# A CRITIC FOR LISP

Gerhard Fischer

Department of Computer Science and Institute of Cognitive Science
University of Colorado, Campus Box 430
Boulder, CO 80309

## Abstract

Our goal is to establish the conceptual foundations for using the computational power that is or will be available on computer systems. Much of the available computing power is wasted, however, if users have difficulty understanding and using the full potential of these systems. Too much attention in the past has been given to the technology of computer systems and not enough to the effects of that technology, which has produced inadequate solutions to real-world problems, imposed unnecessary constraints on users, and failed to respond to changing needs.

We have designed and implemented a critic for LISP as a prototype of an intelligent support system. Critics enhance incremental learning of a system and support learning strategies such as learning on demand. Our LISP-CRITIC has knowledge about how to improve LISP programs locally, following a style as defined by its rules. The advice given is based on the hypothesized knowledge of the user contained in the system's model of the user. Additional tools (e.g., a knowledge browser and visualization support) are available to explain and illustrate the advice.

The LISP-CRITIC has been used by intermediate and expert LISP programmers and has proven to be a valuable tool for incremental learning of LISP and for improving programs.

## 1. Introduction

Our goal is to establish the conceptual foundations for using the computational power that is or will be available on computer systems. We believe that Artificial Intelligence methodologies and technologies provide the unique opportunity to achieve the goal of improving productivity by addressing, rather than ignoring, human needs and potential. We are building systems which *augment human intelligence.* Winograd and Flores [Winograd, Flores 86] argue that the development of *tools for conversation* where the computer serves as a structured dynamic medium for conversation in systematic domains, is a more realistic and more relevant goal to successfully exploit information and communication technologies than the most widely perceived goal of AI *"to understand and to build autonomous, intelligent, thinking machines"*[Stefik 86]. In addition to the fact, that the track record of the latter approach is *not too good* (e.g., high quality fully automatic translation, automatic programming), we believe, that *partial autonomous systems* pose greater design challenges than fully automated do (evidence for this view comes from many sources, e.g., in the development of cockpits for future aircrafts the pilots assistant is a more challenging goal than the electronic copilot [Chambers, Nagel 85]).

In this paper we argue that to successfully learn and use complex computer system, incremental learning, learning on demand, user and task specific advice has to be supported. Intelligent support systems are one approach to provide these support structures. We describe in detail one of these systems, the LISP-CRITIC, which enhances incremental learning and supports learning on demand.

## 2. Incremental Learning

### 2.1 Facts of Life

The functionality of modern computer systems is constantly expanding (see Figure 2-1). This increase in functionality is of little use if we do not find ways for people to take advantage of it. Online help systems usually do not do much more than present the same information found in the printed documentation, although the better ones provide additional assistance like keyword-based access and bookmarks that leave a trace of an information-seeking session.

The existence of documentation and other support information does not guarantee that people know how to use it, or that they read it or understand it. It is a fact of life that people quickly settle on plateaus of suboptimal behavior. There are two reasons for this. One is that people do not want to learn. They use computers because they want to get something accomplished. The positive side of this behavior is that people focus on their work, the negative side is that they are not motivated to spend time learning about the system just to find out what they can do. This phenomena is called the "production paradox" by Carroll and Rosson [Carroll, Rosson 86]. People have a tendency to stick with what they know best. When situations occur that could be more effectively handled by new procedures, they are willing to deal with them in suboptimal ways that they personally consider to be safe. People *intentionally* do things suboptimally; they have a subjective metric for cost-effectiveness.

Another reason for suboptimal behavior is that learning Is often restricted. Even if people want to learn, they may not be able to. For example, if the context for new information is missing, people often do not understand the relevance and applicability of new commands. We verified this effect in small-scale experiments with the editor command "Query-Replace" in EMACS. This command steps through all occurrences of a string of text, and the user has to confirm or deny the replacement for each occurrence. It happens quite frequently that while users are in "Query-Replace mode" they detect an error in the immediate environment that they would like to correct. To do so, either they have to make a note to go back to the

'This paper describes work done in the Department of Computer Science and the Institute of Cognitive Science at the University of Colorada, Boulder. It was supported in part by grant No. N00014-85-K-0842 from the Office of Naval Research, the German Ministry for Research and Technology, and Triumph Adler Corporation, Nuernberg.

incorrect text after they complete the whole replacement cycle, or they have to leave the cycle and set it up again to take care of the rest of the file. An advanced feature provided by EMACS is the "recursive edit", which allows the user to do some modifications and then restart the "Query-Replace" cycle. Our experiments show that people reading through a description of this command do not understand what recursive edit does, what it is good for and in which situations it can be used. Only after using the basic command for some time do people start to appreciate an advanced feature of this kind. It is within the scope of our active help system [Fischer, Lemke, Schwab 85] to give a hint about the usefulness of the recursive edit at a time when the command can be used successfully.

Number of Computational Objects in Systems

EMACS:
- 170 function keys and 462 commands

UNIX:
- more than 700 commands and a large number of embedded systems

LISP-Systems:
- FRANZ-LISP: 685 functions
- WLISP: 2590 LISP functions and 200 ObjTalk classes
- SYMBOLICS LISP MACHINES. 23000 functions and 2600 flavors

Amount of Written Documentation

Symbolics LISP Machines:
- 12 books with 4400 pages
- does not include any application programs

SUN workstations:
- 15 books with 4600 pages
- additional Beginner's Guides: 8 books totaling 800 pages

Figure 2-1: Quantitative Analysis of Some Systems

Our preliminary empirical findings indicate that the following problems prevent many users from successfully exploiting the potential of high-functionality systems:

1. Users do not know about the *existence* of tools (and therefore they are not able to ask for them);
2. Users do not know how to *access* tools ;
3. Users do not know *when* to use these tools;
4. Users do not understand the *results* that tools produce for them;
5. Users cannot combine, adapt, and modify a tool to their *specific* needs.

A consequence of these problems is that many systems are underused. We are strongly convinced that what is needed is not quantitatively more information but qualitatively new ways to structure and present information.

## 2.2 Modes of Learning

In our research we want to determine the balance between supporting an exploratory learning style of *learning by doing* which is the basic philosophy behind the interest worlds in LOGO environments [Papert 80], and a guided learning experience through *coaching assistance,* which is the primary strategy supported by systems in intelligent computer-assisted instruction [Sleeman, Brown 82]. There are different modes of learning that can complement each other depending on whether the user's goal is the completion of an action or the acquisition of new knowledge, and depending on whether users are inexperienced with a system or familiar with it and able to help themselves.

Ungulded, active exploration. The advantage of this mode of learning is that users can fully control what they would like to do and how they would like to do it. It is important that an environment of this kind supports safe experimentation (e.g., UNDO mechanisms are crucial) and that it is intuitively approachable. Examples of systems that support unguided learning are: Looo-based learning environments [Papert 80], spreadsheets and construction kits [Fischer, Lemke 87].

Tutoring. Tutoring is an adequate mode of learning for getting started learning a new system. One can predesign a sequence of microworlds (see section 2.3) and lead a user through them [Anderson et al. 84; Anderson, Reiser 85]. But tutoring is of little help in supporting learning on demand when intermediate users are involved in their "own doing". Tutoring is not task-driven because the total set of tasks cannot be anticipated. Instead, the system controls the dialogue, and the user has little control over what to do next.

Asking for Help. In passive help systems, users actively have to seek for help. In complex systems, even experienced users know only a minority of the large set of commands available [Fischer 87]; their expertise lies not in having learned enough to solve any problem immediately, but in having become skilled in gathering information and supplementing what they know by active use of external sources of information. But finding information in these systems is Tar from easy. Help systems have become large systems in their own right, and finding needed information is a problem because there is usually a huge gap between the initial mental form of the query and the corresponding expression required by the system. Documentation and help are structured at the level of the system modules (commands) and not at the task level; to accomplish a specific task may require reading through a substantial amount of information

Answers first, then Questions. To ask a question, one must know how to ask it, and one cannot ask questions about knowledge whose existence is unknown. Owen [Owen 86] has implemented a program called DYK ("Did You know"), which volunteers information and supports the acquisition of information by chance. It supports an unstructured learning process, but there is a fair chance that users occasionally pick up some relevant piece of knowledge.

Learning on Demand. *Active help systems* and *critics* support *learning on demand* Users are often unwilling to learn more about a system or a tool than is necessary for the immediate solution of their current problem. To be able to successfully cope with new problems as they arise, users require a consultant that generates advice tailored to their specific need. This approach provides information only when It becomes relevant. It eliminates the burden of learning many things in neutral settings when the user does not know whether the information will ever be used and when it is difficult for the learner to imagine an application (see the "Query-Replace" example in section 2.1). Active help systems and critics overcome the problem of asking a question. They allow users to do whatever they want and interrupt only when users' plans, ways of achieving something or products are considered significantly inferior to what the program would have recommended. They offer new information only if it is needed. A potential drawback might be that they offer help only in related areas; this can be overcome by accessing knowledge structures that allow the system to present information in related areas in a goal-directed "DYK" fashion (see Figure 4-5).

Human Assistance. Human assistance, if available on a personal level, is in almost all cases the most useful source of advice and help. The mode of learning can best be characterized as a *cooperative problem-solving process.* Learners and advice seeking persons can ask a question in an infinite variety of ways, they can articulate their problem in the "situation model" rather than being required to express their needs in a "system model" [Dijk, Kintsch 83]. Many systems to support information and advice seeking (with some notable exceptions like RABBIT [Tou et al. 82] and ARGON [Patel-Schneider, Brachman, Levesque 84]) have assumed, that persons know what they are looking for. Studying human advisory dialogues [Webber, Finin 84] has shown, that this assumption does not hold: the most valuable assistance is often in formulating the question.

## 2.3 Increasingly Complex Microworlds: An Architecture to Support Incremental Learning

Over the last several years we have developed a general paradigm for instruction that is best described as a sequence of *"Increasingly Complex Microworlds (ICM)"* [Fischer 81; Burton, Brown, Fischer 83].

The ICM paradigm was developed to capture instructional processes for complex skills that are difficult to learn because the starting state and goal state are too far apart. The student is exposed to a sequence of increasingly complex microworlds, which provide stepping stones and intermediate levels of expertise so that within each level the student can see a challenging but attainable goal. Increasingly complex microworlds can also be used to provide protective shields for novices and prevent them from being dumped into unfamiliar areas of the system. The paradigm requires a precise representation of the knowledge that is learned in a specific microworld and of the method for choosing the next microworld. As a model, it captures the essence of the incremental learning processes.

The LISP CRITIC provides a rich environment for pursuing interesting questions within the ICM paradigm:

1. What is the right grain size for microworlds?
2. How do we generate microworlds? By constructing tools to eliminate the necessity of learning subskills? By providing defaults? By constraining the design space thereby decreasing the objective computability of the system but increasing the subjective computability [Fischer, Lemke 87])?
3. What is the right topology of a sequence of microworlds? For people involved in their own work do we need multiple start states? For people having different goals, do we need multiple goal states?
4. What initiates the transition of one microworld to another one? Does the user initiate it or the system? Why should a transition take place?
5. How do we identify a user with a microworld? By taking into account which rules of the LISP-CRITIC (see section 4.3) fire and how often the user code can be matched against the right-hand side of rules (indicating the actual and suggested use of certain concepts)? By doing a statistical analysis of user's programs? By self-evaluation of the user? By instantiating the user's knowledge structures of LISP and comparing them with stored representations of expert knowledge [Fischer 87]?

By creating and studying the LISP-CRITIC that is codifying an idea as a system, we were able to raise issues such as these, which are difficult to articulate concretely in theoretical terms.

## 2.4 Goals of Incremental Learning

The major goals we can pursue by supporting incremental learning are:

- the elimination of suboptimal behavior, thereby increasing efficiency;
- the enlargement of possibilities, thereby increasing functionality;
- the support of learning on demand by presenting new information when it is relevant;
- the structuring of complex systems so that they have *no threshold and no ceiling.* It should be easy to get started; that is, microworlds should provide *entry points:* but these systems should also offer a rich functionality for experienced users;
- the use of *models of the user* to make systems more responsive to the needs of *individual users* and the tailoring of *explanations* to the user's conceptualization of the task.

## 3. Intelligent Support Systems

In our research work we have used the computational power of modern computer systems to construct a variety of *intelligent support systems* (see Figure 3-1). These support systems are called *intelligent,* because they have knowledge about the task, knowledge about the user and they support communication capabilities which allow the user to interact with them in a more "natural" way. They are used to enhance incremental learning processes, to provide help and documentation, to support the understanding of existing programs and advice given, to assist in

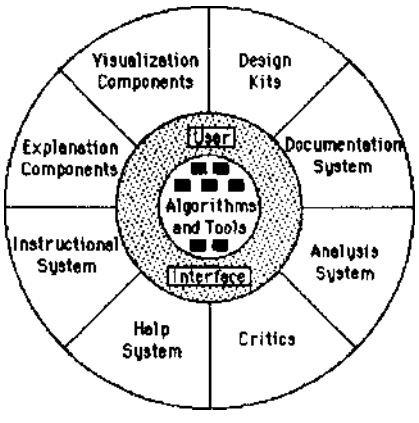


Figure 3-1: An Architecture for Intelligent Support Systems

the construction of new systems. They are described in the following documents:

La documentation system that assists in the incremental development and understanding of a program (Fischer, Schneider 84];

2. a passive and an active help system for an EMACS-like editor [Fischer, Lemke, Schwab 85];

3. components of a "software oscilloscope" that serve as visualization tools [Boecker, Fischer, Nieper 86];

4. design kits which support and guide the designer in the construction of complex artifacts [Fischer, Lemke 87];

5. a critic lor LISP that is described in this paper.

## 4. Description of the LISP-CRITIC

### 4.1 A Critiquing Model

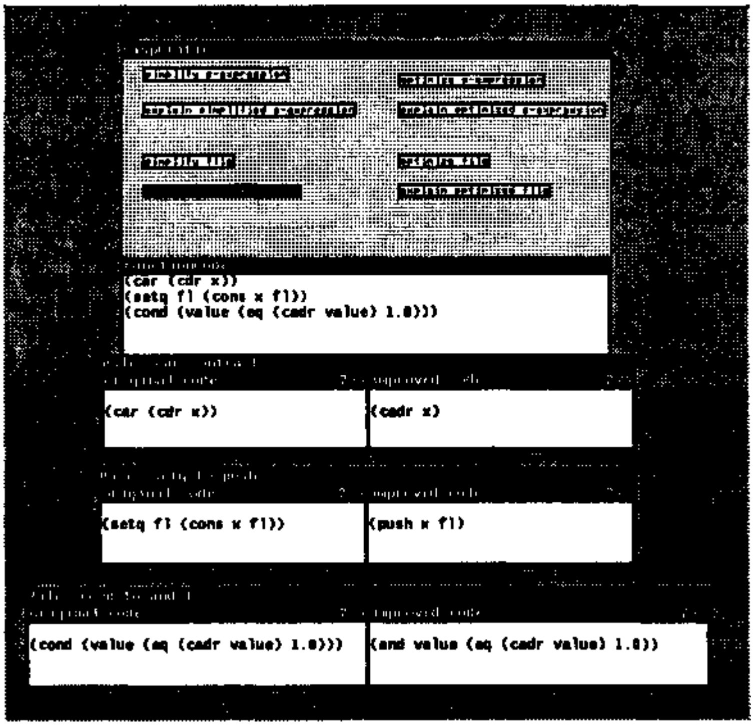
One model frequently used in human-computer systems (e.g., MYCIN [Buchanan, Shortliffe 84]) is the consultation model. From an engineering point of view, it has the advantage of being clear and simple: the program controls the dialogue (much as a human consultant or a tutoring system [Anderson et al. 84; Anderson, Reiser 85] does) by asking for specific items of data about the problem at hand. The disadvantages are that it does not support users in their own doing, it prevents the user from volunteering relevant information and it sets up the program as an "expert", leaving the user in the undesirable position of asking a machine for help. We are in the process of developing a critiquing model which allows users to pursue their own goals and the program interrupts only if the behavior of the user is judged to be significantly inferior to what the program would have done.

The critiquing model will be used to support cooperative problem solving. When a novice and an expert communicate much more goes on than just the request for factual information. Novices may not be able to articulate their questions without the help of the expert, the advice given by the expert may not be understood and/or the advisee requests an explanation for it; persons may hypothesize that their communication partners misunderstood them or the experts may give advice which they were not explicitly asked for (the last aspect we have also explored in our work on active help systems [Fischer, Lemke, Schwab 85]).

### 4.2 The Functionality of the LISP-CRITIC

The USP-CRITIC suggests how to improve LISP code. Improvements can make the code either more cognitively efficient (e.g., more readable and concise) or more machine efficient (e.g., smaller and faster). Users can choose the kind of suggestions they are interested in.

The system is used by two user groups, who have different purposes. One group consists of intermediate users who want to



```
(car (cdr x))
(setq fl (cons x fl))
(cond (value (eq (cadr value) 1.0)))
```

```
(car (cdr x))                            (cadr x)
```

```
(setq fl (cons x fl))                    (push x fl)
```

```
(cond (value (eq (cadr value) 1.0)))     (and value (eq (cadr value) 1.0))
```

Figure 4-1: The LISP-CRITIC In Operation

The "LispCritic" pane provides the basic interface through which the user can initiate an action by clicking a button. The "FunctionCode" pane displays the text of the program that the LISP-CRITIC works on. The other three windows show some of the transformations carried out on the program. The M?" in the title line of the windows is the button for starting the explanation system, which allows the user to browse through additional knowledge structures.

*learn* how to produce better LISP code. We have tested the usefulness of the LISP-CRITIC for this purpose by gathering statistical data on the programs written by students in an introductory LISP course. The other group consists of experienced users who want to have their code "straightened out". Instead of doing that by hand (which in principle, these users can do), they use the LISP-CRITIC to carefully reconsider the code they have written. The system has proven especially useful with code that is under development and is continuously changed and modified.

Figure 4-1 shows *the* system in Operation. The LISP-CRITIC is able to criticize a user's code in the following ways:

- replace compound calls of LISP functions by simple calls to more powerful functions (e.g., (not (evenp a)) may be replaced by (oddp a));
- suggest the use of macros (e.g., (setq a (cons b a)) may be replaced by (push b a));
- find and eliminate 'dead' code (as in (cond (...) (t ...) (dead code)));
- find alternative forms of conditional or arithmetic expressions that are simpler or faster (see Figure 4-2);
- replace garbage-generating expressions by non-copying expressions (e.g., (append (explode word) chars) may be replaced by (nconc (explode word) chars); see Figure 4-4);
- specialize functions (e.g., replace equal by eq; use integer instead of floating point arithmetic wherever possible);
- evaluation or partial evaluation of expressions (e.g., (sum a 3 b 4) may be simplified to (sum a b 7)).

## 4.3 The Architecture of the LISP-CRITIC

The knowledge of the subject domain (concepts, goals, functions, rules and examples) is represented in a network of interrelated nodes. The user can selectively browse through the knowledge. The LISP-CRITIC operates by applying a large set of transformation rules that describe how to improve code. Figure 4-2 shows two of the rules in the system. The user's code is matched against these rules, and the transformations suggested by the rules are given to the user. The modified code is written to a new file, and the user can inspect the modifications and accept or deny them. On demand, the system explains and justifies its suggestions.

The structure of the overall system is given in Figure 4-3. The user's code is simplified and analyzed according to the transformation rules and two protocol files, "people.PR" and "machine.PR", are produced. They contain information (see Figure 4-1) that is used together with conceptual knowledge structures about LISP to generate explanations (see Figure 4-5). The user model (for a more detailed discussion see [Fischer 87]) obtains information from the rules that have fired, from the statistical analyzer and from the knowledge structures that have been visited. In return, it determines which rules should fire and what kind of explanations should be generated. The *statistical analyzer provides* important Information to the user model, for example, which subset of built-in functions the user is using, whether the user is using macros, functional arguments, nonstandard flow of control, etc..

Transform a "COND" Into an "AND"

```
(rule cond-to-and-1                                  the name of the rule
      (cond (?condition Taction))                    the original code
      (and ?condition ?action)                       the replacement
      safe (machine people))                         rule category
```

Example (see Figures ABBCRITIC and EXPL) :
```
   (cond (value (eq (cadr value) 1.0))) —-> (and value (eq (cadr value) 1.0))
```

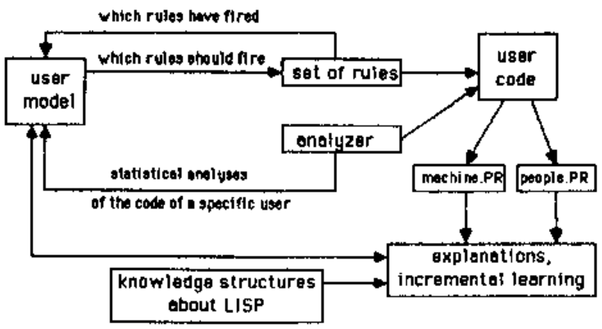
Replace a Copying Function with a Destructive Function

```
(rule append/.1-new.cons.cells-to-nconc/.1...        the name of the rule
   (?foo:(append appendl}
      (restrict ?expr                                the condition
                                                     (rule can only be applied
         (cons-cell-generating-expr expr))           if cons cells
                                                     are generated by "?expr")
      ?b)

   ((compute-it:
      (cdr (assq (get-binding foo)
                 '((append . nconc)                  the replacement
                   (appendl . nconcl)))))
    ?expr ?b)
   safe (machine))                                   rule category
```

Example (see Figure KAESTLE) :
```
   (append (explode word) char)——> (nconc (explode word) char)
```

Figure 4-2: Examples of Rules in the LISP-CRITIC



which rules have fired
which rules should fire
user model
set of rules
user code
analyzer
statistical analyses of the code of a specific user
machine.PR
people.PR
explanations, incremental learning
knowledge structures about LISP

Figure 4-3: The Architecture of the LISP-CRITIC

## 4.4 Support for Understanding the Criticism

Our experience with the LISP-CRITIC in our LISP courses has shown that the criticism it gives is often not understood. Therefore we use additional system components to illustrate and explain the LISP-CRITIC'S advice, KAESTLE, a visualization tool that is part of our software oscilloscope [Boecker, Fischer, Nieper 86], allows us to illustrate the functioning and validity of certain rules. In Figure 4-4 we use KAESTLE to show why the transformation (append (explode word) chars)———> (nconc (explode word) chars) is a safe one (because explode is a cons-generating function; see the rule in Figure 4-2), whereas the transformation (append chars (explode word))———> (nconc chars (explode word)) is an unsafe one (because the destructive change of the value of the first argument by nconc may cause undesirable side-effects).

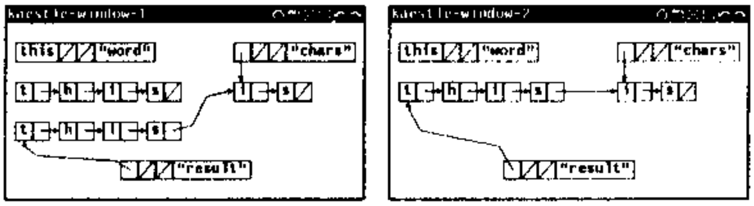
In addition to the visualization support, we have developed an explanation component that operates as a user-directed browser in the semantic net of LISP knowledge. This component contains: textual explanations that justify rules, related functions, concepts, goals, rules and examples (see Figure 4-5). Currently textual explanations are extracts from a LISP textbook [Wilensky 84]). The information structures in the explanation component should help the student to understand the rationale for the advice given by the LISP-CRITIC, and they should also serve as a starting point for a goal-directed "Did you know (DYK)" mode of learning (see section 2.2).

## 4.5 Tutorial Strategies

With these features we can pursue different tutorial strategies in the framework provided by the LISP-CRITIC. The information accumulated in the system's model of the user is used to decide when a user should be criticized, what advice should be given and how the advice should be given (e.g., as a textual explanation from the manual, a KAESTLE visualization, or as a convincing example). These issues are not independent, and have to be perceived from the user's state of knowledge, not the designer's. We must guess and determine the knowledge state of the user in order to make critics such as the LISP CRITIC respond at the user's level of understanding.

Crucial issues in designing the LISP CRITIC were the distribution of initiative between the user and the system and the amount of control over the system. After the LISP-CRITIC has provided an initial starting point for a learning process, we feel that the user can and should be able to proceed in a self-directed mode. A specific user may personally dislike some of the rules and should be able to turn them off. If the system notices that a user never accepts the changes suggested by a rule, it could be turned off automatically [Fischer, Lemke, Schwab 85].

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



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

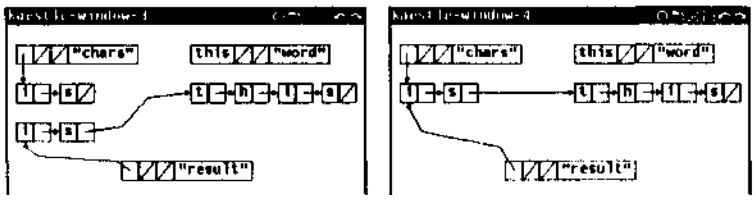
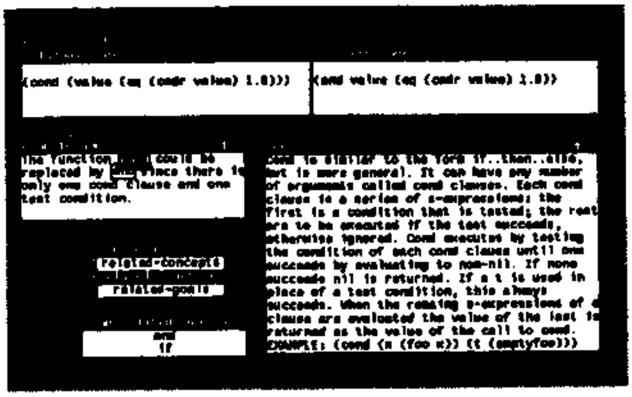


Figure 4-4: Illustration of the Validity of a Rule Using KAESTLE

In the environment shown in the individual screen images, the variable word is bound to the value this and the variable chars is bound to the list (i s).


```
(cond (value (eq (cadr value) 1.0)))
```
```
(and value (eq (cadr value) 1.0))
```
The function [illegible] could be replaced by and since there is only one cond clause and one test condition.

cond is similar to the form if..then..else, but is more general. It can have any number of arguments called cond clauses. Each cond clause is a series of s-expressions; the first is a condition that is tested; the rest are to be executed if the test succeeds, otherwise ignored. Cond executes by testing the condition of each cond clause until one succeeds by evaluating to non-nil. If none succeeds nil is returned. If a t is used in place of a test condition, this always succeeds. When the remaining s-expressions of a clause are evaluated the value of the last is returned as the value of the call to cond. EXAMPLE: (cond (x (foo x)) (t (emptyfoo)))

Figure 4-5: The User Browses through the Knowledge Base

## 5. Experiences with and Evaluation of the LISP-CRITIC

The LISP-CRITIC has been in operation for two years and has been useful for many groups of LISP users. Our informal evaluations indicate that the LISP-CRITIC had an impact on the process of learning LISP. One study showed that as students gained experience in LISP programming, the number of rules fired decreased over time. Students learn new functions, new concepts and ways of structuring their knowledge about LISP. Instructors can use the output of the LISP-CRITIC as a basis for personal advice to their students. Experts also used the LISP-CRITIC to have their code "straightened our. The rules in the knowledge base of the LISP-CRITIC generated an ongoing discussion about the merits of different styles of USP programming. Surprises from our empirical work were, that our statistical analysis showed that even experts only use a relatively small fraction of the total number of primitive LISP functions, and that the LISP-CRITIC suggested many improvements for the implementation of generally used systems (e.g., for the LISP implementation of OPS5 and for the system code of FRANZ-LISP).

To discover suboptimal behavior requires a metric. For our ACTIVIST system [Fischer, Lemke, Schwab 85], we chose a very simple metric: the number of keystrokes needed to perform a specific task. In the LISP-CRITIC, a set of over two hundred rules defines a metric (currently for the FRANZ-LISP dialect). These rules state how a LISP program should be written. Like the UNIX Writers Workbench tools [Cherry 81], they define a style or standard that the authors believe leads to greater clarity and understandably in a program or piece of code. It goes without saying that we do not expect universal agreement on issues of style (we have applied the UNIX writers workbench tools (e.g., DICTION) to the Gettysburg address and the system suggested some modifications).

The LISP-CRITIC in its current form is not restricted to a specific class of LISP functions or domain of application. It accepts any LISP code. Its generality is the reason for some obvious shortcomings. The critic operates only on the code; the system does not have any knowledge of specific application areas or algorithms, and it is naturally limited to improvements that derive from its low-level knowledge about LISP. AS we pointed out in section 2.2, human assistance is by comparison much more powerful than the LISP-CRITIC, because determining and addressing user's goals can extend advisory dialogues far beyond the capabilities of the LISP-CRITIC. Systems like the PROUST system [Johnson, Soloway 84] are able to do a much deeper analysis, but they are very restricted in the range of problems to which they can be applied to. We have designed a module to extend the framework of the LISP-CRITIC by providing expert solutions to problems assigned to students in a course. Comparison of these expert solutions with the work of the students provides additional sources of information for the USP-CRITIC.

The LISP-CRITIC'S understanding has to be extended beyond lines of code or individual functions. Knowledge structures derived from programs like MASTERSCOPE [Teitelman, Masinter 81] and a knowledge-base of cliches (representing intermediate and higher-level programming constructs [Waters 85; Waters 86] would be useful for the LISP-CRITIC to operate on.

## 6. Future Research

To get a better understanding of the empirical consequences of the many design choices which one faces in building a system of this sort, more research in studying naturally occurring advisory situations is urgently needed. The USP-CRITIC has been an interesting starting point towards our long-range goal to build intelligent support systems and to support cooperative problem solving processes between humans and computers. The USP-CRITIC has a few features, which extend the system beyond a "one-shot affair": it allows the user to ask for an illustration of the advice given (see Figure 4-4) and the user can use the advice as a starting point to explore related concepts, functions and goals (see Figure 4-5).

The LISP-CRITIC is a first operational example of the class of systems we are interested in. However, we doubt that the general domain of LISP programming is the best area for a critic. An preliminary analysis of more restricted domains like our user interface construction kit WLISP [Fischer, Lemke, Rathke 87] and the formatting system SCRIBE have lead us to believe that critics may be even more useful for less general systems where a system has a better chance to infer the goals of the users.

## Acknowledgements

Many people have contributed to the development of the USP-CRITIC over the last few years. The author would like to thank especially Heinz-Dieter Boecker, who developed many of the original ideas, the set of rules and the interpreter for it; Andreas Lemke, who contributed to the general framework for incremental learning; Helga Nieper, who developed KAESTLE; Christopher Morel, who implemented the explanation capabilities; Bart Burns, who implemented the statistical analysis package; Catherine Cormack, who analyzed user programs with the statistical package; and Janet Grassia who assisted in editing this paper.

## References

[Anderson et al. 84]
J.R, Anderson, C.F. Boyle, R.G. Farrell, B.J. Reiser, *Cognitive Principles in the Design of Computer Tutors,* Proceedings of the Sixth Annual Conference of the Cognitive Science Society, Boulder, CO, June 1984, pp. 2-9.

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

[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.

[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, Fischer 831
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 Rogoff (ed), Everyday Cognition: Its Development in Social Context, Harvard University Press, Cambridge, MA, 1983.

[Carroll, Rosson 86]
J.M. Carroll, M.B. Rosson, Paradox of the Active User, Technical Report RC 11638, IBM, Yorktown Heights, NY, 1986.

[Chambers, Nagel 85]
A.B. Chambers, D.C. Nagel, Pilots of the Future: Human or Computer?, Communications of the ACM, Vol. 28, No. 11, November 1985.

[Cherry 81]
Lorinda Cherry, Computer Aids for Writers, Proceedings of the ACM SIGPLAN SIGOA Symposion on Text Manipulation, Portland, Oregon, 1981, pp. 61-67.

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

[Fischer 81]
G. Fischer, Computational Models of Skill Acquisition Processes, Computers in Education, Proceedings of the 3rd World Conference on Computers and Education, R. Lewis, D. Tagg (eds.), Lausanne, Switzerland, July 1981, pp. 477-481.

[Fischer 87]
G. Fischer, Enhancing Incremental Learning Processes with Knowledge-Based Systems, in H. Mandl, A. Lesgold (eds.), Learning Issues for Intelligent Tutoring Systems, Springer-Verlag, Berlin - Heidelberg - New York, 1987.

[Fischer, Lemke 87]
G. Fischer, A.C. Lemke, Constrained Design Processes: Steps Towards Convivial Computing, in R. Guindon (ed.), Cognitive Science and its Application for Human-Computer Interaction, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[Fischer, Lemke, Rathke 87]
G. Fischer, A.C. Lemke, C. Rathke, From Design to Redesign, Proceedings of the 9th International Conference on Software Engineering, IEEE, March 1987, pp. 369-376.

[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, Schneider 84]
G. Fischer, M. Schneider, Computer-Supported Program Documentation Systems, Proceedings of INTERACT '84, IFIP Conference on Human-Computer Interaction, IFIP, London, September 1984.

[Johnson, Soloway 84)
W.L. Johnson, E. Soloway, PROUST: Knowledge-Based Program Understanding, Proceedings of the 7th International Conference on Software Engineering, Orlando Florida, March 1984, pp. 369-380.

[Owen 86]
D. Owen, Answers First, Then Questions, 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. 17.

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

[Patel-Schneider, Brachman, Levesque 84]
P.F. Patel-Schneider, R.J. Brachman, H.J. Levesque, ARGON: Knowledge Representation Meets Information Retrieval, Fairchild Technical Report 654, Schlumberger Palo Alto Research, September 1984.

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

[Stefik 86]
M.J. Stefik, The Next Knowledge Medium, AI Magazine, Vol. 7, No. 1, Spring 1986, pp. 34-46.

[Teltelman Masinter 811
W. Teltelman, L Masinter, The Interlisp Programming Environment, Computer, April 1981, pp. 25-33.

[Tou et al. 821
F.N.Tou, M.D. Williams, T.W. Malone, R.E. Fikes, A. Henderson, RABBIT: An Intelligent Interface, Technical Report, Xerox Palo Alto Research Center, 1982.

[Waters 85]
R.C. Waters, The Programmer's Apprentice: A Session with KBEmacs, IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1296-1320.

[Waters 861
R.C. Waters, KBEmacs: Where's the AI?, AI Magazine, Vol. 7, No. 1, Spring 1986, pp. 47-56.

[Webber, Finin 841
B.L. Webber, T.W. Finin, In Response: Next Steps in Natural Language Interaction, in W. Reitman (ed), Artificial Intelligence Applications for Business, Ablex Publishing Corporation, Norwood, NJ, 1984, pp. 211-234, ch. 12.

[Wilensky84]
R wilensky, LISPcraft, W.W. Norton & Company, New York-London, 1984.

[Winograd, Flores 861
T. Winograd, F. Flores, Understanding Computers and Cognition: A New Foundation for Design, Ablex Publishing Corporation, Norwood, NJ, 1986.