformation. The rule name is an abstract reference to a chunk of domain knowledge, in the domain model that chunk is an instance of the class *lcr-rule*, and it may or may not have meaning to users. When it does have meaning, users may be satisfied just by knowing which rule fired and further explanations may not be required. An example of this occurred in the third scenario episode when the user recognized the *cond-to-if* rule because it had previously been explained. We are assuming that the name evoked the appropriate conceptual understanding on his part, seeing the two versions of the code may have also played a role.

2. That second piece of information is precisely those two versions of the code. The user can compare the system-transformed code with his own. The system displays the user's code together with the suggested changes. Sometimes this also triggers an understanding of the underlying concepts and rationale for the transformation. This is what occurred when the *cond-erase-t.nil* rule fired in the second scenario and no explanation was required. A rule based on similar concepts had just been explained and the user can generate his own understanding.
3. The minimal explanation layer is the point where empirical observations come into play. The user is provided with a text description of the system's advice based on the underlying concepts in the domain. The text description is comprised of portions of hypertext associated with each domain concept and rule.
4. A hypertext-based information space is also part of the underlying computational environment. LISP-CRITIC provides access to this information as a source of additional information for users who want to know more or who are not satisfied with the minimalist explanation of the advice. Users navigate through the hypertext space only after the system locates them within it in an appropriate context.

The explanations in Layers 3 and 4 require information from the user model to tailor their presentations to each individual user. The user modelling component can tell the explanation component which concepts users already understands so the system can avoid telling them what they already know.

Layer 4 explanations back up the minimal explanations with access to more detailed information. The system uses hypermedia along with some other available techniques to overcome many of the limitations of prestored text. Some of these techniques are inter-referential input/output [Draper 86]; command completion (the user can type abbreviations and the system completes the command); and mixed initiative dialogue (either participant can take the initiative or volunteer information) [Carbonell 70].

The approach used in the current implementation evolved from rule-tracing and canned text explanations methods attempted in an earlier version of LISP-CRITIC [Frank, Lynn, Mastaglio 87]. Alternative canned explanations for each rule were provided; each designed to accommodate a particular level of expertise. To chose the correct explanation, the system had to classify a user as a novice, intermediate or expert programmer. No user model acquisition was actually attempted because system testing using protocol studies and observations of users pointed out that the explanation approach was inadequate. One result was the finding that a finer grained approach to representing individual user knowledge is required, one that can also support updates as users' expertise changes.

The explanation approach is comprehensive and supports all four layers. An overview of the users' decision-making process, from the point the system makes a recommendation until they decide to accept or reject the suggested transformation is shown in a decision flow chart in Figure 6-3. The user, when informed that LISP-CRITIC recommends a change to his or her program, can get an explanation for that advice, or bypass it, deciding right then to accept or reject the suggestion. In the scenario, the user followed different paths through this decision process in different episodes; except that the user does not activate the hypertext facilities in any of the episodes. If he had used the mouse to select either *cond* or *if* in the text of the explanation in Figure 4-8 the descriptions shown in Figure 6-1 would have been displayed in the LISP-CRITIC window.

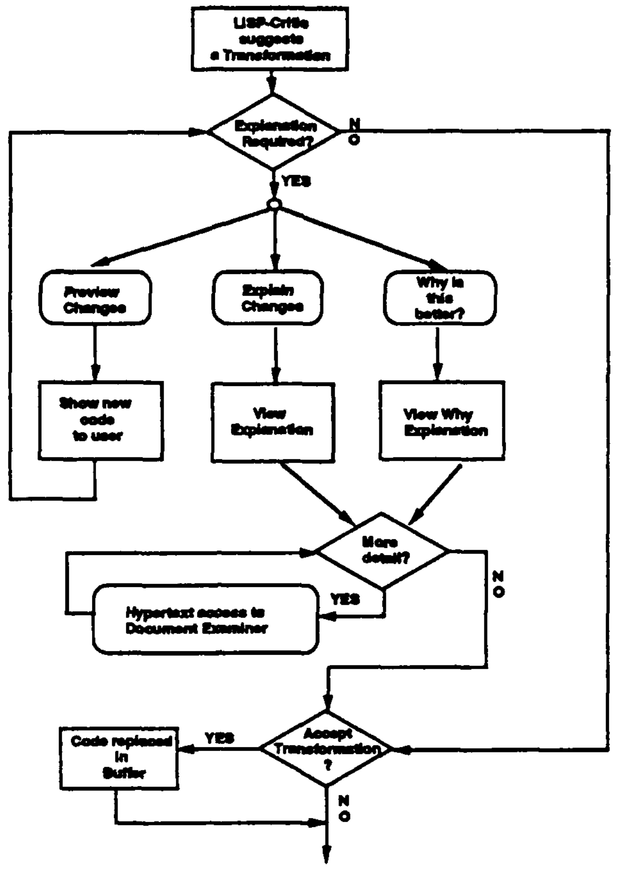


Figure 6-3: User Decision-Making Process in LISP-CRITIC

Explanations use the minimal approach and access the user model to determine what to explain. If users request more detail, the system positions them at an appropriate place in the hypertext information space; once there users have direct control of access to other hypertext nodes to obtain additional information. They terminate the explanation dialog when they are satisfied that they know enough to decide whether to accept or reject the critic's suggestion.

When LISP-CRITIC is invoked, the user's code is examined for ways to simplify it. In the first scenario, the system recommended that *cond* could be replaced by the *if* special form Figure 4-7. Before the user decided whether to accept or reject LISP-CRITIC's suggestion, he asked for an explanation. The system then determined what was required in order to understand this rule, and the aspects of that knowledge that the user lacked.

The explanation component interrogated the domain model and was provided a list of concepts underlying the *cond-to-if-else* rule. This list was generated by traversing the domain model beginning at the node representing the *cond-to-if* rule, and using the *dependent-on* links between domain model objects to accumulate the *concept set.* For the *cond-to-if-else* rule in the first episode of the scenario, (Figures 4-7 through 4-9) traversal of the domain generated a *concept set* of 13 items: *lists, symbolic-expression, evaluation, tests, variables, conditionals, scope, predicates, lisp-atom, arguments, false/empty-list/nil, true/non-nil, and functions.*

That set was personalized for the user in the scenario, to determine the subset of concepts to actually be explained. As discussed in Chapter 3, there are three levels d1, d2, and d3 at which a user can understand a given concept. The current implementation captures users' knowledge in terms of concepts that are well known to the user (d1), known to the user but not well (d2), and unknown (d0). For the concept set, the user model indicated (by their absence from the concepts-known slot in the user model) that the user has no knowledge (level d0) of six of them: *predicates, conditionals, tests, evaluation, symbolic-expression, and lists.* It indicated, based on their markings in the concepts-known slot, some knowledge (d2) of 6 others: *true/non-nil, false/empty-list/nil, arguments, lisp-atom, scope and variable*; and good knowledge (level d1) about just one, *functions.* That information was provided to the explanation component in three sublists, one each for d0, d2, and d1.

simple text fragments, differential explanations, and graphical-based explanations similar to those provided in the KAESTLE system [Nieper 83; Boecker, Fischer, Nieper 86]. Because the current text associated with each is stored using the Concordia hypermedia system, graphics can be integrated easily. Concordia is a hypermedia development and presentation system available on the Symbolics, the Document Examiner uses it.

The problems with current explanation systems are recognized; most efforts to improve them emulate human-to-human communication and, too frequently, attempt to provide a complete explanation in one-shot. Theoretical results in rhetoric, and discourse-comprehension together with empirical observations of human-human collaborative problem solving, indicate that trying to emulate human-to-human human conversational techniques may not be the best approach. This chapter has described the analysis behind a proposed approach to explanation-giving that tries to consider the constraints of the computer interface while taking advantage of capabilities and resources already available in the computational environment.

The suggested approach provides four layers of explanation for the advice given by a knowledge-based system. The first two layers, although they can help users understand, are not explanations in the strictest sense; they are detailed descriptions of what was recommended. The 3rd and 4th layers use a minimal-explanation approach to clarify the recommendations and expose the user to the underlying rationale for that recommendation. Minimalist explanations need a user model to tell the system what is necessary for the user to understand a domain entity. The highest layer, a rich hypertext space, allows users to explore details or examine concepts which they still do not understand. The user model is central to this proposed framework and fundamental for the explanation approach. The next chapter describes that user model.

Furthermore, within each sublist, concepts were ordered according to an implicit hierarchy within the *dependent-on links* in the domain model. The explanation component can ultimately use this information for reasoning about how to generate an explanation for the user, but the current implementation, using a simplified strategy for testing purposes only, selects the first three concepts in that filtered list *predicates, conditionals, and tests.* Ideally, the user finds the explanation adequate; but other concepts fundamental to understanding, or related to, these concepts are shown as mouse-sensitive objects displayed in bold. Selecting any of them will display either an explanation associated with that object in the domain model, or a description from the Symbolics Document Examiner (e.g. Figure 6-1).

The explanation system will not attempt to present explanations as though they were generated by an intelligent agent, but rather use combinations of straightforward, concise, prewritten sentences. What distinguishes this approach from most systems that use canned-text is the role played by the user model in the process of constructing an appropriate explanation.. Each part of the explanation can be chosen using the domain model, the user model, and the explanation strategies. The present implementation is an interim step to determine the efficacy of the user and domain model implementations; it is far from complete and future work should investigate how better to determine exactly which concepts to explain, and how to link descriptions of them together with information about the LISP-CRITIC rule of interest in a coherent discourse structure.

## 6.6. Summary

A number of approaches to structuring explanations could make use of the available domain and user model information. They include: prestoring

# CHAPTER VII

# USER MODELLING COMPONENT

This chapter describes the user modelling component for LISP-Critic. That component acquires the user model and represents the knowledge of each user in an object oriented structure; it provides access to that model, and insures it is persistent. The component was implemented in the Common LISP Objects Systems (CLOS). Access to individual models is provided via a set of generic interface functions; other system components know which functions to call to obtain whatever information from the user model that they might require. The user modelling component invokes the appropriate methods to actually access a user model's contents; it uses the domain model structure to insure that appropriate information is provided. The acquisition subcomponent provides direct methods based on episodes from the user-computer dialog, and indirect methods triggered by changes to individual user models. The primary purpose for which this model was developed is support for explanation-giving.

## 7.1. Design Approach

The design objectives for the user modelling component derived from a goal of supporting the explanation-giving framework discussed in the Chapter 6. The specific implementation approaches selected to achieve these objectives resulted from the analysis of other user modelling research, plus the requirements and framework for user modelling needed to support cooperative problem solving that were presented in Chapter 3.

### 7.1.1. Objectives

The user modelling component design had to provide support for explanation-giving, accommodate various acquisition techniques, and be able to represent a variety of information about the user. The model captures users' domain knowledge and supports implicit updating. An object oriented approach was selected for implementation in order to insure that the model is extensible, can be adapted to accommodate other techniques (such as stereotyping), and can be easily modified to represent new kinds of information about users (such as their preferences). The object oriented approach allows new methods to be defined on existing slots in the model and new slots to be added to the model class definition if necessary.

The component supports both implicit and explicit update. In this research the implicit update techniques were the primary focus however, the functions which modify the content of the user model are general methods designed to support other acquisition approaches as well.

A model that we are able to use only once, or during just a single programming session, is not acceptable; it needs to be retained between sessions and reused the next time a user accesses the system. Methods that save the contents of the model at the termination of a user-system dialog and start with that model the next time a user accesses the system are included in the component.

The model must support changes in users' knowledge over time. It is in this sense that the model is dynamic; its contents change as a user's knowledge improves. In this regard, the emphasis in developing inference methods was on improvements in users' knowledge in the domain. The problem of how to modify the model when users forget something they once knew is an important issue but was not addressed at this point in the research. That the model is at best an approximation of the user implies that the modelling component must be designed to include techniques for improving on that approximation. Whatever information is available to enhance the model has to be used to best advantage. Specifically, what a model represents about any specific user should get better during subsequent interactions between the system and that user.

Three different approaches to representing the users of LISP-CRITIC were considered and, in some cases, partially implemented: classification methods, stereotyping, and an overlay of the systems domain knowledge (the LISP-CRITIC rules). Classification categories and stereotypes are similar, but for discussion purposes they are considered separately, as were the attempts to use them.

Initial attempts to model the user with classification methods in support of explanation-giving [Frank, Lynn, Mastaglio 87] met with only limited success. The problem with classification methods were two-fold. The canned-text explanations directed at a particular level of expertise were found to be unsatisfactory during user testing. They were often too basic to satisfy the user, or too difficult, using concepts not yet understood by a particular user. Part of the problem is how to classify a user into one of a set of prespecified categories. The ones used (novice, intermediate, and expert) did not appear to capture individual expertise in a satisfactory manner. The second problem with classification methods is that they are not fine grained enough to provide adequate fidelity in their representation of individuals. This problem was confirmed by an informal study in which the group of local LISP programmers, generally considered to be the experts, were asked to respond to a questionnaire about their use of, preference for, and opinion about teaching certain language constructs. The responses varied widely, indicating a significant difference in what these *experts* knew and preferred. When establishing the expertise categories in a domain one has to face the same problem as determining the contents of appropriate stereotypes.

A schema for model acquisition using stereotypes of LISP programmers was developed [Fischer, Mastaglio, Rieman 89]. It was based on Rich's approach to stereotyping [Rich 79], and showed promise as a way to leverage analysis of the content of users programs to *stereotype* them, and from that stereotype infer additional characteristics indirectly. Part of this work was a study in which human LISP experts were provided a program and asked to assess the expertise of the programmer. Protocols observations in this study showed that the human experts either looked for or noticed what we called "cues" in the code; cues triggered inferences about the expertise level of programmers who wrote them. This idea of identifying cues in the context of a user's work is something that was carried into the acquisition methods finally implemented. The methods were developed and partially implemented but this line of research had to deal with the problems of what stereotypes to use, where they come from, and how to insure their validity.

Representing a user's knowledge as an overlay of the existing rule base was also considered. It was found to be useful for guiding critiquing (e.g., making it more efficient by enabling or disabling rules). However, a model that only captures user knowledge in terms of the LISP-CRITIC transformations is inadequate; it cannot provide the required support for explanation-giving. There are slots provided in the model for representing rule level information about a user; the implementation therefore does provide such an overlay of the rule base for use in situations where it is of value.

The limitations encountered in considering these other approaches provided a key objective for the design of the user modelling component, to implement a model that represents user knowledge of the domain at a level that is of fine enough granularity to support the explanation of domain entities. The basis for that representation turned out to be the same deep, domain model, as required to accomplish explanation-giving.

### 7.1.2. Implementation Approaches

An object-oriented representation allows the model representing each individual to be idiosyncratic but for all the individual models to confirm to a common format. This requirement, coupled with the need to support easy access and the changing of separate instances of the entire class of models, led to selecting that object oriented representation. The structure of the individual models is defined as a class, and communicating with the instances of that class is facilitated through methods defined on it. The object oriented representation can also support potentials enhancements to the overall user modelling component. The actual language chosen for the object-oriented representation is the Common LISP Objects Systems (CLOS).

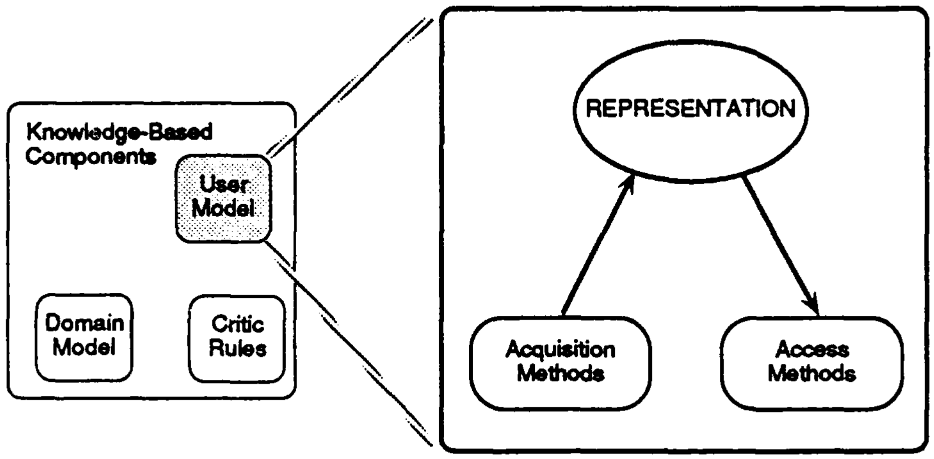
The need to achieve a representation of users' domain knowledge in a more abstract or conceptual form than the LISP-CRITIC rules resulted in basing the user model on the conceptual domain model described in Chapter 5. The domain model did not pre-exist, rather the motivation, in part, for developing it was to provide support for representing users' domain knowledge. The research process concurrently developed both the user modelling approach and the domain model.

To support dynamic update without explicitly querying the user, the implementation makes use of information available in the context of the human-computer dialog. Dialog, in the sense used here, refers to any action that occurs between the system and the user. The idea that the model should be implicitly enhanced based on the dialog led to an analysis of the content of these interactions. This work views the dialog as consistirg of a series of episodes. From a hypothetical scenario of a user interacting with a completed system, the following kinds of episodes in LISP-CRITIC were identified:

- user requests and receives an explanation of critic suggestion
- user decides to accept (or reject) critic suggestion
- user accesses additional on-line documentation to help clarify an explanation
- user informs LISP-CRITIC to disable (enable) a rule
- user adds a personal comment to an argumentation database about the applicability or usefulness of a transformation in the rule base

The implementation uses what takes place in those episodes as a primary source for triggering system inferences about the users' knowledge. One basis for this approach is the fact that users apply their knowledge in constructing their ''side'' of the dialog, therefore the actions they take provide evidence about what they know. Just as significant is that when users act as mere receivers of information there are cues here as to how the user's knowledge should be changing. Specifically, they should now have command of the domain concepts explained by the system. In this second case, users ''learn'' from what the system tells them — this is the basis for the some of direct inference methods that will be described later.

The above objectives and approach guided the manner in which the user model is represented, acquired, and accessed. An architectural overview of the user model component in Figure 7-1 shows the separate subcomponents and functions of LISP-CRITIC's user modelling component; it corresponds to the general architecture developed and shown in Figure 3-2 and is an internal view of the user-model as one of knowledge-based component in the overall system diagram that was shown in Figures 4-4 and 4-5. The representation subcomponent will be discussed next.



The user modelling component is one of the knowledge-based components of LISP-CRITIC. Data are indicated with an oval, collections of processes with rectangles, data flow with directed arrows. The three subcomponents are: a **representation** in object-oriented form (CLOS), **acquisition methods**, and **access methods**. Acquisition methods modify the representation — information flows from them to the representation. Access methods extracts information from the model — information flows from the representation.

Figure 7-1: User Model Component for LISP-CRITIC

## 7.2. User Model Representation

The representation is designed to capture a variety of information about the user. An example instance of the class user model is shown in Appendix B; it is the state of the user model at the conclusion of the three dialog episodes in the Chapter 4 scenario. The interesting part of the model (with respect to this project) are those slots that represent the user's expertise in the domain of LISP: *rules-known*, *functions-known*, and *concepts-known* slots. Conceptually these record the coloring of the domain model graph for the user. An approach that was considered for the representation was an overlay of the domain model; the overlay representing those domain entities a user knows. But the model needs to also capture the levels of the users' knowledge according to the classification framework described in Chapter 3, the approach implemented is to model the user as a coloring of the graph representation of the domain. To demonstrate this and show how that coloring changes over time we will use the previous interaction scenario.

According to the conceptual framework, a user's knowledge about a given concept can be categorized into one of three levels: d1, d2, and d3. That framework was shown in Figure 3-1; it provides a useful scheme for approximating the expertise levels of users. For this research, we adapted it to represent user knowledge of a programming language. For LISP the regions in the graph are interpreted as follows:

$D_1$: The subset of LISP functions and underlying language concepts which users knows and incorporate into their programs regularly, they understand these quite well.
$D_2$: The subset of concepts which users know and will use, but only occasionally. They does not know the details nor perhaps even the specific syntax of functions in this region but are aware of their existence and have a general understanding of their purpose. Users might refer to a LISP text, on-line documentation (e.g. Symbolics Document Examiner), or consult a colleague for help in coding functions in this class. Concepts in this class are less well understood by users than those in $D_1$ but still can be considered a part of their active knowledge.
$D_3$: The conceptual model of LISP held by a user. The concepts and functions that they think exist in the language; this region also includes misconceptions.
$D_4$: The domain knowledge of LISP.

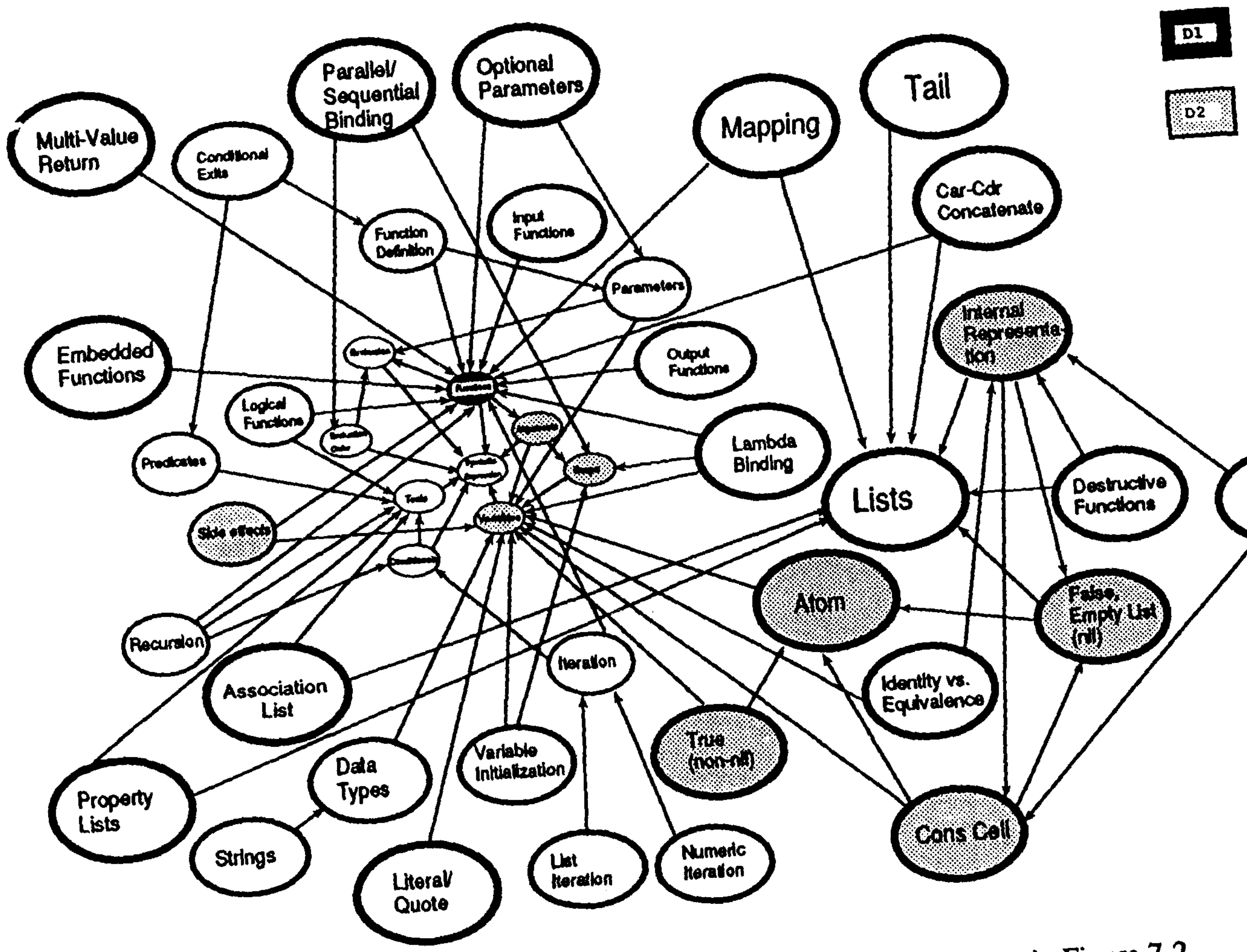
The inference methods that were developed are only able to recognize domain knowledge in d1 and d2, so a simplified scheme was used. It conceptually marks entities in the domain as: well known to the user (d1), known to the user but not well (d2), and unknown (d0). Entities at level d0 are not explicitly listed in the user model but the component infers they are unknown to a user by their absence from the appropriate slot. There is another condition that could be the reason that a domain entity is at d0; it may be that the system has not yet encountered anything to trigger an inference about the level of knowledge — it just does not know how well the user knows the entity, if at all. Discriminating between these two situations could be accommodated with methods that test or query the user. There are situation in which this distinction would probably be beneficial, tutoring for example, but for explanation generation, the processing required to distinguish between them does not provide enough additional information to make it worth the effort, and the present implementation treats both situations identically.

```
Summary data for user model for SCENARIO-USER

Following concepts in D1

FUNCTIONS

Following concepts in D2
INTERNAL-REPRESENTATION
SIDE-EFFECTS
CONS-CELL
VARIABLES
SCOPE
LISP-ATOM
ARGUMENTS
FALSE/EMPTY-LIST/NIL
TRUE/NON-NIL

Following functions in D1

Following functions in D2

Following lcr-rules in D1

Following lcr-rules in D2

Rules-fired by name and number of firings
NIL
```

This is the state of the user model at the beginning of the scenario. The concepts came from user's self-ratings on the initial questionnaire.

Figure 7-2: Initial User Model

Recall that the user model is conceptually a coloring of the graphical domain model, that the graph has concept, function, and LISP-CRITIC rule layers, and that determining what to explain to a user involves extracting information from that user model, the appropriate concepts required to understand a transfor-



Graph colored to represent the user's initial knowledge state, also shown in Figure 7-2.

**Figure 7-3:** Coloring of Conceptual Graph for Initial User Model

mation. Figure 7-3 shows the initial coloring of the concept layer for the scenario-user. Concepts known to the user model are shaded appropriately, depending on whether they are in d1 or d2. Unshaded concepts are at level d0. An equivalent representation that shows the domain knowledge slots for the user model is textually displayed in Figure 7-2.

In the scenario the system suggested a transformation based on the *cond-to-if-else* rule, the situation shown in Figure 4-7. A traversal of the domain model graph generated the 13 items for explanation discussed in Chapter 6. It is now up to the user model to filter that set to provide assistance to the explanation component about what should be explained and how. Most significant is how well (at what level) the user knows each concept. That information is provided to the explanation component in three sublists, one for each level (d0, d1, and d2), which are then used to determine the explanation strategy and presentation approach.

During the dialog episode, the user was satisfied with the explanation and *accepted* this suggestion. The content of this dialog episode was used to update the user model. The cues from this episode which are important to the user modelling component are: the receipt of explanations about certain concepts (e.g. *conditionals, predicates* and *tests*) and the user deciding to accept the *cond-to-if* transformation. Cues triggered direct inferences that changed the user model and these changes in turn triggered indirect inferences that will be explained in the next section. A portion of the updated model is shown textually in Figure 7-4, and its associated graph coloring for the concepts layer in Figure 7-5. The system's design incorporates techniques that recognize that a model has been constructed for this user and makes use of that version of in subsequent dialogs between LISP-CRITIC and this user.

The model was saved between sessions and reused by LISP-CRITIC when

Summary data for user model for SCENARIO-USER

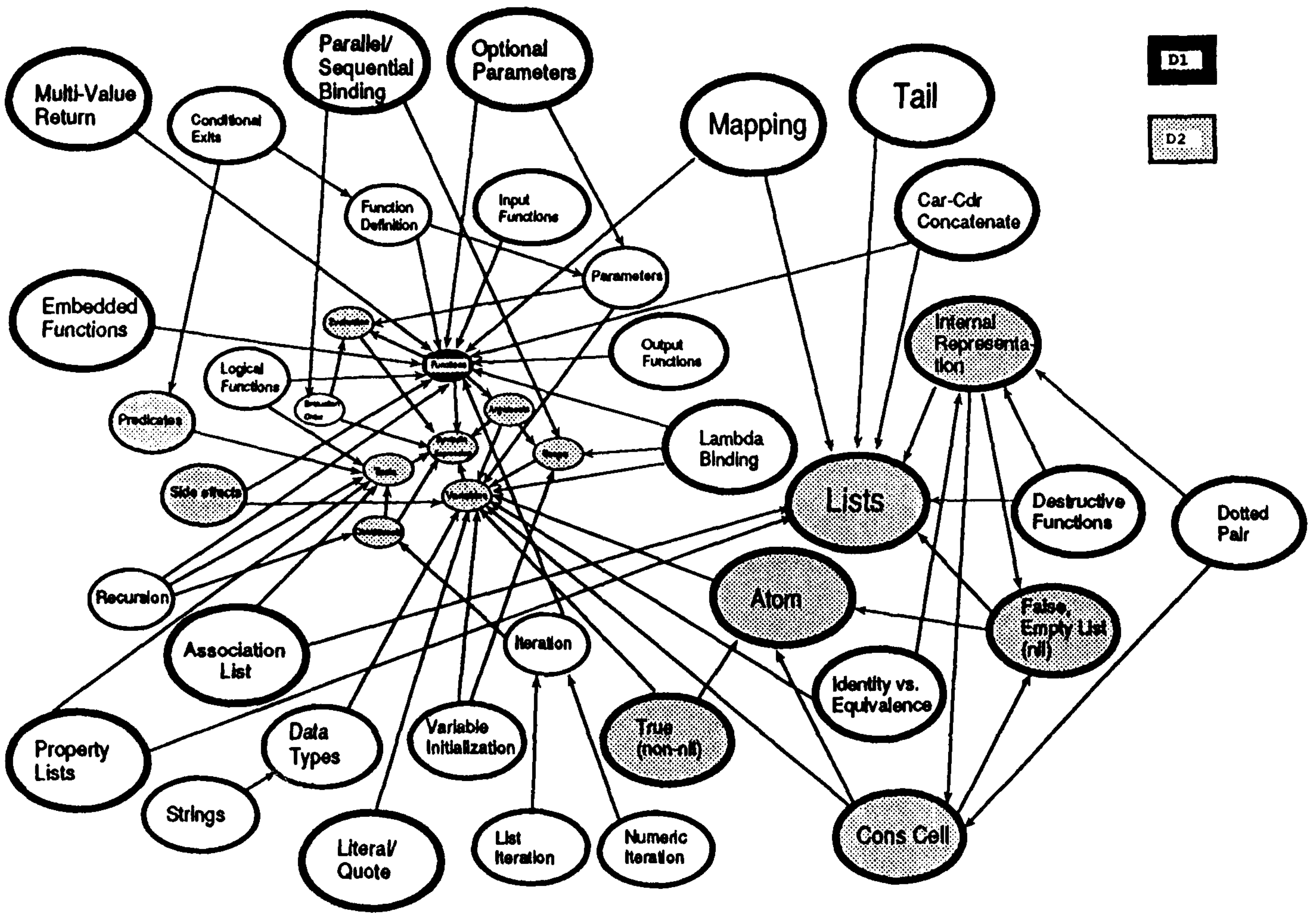
```
Following concepts in D1          Following functions in D1
FUNCTIONS
                                  Following functions in D2
Following concepts in D2          IF
SYMBOLIC-EXPRESSION               COND
LISTS
EVALUATION                        Following lcr-rules in D1
TESTS
CONDITIONALS                      Following lcr-rules in D2
PREDICATES                        USER::COND-TO-IF-ELSE
INTERNAL-REPRESENTATION
SIDE-EFFECTS                      Rules-fired by name and times fired
CONS-CELL
VARIABLES                         USER::COND-TO-IF-ELSE
SCOPE                               TIMES-FIRED    1
LISP-ATOM                           TIMES-ACCEPTED 1
ARGUMENTS                           TIMES-REJECTED 0
FALSE/EMPTY-LIST/NIL
TRUE/NON-NIL
```

The contents user of the model after the first dialog episodes. Changes to the content, when compared to Figure 7-2, are a result of user actions during the episode triggering inference methods that update the model.

Figure 7-4: User Model Contents after First Dialog Episode

the user next requested critiquing. In the scenario, that next episodes occurred when the user requested LISP-CRITIC to look over the code for function *test*, as shown in (Figure 4-10). The first transformation recommended on this code is based on a rule called *de-morgan*, the user requested that this rule be explained. The user model again provided the information to guide the process of developing and presenting that explanation.

When the user accepted the suggestion, his user model was again updated. The system suggested a second transformation for this function, one based on the *cond-erase-pred.t* rule; it was explained in Figure 4-11 based once again on



Graph coloring for user's concept knowledge after first episode, also shown in Figure 7-4.

Figure 7-5: Coloring of Conceptual Graph for User Model after First Dialog Episode

information from the user model about how well the user knows the underlying concepts for this rule. It happens in this case that the concepts incorporated into the explanation belong in the user's d2 level because none of the *concept set* underlying that rule were unknown (at level d0). A third suggestion, based on the *cond-erase-t.nil* rule triggered and the user accepted it, but without an explanation because the prerequisite concepts are similar to those already explained for the previous rule. The user model was dynamically updated throughout the dialog. Each time information was shown or a decision made, inference methods triggered. The domain knowledge portion of the user model at the conclusion of the scenario is shown in Figure 7-6; its associated graph coloring for the concept layer in Figure 7-7. The final internal representation for the instance of the class *user model* that represents *scenario-user* is shown in Appendix B.

Each individual's model is an instance of the class user model. A user's knowledge about each category of domain object is captured in slots in that model. For example one slot contains the LISP functions a user knows as well as their level. Other slots contain personal information and data about the user's background. The information and data slots could be filled during initial start-up of LISP-CRITIC for a particular user (i.e., the first time it is ever invoked by them) by several explicit acquisition techniques. Specific the methods to do this have not yet been implemented. Instead this research concentrated on developing the implicit inference methods that modify the content of those domain knowledge slots during the course of using of the system.

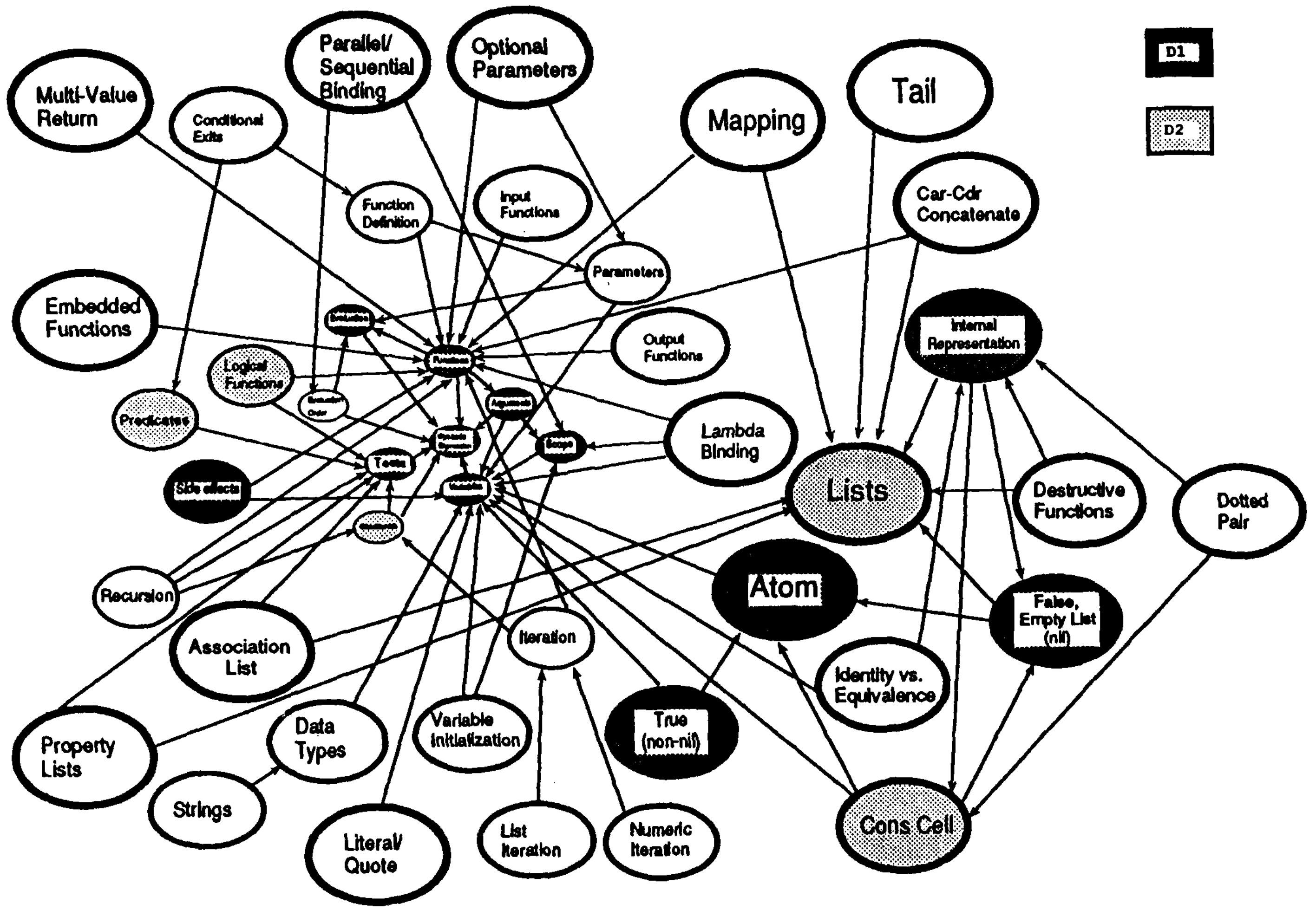
## 7.3. User Model Acquisition

The user modelling component contains a collection of methods that infer which domain concepts belong in the user's model, and the level of that

```
Summary data for user model for SCENARIO-USER

Following concepts in D1
SYMBOLIC-EXPRESSION
EVALUATION
TESTS
INTERNAL-REPRESENTATION
SIDE-EFFECTS
VARIABLES
SCOPE
LISP-ATOM
ARGUMENTS
FALSE/EMPTY-LIST/NIL
TRUE/NON-NIL
FUNCTIONS

Following concepts in D2
LOGICAL-FUNCTIONS
LISTS
CONDITIONALS
PREDICATES
CONS-CELL

Following functions in D1

Following functions in D2
NULL
NOT
OR
AND
IF
COND

Following lcr-rules in D1

Following lcr-rules in D2
USER::COND-ERASE-T.NIL
USER::COND-ERASE-PRED.T
USER::DE-MORGAN
USER::COND-TO-IF-ELSE

Rules-fired by name and times fired

USER::COND-ERASE-T.NIL
   TIMES-FIRED    1
   TIMES-ACCEPTED 1
   TIMES-REJECTED 0

USER::COND-ERASE-PRED.T
   TIMES-FIRED    1
   TIMES-ACCEPTED 1
   TIMES-REJECTED 0

USER::DE-MORGAN
   TIMES-FIRED    1
   TIMES-ACCEPTED 1
   TIMES-REJECTED 0

USER::COND-TO-IF-ELSE
   TIMES-FIRED    1
   TIMES-ACCEPTED 1
   TIMES-REJECTED 0
```

The state of the user model after the second (final) dialog episode.

Figure 7-6: User Model Contents after Second Dialog Episode



Graph coloring of user's knowledge at the end of the scenario, also shown in Figure 7-6.

Figure 7-7: Coloring of Conceptual Graph for User Model after Second Dialog Episode

knowledge. The information that triggers these methods is passed to the user model by other system components using interface functions; in turn, indirect methods are internally triggered by those changes.

This research investigated how a user model for a cooperative problem solving system could be enhanced incrementally over multiple dialog episodes using implicit methods. To review the point made in Table 3-1, one finding was that the system can use two different classes of update methods:

- Direct methods that use a specific dialog item to trigger singular changes in the user model information. They make changes to the user model as a *direct* result of information the user receives from the system, or of an action the user takes.
- Indirect methods that are triggered whenever a change to the user model contents occurs, they cause further updates to the model, one might view these as internal demons.

When a dialog episode triggers direct methods that change the user model, these changes in turn trigger indirect methods that also change the user model. Indirect methods are implemented as after-methods on slots in the user model; this insures that they get run whenever a slot change occurs. An example of this chain of inferences happens in the first episode in the scenario. The user received an explanation of the transformation that included a description of the concept of *conditionals*. That concept is marked in the user model at level d2 based on a direct inference method — when the system explains a domain entity it assumes the user now knows that entity. An indirect inference method is triggered because of the change to the *concepts-known* slot. That indirect method interrogates the domain model and determines that a prerequisite piece of knowledge for understanding the concept conditional is *symbolic-expression*. This causes a change to the model that adds symbolic expression to *concepts-known* slot at the d2 level.

## 7.3.1. Direct Methods

The direct methods are an adaptation of related work on implicit user model acquisition in dialog advisory systems that was discussed in Chapter 3. The implicit acquisition rules developed in [Kass 88] are based on using natural language dialogs. Here dialog is used in a more general way. As previously described, it means any of the different interaction episodes that occur between a user and the system. Using this view it was possible to develop our own set of direct implicit methods by modifying the implicature rules, these methods fall into four major categories:

- techniques based on user decisions,
- methods triggered by information provided to the user,
- those triggered by optional actions on the part of the user, and
- ones activated when users access the hypertext information space.

These categories of information are available to the system through tracking its dialogs with the user. Appendix C shows the implementation code and associated descriptions for each specific type of direct inference method. Here we will describe, in general terms, each category and the rationale behind them.

Dialog episodes, between the user and LISP-CRITIC, terminate with a user decision (unless the session is aborted) to accept or reject the critic's advice. For either decision users requires the same type of knowledge (or level of understanding.) In both cases the system makes the same inference. A decision to accept a suggestion made by LISP-CRITIC causes the system to mark the rule behind the transformation as known to the user at level d2. A user's decision to reject a rule is handled similarly. This ignores the situation where users *reject* a transformation because they do want to bother with it and just go on. The system implementation does provide a method to abort the interaction and it is assumed that users are sophisticated enough to use that command appropriately. If the system served a group of users less computer-knowledgeable than programmers, that assumption could be called into question.

In the dialog the system presents information to the user in the form of explanations, an event that triggers a direct inference that users know the explained entity. When the system explains a LISP concept, the level for that concept in the user model is marked as d2. In the first scenario explanation, this is how the concept *conditionals* came to be marked d2. If a concept just explained was already marked at level d2 then its level is improved to d1. In the second explanation episode the concepts *internal-representation* and *side-effects* migrated to the d1 level in this manner. Similar direct changes occur when functions or rules get explained.

When users encounter explanations that are unsatisfactory they can access a hypertext information space as a fallback technique. Their selection of a mouse sensitive word is also information that can be used to update the user model. The system can capture the selections and relate them to domain model entities where possible. This capability was implemented but has not yet been tested to determine how often the mouse sensitive objects selected match the entities (functions or concepts) in our domain model. When a match is found the system marks the domain entity at level d2 in the user model unless it is already at level d2, in which case its level is improved to d1. The assumption on which this method is based is an optimistic view that users actually read and understand information provided by the document examiner; an assumption that could be subjected to further testing. When a user gets an explanation of the selected domain entity, there is an assumption that they are therefore now familiar with that entity. Their user model should now show that information. The domain model is not necessarily complete and there are may be inconsistencies in the terms used. However, because both the domain model and the document examiner generally use accepted terms for LISP concepts from [Steele 84], the correlation should be high enough to make this a useful approach.

The system is being extended to allow for several optional actions — actions that are not required of users, but allow the system's behavior or the documentation base to be modified.

- Users can change the action taken when a rule fires; they can tell system to ignore it (always reject this transformation) or automatically to make the suggested change to the LISP code (always accept this transformation.) The claim here is that users must understand a rule before they are able to modify what happens when that rule fires. It is analogous to the specific decision to accept or reject a given suggested transformation; the user decides to accept or reject it in all cases. When users change a rule status, the level of that rule is caused to be set to d2 in their user model.
- A recent extension to the system allows users to associate personal comments with any rule in the documentation space. An example might be a programmer who does not like the *cond-to-if* rule because "it creates code that is less general." This argumentation can be attached to the rule and associated with that programmer. These comments then become available to anyone else who uses the system, who can also add their own comments, perhaps disagreeing with the pre-

vious author, because "the argument misses the point that the rule is intended to made the code easier for other programmers (ones involved in maintaining it in the future) to understand." When users involve themselves in generating such argumentation, the system should infer that they understand the rule quite well and consequently mark it at level d1 in the user model.

Direct methods change the user model using explicit information observed in the dialog. These methods alone are inadequate for developing a useful model that becomes sufficiently complete in a reasonable amount of time. Additional methods that leverage this information, in the spirit of stereotypes, were needed. The structure of the domain model provides the basis for such an additional class of methods that do indirect implicit updating.

### 7.3.2. Indirect Methods

The idea for indirect methods developed while implementing the conceptual domain model when it was observed that the links in the model capturing prerequisite knowledge for the domain entities could provide a source for implicit acquisition. These prerequisite were established for use in explanation, but the idea for using them for implicit acquisition resulted from noting that they may tell us something about what the individual knows about the domain — in the spirit of the notion, used in the UMFE system, that user knowledge "propagates" through a set of concepts. The prerequisite relationships indicate that if users knows a given concept, they probably know its *dependent-on* concepts. Based on this observation, there are methods that trigger whenever a change occurs in the knowledge level of an entity in a user model. Therefore the indirect methods leverage the domain model structure, allowing the system to enrich its model of a user without waiting for explicit evidence about each domain entity. The indirect implicit methods belong to a class of model building techniques that includes stereotypes and the short cut methods methods used in human-to-human cooperative problem solving.

Models of communications partners are based on more than the direct evidence provided directly from dialog. Computers, as Suchman pointed out [Suchman 87], do not have access to the rich set of information available to another human partner, therefore this research looked into ways to accomplish a similar enrichment of the model by using available resources. One such resource is the linkages between entities represented in the structure of the domain model, these links form the basis for the indirect update techniques in LISP-CRITIC. A change occurring in the representation for the user's domain knowledge can be used to trigger further changes to the domain model based on the prerequisite knowledge for the entity just changed. The indirect methods infer how well those prerequisites domain entities are known and put that information into the appropriate slot in the user model. Indirect methods exist for each class of domain entity: LISP-CRITIC rules, LISP functions and LISP concepts. Their implementation is shown in Appendix C.

There is a set of functions associated with each LISP-CRITIC rule, these functions are each linked to that rule via the *functions-in-rule* relationship in the domain model; they are the functions used in either the left hand side or right hand side of the rule. Often, rules are also based upon certain LISP concepts in the domain model. When the level for a rule is changed in the user model, indirect methods modify what the user model has to say about how well the user knows those associated functions and concepts. If the rule has been set to level d2 our indirect methods infer that the functions in that rule are also known at level d2, and nothing is inferred about the concepts behind the rule. When the level of a rule is set to d1, both the functions in that rule and the concepts on which it depends are set to level d2 in the user model. A case could be made, in retrospect, that a different level might be inferred for the functions contained in the left hand side of a rule; after all, programmers actually use these in their code, or perhaps that the functions in the right hand side are not yet a part of a user's LISP knowledge and should not be added to their model.

For a LISP-CRITIC rule, the domain model provides information about the functions in that rule and further traversal of the model beginning with those functions provides a list of prerequisite concepts. When a direct method modifies the *rules-known* slot in a model, an indirect inference is triggered by an after-method on that slot. For example, for the *cond-to-if* rule in the first explanation dialog in the scenario, first that rule is added to the appropriate user model slot, then the indirect methods also add the *cond* and *if* functions to the user model and, in turn, the concepts *functions* and *arguments.*

The prerequisite to understanding a LISP function are its *dependent-on* domain concepts. When the level at which a function is known is set to d2, its *dependent-on* concepts are also set to level d2. If the level of a function is set to d1, those concepts are also set to d1.

Concepts themselves are linked to one another via the *dependent-on* relationship. When the level of a concept is changed to d1 in a user model, its prerequisites in that user model are also set to d1. In the situation where the level of a concept is changed to d2 its *dependent-on* concepts are marked at d1 when they were previously marked d2 in the model. If those *dependent-on* concepts were not already in the model (conceptually marked d0) then they are added to it with a d2 marking.

The domain model implementation also contains *related* links that identify functions, concepts, or rules that are similar to another function, concept, or rule. This class of links could be used as the basis for a class of weak inference meth^ds, such as predicting the ease with which a new entity could be introduced to the user. In the current implementation, the similarity relationships were not exploited in the present user model acquisition methods. This possibility is mentioned here to show how the theoretical approach of using the domain model structure to infer useful information extends beyond the techniques that were actually specified and implemented.

The implemented indirect methods were designed conservatively; they are neither complete nor perfect. Their shortcomings, and indications about how to improve them came out during an evaluation that is discussed discussed in Chapter 8. There are two results from this research of general interest and utility:

- It is possible to define a class of implicit user model acquisition approaches that are indirect. These are leverage techniques that use information about users that is not directly observed, but is derived from other knowledge (knowledge of a domain model, stereotypes, or etc) to indirectly enrich the models of those users.
- We can use the deep conceptual domain model that is needed for proper explanation as a source for a set of such indirect implicit user model acquisition methods.

The methodology followed provides an approach that can be used for developing user model acquisition techniques to serve other situations.

In summary, to show how the direct and indirect acquisition techniques work together, let us review a portion of the scenario. The concept *tests* is marked at the d2 level in the .:er model shown in Figure 7-5. This resulted from a direct inference based on a method that claims when a domain entity is explained to a user, that user is now aware of its existence and has a fundamental understanding of it — the user knows of the concept but is not proficient in applying it in every circumstance. The domain model also tells us that, for the concept *tests*, a prerequisite (according to the *dependent-on* links in the domain model) is the concept *symbolic-expression*. Therefore, an indirect inference places *symbolic-expression* at level d2 for this user. Similar direct and indirect user model acquisition methods fire for the *cond-to-if-else* rule, its underlying functions, and its *dependent-on* concepts. More domain entities in the user model in Figures 7-4 and 7-5 get marked at the d2 or d1 level as the result of indirect implicit methods than as a result of direct methods.

## 7.4. Access to the User Model

In developing the architecture for the user modelling component one objective was to insure that other system components can easily access the model. Another consideration was to have a model that supports modification to incorporate additional information. Significant theoretical issues or results were neither addressed or discovered in this aspect of the work, but played a role in deciding to use an object-oriented approach. Access methods support the current explanation component framework while providing access to information likely to be of value for other purposes.

The interface functions support the explanation strategies described in Chapter 6. The user model can be queried to determine which of a set of domain objects a user knows or does not know. The interface functions are shown in Appendix D; some examples are ones that determine how well a user knows a domain entity (i.e., at what level), and whether a domain object was previously explained.

An attempt was made to conjecture the additional information a user model might be asked to provide, and include in the framework functions that might be needed in other situations. Slots in the model record all rules-fired during previous dialog episodes and the number of times a user has invoked the critic. Other such information includes the user's goals and previous programming language experience. The goal can be acquired by explicit query of users during their initial session with LISP-CRITIC; currently it is defaulted to "simplifying" code to make the program easier for others to read and maintain. Previous programming experience in other languages can also be obtained through such an initial information-gathering session or interactive questionnaire.

The system allows the user to modify the manner in which the system presents information and the default action taken when a rule fires. It supports end-user-modifiability. The user model contains slots that record such user preferences and make them available to other system components.

A number of access requirements are internal to the user modelling component itself, the instance-slots can only be directly updated by the modelling component. The component receives information from other components about user actions or explanations, and determines how to use that information. It decides what additions or modifications to make to the user model and calls the internal methods to make them. The user modelling component is notified when a session terminates normally (is not aborted) and a set of cleanup actions invoked. These functions save the user model's current contents in a file so that information is not lost when the user logs out or the system is rebooted, and can be used during subsequent log-ins when LISP-CRITIC gets invoked.

When LISP-CRITIC is called, the system determines whether the user has a model already loaded into the current environment; does not, but the system saved one during a previous session; or have not previously used LISP-CRITIC. In the first case, the system does nothing; in the second case, it loads the most recent version of the user model; and in the third case it must initialize a model for this programmer. It is in the last situation, when the user model is initialized, that the system could use explicit query methods to gather start-up and background information about the user. No explicit methods or start-up user questionnaire were implemented in this research; such an implementation would not add to the theoretical ideas developed here.

## 7.5. Summary

Specification and implementation of the user model component was a major portion of this research effort. The design objectives were established with a goal in mind of generating a framework ultimately able to support explanation in any cooperative problem solving system. Based on these objectives, specific implementation decisions were made for the user modelling component architecture for LISP-CRITIC. That component uses an object-oriented representation scheme for the user model, a set of access methods implemented as generic interface functions that can be called by other components to interrogate the user model, and a set of implicit acquisition methods. The latter are separated into direct methods that use specific information to trigger an inference about the user's domain knowledge and indirect methods that percolate changes through the user model based on relationships between domain entities captured in the domain model graph structure.

The outcome of this theoretical development and implementation is a framework of user model acquisition techniques. That framework can be used to analyze and design user modelling components. The update methods implemented here were the subject of an evaluation that will discussed next.

# CHAPTER VIII

# EVALUATION OF THE USER MODEL

## 8.1. Introduction

The user model developed in this research project was evaluated in two ways. Programs written by students learning LISP were processed using LISP-CRITIC, and the individual user models of each programmer saved. The models were compared to one another, attending particularly to the changes that took place in them over time. Comparisons of the contents of the models at different times permitted an evaluation of the behavior of the user modelling system, and indicated potential system improvements. The models were also compared to questionnaires that users completed prior to each of the three programming assignments. The data collection process is first described, then the results of analyzing those data; finally I discuss what these results imply regarding modifications and enhancements to the system.

This particular evaluation was not a usability study; it did not attempt to assess either the overall effectiveness of LISP-CRITIC nor the ability of programmers to use it effectively. These types of studies, as discussed briefly in Chapter 2, were done for previous systems versions and helped determine the capabilities we want to provide in the current system under development. In this work the emphasis was on developing an approach to user modelling; therefore; the evaluation focused on the effectiveness of the user modelling component. To insure consistent and useful test results, it was necessary to control the test scenario conditions to which the the user model acquisition subcomponent was subjected. The current version of LISP-CRITIC does not have a fully operational explanation component based on the framework described in Chapter 6; because the presentation strategies have not been fully defined or implemented, a total system test was precluded. Informal studies in which other researcher were asked to "test out" the system were conducted, and the results integrated into the interface design during development. That user feedback guided decisions about menu options and names, the type of explanation, and what capabilities should be provided for users who want to modify the system.

If the user modelling acquisition methods work properly then the contents of the user models should both change over time to reflect first, improved representation of a user, and second changes in the students' knowledge itself because they were engaged in a learning process. Controlling for or separating-out these two phenomena was not possible under the scenario in which this evaluation was conducted. However, over time, the user models when compared to earlier ones should reflect a richer representation of the student's knowledge state. Secondly, in spite of the limitations of a self-assessment methodology, there should be some correlation between the model contents and the actual state of students' knowledge.

## 8.2. Data Collection

LISP programs written by undergraduate computer science students were collected throughout the Spring 1989 Semester. These students were enrolled in CS3202, Introduction to Artificial Intelligence, a survey of artificial intelligence techniques which provides an introduction to programming in LISP. Ten students volunteered to participate in the study. We collected the programs which were submitted to fulfill three class assignments. Classroom lectures on LISP preceded the assignments; the lectures introduced LISP syntax and functional programming techniques.

The three assignments were spread over the course of the semester with approximately three weeks between due dates. The total code for all three assignments averaged about 250 lines per student, including comments. The subjects also completed questionnaires, an example of which is shown in Appendix E. The questionnaires accumulated personal and experiential background as related to programming and asked the students to assess their own knowledge of LISP concepts and functions. Three questionnaires were administered, one before each of the programming assignments. The questionnaires asked the student to rate their knowledge of 18 concepts from the LISP domain model (see Chapter 5) and 30 LISP functions. The rating categories were designed to approximate verbally the levels of user knowledge that were discussed in Chapter 3. The descriptive rating categories used on the questionnaire were:

1. the student could define the concept or write an expression using the function (d1),
2. for a concept, this rating means they were familiar with, but could not precisely define it; and for functions that they knew of its existence but would have a problem using correct syntax (d2), and
3. they were not aware of the concept or function (d0).

You might notice that it was possible, on the questionnaire, for students to also classify functions into a category indicating that they had heard of the function but were not entirely sure of its purpose and effects. Functions in this category fall into the student's d3, these data were not used in the evaluation because the user model has no techniques for classifying knowledge of domain entities at that level.

LISP-CRITIC was run on each student's programs in two "test scenarios", one using an "accept" condition and the other using an "explain" condition. Under the "accept" condition, the simulated response to each LISP-CRITIC recommendation was to accept it without requesting an explanation. In the "explain" condition, the scenario called for the user to request an explanation the *first* time a particular LISP-CRITIC rule fired; in this scenario all suggested changes were also accepted. Scenario conditions were established to control the conditions so as to limit the types of user actions to which the user model acquisition subcomponent was exposed. For example, access of the hyptertext documentation space was not called for in any test scenarios.

Programs from four of the students were run through LISP-CRITIC under each test scenario conditions. These four were selected because they completed all questionnaires and provided completed working programs for all three assignments. Six user models for each student were captured; a set was saved after each one of their three programs had been run through a scenario. In the test conditions, the initial (or startup) user models were empty. The user model accumulated during an episode under one of the scenario conditions was retained and used for the succeeding episode under that same condition. For example, the user model the system developed for user1 in programming assignment one under the *accept* condition was the model with which the system began the test scenario dialog for user1 about programming assignment two under the *accept* condition.

## 8.3. Analysis

The contents of the user models were analyzed to determine the total number of domain objects represented at knowledge levels d1 and d2 after each assignment. Recall that any domain entity not explicitly represented in the user model is, by default, considered by the system to belong in d0. The results are summarized in graphical form in Figures 8-1, 8-2, and 8-3.

The first two sets of graphs (Figures 8-1, 8-2) show how the number of objects in the user model increase over time (from assignment 1 to 3). User1 under the accept condition scenario, after completing the first assignment, knew seven LISP functions, according to his user model; after completing assignment two, eight functions; and after assignment three, ten functions. Under the *accept* condition, domain objects never get ranked higher than level d2 because collectively the acquisition method will only allow a domain model object to move to level d1 once it has already been ranked d2 and either is explained explicitly (which of course never happened under this condition) or migrates to d1 because it is linked via the dependent-on relations to another domain entity that moves to d1. Since no explanations were included in this test scenario no entities ever got marked d1 to begin this chain of inferences. It is therefore impossible for any domain entity to indirectly migrate to the d1 level. A similar circumstance exists for LISP functions even in the *explain* scenario, but here it is not an attribute of the control conditions, but rather indicates a possible shortcoming in the acquisition methods that will be discussed later.

Figure 8-3 shows cumulative results for all three types of objects for these four students over the three assignments. There is only one curve for the accept condition, the total for the number of entities ranked d2, for the reason discussed above. The shape of the curves in this graph are probably what would be expected from programmers learning a new language. They initially learn a few functions and concepts to get them familiar with the language and able to write some code, and after some experience they begin to acquire new knowledge at a faster rate. This is the type of learning curve one would expect for students learn-

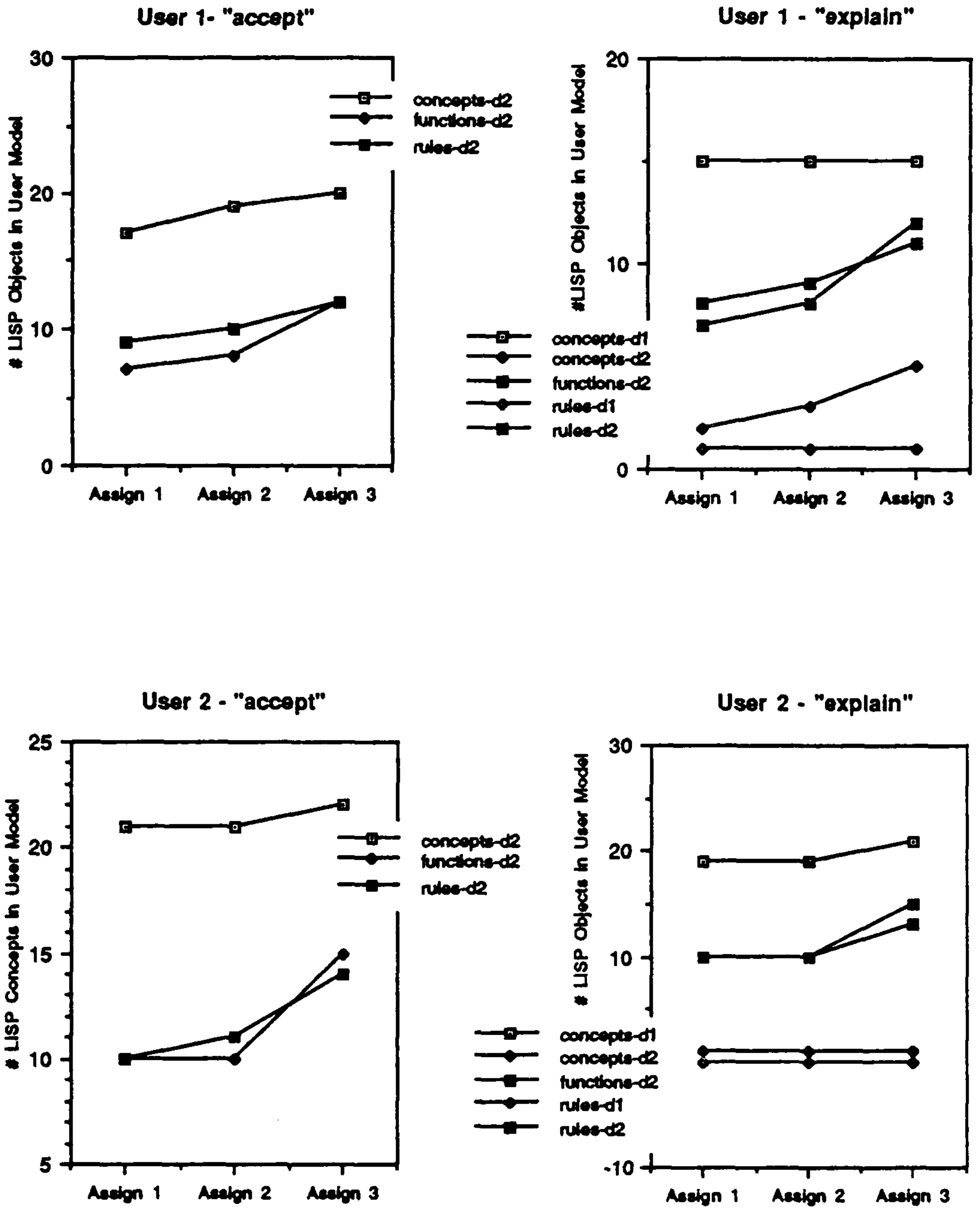


Figure 8-1: User Model Test Results

These graphs show changes in the number of domain entities recorded in the user models of two students under the two different test conditions.

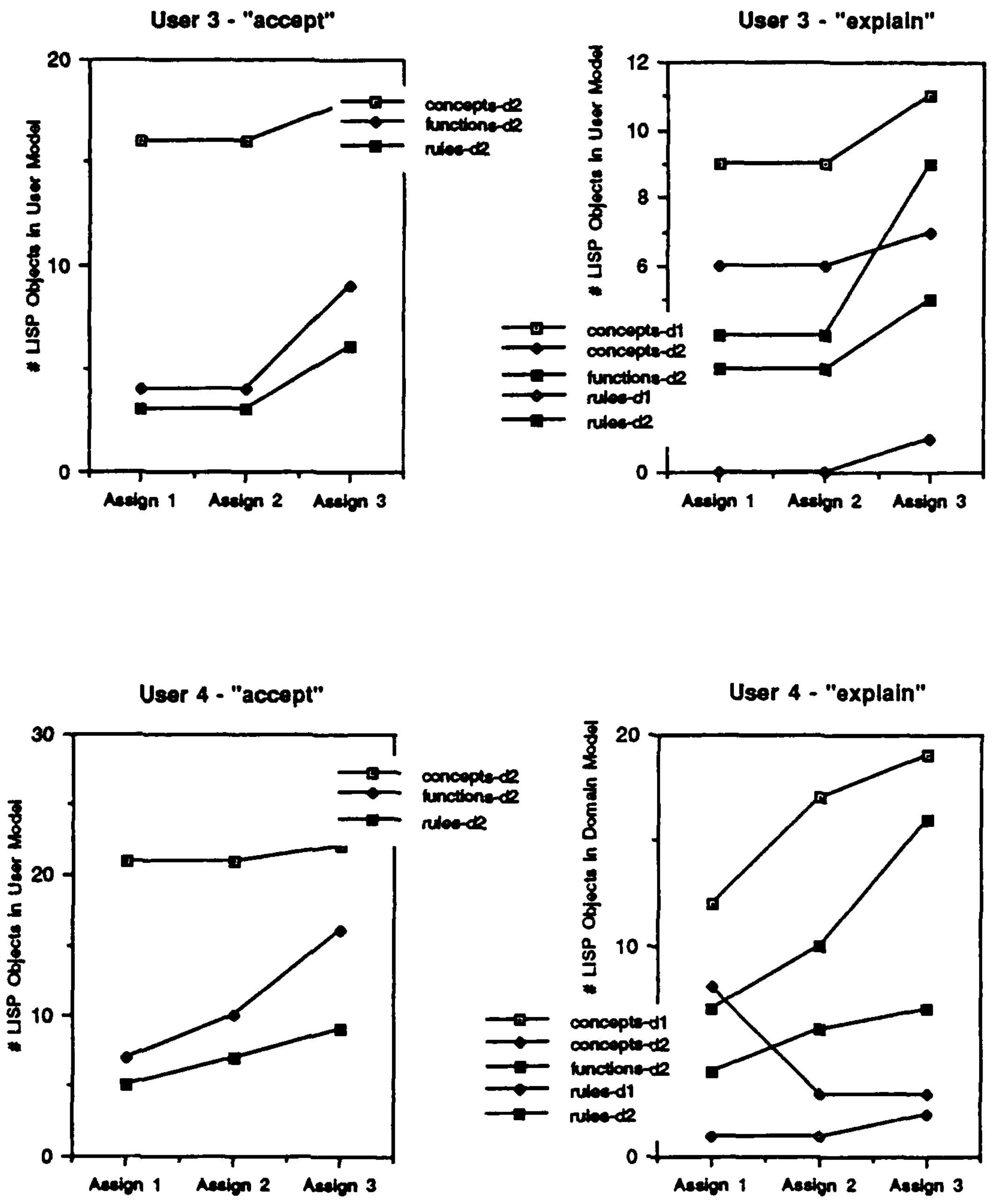


Figure 8-2: User Model Test Results

Graphs similar to the ones in Figure 8-1 showing changes in the user models for two more students.

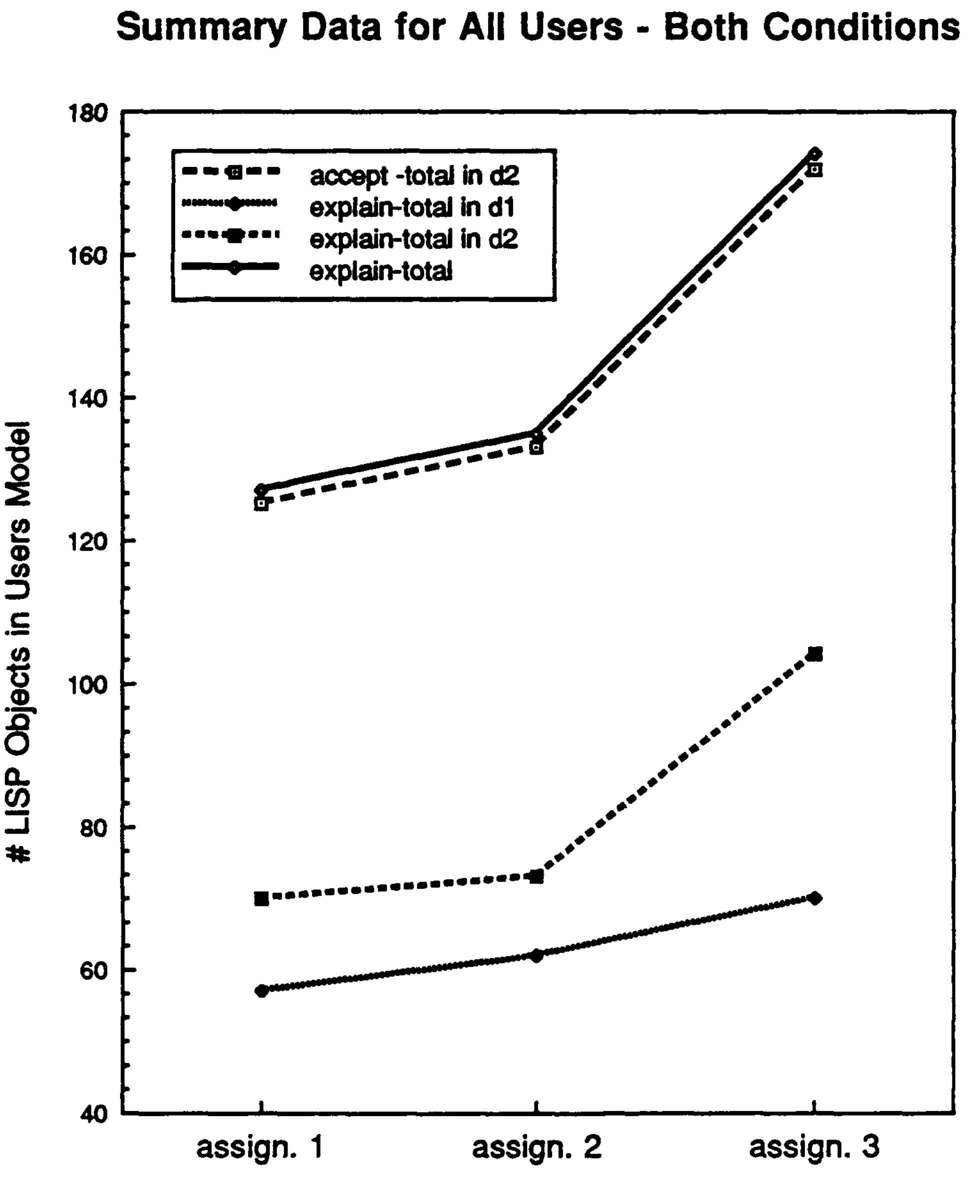


Figure 8-3: Summary of Test Results

ing LISP for the first time. It rises only slightly between assignments 1 and 2, then more sharply between the second and third programming assignments. The curve for objects being ranked at level d1 rises less sharply overall because students do not gain complete understanding of that many concepts and functions over the course of just three programming assignments, but they are likely to become quickly familiar (level d2) with a greater number.

Recall that the semantics of domain entity ranked at level d2 is that users know about such entities, but would need assistance in applying them in their work. This situation matches both the course objective, to introduce LISP and functional programming to the students; and reasonable expectations, students do not become experts after three programming assignments but do gain a more general understanding about some number of the central concepts in the domain. In general, this analysis indicates that the contents of the user models correspond to expectations about user knowledge under the conditions set for the evaluation. No claim is made that this data guarantees that the model representation or the acquisition methods are valid; instead the assessment here is that the user modelling component works in a predictable and reasonable fashion.

## 8.4. Results of Analysis

The analysis provides two observations. The first one examines the effectiveness of the implemented user modelling component; the other one considers the accuracy of the user model contents.

### 8.4.1. Efficacy of the User Model Component

As discussed in the scenario presented in Chapter 7, one way to view the content of the user model is as a coloring of the conceptual graph representation of the domain model. These graph colorings together with data discussed above (shown in Figures 8-1, 8-2, and 8-3), point out some potential shortcomings in the user model acquisition methods. The results indicate some types of refinements that might be made to improve the inference methods.

- It is possible for the acquisition methods to infer *(color)* certain objects from the domain model as well known to the user (level d1) even though they have never actually been explained by the system. This inference is perhaps too optimistic. Certainly some users generate self-explanations for some concepts or rules without ever consulting other material, but there is no guarantee of that happening and the inference methods need to be changed to wait for outside confirmation of that knowledge.
- The previous situation is acceptable under some conditions, such as after observations of users applying the given concept or function correctly in a subsequent program. The *a priori* conditions for these types of indirect inferences should be made more stringent; for example we might require corroborating evidence from other sources, such as a report from the *statistical analysis component* of how frequently a programmer uses a function.
- LISP functions were never *colored* at the d1 level because they do not get explained directly under the present strategy. There presently are no methods to *infer* indirectly that a user knows them at that level. This is partially a phenomenon of test scenario conditions which did not call for using the hypertext capability as a fallback technique. Actual users would most likely have used that facility, for example calling up the Document Examiner descriptions shown in Figure 6-1 as a fallback to the explanation for the *cond-to-if* rule shown in Figure 4-8. Again, an outside source could provide corroborating evidence showing application of that knowledge (e.g., using an *if* in a follow-on assignment in the situation above).

### 8.4.2. Comparison of the User Models with the Questionnaires

In an attempt to validate the inferred user models, the contents of those models were compared to the responses from the student questionnaires. Specifically, concepts appearing in the user model after each scenario episode were compared to the students assessment of their own knowledge of those concepts at the same approximate stage of learning.

Table 8-1 shows, for all users and as a total, the correlation of the contents of the models after processing the first program under the two conditions, *accept* and *explain*, with the second questionnaire. Similarly, the contents of the models resulting after the scenario episode for assignment two were correlated with the third questionnaire. The questionnaires were administered immediately *before* the students received their programming assignments. The second questionnaire, completed in class just before assignment two was given, therefore reflects what students learned about LISP while completing assignment one. The classroom lectures on LISP were formally presented at the beginning of the semester while students worked on the first assignment, and should not effect these correlations.

Only correlations for concepts were computed because, as it turned out, the portion of the questionnaire dealing with functions was not well designed. There was minimal overlap between the set of LISP functions on the questionnaire and the set used by the students in their three assignments — functions acquired by the user model acquisition methods. The questionnaire was developed and administered before getting the students' programs and we failed to include many of the functions they actually used in the assigned problems. A better prior analysis of the programming problems, and conjecture about what functions might be used could have been done; the functions used on the questionnaire were ones that we

Table 8-1: Summary of Correlation Results

| Correlations of User Model Contents to Questionnaires | | | |
|---|---|---|---|
| Condition | User | User Model 1 vs Questionnaire 2 | User Model 2 vs Questionnaire 3 |
| Accept | User1 | .25 | .15 |
| | User2 | .08 | .17 |
| | User3 | .18 | .18 |
| | User4 | .92 | .23 |
| | Total | .38 | .18 |
| Explain | User1 | .67 | .77 |
| | User2 | .92 | .83 |
| | User3 | .45 | .36 |
| | User4 | .23 | .69 |
| | Total | .56 | .67 |

This table shows correlations of the user models' contents with self-assessment questionnaires. When the scenario call for users to receive explanations of LISP-CRITIC suggestions, correlations are the best, greater than 50% for most students and overall.

thought were fundamental. The questionnaires did not ask about all of the 45 domain concepts and 103 functions in the domain model because of a desire to keep it within a reasonable length, in retrospect it might have been better to ask about all of them. LISP-CRITIC rules were not asked about on the questionnaires because they would not have had any meaning to the students in an abstract form (e.g., by name).

The correlations under the *explain* condition are significantly higher. The models acquired under this condition correspond better to the self assessments. The *explain* scenario is probably closer to the process student programmers follow. They probably engage in active learning while in the process of doing the assignments. They encounter and learn those concepts represented in the user model, and on the questionnaire, through classroom instruction, or by consulting additional information sources (e.g., textbooks, human advisors, or peers). While in the process of writing their programs, they seek out "explanations" that help them to build a mental model for the domain comprised of the concepts and LISP functions. Questionnaires 2 and 3 probably should have asked the respondents the types of source materials they used (e.g., textbook, on-line documentation, etc.) when writing their LISP programs.

## 8.5. Limitations

A criticism of the testing process is that because the students themselves did not use LISP-CRITIC, there is not evidence that they would have *learned* the transformations in our scenario. Neither do we know if students actually learned the new functions in the right hand side of the transformation rules or the concepts are behind them. There is evidence that they knew some of the functions (the ones in the left hand side of the transformation rules) as these were used to complete the assignments. The evaluation depends on the assumption that the subjects became more knowledgeable in the domain of LISP because they completed the assignments, and on the assumption that the code used in the test scenarios to infer how their knowledge changed reflects that new knowledge.

The domain knowledge required for any single transformation requires understanding the functions in, and concepts underlying, both the old code and the new code from a transformation. Therefore, some of the knowledge attributed to subjects by their user model, came from code the students wrote; other knowledge is that captured in the LISP-CRITIC rule for each transformation. In principle, more than half of what the system infers is based on the students' code, that half inferred from the left hand side of the transformation rules. If the subjects had in fact experienced our test scenario episodes, one could hypothesize that the rule firings (and explanations if requested) would have caused them to learn new functions and concepts, those in the right hand side; they would have appeared in their questionnaire responses and there would be improved correlation between the user model contents and the questionnaire data.

## 8.6. Shortcomings in System Pointed Out by the Evaluation

There are some shortcomings in the system pointed out by the evaluation and some pointed out by the scenario. The scenario was presented in Chapter 3 as a vehicle for understanding the context of this work. However, because it is based on a set of programs produced under actual circumstances, it provides insight into how the user modelling component works in such a real scenario, and provides another way to assess its effectiveness. The explanation strategies require additional implementation work. The user model and the LISP domain model can support richer explanation strategies than simply the display of hypertext descriptions for the underlying concepts. For example, the domain model also links related objects, such as similar LISP-CRITIC rules. This information could be used for other types of explanation strategies, as previously discussed. The explanation component could consult the domain model for the set related domain entities and then interrogate the user model to determine if any of the entities in this set are known to the user. Given this knowledge, the explanation component could describe the new entity in terms of its differences from, and similarities to an already known entity. Examples occur both at the rule level, now that the reader here knows the *cond-to-if* rule, a reasonable strategy for explaining the *cond-to-when* rule would be to use this differential approach; in a similar fashion two related-concepts or functions can also be described.

Another observation is that domain objects migrate to the user's d1 level of understanding in the user model too easily. One can see this graphically by comparing the graph colorings from the three scenario dialog episodes in Figures

7-3, 7-5, and 7-7. The user model acquisition methods should be changed, constraining inferences to "percolating" knowledge to the d1 level through no more than one level of dependent-on links. Another indicated modification is to the indirect acquisition rule; instead of marking dependent-on concepts at level d1, when the base concept is at d1 use a weaker condition that marks them only at level d2. We should consider modifying any method that allows the user model to indirectly infer a piece of knowledge is well known (level d1) to a user.

## 8.7. Implications for System Modifications and Further Development

Three major findings resulted from the evaluations. The user model implementation, particularly the acquisition methods, can be refined. Using a startup or initial user model is likely to provide a more accurate evaluation, and the domain model itself could probably be iteratively refined using analyses of additional test cases.

Refining the user model acquisition process means that methods should be modified to apply less optimistic inferences, as just discussed in the previous section, and that additional methods should be added. The indirect methods need to be modified so that LISP objects are added to the domain model at no better than the d2 level. Presently the indirect methods can cause a domain model entity to be ranked at level d1 and that is probably too optimistic a point of view. These methods should be modified, and the test data rerun, to see if better correlations result.

Using a startup model rather than beginning "from scratch" would establish a more realistic test scenario. One approach would be to use stereotyping or classification approaches to provide information about users that is likely to be true even if not provided directly in initial questionnaires, or later through implicit methods. Observations of cooperative activities between humans [Reeves 90] showed that people apply certain "leverage" techniques, such as stereotypes or explicit questioning of their partners, to provide an initial or default model to guide their first interactions with another person. Implementing these types of initial modelling techniques was not part of this work, but some simple techniques could probably be used to establish initial models which the implicit inference methods could then improve upon during the test scenarios. The questionnaires already completed by the subjects could be the source of initial information for a startup model. A test of the system starting with models initialized from those questionnaires would probably provide a more realistic scenario of how the students' knowledge changed during the programming exercises.

An analysis of the graph colorings for domain concepts indicate that some groups of concepts (a grouping being indicated by the oval size) are more frequently colored and perhaps more fundamental. They migrate to level d1 the most quickly; they are the ones the model claims are best known to the user. A more detailed analysis of the models generated for a larger population of users might provide some insight into how to refine and improve the domain model to more accurately portray programmers' mental models of the domain.

## 8.8. Summary

Testing the implemented user modelling component demonstrates that the techniques work approximately as was expected. Some parts of the user model component can be made to more accurately predict user knowledge, and any subsequent evaluations should employ a startup model to make it approximate more closely the approaches people use in similar cooperative problem solving situations.

To fully validate a model, such as the one proposed and implemented here, will require significant additional iterative development together with extensive testing of a complete critiquing system on actual users. No matter how well the user modelling component works, it produces only an ''approximation'' of a user's knowledge state. People themselves make do with similar approximations of their communication partners. A complete user modelling system will need to use a range of comprehensive acquisition methods that include multiple techniques, as will be described in the next chapter. These, together with detailed domain and, perhaps, task models, are needed if user modelling is to become a mature technology.

# CHAPTER IX

# APPLICATIONS FOR, AND EXTENSIONS TO, THE WORK

This chapter analyzes the contributions of this work in a larger context of research problems and application systems. Primary focus was developing a user modelling approach for critiquing. But, the research area of user modelling is important in a more general sense; my results can be of use in several other paradigms: advice giving systems [Wahlster, Kobsa 88], intelligent computer-aided instruction [Wenger 87], and human-computer interaction in general [Murray Benyon 89]. A primary contribution of this research is a conceptual framework for approaches to acquiring user models; a framework that can also guide future research. A possible extension is to apply our specific model to support applications other than critiquing. Other critiquing applications could benefit from the addition of a user modelling component; it may be possible to develop such a component in fashion similar to the approach followed in this dissertation. Lastly, there are also several interesting directions in which this approach to user modelling can be continued and extended.

## 9.1. A Framework for User Model Acquisition Techniques

In the course of these investigations a comprehensive framework for classifying approaches to user model acquisition was developed. It is a framework that integrates conceptual discussions [Wenger 87] together with ideas put forth in research attempting to identify the acquisition requirements in a general user modelling architecture [Kass 88]. The framework contains four categories of acquisition techniques. Each category is a collection of methods which may or may not be appropriate in a specific system; this depends upon the domain and type of application that the user model will support. The purpose for the framework is to aid system developers and researchers in identifying appropriate techniques for a particular application. The framework can also help us to categorize research efforts and identify problems worthy of further investigation. This framework helped guide the implementation of the user modelling component for LISP-CRITIC.

### 9.1.1. Background

In the process of developing the user modelling component for LISP-CRITIC, we investigated a diverse set of acquisition strategies, but there was no methodology that could be used easily to correlate them. A classification scheme that helped to organize the ideas, and to understand research on user model acquisition by others, was developed. One significant finding during this process was that user modelling for many types of systems and applications require abstract, conceptual domain representations, and furthermore, a number of the techniques in the specified framework depend on that deep domain model.

A motivating factor in this work was the intuition that considerable information about the user is available within the computational environment, and a desire to explore how to make use of that information for user model acquisition. It was specifically observed that users demonstrate their understanding through actions that they take and by the decisions that they make. Also noted was that their knowledge is enhanced whenever they are exposed to system provided information in the form of explanation or advice. The central issue is how to use that information in the acquisition process. At a general level I was interested in ''evidence-based'' approaches. Systems have available evidence about what users know, they need methods that tell them how use that evidence to infer models of those users. The indirect implicit acquisition developed in this research also had to fit into the framework.

The acquisition framework was also motivated by what was observed in human-to-human cooperative problem solving, specifically those situations in which one person has a greater understanding of the task itself, while a second is more of a domain expert — knows more potential solution approaches. This role distribution is similar to the one between users and knowledge-based computer systems. In the study of sales agents assisting customers [Reeves 90], when interviewed the experts related that they use direct, questioning-types of approaches to acquiring a model of their clients and, more interestingly, several consciously recognized short-cut techniques (and others, we suspect, that are not). The triggering conditions for the inferences are interesting; they ranged from physical characteristics of the customer, the way the client is dressed, to cognitive traits, how conversant they were in expressing the problem, and even to the local weather, was there a significant winter storm likely to cause certain problems for homeowners, automobile operators, etc. The classification framework attempts to account for as many of these observed techniques as possible.

There are four classes of update techniques: explicit acquisition, tutoring, statistical techniques, and implicit methods. With present technology it is unlikely that a single system will be able to use methods in every class but, in building a specific user modelling component, this framework can help in the selection of appropriate and feasible approaches. This is not meant to contradict the long term goal of a comprehensive system which uses multiple approaches, but rather accounts for what is possible to do in systems in the immediate future.

### 9.1.2. Explicit Acquisition Methods

Explicit user model acquisition methods are based on direct query of the user. They acquire specific information about the user that will assist in determining an initial user model. In some cases they are used to clarify conflicting information in a model, in others to add information that is missing but needed by the system. Explicit acquisition techniques can be used in conjunction with stereotypes to construct the initial model of a user.

Three explicit acquisition methods are a prescriptive set of questions prestored in the system, dynamic selection of questions for the user, and free-form descriptive user input. In the first approach, a system developer determines what the information is needed for the initial model, then specifies and prestores questions to ask of the user. The answers provide direct information to enter into the model, for example which LISP functions they already know, or can be designed to trigger richer inferences that are represented in sets of rules, procedures, or a decision table.

A similar approach dynamically generates the questions. It is possible to use a decision tree, or the structure of the domain, especially if it is hierarchical, in conjunction with previous answers to select a minimal set of queries to establish an adequate startup model. An example from the UNIX operating system domain is to ask users if they know the *diff* command; a positive answer would allow the system to infer they have command of concepts like the UNIX file system, types of files, and that they probably know more basic, related commands, like *cat*, *ls*, and *more*. This type of approach was explored in related work where we explored building an initial user model for a learning environment, a learning environment designed to assist new users of a workstation [Mastaglio, Turnbull 87].

The third technique was used in the stereotyping research conducted by

[Rich 79]. Here users describe themselves (their interests in the case of GRUNDY); the terms they use are compared to stereotypes the systems knows, if a favorable match is found the content of the stereotype becomes the default contents for the initial user model.

### 9.1.3. Tutoring-based Methods

Tutoring-based methods use instructional episodes as an information source to inform the user model contents. An assumption here is that after individuals receive tutoring on some domain aspect, they now know it, and their model should reflect that fact. Tutoring-based methods are used, in intelligent tutoring system, to infer student models, possibly in conjunction with additional methods which observe a student subsequently using that knowledge correctly. It is possible to use these same techniques to build a user model that is able to accommodate a more comprehensive system, one that includes a tutoring and other components. Student models are primarily used to determine knowledge that is "missing", which the system can then teach; critics are interested in offering explanations for similar missing knowledge, when it is required to understand a critique. These related needs indicate that there is a possibility to share both the user model and the acquisition methods.

Some acquisition methods in tutoring attempt to determine the parts of a domain that are misunderstood, these are the bug approaches. Knowing the student's bugs is of significant value in guiding an instructional process but it is of limited use in applications more general than tutoring, such as critiquing

An ultimate objective of some research in human-computer interaction is to provide a comprehensive knowledge-based system, with multiple components all supporting users in an interactive working context — an intelligent support system [Fischer 86]; one that can support them with advice, help, tutoring, critiquing, etc. The components of such a system should be able to share a common user and domain model. Tutoring episodes provide an important source of information that can be used to enrich the common user model; this is information that is useful to other system components.

### 9.1.4. Statistical Analysis of User's Work

Statistical methods can provide a measure of the sophistication of a user's knowledge. An analysis of work produced by the user could be accumulated and reported to the system. The reported statistics provide the system triggers for inferences about the sophistication of a user's knowledge. For example, in LISP the type of functions used (destructive versus cons-generating) might provide a system evidence about the user's overall expertise. Acquisition methods based on statistical approaches can apply machine learning paradigms such as learning by example [Fain-Lehman, Carbonell 87]. The examples used in the machine learning process are the users' work. The analysis of what a user produces could take place as a separate off-line system activity [Fischer 87b]. Alternatively, it may be best accomplished, in some situations, by interpreting cumulatively observed data about the user over time, like in the ACTIVIST system [Fischer, Lemke, Schwab 85]. Statistical and mathematical techniques, such as Bayesian inference and fuzzy set theory, can provide theoretical bases for methods in this class.

In the domain model described in Chapter 5, some categories of concepts (or functions) appear to require more sophisticated domain knowledge on the part of the user. An analysis of code could inform the system about the programmer's usage, by category, of both functions and concepts. This information could be used by the system to trigger a stereotype, or select an expertise category for the programmer, an initial user model. In the KNOME system, double stereotypes were used in this manner [Chin 89], one set of stereotypes represents canonical users, the other set is a categorization of UNIX concepts and commands that is similar to the groups in the LISP domain model.

The major problems for developers is determining which statistics are important, and how to use them. Accumulating statistical data about a user is not difficult; the issue is the inferences to make with those data. The conditions under which these statistical methods might work is possibly domain and application system dependent; they may not conform to a general theory: this is certainly an open research question. Extensive studies of user populations for specific systems will be required. The results of the data collection have to be correlated with known characteristics of the users to determine how to best use specific pieces of statistical information. Statistical methods are similar to the implicit acquisition methods, discussed next, in that both operate without specific input from the user: they make use of information that is already available, information generated during the course of the user's normal work. Like stereotyping, they require prior analysis of a sample user population to determine what certain analytic results might imply about any user. Only then can those results be used as a triggering condition for an inference method.

### 9.1.5. Implicit Acquisition

Implicit acquisition methods use the contents of user-system interactions to make inference about users. They are designed to avoid having to subject users explicit methods that interrupt their work. Implicit acquisition methods fall into into two subcategories direct and indirect methods.

**Direct methods:** The direct implicit inference methods observe or note user actions that are part of the ongoing user-system dialog, and then use that observation to add to, or change information already in the user model. The set of implicature rules developed by Kass are an example of methods in this class [Kass, Finin 89]. Direct methods are based on the idea that user-computer interaction is a dialog. Depending upon the specific application, these dialogs have different goals and formats. The dialog may seek to achieve a shared understanding between the system and the user, or to negotiate a common goal: advisory type systems are a canonical example of this. The human and the computer might also seek agreement on whether a certain course of action is appropriate; knowledge-based decision support systems are an architecture for achieving this type of collaboration [Turban, Watkins 86].

When users communicate with a system in any form, ranging from natural language to menu selections, there is information within the context of those interactions that the system can use to infer their user models. Conversely, when the system explains something to users, that information should now be known, and it can be added to their user models. In LISP-CRITIC the direct methods use the acceptance of critic suggestions to trigger one type of direct method, and the request for and receipt of an explanation to trigger another. The degree of the system's confidence in that part of the model, and the way it determines how well a user "knows" that information are open questions, the answers to which may also turn out to be domain and application dependent.

**Indirect methods:** The indirect methods operate like internal demons; they use changes to a user model to trigger further changes. In general, any inference method that adds not-directly-observed information to the user model belongs in this category. Stereotypes are sometimes used in this fashion. In this work, the indirect methods depend on the support of a deep domain model. An example of an indirect method occurs when the user model is updated to include the fact that a user knows a certain aspect of the domain: indirect methods in LISP-CRITIC use that change to the user model to infer that the user also knows the prerequisite knowledge for the aspect just added. Consider an example from another domain that demonstrates the generality of this idea, the domain is mathematics and here the system observes a student summing two negative numbers correctly. From this observation, an inference is made that the student knows how to sum negative numbers. This direct inference changes the user model, which in turn triggers other changes. One inference might be that the student knows the concept negative numbers, another that he or she knows the concept addition. That information can now be added to the user model if it is not already present.

## 9.2. Employing the Approach in Other Applications

A potential application of this research is to use the approach reported on here in other types of applications besides critiquing. The concept-based user model developed here has the characteristics required to support tutoring, advisory systems, and human-computer interaction systems in general.

Intelligent tutors frequently represent their students in terms of productions contained in a system rule base, or in terms of their misconceptions. The concept-based user model representation provides an alternative method for guiding the tutor; the user model can provide a list of those concepts that a user does not know. Concepts that the tutor can now focus on teaching. Concepts which a user already knows provide a source for selecting pedagogical strategies; this is similar to the methodology used in the genetic graph approach. Opportunities to teach new concepts using analogy, generalization, or specialization, can be selected by comparing the user and domain models. To actually accomplish this will require the tutor to have greater understanding about the semantics of the domain model structure, specifically the links between entities, as well as knowledge about what didactic approaches are suitable under what conditions.

Advisory systems give advice and, in some cases, are also designed to assist users with understanding the rationale for that advice. User modelling components in advisory systems focus on supporting the giving of advice, they infer user goals and plans to insure that it is appropriate. In a financial advising system, advice would consist of suggesting to users where to invest their money (e.g., in mutual funds or municipal bonds). A concept-based user model could assist the system to explain such advice — answer questions such as why are municipal bonds a good investment for me at this time. A user model that is able to support both the advice giving and explaining roles in advisory systems will have to be more general, capturing both situation specific conditions for users, such as their goals, and their domain expertise.

An advisory system that will be consulted on multiple occasions by the same user is a better candidate for such a comprehensive model, for example the financial advisor above, than a system designed to provide one-shot advice, such as one that suggests which train to take. A hypothetical example is an advisor for LISP that can, in the spirit of the Programmer's Apprentice, suggest software cliches that will accomplish a specified task (e.g., print an item). The system has to know about the code in a program library to make an appropriate recommendation and have the ability to explain that code when asked. A concept-based user model could inform the system during the advising phase to help it select a cliche the user is more likely to understand (e.g., one using *print* instead of *format* ), and during the explanation phase to help it formulate an explanation in a manner similar to that envisioned for LISP-CRITIC.

Research efforts in human-computer interaction often claim that an idiosyncratic user model will enhance that interaction [Murray 88], but focused investigations along these lines are not reported in the literature. The fundamental issues are determining what information those models must provide, and how that information will be used by the system. One direction is to use a modelling approach, like the one developed here for critiquing, as a starting point for investigating how system adaptivity in the general class of human-computer interaction systems can be supported by user models [Murray Benyon 89].

## 9.3. Support for Critiquing in Other Domains

In Chapter 2 we covered the application domains for which critiquing system have been developed; it is intriguingly diverse. Enhancing the effectiveness and utility of critics with a user modelling component is an indicated future research direction in several of the system descriptions. A useful application of this research would be to enhance a different existing critic system using an approach similar to the one followed for the work on LISP-CRITIC.

Design environments [Lemke 89] include a critic component and some use hypertext issue-based information systems as a source of information for helping designers understand a critique, as well as to precipitate reflective practice [Fischer, McCall, Morch 89a; McCall, Fischer, Morch 89; Fischer, McCall, Morch 89b]. No attempt is made to adapt, or tailor, the information to the individual designer, but rather the methodology focuses on presenting it in a structured manner, and insuring it is contextually related to current work. It would be worth investigating whether user models, such as the one in LISP-CRITIC, can be integrated with these techniques.

In a more general sense, systems based on critiquing have been developed to support domains such as software engineering [Fickas, Nagarajan 88], VLSI design [Steele 87], and decision making [Mili 88]; systems that support knowledge workers who use them repeatedly. Users expect to learn from the critiques to perform their tasks better, this means that there exists a need for system explanations; this is an argument similar to the one given as motivation for the work on explanation-giving in LISP-CRITIC. Critiquing is also used to support medical applications, and several of these research efforts suggest that having a user model would enhance their systems [Langlotz, Shortliffe 83; Miller 86; Rennels 87]. Application of the user modelling methodology followed here to enhance some existing critics would serve to validate and refine the techniques; it is also sure to provide additional insight to motivate improved theory.

## 9.4. Issues Warranting Further Research

The other potential direction for future research is to enhance the work accomplished thus far. Some ideas were previously mentioned in the context of describing the approach, the implementation, and the evaluation. One such area is to learn how to use statistical information that can be obtained from a computer analysis of users' work or actions; another is to integrate tutoring with cooperative problem solving systems to determine more specifically what is needed in a model designed to serve the needs of both. One extension along these lines might be to build a comprehensive system from scratch, or to integrate two such already existing systems. Some other potential research directions, not previously mentioned, are investigating how to make better use of networked computing environments, enhancing the domain modelling approach, and sharing the user model between multiple applications, or even different domains with shared conceptual spaces, for example different programming languages.

**Distributed User Modelling.** Present computing environments are almost always part of larger networks. Having other machines available on that network should allow us consider how to introduce concurrency into the user modelling process. Domain models will come to have greater fidelity and a richer representation, and user models will, likewise, become more comprehensive; this may cause them to tax the computational power in a single workstation. Also we should consider the situation where users run applications on remote machines, applications that might benefit from access to their user model.

One direction for research is to investigate how to provide access to the user model stored on a "personal workstation" to applications running at remote sites. If our goal is a truly comprehensive and complete user model, of use to multiple systems, then it follows that they should share a single version of that model. Some issues that must be considered are privacy, concurrent updating, and simultaneous access by more than one remote server. Research into using the approach in this manner could begin by determining which of the problems involved can be solved using techniques already developed in other concurrent systems research and identifying any new ones that are generated.

A more futuristic idea is to consider having a user modelling machine, either virtual or actual; one dedicated to performing implicit user model acquisition in parallel with other applications to achieve concurrency. Similarly, a machine could be dedicated to the role of domain model server. This might be a particularly useful approach for providing reasonable access to the large models that are growing out of research into representing general common sense knowledge [Lenat, Prakash, Shepherd 86].

**Generality of the Domain Model.** The domain model was developed specifically for LISP, in order to meet explanation and user modelling needs. During this research it was observed that our domain model is an instance of what other researchers in explanation-giving have called "deep domain models" [Chandrasekaran, Tanner, Josephson 88]. It may be able to provide support for more general applications. The research issue is, can it usefully serve other applications or interaction paradigms, other than critiquing, and if not, can it be modified in some way to accomplish this?

Predominantly, past computer-based systems for instruction have been one of three types: drill and practice computer-aided instruction that captures domain and pedagogical knowledge directly in the course material, intelligent tutoring approaches that capture domain and pedagogical knowledge in productions and exercises, and simulation systems that capture domain knowledge in the behavior of the simulated devices; pedagogy is implicit in the simulation process. This is not to say that these will be the dominant future approaches, in fact we happen to believe critiquing [Mastaglio 89] will replace or augment all them in certain situations; these three are just historically the most common. The concept-based domain model allows pedagogy to be derived from a traversal of its structure, links provide the pedagogy for teaching new concepts from already familiar ones. It would be worth investigating if the domain model could be used to help direct a didactic computer agent, such as a coach, that knows the learning objectives and has available to it a set of exercises or simulations that are linked to domain model entities and to pedagogical knowledge.

**Sharing the User Model.** An ultimate goal of some work in human-computer interaction research is a comprehensive system which supports multiple interaction approaches through multiple components. In terms of user modelling, it would be ideal if the model could be shared by these components, such as a system incorporating a critic and a tutor like GRACE. It would be worth investigating what type of user model is needed to support a larger class of applications, and if the approach discussed here needs to be modified to achieve this goal. As a simple example, some of the knowledge of LISP captured in our user model (e.g., conditionals, scope, tests, etc) would also be useful to a critic in a related domain, such as one for another programming language. Identifying the common domain characteristics to capture in such a shared model warrants further investigation.

## 9.5. Summary

This chapter indicated how this specific work fits into a broader perspective of user modelling and related research. The acquisition framework can help developers of systems that will contain a user modelling component, and it provides a guide for future research. The methodology used in this project has potential for use in other applications, and to support critics in other domains. What is required to achieve a system that a user will perceive as meeting our goal of being a cooperative problem solving system is still an open research issue: the work done here provides a starting point for individualizing the types of environments in which we eventually hope to find these systems. One thing this dissertation research has shown is that user modelling is complex, perhaps one of the more complex applications yet encountered in applied computer science and artificial intelligence research. It is not possible to provide complete approaches in a single research effort, and solutions are neither singular nor simple. This does not mean the effort is not important nor should it be abandoned; personalized computer systems that adapt to our needs are able to give and explain meaningful advice, and can interpret our actions are a consistent image in science fiction and futuristic scenarios studied by serious researchers [Skulley 88]. User modelling is one of the important enabling technologies needed to reach that goal; but arriving at a common, useful theory will require multiple efforts and the synthesis of results in order to understand all the cognitive and computational issues involved.

# CHAPTER X

# SUMMARY AND CONCLUSIONS

This chapter is a summary of this dissertation and identifies its major contributions. The general scheme of this research was to study user modelling research in other areas; to develop an understanding of what is required for a user model to support cooperative problem solving; and, from those analyses, to develop an approach for supporting a computer-based critic. A user model that meets those requirements was implemented for LISP-CRITIC, and subjected to an evaluation. The results of the system development work and analysis suggested ideas about the generalizability of the methodology, and indicated possible extensions to the approach as well as directions for future research.

## 10.1. Summary

This dissertation research was accomplished in a context of developing a paradigm for cooperative problem solving, and in the context of knowledge-based systems that support user learning. In principle all cooperative systems should also support learning. Users need access to system-provided explanations in order for that learning to take place. Furthermore, those explanations should be tailored to their individual expertise in the application domain. This need for individualized explanations motivates a requirement for idiosyncratic user models. These characteristics of knowledge-based computers systems, that they support collaborative human-computer effort, and also, that they provide learning opportunities, determine the general requirements for the user modelling approach.

What it means for a system to be cooperative, and the theoretical characteristics of learning environments were discussed in Chapter 1.

Critiquing is one way to use computer knowledge bases to aid users in their work and at the same time support their learning needs. Research investigations into critiquing by the Human-Computer Communications Research Group has included system building efforts, the integration of cognitive and design theories, empirical observations, and the evaluation of prototypes. That collective experience was integrated with a study of other research to determine the theoretical foundations and characteristics of critiquing.

Chapter 2 presented those theoretical foundations and the theory behind the present critiquing framework. Critiquing systems, also called critics, are an alternative to traditional experts systems. The generality of the approach is demonstrated by a study of the literature which shows that critiquing has been successfully used in diverse application domains. In order to enhance current critiquing approaches so that these systems move from simple ''suggestors'' of how to improve a user's work, to ones which can interact with them in a collaborative style, will require models of users. These models will help systems adapt explanations for their domain knowledge to individual users. Critics are not the only approach to building better knowledge-based systems, but a growing number of such systems will contain a critiquing component. Some of them need detailed understanding of users' problems, tasks and goals; but more commonly they will have limited yet helpful capabilities, one of which is to model the knowledge of individual users.

A general approach was chosen as a result of studying related research in user modelling. Chapter 3 discussed user modelling in other research areas and the foundations for user modelling to support the types of cooperative systems in which we are interested. The approach includes an architecture for a user modelling component comprised of a representation scheme for the models, acquisition techniques, and methods for accessing the models. An analysis of reported work on student models to support Intelligent Computer-aided Instruction, and user models for advice-giving dialog systems determined that both areas provide some important concepts to help us establish the foundations for the models we want to have. A user modelling approach for cooperative problem solving can use ideas developed in this other work, but it was not possible to find an approach from other research that could be adapted directly to meet the needs of collaborative systems. Therefore, the methodology followed was to identify the requirements for a user modelling component to support explanations based on a theoretical model of users' expertise, a conceptual model of the domain, the need to acquire the model using implicit methods, and all the while keeping in mind a goal of generality. The resulting conceptual architecture was instantiated in a specific system.

In Chapter 4 LISP-CRITIC was described: it is the environment in which the implementation work was performed. LISP-CRITIC provides a suitable context for investigating user modelling because, in the past, it has been a development platform for investigating various notions of how knowledge-based computer systems can be better designed to accommodate their users. Some of these ideas were knowledge representation and application, user access to the systems actions, and explanation of advice. Integrating a user modelling component to support explanation giving was a natural extension of that previous work.

A domain model was required to support both explanation-giving and user modelling. It links the system's operational knowledge (LISP-CRITIC rules) to the domain knowledge necessary for explanation-giving and representing users' expertise. Chapter 5 covers the analysis of LISP that determined what to represent in the domain model, and then selected some appropriate techniques for achieving that representation. The implemented domain model captures knowledge of LISP in a conceptual structure. From our analysis of the domain it was determined that the fundamental domain entities are LISP concepts, LISP functions and LISP-CRITIC rules, they are all interconnected via semantic relationships. Conceptual graph notation was used to visualize the domain structure, and the domain model in LISP-CRITIC was implemented using the Common LISP Objects System (CLOS).

Cooperative knowledge-based systems take advantage of the different strengths of users and computer systems. Computers are potential sources of expert domain knowledge and can be used to make suggestions; their role must also include the ability to explain those suggestions. Explanation systems often fail because they are based on implicit assumptions that explaining is a one-shot affair, and that artificially intelligent systems will be able to retrieve or produce complete and individualized text. Another approach is to take advantage of information and present computer technology. The explanation approach discussed in Chapter 6 focuses on determining which concepts to explain to a user rather than on choosing a prestored explanation. Executing that process requires a domain model that can provide the set of concepts needed for a given explanation situation, and a user model that can help tailor the explanation to a given individual. When explanation follows this approach, the process is one of constructing, rather than selecting, information that will be presented to a user.

The approach used provides four layers of explanation that can be accommodated in LISP-CRITIC. The first two layers are not explanations in the strictest sense but rather techniques for presenting the critic's advice that facilitate user understanding; they are detailed descriptions of what the system suggests. Rhetoric principles and discourse comprehension research provide foundations for a minimal approach that make up the 3rd layer. Such minimal explanations are guided by a domain and user model that provide, to the system, information about what needs to be explained in order for a user to understand a particular domain entity. The highest layer is a rich hypertext information space that provides a fallback capability for situations in which users need more details. In that hypertext space, users can investigate LISP functions or examine concepts that they still do not understand.

The user modelling component developed for LISP-Critic is described in Chapter 7; it represents what the system knows about each user in an object oriented structure, acquires those user models, provides access to them and retains them for future use. The user model is also implemented in CLOS. The design objectives were based on what is required to support explanation-giving; these objectives guided specification of the architecture for the user modelling component.

Representation of the model is an enhancement of overlay modelling techniques. The approach captures the domain entities a user knows, a subset of those represented in the domain model, but also marks them in accordance with how well they are known. Conceptually, there is a coloring of the domain model graph unique to each individual.

Access to the individual models for other system components is provided for with a set of generic interface functions. In this research, access for the explanation component has been emphasized; but we have attempted to make the methods general so that current system components, such as the critiquing engine, or even new components that are added, like a tutor, can use them.

The acquisition subcomponent contains direct methods that make use of episodes in the user-computer dialog. These are "evidence-based" methods that resulted from the intuition that the interaction context contains useful information for inferring the knowledge state of a user. The subcomponent also contains indirect methods that are triggered by changes to individual user models. One outcome of the LISP-CRITIC system development and implementation work is a framework of user model acquisition techniques.

The development of the implicit methods is considered one of the significant contributions of this research; an evaluation of their effectiveness and possible modifications was undertaken. That evaluation process is covered in Chapter 8. The acquisition methods were evaluated in two ways. Programs written by students learning LISP were processed by LISP-CRITIC; and the individual user models for each programmer compared with one another, attending particularly to the changes that took place over time. In this way the behavior of the user modelling system could be analyzed to determine potential modifications. The user models developed for each student were correlated with questionnaires assessing the students' expertise according to the topology of the domain model; the questionnaires were completed before each programming assignment.

The evaluation demonstrated that the models conform to expectations about how user knowledge might change under the conditions these programs were produced. The models contents were modified by the application of the acquisition methods in a manner similar to what was expected: the models became more detailed as the system was exposed to more of the users' work; and they captured new concepts as students learned them during the course of completing three programming assignments. The evaluation pointed out opportunities for improving the acquisition methods. It also resulted in the observation that using a startup or initial model would probably improve the models' fidelity. The availability of an initial model had been an underlying assumption in this research. This finding confirmed the importance of having that model; regardless of the effectiveness of implicit methods, there is a need for explicit acquisition of initial models of users.

To achieve a completely operational model will require significant additional development and extensive testing. A limitation is that any user model is at best an "approximation" of a user's knowledge state, therefore, it will be difficult to determine when an acquisition methodology is as complete as possible. An outcome of this work is an awareness that a comprehensive user modelling system will be extremely complex; the problem will not yield to singular solutions or simple methods alone. Research efforts to date have tackled only a parts of the problem, usually in isolation. A complete implementation will have to integrate multiple techniques (e.g., stereotyping, explicit questioning and implicit acquisition methods) with detailed domain and perhaps task models. The work undertaken here focused on developing a domain model and implicit acquisition techniques.

In chapter 9, I tried to demonstrate how this work contributes to a broader scope of research. One such contribution is the acquisition framework; it provides a pretheoretic scheme which can be used when developing user modelling components for human-computer interaction systems in general, and can serve as a guide for future research. The development methodology followed here can be used for developing user models for applications other than critiquing; and to extend those critics already developed in other research. In this research the approach used was specifically developed for critiquing, but it provides a starting point for individualizing a more general class of cooperative problem solving systems. There are possibilities to share both the model in its current form with other interaction approaches, like advising or tutoring, and to use the methodology as a guide for developing new models to serve a range of applications, critiquing computer programs being just one them.

## 10.2. Conclusions

The work in this dissertation project contributes to research in user modelling, explanation-giving, and cooperative knowledge-based systems. The use of a common deep conceptual domain representation for both explanation generation and user modelling is unique. Using the inherent structure of that deep domain model to perform implicit acquisition is a technique that enhances a system's ability to build up more complete idiosyncratic models of users, and should be explored for other domains, and using different relational links between the domain entities. The explanation process begins with a single piece of procedural system knowledge, e.g., a rule that a user wants described. It serves as a starting point to extract the appropriate domain concepts; these are filtered through the user model and some of them are eventually explained to the user. This approach could potentially be used in a large class of human-computer interaction systems but depends upon domain and user models to inform the process. This work is a first documented implementation of a model of users' domain expertise in a critiquing system. The possibilities and limitations uncovered here can aid developers of other computer-based critics. The framework for user modelling acquisition methods proved useful in developing a specific user modelling component, and it can be used to guide design analysis and architectural specification for others.

It would be premature to claim that a general theory of user modelling is forthcoming, but this effort has provided a better understanding about some significant aspects of such a theory. Specifically, we now know more about what is required of a user model that supports explanation-giving; the sort of techniques an interactive system can use to implicitly acquire such a model; and how a concept-based domain model can serve as a basis for user model representation, and at the same time support user model acquisition. These ideas expose a new range of issues and directions for research into user modelling that may eventually provide general methods able to accommodate a broad class of human-computer interaction systems.

## REFERENCES

[Aaronson, Carroll 87]
A. Aaronson, J.M. Carroll, *Intelligent Help in a One-Shot Dialog: A Protocol Study*, Human Factors in Computing Systems and Graphics Interface, CHI+GI'87 Conference Proceedings (Toronto, Canada), ACM, New York, April 1987, pp. 163-168.

[Anderson, Conrad, Corbett 89]
J.R. Anderson, F.G. Conrad, A.T. Corbett, *Skill Acquisition and the LISP Tutor*, Cognitive Science, Vol. 13, 1989, pp. 467-505.

[Anderson, Reiser 85]
J.R. Anderson, B.J. Reiser, *The LISP Tutor*, BYTE, Vol. 10, No. 4, April 1985, pp. 159-175.

[Anderson, Thompson 86]
J.R. Anderson, R. Thompson, *User of Analogy in a Production System Architecture*, 1986, Paper presented at the Illinois Workshop on Similarity and Analogy, Champaign-Urbana, June 1986.

[Atwood et al. 90]
M.E. Atwood, W.D. Gray, B. Burns, A.I. Morch, B. Radlinski, *Cooperative Learning and Cooperative Problem Solving: The Case of Grace*, Working Notes, 1990 AAAI Spring Symposium on Knowledge-Based Human-Computer Communication, AAAI, Menlo Park, CA, 1990, pp. 6-10.

[Bloom 84]
B.S. Bloom, *The Search for Methods of Group Instruction as Effective as One-to-One Tutoring*, Educational Leadership, May 1984, pp. 4-17.

[Boecker 84]
H.-D. Boecker, *Softwareerstellung als wissensbasierter Kommunikations- und Designprozess*, Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, April 1984.

[Boecker, Fischer, Nieper 86]
H.-D. Boecker, G. Fischer, H. Nieper, *The Enhancement of Understanding Through Visual Representations*, Human Factors in Computing Systems, CHI'86 Conference Proceedings (Boston, MA), ACM, New York, April 1986, pp. 44-50.

[Brech, Jones 88]
B. Brecht, M. Jones, *Student Models: the Genetic Graph Approach*, International Journal of Man-Machine Studies, Vol. 28, 1988, pp. 483-503.

[Britton, Black 85]
Bruce K. Britton, John B. Black (eds.), *Understanding Expository Text*, Lawrence Erlbaum Associates, London, 1985.

[Brown, Burton 86]
J.S. Brown, R.R. Burton, *Reactive Learning Environments for Teaching Electronic Troubleshooting*, in W.B. Rouse (ed.), *Advances in Man-Machine Systems Reasearch, Vol 3*, JAI Press, Inc, Greenwich, CT, 1986.

[Brown, Burton, Kleer 82]
J.S. Brown, R.R. Burton, J. de Kleer, *Pedagogical, Natural Language and Knowledge Engineering Techniques in SOPHIE I, II and III*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 227-281, ch. 11.

[Brown, VanLehn 80]
J.S. Brown, K. VanLehn, *Repair Theory: A Generative Theory of Bugs in Procedural Skills*, Cognitive Science, Vol. 4, 1980, pp. 379-426.

[Buchanan, Shortliffe 84]
B.G. Buchanan, E.H. Shortliffe, *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984.

[Burton, Brown 82]
R.R. Burton, J.S. Brown, *An Investigation of Computer Coaching for Informal Learning Activities*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 79-98, ch. 4.

[Burton, Brown, Fischer 84]
R.R. Burton, J.S. Brown, G. Fischer, *Analysis of Skiing as a Success Model of Instruction: Manipulating the Learning Environment to Enhance Skill Acquisition*, in B. Rogoff, J. Lave (eds.), *Everyday Cognition: Its Development in Social Context*, Harvard University Press, Cambridge, MA - London, 1984, pp. 139-150.

[Carbonell 70]
J.R. Carbonell, *AI in CAI: An Artificial-Intelligence Approach to Computer-Assisted Instruction*, IEEE Transactions on Man-Machine Systems, Vol. MMS-11, No. 4, December 1970.

[Card, Moran, Newell 83]
S.K. Card, T.P. Moran, A. Newell, *The Psychology of Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Carroll, Carrithers 84]
J.M. Carroll, C. Carrithers, *Training Wheels in a User Interface*, Communications of the ACM, Vol. 27, No. 8, August 1984, pp. 800-806.

[Carroll, McKendree 87]
J.M. Carroll, J. McKendree, *Interface Design Issues for Advice-Giving Expert Systems*, Communications of the ACM, Vol. 30, No. 1, January 1987, pp. 14-31.

[Carver, Lesser, McCue 84]
N.F. Carver, V.R. Lesser, D.L. McCue, *Focusing in Plan Recognition*, Proceedings of AAAI-84, Forth National Conference on Artificial Intelligence (Austin, TX), William Kaufmann, Los Altos, CA, 1984, pp. 42-48.

[Chandrasekaran, Tanner, Josephson 88]
B. Chandrasekaran, C. Tanner, J.R. Josephson, *Explanation: The Role of Concept Strategies and Deep Models*, in J.A. Hendler (ed.), *Expert Systems: The User Interface*, Ablex Publishing Corp, Norwood, NJ, 1988.

[Chandrasekaran, Tanner, Josephson 89]
B. Chandrasekaran, C. Tanner, J.R. Josephson, *Explaining Control Strategies in Problem Solving*, IEEE Expert, Vol. 4, No. 1, Spring 1989, pp. 9-23.

[Chin 89]
D.N. Chin, *KNOME: Modeling What the User Knows in UC*, in A. Kobsa, W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag, New York, 1989, pp. 74-107.

[Clancey 84]
W. Clancey, *Use of MYCIN's Rules for Tutoring*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 464-489, ch. 26.

[Clancey 86]
W.J. Clancey, *Qualitative Student Models*, Annual Review of Computing Science, Vol. 1, 1986, pp. 381-450.

[Clancey 87]
W.J. Clancey, *Knowledge-Based Tutoring: The Guidon Program*, MIT Press, Cambridge, MA, 1987.

[Coombs, Alty 84]
M.J. Coombs, J.L. Alty, *Expert Systems: An Alternative Paradigm*, International Journal of Man-Machine Studies, Vol. 20, 1984.

[Danlos 87]
L. Danlos, *The Linguistic Basis of Text Generation*, University of Cambridge Press, Cambridge, 1987.

[Dews 89]
S. Dews, *Developing an ITS in a Corporate Setting*, Proceedings of the 33rd Annual Meeting of the Human Factors Society, 1989, pp. 1339-1342.

[Dijk, Kintsch 83]
T.A. van Dijk, W. Kintsch, *Strategies of Discourse Comprehension*, Academic Press, New York, 1983.

[Doane, Pellegrino, Klatsky 89]
S.M. Doane, J.W. Pellegrino, R.L. Klatsky, *UNIX System Mental Models and UNIX System Expertise*, Proceedings of the 22nd Annual Hawaii Conference on System Sciences, Vol. II: Software Track, IEEE Computer Society, January 1989, pp. 457-467.

[Draper 86]
S.W. Draper, *Display Managers as User Interfaces*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, ch. 16.

[Dreyfus, Dreyfus 86]
H.L. Dreyfus, S.E. Dreyfus, *Mind Over Machine*, The Free Press, New York, 1986.

[Duchastel 88]
P.C. Duchastel, *Models for AI in Education and Training*, Artificial Intelligence Tools in Education: Proceedings of the IFIP TC3 Working Conference, IFIP, 1988, pp. 17-28.

[Fabian, Lemke 85]
F. Fabian Jr., A.C. Lemke, *WLisp Manual*, Technical Report CU-CS-302A-85, Department of Computer Science, University of Colorado, Boulder, CO, February 1985.

[Fain-Lehman, Carbonell 87]
J. Fain-Lehman, J.G. Carbonell, *Learning the User's Language: A Step Toward Automated Creation of User Models*, Technical Report, Carnegie-Mellon University, March 1987.

[Feigenbaum, McCorduck 83]
E.A. Feigenbaum, P. McCorduck, *The Fifth Generation. Artificial Intelligence and Japan's Computer Challenge to the World*, Addison-Wesley Publishing Company, Reading, MA, 1983.

[Fickas, Nagarajan 88]
S. Fickas, P. Nagarajan, *Critiquing Software Specifications*, IEEE Software, Vol. 5, No. 6, November 1988, pp. 37-47.

[Finin 83]
T.W. Finin, *Providing Help and Advice in Task Oriented Systems*, Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983, pp. 176-178.

[Fischer 83]
G. Fischer, *Symbiotic, Knowledge-Based Computer Support Systems*, Automatica, Vol. 19, No. 6, November 1983, pp. 627-637.

[Fischer 84]
G. Fischer, *Formen und Funktionen von Modellen in der Mensch-Computer Kommunikation*, in H. Schauer, M.J. Tauber (eds.), *Psychologie der Computerbenutzung*, R. Oldenbourg Verlag, Wien - Muenchen, Schriftenreihe der Oesterreichischen Computer Gesellschaft, Vol. 22, 1984, pp. 328-343.

[Fischer 86]
G. Fischer, *Cognitive Science: Information Processing in Humans and Computers*, in H. Winter (ed.), *Artificial Intelligence and Man-Machine Systems*, Springer-Verlag, Berlin - Heidelberg - New York, 1986, pp. 84-112.

[Fischer 87a]
G. Fischer, *Learning on Demand: Ways to Master Systems Incrementally*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1987.

[Fischer 87b]
G. Fischer, *A Critic for LISP*, Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), J. McDermott (ed.), Morgan Kaufmann Publishers, Los Altos, CA, August 1987, pp. 177-184.

[Fischer 88a]
G. Fischer, *Enhancing Incremental Learning Processes with Knowledge-Based Systems*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, New York, 1988, pp. 138-163, ch. 7.

[Fischer 88b]
G. Fischer, *Cooperative Problem Solving Systems*, Proceedings of the 1st Simposium Internacional de Inteligencia Artificial (Monterrey, Mexico), October 1988, pp. 127-132.

[Fischer 90]
G. Fischer, *Communications Requirements for Cooperative Problem Solving Systems*, The International Journal of Information Systems (Special Issue on Knowledge Engineering), 1990.

[Fischer et al. 88]
G. Fischer, S.A. Weyer, W.P. Jones, A.C. Kay, W. Kintsch, R.H. Trigg, *A Critical Assessment of Hypertext Systems*, Human Factors in Computing Systems, CHI'88 Conference Proceedings (Washington, D.C.), ACM, New York, May 1988, pp. 223-227.

[Fischer et al. 90]
G. Fischer, A.C. Lemke, T. Mastaglio, A. Morch, *Using Critics to Empower Users*, Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA), ACM, New York, April 1990, pp. 337-347.

[Fischer, Girgensohn 90]
G. Fischer, A. Girgensohn, *End-User Modifiability in Design Environments*, Human Factors in Computing Systems, CHI'90 Conference Proceedings (Seattle, WA), ACM, New York, April 1990, pp. 183-191.

[Fischer, Lemke 88]
G. Fischer, A.C. Lemke, *Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication*, Human-Computer Interaction, Vol. 3, No. 3, 1988, pp. 179-222.

[Fischer, Lemke, Mastaglio, Morch 90]
G. Fischer, A. Lemke, T. Mastaglio, A. Morch, *Critics: An Emerging Approach to Knowledge-Based Human Computer Interaction*, International Journal of Man-Machine Studies, 1990, to be published.

[Fischer, Lemke, Nieper-Lemke 88]
G. Fischer, A.C. Lemke, H. Nieper-Lemke, *Enhancing Incremental Learning Processes with Knowledge-Based Systems (Final Project Report)*, Technical Report CU-CS-392-88, Department of Computer Science, University of Colorado, Boulder, CO, March 1988.

[Fischer, Lemke, Schwab 84]
G. Fischer, A.C. Lemke, T. Schwab, *Active Help Systems*, Readings on Cognitive Ergonomics - Mind and Computers, Proceedings of the 2nd European Conference (Gmunden, Austria), G.C. van der Veer, M.J. Tauber, T.R.G. Green, P. Gorny (eds.), Springer-Verlag, Berlin - Heidelberg - New York, September 1984, pp. 116-131.

[Fischer, Lemke, Schwab 85]
G. Fischer, A.C. Lemke, T. Schwab, *Knowledge-Based Help Systems*, Human Factors in Computing Systems, CHI'85 Conference Proceedings (San Francisco, CA), ACM, New York, April 1985, pp. 161-167.

[Fischer, Mastaglio 89]
G. Fischer, T. Mastaglio, *Computer-Based Critics*, Proceedings of the 22nd Annual Hawaii Conference on System Sciences, Vol. III: Decision Support and Knowledge Based Systems Track, IEEE Computer Society, January 1989, pp. 427-436.

[Fischer, Mastaglio 90]
G. Fischer, T. Mastaglio, *A Conceptual Framework for Knowledge-based Critic Systems*, The International Journal of Decision Support Systems, Vol. Special Issue on Active, Symbiotic Systems, 1990, to be published.

[Fischer, Mastaglio, Reeves, Rieman 90]
G. Fischer, T. Mastaglio, B. Reeves, J. Rieman, *Minimalist Explanations in Knowledge-Based Systems*, Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr (ed.), IEEE Computer Society, 1990, pp. 309-317.

[Fischer, Mastaglio, Rieman 89]
G. Fischer, T. Mastaglio, J. Rieman, *User Modeling in Critics Based on a Study of Human Experts*, Proceedings of the Fourth Annual Rocky Mountain Conference on Artificial Intelligence, RMSAI, Denver, CO, June 1989, pp. 217-225.

[Fischer, McCall, Morch 89a]
G. Fischer, R. McCall, A. Morch, *Design Environments for Constructive and Argumentative Design*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 269-275.

[Fischer, McCall, Morch 89b]
G. Fischer, R. McCall, A. Morch, *JANUS: Integrating Hypertext with a Knowledge-Based Design Environment*, Proceedings of Hypertext'89, ACM, New York, November 1989, pp. 105-117.

[Flesch 49]
R. Flesch, *The Art of Readable Writing*, Harper & Brothers, New York, 1949.

[Forbus 84]
K. Forbus, *An Interactive Laboratory for Teaching Control System Concepts*, Report 5511, BBN, Cambridge, MA, 1984.

[Fox 88]
B.A. Fox, *Robust learning environments -- the issue of canned text*, Technical Report, Institute of Cognitive Science, University of Colorado, Boulder, Colorado, 1988.

[Frank, Lynn, Mastaglio 87]
J. Frank, P. Lynn, T. Mastaglio, *Using A Critic Methodology as a Computer-aided Learning Paradigm: extending the concepts*, 1987, Final Project Report for CS659 - Fall Term 1987.

[Friedman 87]
M.P. Friedman, *WANDAH - A Computerized Writer's Aid*, in D.E. Berger, K. Pezdek, W.P. Banks (eds.), *Applications of Cognitive Psychology, Problem Solving, Education and Computing*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987, pp. 219-225, ch. 15.

[Gentner, Stevens 83]
D. Gentner, A.L. Stevens (eds.), *Mental Models*, Lawrence Erlbaum Associates, Hillsdale, NJ, Cognitive Science Series, 1983.

[Glaser, Raghaven, Schauble 88]
R. Glaser, K. Raghavan, L. Schauble, *Voltaville: A Discovery Environment to Explore the Laws of DC Circuits*, Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), June 1988, pp. 61-66.

[Goldstein 82]
I.P. Goldstein, *The Genetic Graph: A Representation for the Evolution of Procedural Knowledge*, in D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, 1982, pp. 51-77, ch. 3.

[Gray 88]
W.D. Gray, *PUPS Analysis of LISP (PAL)*, 1988, Draft Hypercard document available from author.

[Gray, Corbet, VanLehn 88]
W.D. Gray, A.T. Corbett, K. VanLehn, *Planning and Implementation Errors in Algorithm Design*, Submitted to 1988 AAAI National Conference, 1988.

[Hansen, Hass 88]
W.J. Hansen, C. Haas, *Reading and Writing with Computers: A Frameworks for Explaining Differences in Performance*, Communication of the ACM, Vol. 231, No. 9, September 1988, pp. 1080-1089.

[Hefley 90]
W. Hefley, *Architectures for Adaptable Human-Machine Interfaces*, in Karwowski, Rahimi (eds.), *Ergonomics of Advanced Manufacturing and Hybrid Automation Systems II*, Elsevier, N.Y., 1990, forthcoming.

[Hempel 65]
C.G. Hempel, *Aspects of Scientific Explanation and Other Essays in the Philosophy of Science*, The Free Press, New York,, 1965.

[Illich 73]
I. Illich, *Tools for Conviviality*, Harper and Row, New York, 1973.

[Johnson, Soloway 84]
W.L. Johnson, E. Soloway, *PROUST: K owledge-Based Program Understanding*, Proceedings of the 7th International Conference on Software Engineering (Orlando, FL), IEEE Computer Society, Los Angeles, CA, March 1984, pp. 369-380.

[Kass 87a]
R. Kass, *Implicit Acquisition of User Models in Cooperative Advisory Systems*, Technical Report MIS-CIS-87-05, LINC LAB 49, University of Pennsylvania, 1987.

[Kass 87b]
R. Kass, *Modelling User Beliefs for Good Explanations*, Technical Report MIS-CIS-87-77, LINC LAB 82, University of Pennsylvania, 1987.

[Kass 88]
R. Kass, *Acquiring a [illegible]del of the User's Belief from a Cooperative Advisory Dialog*, Unpublis[illegible] n.D. Dissertation, University of Pennsylvania, 1988.

[Kass, Finin 87a]
R. Kass, T. Finin, *Rules for the Implicit Acquisition of Knowledge about the User*, 6th National Conference on Artificial Intelligence, AAAI, 1987, pp. 295-300.

[Kass, Finin 87b]
R. Kass, T. Finin, *Modeling the User in Natural Language Systems*, Computational Linguistics, Special Issue on User Modeling, Vol. 14, 1987, pp. 5-22.

[Kass, Finin 88a]
R. Kass, T. Finin, *A General User Modelling Facility*, CHI '88 Conference Proceedings, Human Factors in Computing Systems, ACM, 1988, pp. 145-150.

[Kass, Finin 88b]
R. Kass, T. Finin, *The Need for User Models in Generating Expert System Explanations*, International Journal of Expert Systems, Vol. 4, 1988, pp. 345-375.

[Kass, Finin 89]
R. Kass, T. Finin, *The Role of User Models in Cooperative Interactive Systems*, International Journal of Intelligent Systems, Vol. 4, 1989, pp. 81-112.

[Kelleher 88]
G. Kelleher, *Helping Learning through Explanation and Advice: an overview of EUROHELP*, Artificial Intelligence Tools in Education: Proceedings of the IFIP TC3 Working Conference, IFIP, 1988, pp. 67-72.

[Kelly 85]
V.E. Kelly, *The CRITTER System: Automated Critiquing of Digital Circuit Designs*, Proceedings of the 21st Design Automation Conference, 1985, pp. 419-425.

[Kennedy etal. 88]
A. Kennedy, A. Wildes, L. Elder, W.S. Murray, *Dialogue with Machines*, Cognition, Vol. 30, 1988, pp. 37-72.

[Kieras, Polson 85]
D.E. Kieras, P.G. Polson, *An Approach to the Formal Analysis of User Complexity*, International Journal of Man-Machine Studies, Vol. 22, 1985, pp. 365-394.

[Kintsch 89]
W. Kintsch, *The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension*, in R. Dietrich, C.F. Graumann (eds.), *Language Processing in Social Context*, North Holland, Amsterdam, 1989, pp. 185-209, also published as Technical Report No. 152, Institute of Cognitive Science, University of Colorado, Boulder, CO.

[Kobsa, Wahlster 89]
A. Kobsa, W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag, New York, 1989.

[Langlotz, Shortliffe 83]
C.P. Langlotz, E.H. Shortliffe, *Adapting a Consultation System to Critique User Plans*, Int. J. Man-Machine Studies, Vol. 19, 1983, pp. 479-496.

[Lemke 89]
A.C. Lemke, *Design Environments for High-Functionality Computer Systems*, Unpublished Ph.D. Dissertation, Department of Computer Science, University of Colorado, July 1989.

[Lemke 90]
A.C. Lemke, *Framer: A Knowledge-Based Design Environment for User Interface Design*, IEEE Software (Tools Fair Issue), May 1990.

[Lenat, Prakash, Shepherd 86]
D. Lenat, M. Prakash, M. Shepherd, *CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlenecks*, AI Magazine, Vol. 6, No. 4, Winter 1986, pp. 65-85.

[Lewis 89]
C.H. Lewis, *Explanation and Learning in Procedural Skills*, Technical Report CS-CU-436-89, Department of Computer Science, University of Colorado, Boulder, CO, April 1989.

[London, Clancey 82]
B. London, W.J. Clancey, *Plan Recognition Strategies in Student Modeling: Prediction and Description*, Proceedings of AAAI-82, Second National Conference on Artificial Intelligence (Pittsburgh, PA), 1982, pp. 335-338.

[Manheim, Srivastava, Vlahos, Hsu, Jones 90]
M.L. Manheim, S. Srivastava, N. Vlahos, J. Hsu, P. Jones, *A Symbiotic DSS for Production Planning and Scheduling: Issues and Approaches*, Proceedings of the 23rd Annual Hawaii International Conference on System Sciences, Vol III, J.F. Nunamaker, Jr. (ed.), Jan 1990, pp. 383-390.

[Mastaglio 89]
T. Mastaglio, *Computer-based Critiquing: A Foundation for Learning Environments*, Proceedings TITE '89, 1989 Conference on Technology and Innovations in Training and Education, March 6-9, 1989, Atlanta, GA, Linda Wiekhorst (ed.), 1989, pp. 125-136.

[Mastaglio 90a]
T. Mastaglio, *Paradigms for Intelligent Learning Environments: Tutoring, Coaching and Critiquing*, Proceedings TITE '90, 1990 Conference on Technology and Innovations in Training and Education, March 12-16, 1990, Colorado Springs, CO, 1990, pp. 190-204.

[Mastaglio 90b]
T. Mastaglio, *User Modelling in Computer-Based Critics*, Proceedings of the 23rd Hawaii International Conference on System Sciences, Vol III: Decision Support and Knowledge Based Systems Track, Jay F. Nunamaker, Jr (ed.), IEEE Computer Society, 1990, pp. 403-412.

[Mastaglio, Turnbull 87]
T. Mastaglio, W. Turnbull, *A Learning Environment for the HP Bobcats*, 1987, Final Project Report for CS614 - Spring Term 1987.

[McCall, Fischer, Morch 89]
R. McCall, G. Fischer, A. Morch, *Supporting Reflection-in-Action in the Janus Design Environment*, Proceedings of the CAAD Futures '89 Conference, Harvard University, Cambridge, June 1989, Pre-Publication Edition.

[Mili 88]
F. Mili, *A Framework for a Decision Critic and Advisor*, Proceedings of the 21st Hawaii International Conference on System Sciences, Jan 1988, pp. 381-386.

[Mili, Manheim 88]
F. Mili, M.L. Manheim, *And What Did Your DSS Have to Say About That: Intoduction to the DSS Minitrack on Active and Symbiotic Systems*, Proceedings of the 21st Hawaii International Conference on System Sciences, Jan 1988, pp. 1-2.

[Miller 79]
M.L. Miller, *A Structured Planning and Debugging Environment for Elementary Programming*, in D.H. Sleeman, J.S. Brown (eds.), *International Journal of Man-Machine Studies*, Academic Press, 1979, pp. 79-95.

[Miller 86]
P. Miller, *Expert Critiquing Systems: Practice-Based Medical Consultation by Computer*, Springer-Verlag, New York - Berlin, 1986.

[Moore 87]
J. Moore, *Explanations in Expert Systems*, Technical Report, USC/Information Sciences Institute, 9 December 1987.

[Moore 89]
J. Moore, *Responding to 'HUH': Answering Vaugely Articulated Follow-up Questions*, Human Factors in Computing Systems, CHI'89 Conference Proceedings (Austin, TX), ACM, New York, May 1989, pp. 91-96.

[Moran 81]
T.P. Moran, *An Applied Psychology of the User*, ACM Computing Surveys, Vol. 13, No. 1, March 1981, pp. 1-31.

[Murray 88]
D. Murray, *A Survey of User Cognitive Modelling*, Technical Report NPL Report 92/87, DITC, National Physical Laboratory, Teddinton, Middlesex TW11 0LW, UK, 1988.

[Murray Benyon 89]
D. Murray, D. Benyon, *Models and Designer's Tools for Adaptive Systems*, Technical Report, DITC, National Physical Laboratory, Teddinton, Middlesex TW11 0LW, UK, 1989.

[Neches, Swartout, Moore 85]
R. Neches, W.R. Swartout, J.D. Moore, *Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development*, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1337-1351.

[Nieper 83]
H. Nieper, *KAESTLE: Ein graphischer Editor fuer LISP-Datenstrukturen*, Studienarbeit 347, Institut fuer Informatik, Universitaet Stuttgart, 1983.

[Norman 82]
D.A. Norman, *Five Papers on Human-Machine Interaction*, CHIP Report 112, University of California, San Diego, May 1982.

[Norman 86]
D.A. Norman, *Cognitive Engineering*, in D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 31-62, ch. 3.

[Papert 80]
S. Papert, *Mindstorms: Children, Computers and Powerful Ideas*, Basic Books, New York, 1980.

[Paris 87]
C.L. Paris, *The Use of Explicit User Models in Text Generation: Tailoring to a User's Level of Expertise*, Unpublished Ph.D. Dissertation, Columbia University, 1987.

[Paris 89]
C.L. Paris, *The Use of Explicit User Models in a Generation System for Tailoring Answer to a User's Level of Expertise*, in A. Kobsa, W. Wahlster (eds.), *User Models in Dialog Systems*, Springer-Verlag, New York, 1989, pp. 200-232.

[Polson, Richardson 88]
M.C. Polson, J.J. Richardson (eds.), *Foundations of Intelligent Tutoring Systems*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[Psotka, Massey, Mutter 88a]
J. Psotka, L.D. Massey, S. Mutter, *Intelligent Instructional Design*, in J. Psotka, L.D. Massey, S. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ , 1988, pp. 113-118.

[Psotka, Massey, Mutter 88b]
J. Psotka, L.D. Massey, S.A. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1988.

[Raghaven, Schultz, Glaser, Schauble 90]
K. Raghaven, J. Schultz, R. Glaser, L. Schauble, *A Computer Coach for Inquiry Skills*, 1990, draft submission to Intelligent Learning Environments Journal.

[Reeves 90]
B. Reeves, *Finding and Choosing the Right Object in a Large Hardware Store -- An Empirical Study of Cooperative Problem Solving among Humans*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1990, forthcoming.

[Rennels 87]
G.D. Rennels, *A Computational Model of Reasoning from the Clinical Literature*, Springer Verlag, Lecture notes in medical informatics, 1987.

[Rennels, Shortiliffe, Stockdale, Miller 89]
G.D. Rennels, E.H. Shortliffe, F.E. Stockdale, P.L. Miller, *A Computational Model of Reasoning from the Clinical Literature*, AI Magazine, Vol. 10, No. 1, Spring 1989, pp. 49-56.

[Rich 79]
E. Rich, *Building and Exploiting User Models*, Unpublished Ph.D. Dissertation, Carnegie-Mellon University, 1979.

[Rich, Waters 88]
C.H. Rich, R.C. Waters, *Automatic Programming: Myths and Prospects*, Computer, Vol. 21, No. 8, August 1988, pp. 40-51.

[Rich, Waters 90]
C. Rich, R.C. Waters, *The Pogrammer's Apprentice*, ACM Press, New York, 1990.

[Riemann, Raghaven, Glaser 88]
P. Riemann, K. Raghaven, R. Glaser, *Refract, a Discovery Environment for Geometrical Optics*, Technical Report, Learning Research & Development Center, University of Pittsburgh, 1988.

[Sacerdoti 75]
E.D. Sacerdoti, *A Structure for Plans and Behavior*, Technical Note 109, Stanford Research Instititute, Stanford, CA, 1975.

[Schank 86]
R.G. Schank, *Explanation Patterns: Understanding Mechanically and Creatively*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[Schank, Abelson 77]
R.C. Schank, R.P. Abelson, *Scripts, Plans, Goals, and Understanding*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.

[Schiff, Kandler 88]
J. Schiff, J. Kandler, *Decisionlab: A System Designed for User Coaching in Managerial Decision Support*, Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), June 1988, pp. 154-161.

[Schmidt, Sridharan, Goodson 78]
C.F. Schmidt, N.S. Sridharan, J.L. Goodson, *The Plan Recognition Problem: An Intersection of Psychology and Artificial Intelligence*, Artificial Intelligence, Vol. 11, 1978, pp. 45-83.

[Scott, Clancey, Davis, Shortliffe 84]
A.C. Scott, W.J. Clancey, R. Davis, E.H. Shortliffe, *Methods for Generating Explanations*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 338-362, ch. 18.

[Seidel, Weddle 87]
R.J. Seidel, P.D. Weddle, *Computer-Based Instruction in Military Environments*, Plenum Press, New York, 1987.

[Simon 81]
H.A. Simon, *The Sciences of the Artificial*, The MIT Press, Cambridge, MA, 1981.

[Skulley 88]
J. Skulley, *The Relationship Between Business and Higher Education: A Perspective on the 21st Century*, CACM, Vol. 32, No. 9, September 1988, pp. 1056-1061.

[Sleeman 83]
D.H. Sleeman, *Inferring Student Models for Intelligent Computer-Aided Instruction*, in R.S. Michalski, J.G. Carbonell, T.M. Mitchell (eds.), *Machine Learning: An Artificial Intelligence Approach*, Morgan Kaufmann Publishers, Los Altos, CA, 1983, pp. 483-508, ch. 16.

[Sleeman 84]
D.H. Sleeman, *UMFE: A User Modeling Front End Subsystem*, Working Paper HPP-84-12, Heuristic Programming Project, Department of Computer Science, Stanford University, April 1984.

[Sleeman, Brown 82]
D.H. Sleeman, J.S. Brown (eds.), *Intelligent Tutoring Systems*, Academic Press, London - New York, Computer and People Series, 1982.

[Sowa 84]
J.F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, MA, 1984.

[Steele 84]
G.L. Steele, *Common LISP: The Language*, Digital Press, Burlington, MA, 1984.

[Steele 87]
R.L. Steele, *An Expert System Application in Semicustom VLSI Design*, Proceedings of the 24th IEEE/ACM Design Automation Conference (Miami Beach, FL), IEEE Computer Society Press, Los Angeles, CA, 1987, pp. 679-686.

[Steele 88]
R.L. Steele, *Cell-Based VLSI Design Advice Using Default Reasoning*, Proceedings of 3rd Annual Rocky Mountain Conference on AI, Rocky Mountain Society for Artificial Intelligence, Denver, CO, 1988, pp. 66-74.

[Strunk, White 57]
W. Strunk, E.B. White, *The Elements of Style*, Harcourt-Brace, New York, 1957.

[Suchman 87]
L.A. Suchman, *Plans and Situated Actions*, Cambridge University Press, New York, 1987.

[Sussman 75]
G.J. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, New York, 1975.

[Swartout 81]
W.R. Swartout, *Explaining and Justifying Expert Consulting Programs*, Proceedings of the Seventh International Joint Conference on Artificial Intelligence, A. Drinan (ed.), 1981, pp. 815-822.

[Swartout 83]
W.R. Swartout, *XPLAIN: A System for Creating and Explaining Expert Consulting Programs*, ISI Reprint Series ISI/RS-83-4, Information Sciences Institute, University of Southern California, Marina del Rey, CA, July 1983.

[Teach, Shortliffe 84]
R.L. Teach, E.H. Shortliffe, *An Analysis of Physicians' Attitudes*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 635-652, ch. 34.

[Turban, Watkins 86]
E. Turban, P.R. Watkins, *Integrating Expert Systems and Decision Support Systems*, MIS Quarterly, Vol. , June 1986, pp. 120-136.

[VanLehn 88]
K. VanLehn, *Toward a Theory of Impasse-Driven Learning*, in H. Mandl, A. Lesgold (eds.), *Learning Issues for Intelligent Tutoring Systems*, Springer-Verlag, New York, 1988, pp. 19-41, ch. 2.

[Wahlster, Kobsa 88]
W. Wahlster, A. Kobsa, *User Models in Dialog Systems*, Technical Report 28, Universitaet des Saarlandes, FB 10 Informatik IV, Sonderforschungsbereich 314, Saarbruecken, FRG, 1988.

[Wallis, Shortliffe 84]
J.W. Wallis, E.H. Shortliffe, *Customized Explanations Using Causal Knowledge*, in B.G. Buchanan, E.H. Shortliffe (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, Addison-Wesley Publishing Company, Reading, MA, 1984, pp. 371-388, ch. 20".

[Waterman etal. 86]
D.A. Waterman, J. Paul, B. Florman, J.R. Kipps, *An Explanation Facility for the ROSIE Knowledge Engineering Language*, RAND Corporation, Santa Monica, Calif., 1986.

[Weiss 88]
E.H. Weiss, *Breaking the Grip of User Manuals*, Asterisk -- Journal of ACM SIGDOC, Vol. 14, Summer 1988, pp. 4-11.

[Wenger 87]
E. Wenger, *Artificial Intelligence and Tutoring Systems*, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Wiener 80]
J.L. Wiener, *BLAH, A System Which Explains its Reasoning*, Artificial Intelligence, Vol. 15, 1980, pp. 19-48.

[Wilensky 84]
R. Wilensky, *LISPcraft*, W.W. Norton & Company, New York - London, 1984.

[Wilkins, Clancey, Buchanan 88]
D.C. Wilkins, W.J. Clancey, B.G. Buchanan, *Using and Evaluating Differential Modelling in Intelligent Tutoring and Apprentice Learning Systems*, in J. Psotka, L.D. Massey, S. Mutter (eds.), *Intelligent Tutoring Systems: Lessons Learned*, Lawrence Erlbaum Associates, Hillsdale, NJ , 1988, pp. 257-277.

[Williams 90]
M.D. Williams, *The Pragmatics of Knowledge-based Interface Design*, Working Notes of AAAI Spring Symposium Series: Knowledge-Based Human-Computer Interaction, AAAI, 1990, pp. 132-135.

[Winograd, Flores 86]
T. Winograd, F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*, Ablex Publishing Corporation, Norwood, NJ, 1986.

[Winston, Horn 81]
P.H. Winston, B.K.P. Horn, *LISP*, Addison-Wesley Publishing Company, Reading, MA, 1981.

[Wipond, Jones 88]
K. Wipond, M. Jones, *Curriculum and Knowledge Representation in a Knowledge-Based System for Curriculum Development*, Proceedings of the International Conference on Intelligent Tutoring Systems (Montreal, Canada), June 1988, pp. 97-102.

[Young, Barnard, Simon, Whittington 89]
R.N. Young, P. Barnard, T. Simon, J. Whittington, *How Would Your Favorite User Model Cope with these Scenarios*, SIGCHI Bulletin, Vol. 20, No. 4, 1989, pp. 51-55.

# APPENDIX A

## USER MODELS REFERENCED IN DISSERTATION

This is a table of the different user modelling systems discussed throughout the dissertation that provided insight or ideas for this work.

**Characteristics of User Models Referenced in the Dissertation**

| SYSTEM NAME | APPLICATION DOMAIN | SYSTEM PURPOSE | SYSTEM PARADIGM | USER MODEL PURPOSE | INFORMATION REPRESENTED | ACQUISISITON METHODOLOGY |
|---|---|---|---|---|---|---|
| GRUNDY | literature | suggests books | advisory-dialog | guide advice | Personality traits | explicit query & stereotyping |
| GUMAC | financial | investment counselling | advisory-dialog | aid NL generation | user goals & beliefs | implicit parse of dialog |
| UMFE | generic | explanation | tutoring | guide explanation | domain concepts | implicit |
| WEST | mathematics | learning | coaching | intervention & advice | rules | implicit |
| WUSOR-II | WUMPUS computer game | learning | coaching | advice & explanation | genetic graph of rules | implicit & explicit |
| LISP TUTOR | LISP programming | learning | tutoring | select tutoring strategies | production rules | implicit |
| KNOME | UNIX Operating System | intelligent help | advisory | guide responses | expertise level | stereotyping |
| LISP-CRITIC | LISP programming | improve programs | critiquing | guide explanation | domain entities | direct & indirect implicit |

# APPENDIX B

## SAMPLE USER MODEL

This appendix shows the final user model at the conclusion of the scenario presented in Chapter 4:

```
#<USER-MODEL 54237715> is an instance of class #<Standard-Class USER-MODEL 263574537>:
 The following slots have :INSTANCE
 SHOW-WHY-BETTER?              NIL
 SHOW-EXPLANATION?             T
 SHOW-NEW-CODE?                NIL
 SHOW-OLD-CODE?                T
 PROMPT-RULES                  NIL
 DEFAULT-ACCEPT-RULES          NIL
 RULES-TURNED-OFF              NIL
 RULES-EXPLAINED               (COND-ERASE-PRED.T DEMORTGAN COND-TO-IF-ELSE)
 FUNCTIONS-EXPLAINED           NIL
 CONCEPTS-EXPLAINED            (LISTS LISP-ATOM FALSE/EMPTY-LIST/NIL TRUE/NON-NIL TESTS
                                CONDITIONALS PREDICATES)
 RULES-FIRED                   ((COND-ERASE-T.NIL (TIMES-FIRED . 1) (TIMES-ACCEPTED . 1)
                                 (TIMES-REJECTED . 0))
                                (COND-ERASE-PRED.T (TIMES-FIRED . 1) (TIMES-ACCEPTED . 1)
                                 (TIMES-REJECTED . 0))
                                (DE-MORGAN (TIMES-FIRED . 1) (TIMES-ACCEPTED . 1)
                                 (TIMES-REJECTED . 0))
                                (COND-TO-IF-ELSE (TIMES-FIRED . 3) (TIMES-ACCEPTED . 3)
                                 (TIMES-REJECTED . 0)))
 RULES-KNOWN                   ((COND-ERASE-T.NIL . D2) (COND-ERASE-PRED.T . D2)
                                (DE-MORGAN . D2) (COND-TO-IF-ELSE . D1))
 FUNCTIONS-KNOWN               ((NULL . D2) (NOT . D2) (OR . D2) (AND . D2) (IF . D2)
                                (COND . D2))
 CONCEPTS-KNOWN                ((LOGICAL-FUNCTIONS . D2) (SYMBOLIC-EXPRESSION . D1)
                                (LISTS . D1) (EVALUATION . D1) (TESTS . D1)
                                (CONDITIONALS . D1) (PREDICATES . D1)
                                (INTERNAL-REPRESENTATION . D2) (SIDE-EFFECTS . D2)
                                (CONS-CELL . D2) (VARIABLES . D1) (SCOPE . D1)
                                (LISP-ATOM . D1) (ARGUMENTS . D1) (FALSE/EMPTY-LIST/NIL . D1)
                                (TRUE/NON-NIL . D1) (FUNCTIONS . D1))
 PROGRAMMING-LANGUAGE-EXPERIENCE
                               (PASCAL C)
 USER-GOAL                     SIMPLIFY
 TIMES-LCR-INVOKED             3
 DATE-LAST-UPDATED             "2/23/90 13:52:27"
 USER-HOME-DIRECTORY           #P"MUNCH:>SCENARIO-USER>"
 NAME                          SCENARIO-USER
```

# APPENDIX C

## INFERENCE METHODS IN USER MODELLING COMPONENT

This appendix shows some of the code that implements the acquisition subcomponent of the user modelling component for LISP-CRITIC. The implemented methods to implicitly infer information about the user's domain knowledge are shown below.

```
;;;;        DIRECT METHODS
;;;These are the interface functions that other system components call to
;;;let the user model know that a user has taken certain actions.  The first
;;;set of functions are the direct methods;  these infer that certain
;;;information should be added to the user model.  These functions can be
;;;viewed as the implementation of a message passing protocol between modules.

(defun TELL-USERMODEL-RULE-ACCEPTED (rule-name)
  ;;if a user accepts a rule we infer that they understand that rule at the d2 level
  (add-to-user-model *current-usermodel* rule-name 'd2)
  ;;we also add the fact that this rule has fired and was accepted to rules-fired slot of user model
  (update-rules-fired-slot rule-name 'accepted))

(defun TELL-USERMODEL-RULE-REJECTED (rule-name)
  ;;if a user rejects a rule we infer that they understand that rule at the d2 level
  (add-to-user-model *current-usermodel* rule-name 'd2)
  ;;we also add the fact that this rule has fired and was rejected to rules-fired slot of user model
  (update-rules-fired-slot rule-name 'rejected))

(defun TELL-USERMODEL-CONCEPT-EXPLAINED (concept)
  ;;after a concept is explained we place that concept in the user model at
  ;;level d2, unless it is already present, in that case we upgrade its level to d1
  ;;when the concept is  already known at level d1, we do nothing
  (case (cdr (assoc concept
                    (concepts-known *current-usermodel*)))
    ((nil) (add-to-user-model *current-usermodel* concept 'd2))
    ('d2 (add-to-user-model *current-usermodel* concept 'd1)))
  (pushnew concept (concepts-explained *current-usermodel*)))

(defun TELL-USERMODEL-FUNCTION-EXPLAINED (function)
  ;;after a function is explained we place that function in the user model at
  ;;level d2, unless it is already present, in that case we upgrade its level to d1
  ;;when the function is  already known at level d1, we do nothing
  (case (cdr (assoc function
                    (functions-known *current-usermodel*)))
    ((nil) (add-to-user-model *current-usermodel* function 'd2))
    ('d2 (add-to-user-model *current-usermodel* function 'd1)))
  (pushnew function (functions-explained *current-usermodel*)))
```

```
(defun TELL-USERMODEL-RULE-EXPLAINED (rule)
  ;;after a rule  is explained we place that rule  in the user model at
  ;;level d2, unless it is already present, in that case we upgrade its level to d1
  ;;when the rule is  already known at level d1, we do nothing
  (case (cdr (assoc rule
                    (rules-known *current-usermodel*)))
    ((nil) (add-to-user-model *current-usermodel* rule 'd2))
    ('d2 (add-to-user-model *current-usermodel* rule 'd1)))
  (pushnew rule (rules-explained *current-usermodel*)))


(defun TELL-USERMODEL-HYPERTEXT-ACCESS (string)
  ;;function that informs the user model that the user has selected a mouse
  ;;sensitive object in the context of an explanation to request access
  ;;document examiner information on that object.  It acts as a filter to
   ;;determine if the object is something that exists in our domain model; if
  ;;so the user model is informed that an update is appropriate
  (let ((object (intern (string-upcase string))))
    (cond
      ((find-concept-by-name object)    ;;object is a concept
       (tell-usermodel-concept-hypertext-access object))
      ((find-function-by-name object)   ;;object is a function
       (tell-usermodel-function-hypertext-access object))
      (t nil)                           ;;object not in domain model
      )))                               ;;for now do nothing

(defun TELL-USERMODEL-CONCEPT-HYPERTEXT-ACCESS (concept)
  ;;if a user accesses the document examiner documentation associated with
  ;;a particular concept then we also add that concept to their user model
  ;;in a similar manner to when the concept is explained directly
  (case (cdr
          (assoc concept (concepts-known *current-usermodel*)))
    ((nil) (add-to-user-model *current-usermodel* concept 'd2))
    ('d2 (add-to-user-model *current-usermodel* concept 'd1))
    ))

(defun TELL-USERMODEL-FUNCTION-HYPERTEXT-ACCESS (function)
  ;;like the previous function but the hypertext access access is for a function
  (case (cdr
          (assoc function (functions-known *current-usermodel*)))
    ((nil) (add-to-user-model *current-usermodel* function 'd2))
    ('d2 (add-to-user-model *current-usermodel* function 'd1))
    ))

(defun TELL-USERMODEL-RULE-STATUS-CHANGED (rule status)
  ;;if the user makes use of the capability in the ILC interface to specify the
  ;;default action taken for a rule (e.g., always-execute, turn-off, prompt) then
  ;;we infer they have a some understanding of that rule; its level in
  ;;the user model is set to d2.
  (add-to-user-model *current-usermodel* rule 'd2)
  (change-rule-status-for-user *current-usermodel* rule :state status ))

(defun TELL-USERMODEL-RULE-COMMENT-ADDED (rule)
  ;;if the user has something to say about a rule that they add to the system
  ;;documentation for that rule (using the "add rule comment" capability in
  ;;ILC's interface), then we infer that they have a sophisticated understanding
  ;;of that rule; its level in the user model is set to d1
  (add-to-user-model *current-usermodel* rule 'd1))
```

```
;;;;        INDIRECT METHODS
;;;The second set of methods are the indirect methods.  These
;;;note changes to the user model and use that information to
;;;infer changes to other objects in the user model.  They propagate
;;;any changes through the model by using that change as a
;;;cue and making use of "dependent-on" links in the domain model.
;;;These methods are implemented as around methods on updates to the
;;;"knowledge" slots (rules, functions and concepts known) in the user model.

(defmethod (SETF rules-known) :before (new-rules-known (user user-model))
   ;;This method implements the notion that any rule which a user knows implies
   ;;that they also know the functions and concepts underlying that rule, if the
   ;;level of knowledge is "d1" then the knowledge level of both underlying functions
    ;;and concepts is "d2"; if the rule knowledge level is "d2" then only underlying functions
   ;;are inferred to be at level "d2" and nothing is inferred about concepts for now
          (let* ((rule-cons (find-update
                        (rules-known user) new-rules-known))
                 (rule (car rule-cons))
                 (level (cdr rule-cons))
                 (rule-instance  (find-rule-by-name rule))
                 (functions-list (functions rule-instance))
                 (concepts-list
                   (find-all-dependent-on-concepts rule-instance))
                 (new-level 'd2))
            (if (eq level 'd1)
                ;;change model to reflect "d2" knowledge level for concepts underlying this rule
                (dolist (each-concept concepts-list)
                  (add-to-user-model *current-usermodel*
                                    (name each-concept) new-level)))
            ;;for both knowledge levels "d1" and "d2" for this rule in the user model
            ;;set knowledge level for underlying functions to "d2"
            (dolist (each-function functions-list)
                    (add-to-user-model *current-usermodel* each-function new-level))))


(defmethod (SETF functions-known) :before (new-functions-known (user user-model))
   ;;This method is similar to the previous one, except it runs after updates to the
   ;;functions known slot in the user model.  It implements indirect methods that allow
   ;;us to set the knowledge level for concepts a given function is dependent on to
   ;;"d1" if the function's knowledge level has gone to "d1" and "d2" if the function's
   ;;knowledge level has gone to "d2"
          (let* ((function-cons (find-update
                                 (functions-known user) new-functions-known))
                 (fun-name (car function-cons))
                 (level (cdr function-cons))
                 (concepts-list (find-all-dependent-on-concepts
                                 (find-function-by-name fun-name)))
                 (new-level (if (eq level 'd1) 'd1 'd2)))
            (dolist (each-concept concepts-list)
                  (add-to-user-model *current-usermodel*
                                    (name each-concept) new-level))))

(defmethod (SETF concepts-known) :before (new-concepts-known (user user-model))
   ;;This method is for concepts.  It uses the an update to the "concepts-known" slot in
   ;;the user model to infer that dependent-on concepts are also known at the same level
          (let* ((concept-cons (find-update
                                (concepts-known user) new-concepts-known))
                 (concept (car concept-cons))
                 (level (cdr concept-cons))
                 (concepts-list (find-all-dependent-on-concepts        ;;list of instances
                                 (find-concept-by-name concept))))    ;;of these concepts
            (if (eq level 'd1)
                ;;if a concept belongs to  a user's "d1" then its dependent-on concepts
                ;;also belong in this user's "d1"
                (dolist (each-concept concepts-list)
                  (add-to-user-model *current-usermodel*
                                    (name each-concept) 'd1))
                ;;on the other hand if a concepts belongs to a user's "d2" then we infer that
                ;;dependent-on concepts are known at level "d2" or better -- concepts
                ;;already in the model at "d2" go to "d1" those absent are added at level "d2"
                (dolist (each-concept concepts-list)
                  (let ((this-concept (name each-concept)))
                    (if (null (how-well-does-user-know *current-usermodel* this-concept))
                        (add-to-user-model *current-usermodel*
                                         this-concept 'd2)
                        (add-to-user-model *current-usermodel*
                                         this-concept 'd1)))))))
```

The following portion of the framework for the acquisition subcomponent is provided so that the other three categories of acquisition techniques (as described in Section 3-2) could be integrated into the user modelling component. Methods implemented in this dissertation are those shown above; all are implicit acquisition methods.

```
;;;;        STATISTICAL METHODS
;;;This set of methods uses information that the statistical
;;;analysis module accumulates for the user.  They use the statistical

;;;data to infer specific changes or additions to the user model.
;;;These are incomplete and would require a significant effort
;;;"analyzing" the statistical information to see exactly what
;;;inference it should trigger.

(defun FUNCTION-USED (function) (ignore function))

;;;;        TUTORING METHODS
;;;This set of methods is the interface with a tutoring component
;;;for updating the model contents. Subjects on which the user
;;;has received specific tutoring episodes either at their request
;;;or as the result of suggestions from LISP-Critic will be used
;;;to infer knowledge levels in the user model

(defun TELL-USER-MODEL-CONCEPT-TUTORED (concept) (ignore concept)
  ;;any concept on which a user has been tutored will be set to level 'd2
  )

(defun TELL-USER-MODEL-FUNCTION-TUTORED (function) (ignore function)
  ;;any function on which a user has been tutored will be set to level 'd2
  )

;;;;        EXPLICIT METHODS
;;;This set of methods uses explicit acquisition techniques to either
;;;establish an initial user model or to interactively query the user
;;;when the system needs additional information or clarification
;;;regarding the knowledge state of the user.

(defun ASK-USER-ABOUT-THEIR-KNOWLEDGE ()
  ;;set of interactive queries that question user about their expertise
  )
```

# APPENDIX D

## ACCESS METHODS IN THE USER MODELLING COMPONENT

This appendix shows the set of access functions that are available for the user modelling component itself as well as for other system components to "interrogate" the contents of model instances. The scheme for implementing access is to provide general accessor functions to other system components which when called cause the user modelling component to invoke the appropriate CLOS method on the user model instance representing the programmer presently using the system — that instance is bound to the global variable **current-usermodel*.*

```
;;;;                    FUNDAMENTAL METHODS
;;;; Methods that are used to access the user model by other system
;;;; components and processes.  These are sometimes referred to as
;;;; the uniformed methods -- they contain no knowledge about how
;;;; to modify or update the user model.

(defmethod WHICH-DOES-USER-KNOW? ((user user-model) lisp-objects)
  ;; Returns a list of those concepts, rules or functions that a user knows
  ;; that are in the argument list "lisp-objects"
  (cond
    ((find-concept-by-name (car lisp-objects))
     (intersection lisp-objects (extract-names
                                 (concepts-known user))))
    ((find-function-by-name (car lisp-objects))
     (intersection lisp-objects (extract-names
                                 (functions-known user))))
    (t (intersection lisp-objects (extract-names
                                   (rules-known user))))))

(defmethod WHICH-DOES-USER-NOT-KNOW? ((user user-model) lisp-objects)
  ;;Returns a list of those concepts, rules, or functions that a user DOES NOT KNOW
  ;;that are in the argument list "lisp-objects"
  (cond
    ((find-concept-by-name (car lisp-objects))
     (set-difference lisp-objects (extract-names
                                   (concepts-known user))))
    ((find-function-by-name (car lisp-objects))
     (set-difference lisp-objects (extract-names
                                   (functions-known user))))
    (t (set-difference lisp-objects (extract-names
                                     (rules-known user))))))
```

```
(defmethod WHICH-DOES-USER-NOT-KNOW-WELL? ((user user-model) lisp-objects)
  ;;From the argument list, lisp-objects, this methods returns any concept, function or
  ;;critic rule that the user does not know at level d1
  (cond
    ((find-concept-by-name (car lisp-objects))
     (mapcan #'(lambda (concept)
                 (unless (eq (how-well-does-user-know *current-usermodel* concept) 'd1)
                   (list concept)))
             lisp-objects))
    ((find-function-by-name (car lisp-objects))
     (mapcan #'(lambda (function)
                 (unless (eq (how-well-does-user-know *current-usermodel* function) 'd1)
                   (list function)))
             lisp-objects))
    (t (mapcan  #'(lambda (lcr-rule)
                 (unless (eq (how-well-does-user-know *current-usermodel* lcr-rule) 'd1)
                   (list lcr-rule)))
                lisp-objects))
    ))


(defun TOPICS-USER-DOES-NOT-KNOW (list-of-topics)
  ;;Filters the topics (concepts) in list-of-topics through the user model and
  ;;returns those the user does not know at all
  (which-does-user-not-know? *current-usermodel* list-of-topics))

(defun EXTRACT-NAMES (a-list)
  ;;slots in usermodel: concepts, functions, and rules-known are a-lists; this function extracts
  ;;the names (car of each item) from that list so comparison operations can be performed
  (mapcar 'car a-list))


(defun HOW-WELL-DOES-USER-KNOW-THESE (objects-list)
  ;;accepts a list of LISP-OBJECTS, determines if each one is in the usermodel and at what level
  ;;the user knows the object.  returns three lists: objects known at levels d0, d1 and d2.
  (let ((d1-list nil)
        (d2-list nil)
        (d0-list nil))
    (dolist (object objects-list)
      (case (how-well-does-user-know *current-usermodel* object)
        (d1 (push object d1-list))
        (d2 (push object d2-list))
        (d0 (push object d0-list))))
    (list (list 'd0 d0-list) (list 'd2 d2-list) (list 'd1 d1-list))))

(defmethod HOW-WELL-DOES-USER-KNOW ((user user-model) lisp-object-name)
  ;;returns the "d" level of the argument "lisp-object"
  (cond
    ((cdr (assoc lisp-object-name (concepts-known user))))
    ((cdr (assoc lisp-object-name (functions-known user))))
    ((cdr (assoc lisp-object-name (rules-known user))))
    (t 'd0)))                                      ;;"d0" means not known at all

(defmethod ADD-TO-USER-MODEL ((user user-model) lisp-object level)
     ;; Adds argument lisp-object to appropriate slot in this individuals user model
  (cond
    ((find-concept-by-name lisp-object)                    ;;this is a LISP concept
     ;;if a new concept for this user then add it to the concepts-known slot in the user model
     (if (null (assoc lisp-object (concepts-known user)))
         (setf (concepts-known user)
               (acons lisp-object level (concepts-known user)))
         ;;else replace the current level with the new level - do this even if they are the same
         (unless (eq 'd1 (cdr (assoc lisp-object (concepts-known user))))
           (rplacd (assoc lisp-object (concepts-known user)) level)))
     )
    ((find-function-by-name lisp-object)                   ;;this is a LISP function
     ;;if a new functions for this user, add it to functions-known slot in the user model
     (if (null (assoc lisp-object (functions-known user)))
         (setf (functions-known user)
               (acons lisp-object level (functions-known user)))
         ;;else replace the current level with the new level - do this even if they are the same
         (unless (eq 'd1 (cdr (assoc lisp-object (functions-known user))))
           (rplacd (assoc lisp-object (functions-known user)) level)))
     )
    (t                                                      ;;default case - its a rule
     ;;if a new rule for this user, add it to rules-known slot in the user model
     (if (null (assoc lisp-object (rules-known user)))
         (setf (rules-known user)
               (acons lisp-object level (rules-known user)))
         ;;else replace the current level with the new level - do this even if they are the same
         (unless (eq 'dl (cdr (assoc lisp-object (rules-known user))))
           (rplacd (assoc lisp-object (rules-known user)) level)))
     )
    ))

(defmethod ALREADY-EXPLAINED? ((user user-model) lisp-object-name)
  ;; Returns true if the argument "lisp-object" has been explained to the user
  ;; in the past otherwise returns nil
  (cond
    ((find-concept-by-name lisp-object-name)
     (member lisp-object-name (concepts-explained user)))
    ((find-function-by-name lisp-object-name)
     (member lisp-object-name (functions-explained user)))
    (t
     (member lisp-object-name (rules-explained user)))
    ))


(defmethod CHANGE-RULE-STATUS-FOR-USER ((user user-model)  rule &key state)
  ;; Changes the state of the argument "rule" depending on the value of argument "state"
  ;; by updating the appropriate slots -- probably should be using a single slot with an a-list
  (case state
    (:always-accept
     ;; enable this rule by removing it from the list of rules turned off or prompted for
     (if (not (member rule (default-accept-rules user)))
         (push rule (default-accept-rules user)))
     (setf (rules-disabled user)
          (delete rule (rules-disabled user)))
     (setf (prompt-me-rules user)
           (delete rule (prompt-me-rules user))))
    (:always-reject
     ;; disable this rule, add it to the list of rules turned off, remove it from others
     (if (not (member rule (rules-disabled user)))
           ;;but only if it is not already there
           (push rule (rules-disabled user)))
     (setf (prompt-me-rules user)
           (delete rule (prompt-me-rules user)))
     (setf (default-accept-rules user)
           (delete rule (default-accept-rules user))))
    (:prompt-me
     ;;add this rule to list of those which system will ask the user about
     (if (not (member rule (prompt-me-rules user)))
         ;;but only if it is not already on that list
         (push rule (prompt-me-rules user)))
     (setf (rules-disabled user)
           (delete rule (rules-disabled user)))
     (setf  (default-accept-rules user)
            (delete rule (default-accept-rules user))))
     ))

;;;; Methods that provide a persistent capability to the user modelling
;;;; component.  These allow the user model to be saved to and
;;;; retrieved from a file so that it can be reused and iteratively
;;;; enhanced during subsequent uses of LISP-Critic.

(defmethod SAVE-OBJECT ((self save-mixin) &optional stream)
  ;;;method on the mixin class to save an individual model to a file that can
  ;;;can be loaded during the next session with the system. This actually works
  ;;;by creating code and putting it into that file so that when the file is loaded
  ;;;it will make an instance of the user model in the local environment.  That
  ;;;instance contains the information about the user built-up over time.
  (let ((initargs nil)
        (other-inits nil)
        (class (class-of self)))
    (with-slots (name) self
      (dolist (slot (pcl::class-slots class))
        (let ((slot-name (pcl::slotd-name slot)))
          (when (slot-boundp self slot-name)
            (let ((value (slot-value self slot-name))
                  (initarg (car (pcl::slotd-initargs slot))))
              (if initarg
                  (setf initargs '(',initarg ',value . ,initargs))
```

```
                    (push (list slot-name value) other-inits))))))
          (print `(setf (get ',name 'instance)
                         ;; (type-of self) does not work properly in PCL
                         ;; use (class-name (class-of self)) instead
                         (make-instance ',(class-name class)
                                        ,@initargs))
                 stream)
          (dolist (init other-inits)
            (print `(setf (slot-value (get ',name 'instance) ',(first init))
                          ',(second init))
                   stream)))))


(defmethod SAVE-USERMODEL-TO-FILE ((user user-model))
  ;; save an image of the user's model in his directory so that it can be used the
  ;; next time that LISP-Critic is invoked -- implements the persistent user model
  (let ((*package* (find-package 'tums)))
    (with-open-file (stream (merge-pathnames (user-directory user) "zlc-usermodel.lisp.newest")
                            :direction :output
                            :if-does-not-exist :create)
      (format stream ";;; -*- Mode: LISP; Syntax: Common-Lisp; Base: 10; Package: TUMS -*-")
      (save-object user stream))))

(defun LOAD-USER-MODEL-FROM-FILE ()
  ;; retrieves from the user's Symbolics home directory his user model for use by LISP Critric
    (load (merge-pathnames (fs:user-homedir) "zlc-usermodel.lisp.newest")
          :if-does-not-exist nil))


(defun CREATE-NEW-USER-MODEL ()
  ;; use this function to instantiate a new user model when one does not already exist
  ;; it can be expanded to use either a standard default model for all new users or to
  ;; initiate explicit acquisition by asking the users questions in order to classify him or
  ;; her into a stereotype to use as a starting point -- for now we will use an initial
  ;; default model that is empty
          (setf *current-usermodel* (make-instance 'user-model)))

(defun ACTIVATE-USER-MODEL ()
  ;; This function is provided to the main critic engine to either retrieve a user model
  ;; from a file for this user (one that was created during previous interactions with
  ;; the system) or create a default for a first time user.
  (if (null (load-user-model-from-file))
      (create-new-user-model)
      (setf *current-usermodel*
            (get (intern
                   (string-upcase zl:user-id) 'tums) 'instance))))

(defun TELL-USERMODEL-CRITIQUE-SESSION-COMPLETE ()
  ;; Interface function that is used to inform the usermodel that the critiquing component
  ;; has just completed a session with the user and certain "clean-up" processes can be run
  (incf (times-invoked *current-usermodel*))
  (setf (date-updated *current-usermodel*)
        (time:print-current-time nil))
  (let* ((zlc-window (dw::find-program-window 'zlc::zlc-frame :selected-ok t))
         (program (scl:send zlc-window :program)))
    (setf (show-old-code? *current-usermodel*)
          (zlc::zlc-frame-show-old-code? program))
    (setf (show-new-code? *current-usermodel*)
          (zlc::zlc-frame-show-new-code? program))
    (setf (show-explanation? *current-usermodel*)
          (zlc::zlc-frame-show-explanation? program)))
  (save-usermodel-to-file *current-usermodel*))
```

## APPENDIX E

## QUESTIONNAIRE ON LISP

**Name:**

1. Which programming languages are you familiar with?
2. In which of the above language are you the most proficient?
3. Please provide the following information about previous experience with LISP:

    a. Number of LISP programs you have written:

    b. Approximate lines of LISP code written (circle answer):

    - None
    - 10 - 100
    - 100-1000
    - 1000-10,000
    - over 10,000

    c. Previous formal LISP instruction (circle those that apply):

    - Took a short course on LISP.
    - Introduced to LISP in a Programming Language Course
    - Have had LISP as part of an AI course
    - Received individual tutoring on LISP
    - Other:

    d. Any Self Directed instruction on LISP:

    - Computer-based instruction
    - Books used on my own to study LISP (which ones)

4. How much do you know about the following LISP concepts?

1=could define
2= am familiar with
3= never heard of it

| | |
|---|---|
| Symbolic Expression | Quote |
| Functions | Conditionals |
| Variables | Side effects |
| Scoping | Property Lists |
| Lists | Mapping |
| Recursions | Multi-value return |
| Cons cell | List iteration |
| Evaluation | True (non-nil) |
| Nil | Atom |

5. How much do you know about the following LISP functions?

1= could probably write a correct expression using the function,
2= am aware this function exists but need help with its syntax
3= have heard of the function but not sure what it does
4= never heard of the function before

| | |
|---|---|
| COND | LAMBDA |
| DO | APPLY |
| DEFUN | FORMAT |
| MAPCAR | EQUAL |
| LET | NULL |
| EVAL | NCONC |
| MEMBER | SETF |
| SYMBOLP | LET* |
| CAR | CASE |
| NTH | AND |
| CONS | ELT |
| LIST | INTERN |
| NCONC | ASSOC |
| APPEND | PROG |
| AND | STRING |
| PRINT | READ |

## INDEX