COMPUTER SOCIETY PRESS REPRINT

MINIMALIST EXPLANATIONS IN KNOWLEDGE-BASED SYSTEMS

Gerhard Fischer
Thomas Mastaglio
Brent Reeves
John Rieman

Reprinted from PROCEEDINGS OF THE TWENTY-THIRD ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEMS SCIENCES
Kailua-Kona, HI, January 2-5, 1990

The Computer Society of the IEEE
1730 Massachusetts Avenue NW
Washington, DC 20036-1903

Washington • Los Alamitos • Brussels

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC

IEEE

COMPUTER SOCIETY PRESS

Minimalist Explanations in Knowledge-Based Systems

Gerhard Fischer, Thomas Mastaglio, Brent Reeves, John Rieman

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

ABSTRACT

Research in discourse comprehension and human-computer interaction indicates that good explanations are usually brief. A system that provides brief explanations, however, must plan for the case where brevity comes at the expense of understanding. Human to human dialog is, to a large part, concerned with conversational repair and question-answer episodes; computer systems need to provide similar fallback techniques to their users. We have designed such an explanation system in the context of a knowledge-based critiquing system, LISP-CRITIC. The system provides several levels of explanations, specifically tailored to the user. If the initial, brief explanation is insufficient, the system positions the user at an appropriate point within a more complete, hypertext-based documentation system. Rather than attempting to design a system that can generate a perfect, one-shot explanation for any given situation, this approach concentrates on matching the communication abilities provided by current computer technology to the cognitive needs of the human user.

INTRODUCTION

The goal of our research is to build cooperative knowledge-based systems. By *cooperative* we mean specifically that a user and a knowledge-based computer system collaborate to create a design artifact or solve a problem. We want to recognize and take advantage of the strength of each participant. The computer system's role in this dyad must include the capability to explain its actions or recommendations.

For a variety of reasons, current explanation systems often fail to satisfy the user. Much of the work directed toward improving explanations has focused on natural language and human-to-human communication paradigms. Even advanced systems are frequently based on an implicit assumption that a sufficiently powerful computer system should be able to generate an acceptable explanation as a "one-shot affair." Our approach is to provide multiple levels of explanation, accessible under the user's control. Rather than emphasizing natural language, we emphasize natural communication, a concept that involves choosing the most appropriate communication technique for the explanation at hand. The technique may be natural language, but it may also involve capabilities unique to computer-human communication, such as real-time computer graphics, hypertext, and runnable examples.

The work described in this paper involves primarily the initial levels of a multiple-level explanation approach, where we provide what we call "minimalist" explanations. These are textual explanations that have been trimmed to a bare minimum, both in terms of "chunks" of knowledge from the underlying domain and length of the presented text. Where possible, new concepts are described in terms of concepts the user already understands. The theoretical foundations for this approach are found in research in discourse comprehension and human-computer interaction. Brief explanations will not always be sufficient, so we address this problem by positioning users in an extensive hypertext documentation system, where they can find more detailed information on topics mentioned in the brief text.

The next section is a description of the theoretical context for this approach. The current design was prefaced by an examination of other research in explanations, which is described next. The operational context for the actual system is LISP-CRITIC, a knowledge-based critiquing system used by LISP programmers for code development. We describe the functional modules, show an example of the system's operation in detail and discuss future plans.

THEORETICAL FOUNDATIONS

This work is founded on investigations of how to best implement the critiquing paradigm [10]. Those investigations led to attempts to provide explanation capabilities within the LISP-CRITIC system, a goal also motivated by empirical studies of the attitudes of users of other expert systems. The identifiable shortcomings of existing explanation systems led to the work described in this paper. This section treats each of these topics, as well as the cognitive theory underlying the current system.

The critiquing paradigm provides a way to make knowledge-based systems more cooperative. Such systems are useful in settings where there is no best solution, where it is easier to evaluate solutions rather than to generate them. Users in these situations should be familiar with the domain and capable of generating at least partial solutions. There are many applications where autonomous expert systems cannot fulfill the needs of the user, sometimes because of their inherent limitations, and sometimes because ethical considerations demand that the human user retain control and responsibility for the ultimate product. Computer-based critics can help in these situations by analyzing the activity of the user, finding inconsistencies, and suggesting improvements. Essentially, the spirit of computer critiquing is to provide the user with context-sensitive advice in a particular domain. The user faces the choice of accepting or rejecting that advice, a decision that may require an explanation of the advice.

The need for good explanations was identified in a study where physicians' attitudes towards expert systems were evaluated:

> *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. [27, p. 681]

On-line documentation, the most common form of computerized explanation, has generally used prewritten paragraphs or sentences, sometimes referred to as "canned text." This material has been criticized as difficult to understand, incomplete, and hard to navigate [29]. The insufficiency of canned text in tutoring, a paradigm closely related to critiquing, was noted in the empirical studies described in [12]. We agree that the use of canned text alone is not an adequate approach to providing acceptable explanations.

At a superficial level, there are several shortcomings of explanations based on canned text. Explanations are too long; users get lost, bored or confused; they cannot afford the time and effort to extract the information they need. This is characteristic of many on-line help systems, such as the Unix "man" command. Even long explanations are seldom entirely complete. Moving among several on-line pages of documentation is both time-consuming and difficult. Furthermore, systems do not have the capability to respond to follow-up questions or engage in a dialog with the user. Most systems have been designed to treat the explanation of each topic as a one-shot affair. Lack of appropriate examples is an additional shortcoming. Where examples are provided, they are often incomplete, and there is seldom a runnable version of the example provided for modification or testing. Finally, the canned text is written from the perspective of its author. It is based on his or her conceptual model of the domain, and may not reflect an in-depth investigation into the domain's conceptual structure or the user's perspective of that domain.

Theoretical results support the common-sense observations noted in the previous paragraph. Short-term memory has been identified as a fundamental limiting factor in reading and understanding text [6; 2]. The best explanations are those that contain no more information than absolutely necessary, since every extra word increases the chances that the reader will lose essential facts from memory before the explanation is completely processed.

Recently, [16] and [7] have noted the importance of relating written text to the reader's existing knowledge. This is related to the idea that an expert's knowledge is stored as a collection of "chunks" [4]. Remembering and understanding a new explanation will be easier if much of it calls to mind existing chunks of knowledge. One way to address these issues is to provide *differential explanations* based on what the user already knows, i.e., to explain new concepts concisely by distinguishing them from known concepts.

Interestingly, similar guidelines have long been championed by teachers of rhetoric. Since the 1930s, various formulae have been proposed for evaluating the readability of written text [17]. These formulae are frequently used to evaluate documentation and instructional text. They generally rank short sentences and known vocabulary as especially important. Strunk and White's manual, familiar to most American college students, contains similar advice. In a brief section dealing specifically with explanatory text they state, "Don't explain too much" [24].

Observations of spoken language are another source of supporting evidence. As an example of this, [23] describes a conversation fragment in which a customer asks a sales person, "What's the code to get out on this phone?" The sales person replies, "Nine -- sometimes the outside lines are all kind of busy." From the customer's response it appears that this brief explanation was sufficient. There was no need for a more complete and detailed explanation, such as: "Pick up the handset. Listen for the dial tone. When you hear the dial tone, press the 'nine' button. ..." Yet similar explanations are often produced by computer systems. By attempting to provide complete information for every user, they fail to provide a good explanation for anyone but the most atypical novice.

RELATED WORK

Natural Language

Several research efforts attempt to provide computer-based explanation facilities that generate unique strings of natural language. Some of these programs rely on a user model for tailored explanations [14] while others generate the same explanation for any user [5; 28]. Another approach, developed by Moore, directly addresses the need for a system that can respond to follow-up questions. Moore's "reactive" model provides the user with a brief initial explanation, but is prepared to expand on the initial text with increasingly informative fall-back explanations [20; 21]. Her work emphasizes natural-language generation and dialog analysis techniques, coupled with a knowledge base that provides a detailed description of the topic being explained. This is a powerful technique which could supplement our work.

One shortcoming of the natural-language based systems is their focus on human-to-human communication as the primary model for human-computer communication. This fails to take advantage of the unique capabilities of computers, such as real-time animated graphics, runnable example code, and text-search tools. It also fails to address the problem that traditional screen-and-keyboard computer interfaces fall short of the communication bandwidth available between humans [15]. Longer natural language dialogs are further limited by the poor readability of long texts on computer displays [13]. To enable natural communications between computers and users, the designers of knowledge-based systems must respond to these special capabilities and limitations. [9].

Explaining and understanding

It seems that the ability to explain a given concept depends directly on one's ability to understand it. In order to build systems that do understand what they are are asked to explain, Swartout proposed that we provide a "domain model," a representation of deep knowledge, and "domain principles," a representation of problem-solving domain control strategies. Given these two components, a system could then be automatically generated [26].

Chandrasekaran and colleagues propose that an explanation system needs to be able to explain three different kinds of things [3]:

- Why specific desisions were made,
- Elements in the knowledge base, and
- The problem solving strategy currently in use.

Our rationale for using LISP-CRITIC was to make use of existing work rather than build a system from scratch. The rule-base for the critiquing component already existed; we were interested in adding components necessary to provide good explanations about the source code transformations. The first item mentioned above is the main focus of this work. We use hypertext to address the second item. The problem solving strategy is pattern-matching. Users know that the system does not address domain-specific issues, e.g. how to write a recursive-descent parser.

Tailoring

McKeown and colleagues motivate the need for tailoring by showing that different users have different views of the same problem [19]. The explanation component needs to be aware of the fact that there are different reasons for asking a given question.

We suggest that not only should explanations be tailored to the individual, they should also be constructed so that they explain new concepts in terms of already known concepts. We call this *differential explanations*. Notice that having several explanations for the same concept is not the same as having an explanation that presents new concepts in terms of old ones.

LISP-CRITIC

LISP-CRITIC is a system that has knowledge about a specific domain (LISP programming) and a critiquing component that analyzes LISP source code and suggests improvements [10]. The intended user is a programmer who writes LISP source code and would like to improve that code, either for ease of interpretation by other programmers (readability), or for efficiency of execution and memory use. LISP-CRITIC operates as part of the Symbolics 3600 Workstation environment. It is directly accessible in the user's editing environment, an EMACS-like editor called Zmacs. The user asks LISP-CRITIC to critique a block of code by pressing a control-key combination. As part of the critique, the system might recommend source code changes that are not understood by the user. The system should be able to explain these recommendations.

Previous explanation strategies used by LISP-CRITIC involved rule-tracing and prestored text descriptions for individual transformations. Alternative canned text explanations of a rule were provided, each intended to meet the needs of a particular user expertise level. During the process of providing these explanations, the system took into account the user's level of LISP knowledge as recorded in the user model. The user's LISP knowledge was expressed using a simple classification approach, where a particular user fit a novice, intermediate, or expert programmer stereotype. A user's classification was deduced from rules that had fired in the past. We found this approach to be of limited success. Appropriate explanations were difficult to construct, and users did not find the explanations particularly helpful. It became clear that a better approach was needed, one that was somehow tailored to an individual user rather than to a level of expertise.

Our current approach to explanations is designed to meet the requirements identified in the Teach and Shortliffe study described above [27]. The issue we address is how to provide an explanation module for LISP-CRITIC that adapts its explanations to the needs and characteristics of the user. The module generates explanations that aid the user in deciding whether to accept LISP-CRITIC's suggestions. To improve the quality of the explanations, the module determines the user's current knowledge of LISP by checking the user model.

The user model is stored in a database of information the system acquires and maintains of individual programmers. That database contains information acquired about the user from statistical analysis of his or her code, individual preference for particular programming constructs, a record of which LISP-CRITIC rules have fired for this user in the past, and a history of previous explanation dialogs. This information is used to acquire and perform consistency checks on a user model that represents what the user knows about the domain of LISP. The representation is based on a conceptual structure that we have identified for the LISP language. As an example, a given user model could record that the user knows the LISP concepts *conditional, expression evaluation, functions, symbols, s-expressions* and the *cond* function, but knows neither the *if* function nor the concept *predicates*. For a more detailed description of the user modelling component see [18].

OVERVIEW OF LISP-CRITIC EXPLANATION SYSTEM

Figure 1 shows the user's decision-making process during interaction with LISP-CRITIC. The user begins this process by asking LISP-CRITIC to critique a section of code. LISP-CRITIC will suggest transformations to improve the code. For each of these transformations, the system can provide explanations at several layers:

- the name of the rule and a preview of the suggested change;
- a brief description of the operation of the transformed code, focusing on how it performs the same function as the original;

- a brief explanation of why the transformed code is an improvement over the original;
- several hypertext entry points into the Symbolics on-line documentation, which is itself in hypertext form.

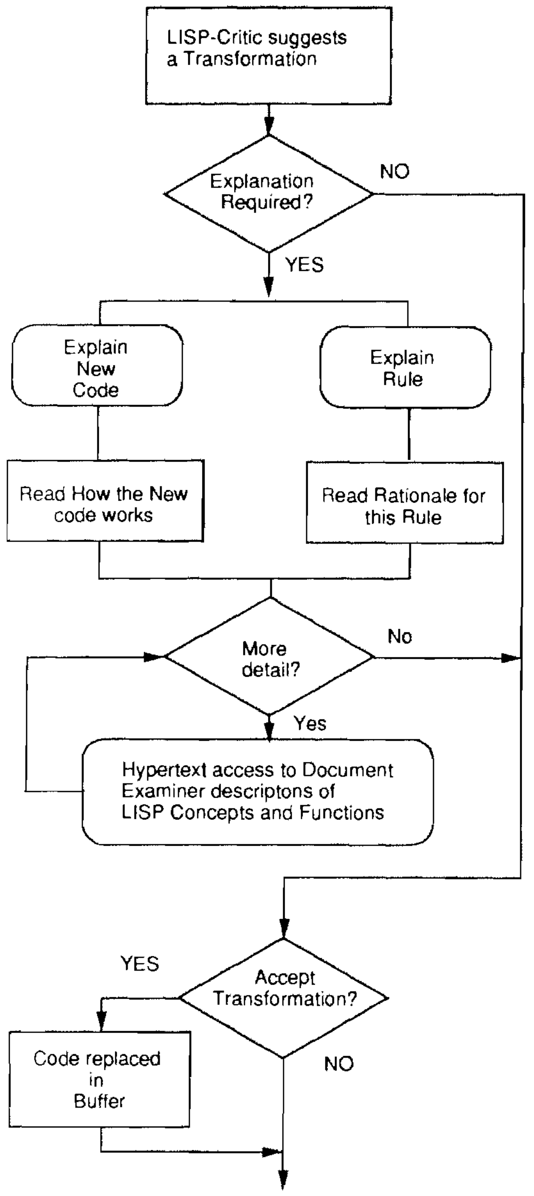

LISP-Critic suggests a Transformation
Explanation Required?
NO
YES
Explain New Code
Explain Rule
Read How the New code works
Read Rationale for this Rule
More detail?
No
Yes
Hypertext access to Document Examiner descriptons of LISP Concepts and Functions
YES
Accept Transformation?
NO
Code replaced in Buffer

Figure 1: Decision Making/Interaction Process in LISP-CRITIC

In LISP-CRITIC's explanation system, the system does not present the minimal explanations as though they were part of a dialog generated by an interactive intelligence agent. Instead, explanations are combinations of straightforward, concise, prewritten sentences, presented in a window. What distinguishes this approach from most standard canned-text systems is the use of a user model. The model is checked to make sure that an explanation does not cover topics that the user already understands, and to help select an explanation that relates new topics to the user's existing knowledge.

Since our system tries not to "explain too much," some explanations will fail to satisfy a given user. To manage this problem, we take advantage of a unique capability of computers: hypertext. Mouse-selectable words within each explanation give immediate access to more detailed explanations of important topics. These detailed explanations are contained in the Symbolics Document Examiner system, a hypertext system that contains the entire Symbolics system documentation as well as complete descriptions of most COMMON LISP functions and underlying language concepts. The transition between the initial minimalist explanation and the entire Document Examiner system is transparent to the user, who sees only a hypertext system that starts with a brief explanation of the current LISP-CRITIC transformation and continues along whatever path the user chooses.

Several components of LISP-CRITIC are important to the process of providing good explanations: the user model, the module that builds explanations appropriate to a user's individual needs, the fall-back access to additional explanatory information, and the user interface. We have touched on the user model; in the following paragraphs we will discuss the other three components.

Explanation Module

The explanation module constructs custom explanation sequences from short explanations of low-level concepts and functions. Low-level means they are fundamental concepts of the programming language LISP. The module's knowledge base consists of the following:

- for each LISP-CRITIC rule, a list of simple concepts and functions that the user must know to understand the rule;
- several possible explanations for each LISP concept and function;
- for each explanation, an associated list of background knowledge indicating which concepts and functions a user must know in order to understand the explanation.

The explanation module receives a rule name -- *cond-to-if*, for example. The module checks its knowledge base to determine what knowledge is required to understand that rule. For *cond-to-if*, the user must understand language concepts *Predicates* and *Functions* as well as LISP functions *Cond* and *If*. The module queries the user mode to see which of these the user already understands. As an example, if the user understands *Functions* and *Cond*, then the module will generate an explanation sequence describing the remaining items, *Predicates* and *If*.

For most concepts and functions, several explanations are available. The *If* function could be explained *differentially* in terms of a similar construct in another programming language, such as Pascal:

IF is like Pascal's IF, but it doesn't require THEN or ELSE keywords.

It could also be explained in terms of the LISP function *Cond*:

> IF is like COND but it only has one <predicate> <action> pair and an optional <else-action>. Also, it uses fewer parentheses.

Another way to explain this is *descriptively* ("from scratch") to the user who has inadequate background knowledge. The explanation module queries the user model about the user's background knowledge and selects the explanation that should be most easily understood by that user. The system can also function adequately with no user model at all. In the case of a new user, the explanations will be based on a default model. In this case the fall-back capability of the hypertext system becomes even more important, since we are more likely to miss the mark with a new user than one who has used the system extensively in the past.

Fall-Back Explanations

In order to provide information that the brief explanations omit, important words are mouse sensitive. Selecting them provides access to the full Symbolics documentation on the topic. That documentation itself is in hypertext form, with mouse-selectable items throughout. (The Symbolics documentation is generally quite lengthy, making it unsuitable as our initial minimalist explanation.) Notice that this fall-back is *not* like handing someone a thick manual and saying, "Here, look it up!" It is more like turning to the appropriate page and saying, "Start here and browse around if you'd like."

User Interface

The goal of the user interface to our explanation component was to allow the user complete control over code transformations and explanations, while maintaining simplicity. The LISP-CRITIC interface provides a single window and a single menu, described in the next section. Any menu item can be selected whenever the LISP-CRITIC window is showing, and previously selected options can be reselected, or the window can be scrolled back to display earlier interactions. This allows the user to dynamically react to the situation, rather than forcing adherence to a predetermined plan (see [25] for discussion of this approach). The user is in charge at all times and can exit at any point in the interaction. Except for the mouse-selectable items, there are no hidden options, modes, or submenus. So the user can operate the interface using recognition memory alone.

SCENARIO

We describe here the program in interaction with a user. The basic unit of critique is a LISP function definition.

Asking for criticism. When the user wants a critique from LISP-CRITIC, he or she places the text-cursor anywhere in the context of a function definition and presses the key combination Super-C. In this scenario, the user asks for a critique of the function *combine*.

Initial display. When Super-C is pressed, LISP-CRITIC examines code in the Zmacs buffer and looks for possible improvements. Figure 2 shows the screen when the LISP-CRITIC window first appears. LISP-CRITIC has found a *COND* statement that it suggests should be changed to an *IF* for readability. The name of the transformation rule, the user's original code, and the suggested transformation are displayed in the window. This is the first layer of explanation in the system, a simple abstract reference to the chunk of domain knowledge applied by the critic. A user who already has a general understanding of the functions used in the transformed code might not be interested in any further explanation.

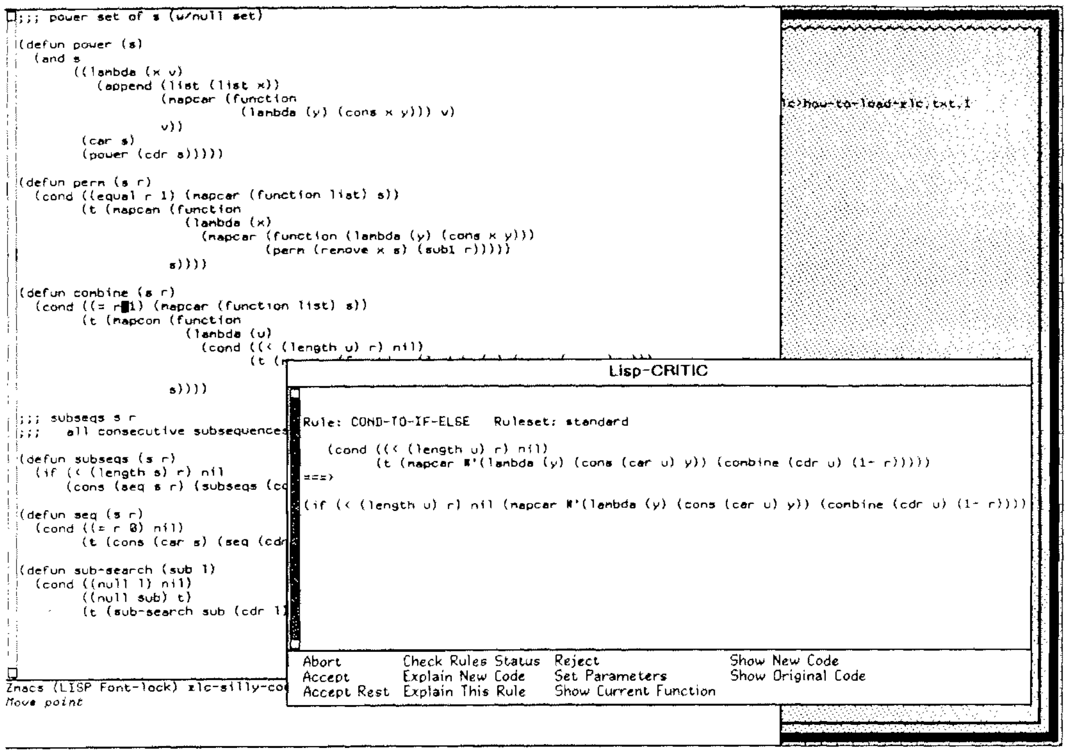
Minimal explanations. If the user doesn't understand the transformation, he or she can select the *Explain New Code* menu item to get a brief explanation of the functions and concepts on which the transformation is based. This is shown in Figure 3. The explanation module has determined that this user needs explanations for *If* and *Predicates*. A differential description of *If* and a descriptive explanation for *Predicates* are chosen and displayed in the window. This text explains how the transformed code performs the same function as the original.

Justification. So far, the explanations described how the transformed code operates. To understand why the new code is an improvement, the user can choose the menu item *Explain this Rule.* The resulting display is shown in Figure 4.

Extended explanations. The user who wants more information on a topic addressed in either minimal explanation can choose one of the mouse-selectable words to access the Document Examiner text covering that item. The idea is not only to provide the user easy access to an extensive on-line resource, but to position him or her within that information space at a logical point. Another approach that uses the PHI methodology to explain critiquing advice in terms of design issues and answers for a kitchen design critic, JANUS, is presented in [11]. In Figure 5 the user has selected the word *Predicates* in the text of a minimal explanation, and the system has brought up the Document Examiner entry on predicates.

Accept or Reject. At several junctures in this process the user might decide to accept or reject LISP-CRITIC'S recommendation. To change the original code to the improved form, the user can click the *Accept* menu item. To retain the original code, the *Reject* item is selected. Again, either option can be selected at any time during the interaction between the user and LISP-CRITIC; it is not necessary for the user to first view any of the explanations. In fact, we expect that the most common mode of operation will be for programmers to quickly go through their code with LISP-CRITIC, deciding whether to accept or reject recommended transformations after seeing only the rule name and the transformed code. This is similar to the way a spelling checker is used -- only rarely does an author go beyond the usual spelling checker's display to confirm that the corrected word has the intended meaning.

Replacement of Original code. Once a decision is made to accept or reject a transformation, LISP-CRITIC continues to process the remainder of the code (since a single block of code may trigger several LISP-CRITIC suggestions.) If additional transformations are possible, they are displayed one at a time to the user in the LISP-CRITIC window and the process described above is repeated. When all possible transformations have been accepted or rejected, the resulting code



```
;;; 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 combine (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (

                  s))))

;;; subseqs s r
;;;   all consecutive subsequences

(defun subseqs (s r)
  (if (< (length s) r) nil
      (cons (seq s r) (subseqs (cc

(defun seq (s r)
  (cond ((= r 0) nil)
        (t (cons (car s) (seq (cdr

(defun sub-search (sub l)
  (cond ((null l) nil)
        ((null sub) t)
        (t (sub-search sub (cdr l
```

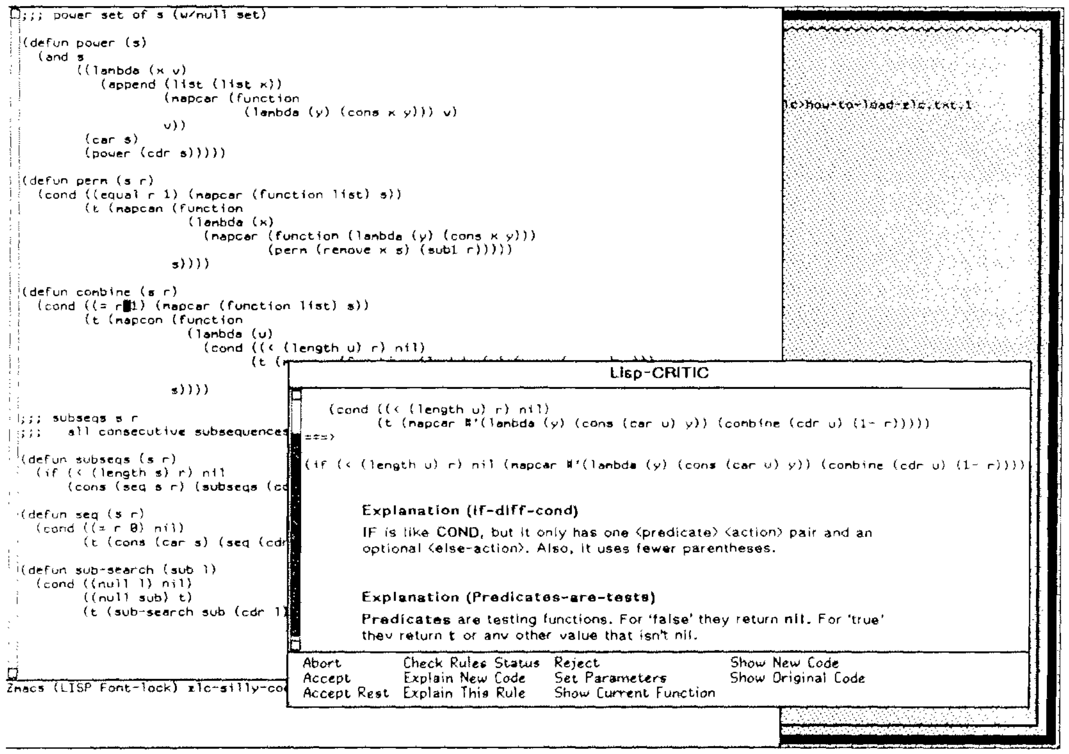
```
Zmacs (LISP Font-lock) zlc-silly-co
Move point
```

```
c>how-to-load-zlc.txt.1
```

Lisp-CRITIC

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

   (cond ((< (length u) r) nil)
         (t (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r)))))
===>

(if (< (length u) r) nil (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r))))
```

Figure 2: LISP-CRITIC display of the Rule name, original code and new code



```
;;; 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 combine (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (

                  s))))

;;; subseqs s r
;;;   all consecutive subsequences

(defun subseqs (s r)
  (if (< (length s) r) nil
      (cons (seq s r) (subseqs (cc

(defun seq (s r)
  (cond ((= r 0) nil)
        (t (cons (car s) (seq (cdr

(defun sub-search (sub l)
  (cond ((null l) nil)
        ((null sub) t)
        (t (sub-search sub (cdr l
```

```
Zmacs (LISP Font-lock) zlc-silly-co
```

```
c>how-to-load-zlc.txt.1
```

Lisp-CRITIC

```
   (cond ((< (length u) r) nil)
         (t (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r)))))
===>

(if (< (length u) r) nil (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r))))
```

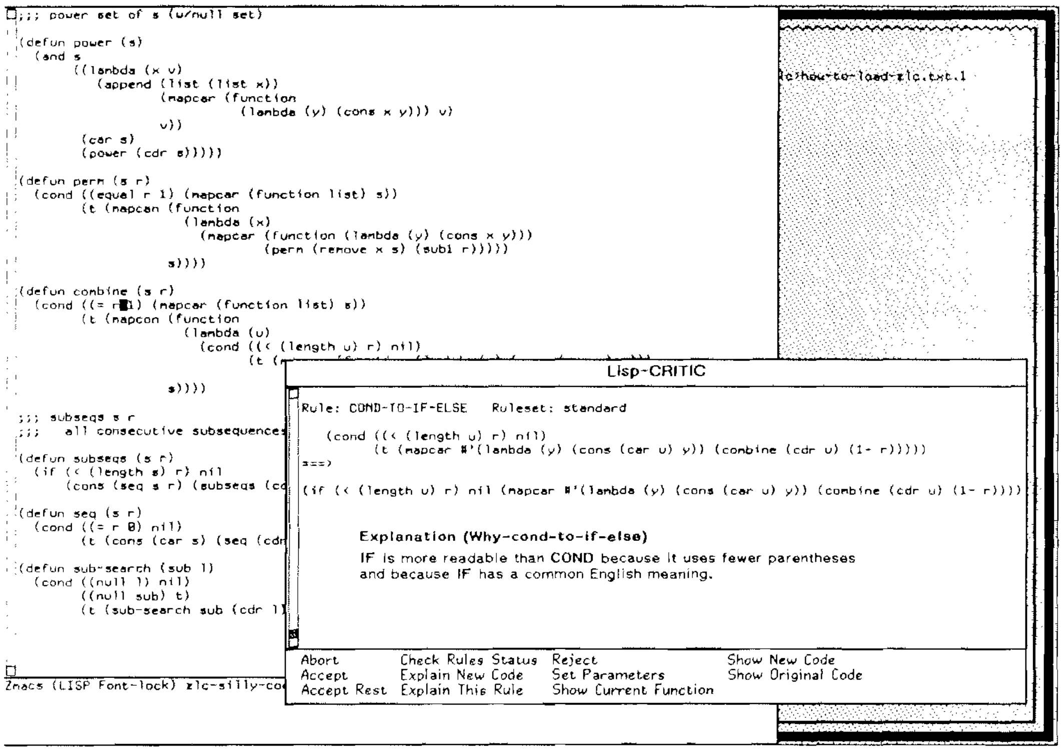
Explanation (if-diff-cond)

IF is like COND, but it only has one <predicate> <action> pair and an optional <else-action>. Also, it uses fewer parentheses.

Explanation (Predicates-are-tests)

Predicates are testing functions. For 'false' they return **nil**. For 'true' they return **t** or any other value that isn't nil.

Figure 3: User has selected *Explain New Code*



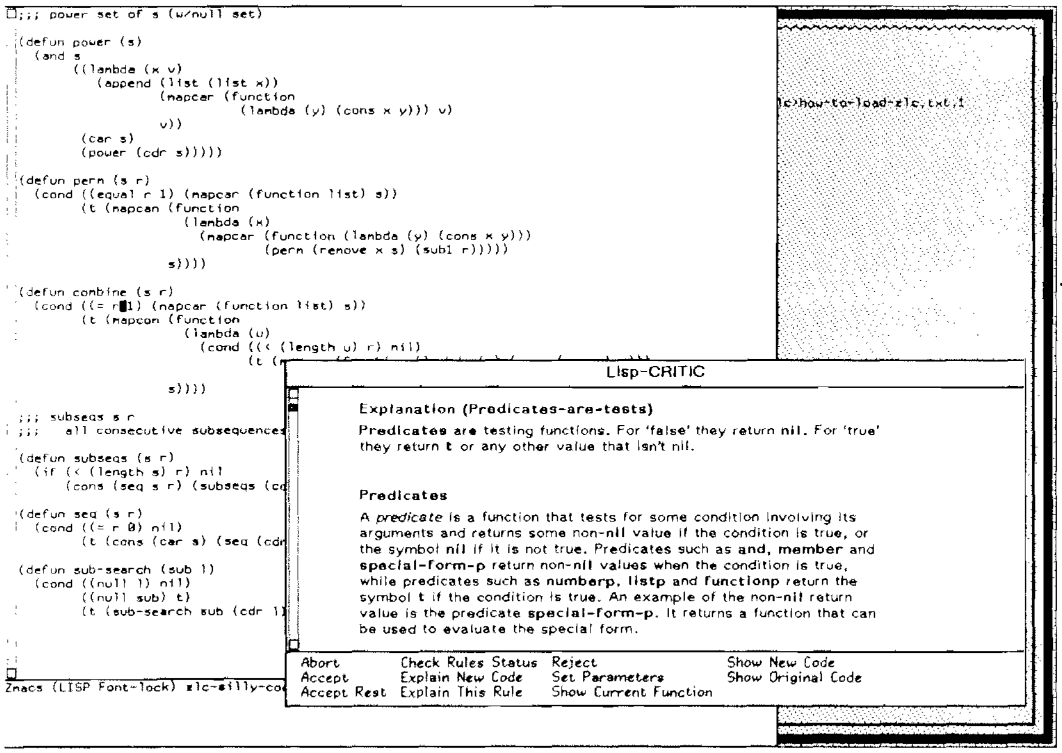
```
;;; power set of s (w/null set)

(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 combine (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (

                   s))))

;;; subseqs s r
;;;   all consecutive subsequences

(defun subseqs (s r)
  (if (< (length s) r) nil
      (cons (seq s r) (subseqs (cd

(defun seq (s r)
  (cond ((= r 0) nil)
        (t (cons (car s) (seq (cdr

(defun sub-search (sub l)
  (cond ((null l) nil)
        ((null sub) t)
        (t (sub-search sub (cdr l

Zmacs (LISP Font-lock) zlc-silly-co
```

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

   (cond ((< (length u) r) nil)
         (t (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r)))))
===>

(if (< (length u) r) nil (mapcar #'(lambda (y) (cons (car u) y)) (combine (cdr u) (1- r))))
```

Explanation (Why-cond-to-if-else)

IF is more readable than COND because it uses fewer parentheses and because IF has a common English meaning.

Figure 4: User has selected *Explain this Rule*



```
;;; 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 combine (s r)
  (cond ((= r 1) (mapcar (function list) s))
        (t (mapcon (function
                     (lambda (u)
                       (cond ((< (length u) r) nil)
                             (t (

                   s))))

;;; subseqs s r
;;;   all consecutive subsequences

(defun subseqs (s r)
  (if (< (length s) r) nil
      (cons (seq s r) (subseqs (cd

(defun seq (s r)
  (cond ((= r 0) nil)
        (t (cons (car s) (seq (cdr

(defun sub-search (sub l)
  (cond ((null l) nil)
        ((null sub) t)
        (t (sub-search sub (cdr l

Zmacs (LISP Font-lock) zlc-silly-co
```

Explanation (Predicates-are-tests)

Predicates are testing functions. For 'false' they return **nil**. For 'true' they return **t** or any other value that isn't nil.

Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns some non-**nil** value if the condition is true, or the symbol **nil** if it is not true. Predicates such as **and**, **member** and **special-form-p** return non-**nil values** when the condition is true, while predicates such as **numberp**, **listp** and **functionp** return the symbol **t** if the condition is true. An example of the non-**nil** return value is the predicate **special-form-p**. It returns a function that can be used to evaluate the special form.

Figure 5: User has clicked on *Predicates*

replaces the original in the edited file. The LISP-CRITIC window then goes into the background, and the Zmacs window comes to the front.

FUTURE DEVELOPMENTS

We encountered two stumbling blocks in the development of a working explanation component, and future work will be needed to overcome both. First, when the LISP-CRITIC was first written, no attempt was made to capture the concepts underlying the transformation rules [8]. The metrics used to define a transformation that produces "cognitively simpler" code were not specified. The knowledge base was built using traditional knowledge acquisition methods, i.e. interviewing expert LISP programmers. Some principles based on theories of cognitive psychology, software engineering, and programming language should be identified. One example is that "English words make better function names than made-up words." Eventually, only rules that can be shown to support one of these principles should be contained in the knowledge base.

Second, no well-defined taxonomy of what concepts and functions underlie the LISP forms used in the rules is available. We have developed a model of the conceptual structure of LISP based on a study of popular LISP texts. This model appears to be sufficient to serve our needs, but we need to verify that model. This model is the key to linking the rule-base, set of minimal explanations, and the user model representation of the user's understanding of LISP.

An unanswered issue is how well this approach will scale up. The LISP-CRITIC rule base contains about 100 rules for which explanations would be useful. Providing a few different differential explanations for each of these is time consuming, yet possible. The problem becomes difficult when there is no clear way of identifying how to differentiate given new concepts in terms of old concepts. By choosing artificial languages as a basis (such as C and Pascal), we have shown here how it is possible to generate differential explanations. A long term approach is to build a knowledge-base that is large enough and fine-grained enough to allow an explanation component to describe how two given concepts differ.

There are a number of ways to improve the system's minimal explanations and fallback capabilities. Graphical explanations, similar to those provided in the KAESTLE system [22; 1], are particularly appropriate to explanations of LISP data structures. Since the explanations are stored in the Symbolics Document-Examiner, which relies on the Concordia hypermedia system, graphics could easily be integrated. Runnable code examples are another option in Concordia. Finally, natural-language generation might be used to produce the minimal explanations themselves, or to provide a dialog fallback capability like that described by [20].

SUMMARY

The problems with current on-line explanation systems are widely recognized, and are in large part due to attempts to provide complete explanations as one-shot affairs. Many of the efforts to improve these systems have concentrated on emulating the natural-language dialogs observed in human-to-human communication. However, theoretical results in discourse comprehension and human-computer interaction indicate that human-to-human communication techniques are not entirely appropriate to computer-based explanations. This paper has identified an approach to explanation giving that applies those theoretical ideas while taking advantage of existing capabilities and resources in a workstation computing environment.

The approach provides several layers of explanation for the advice provided to a user by a knowledge-based critiquing system. The most fundamental layer is simply the name of the transformation rule and its result. The next layers supply a brief textual description and the underlying rationale for the critic's recommendation. The usual problems with canned text are minimized by accessing a user model to tailor these explanations. The final layer, supplementing the minimal explanations, is a rich hypertext information space in which the user can explore further details and concepts.

Acknowledgements We thank our colleages in the Human-Computer Communication group for feedback and ideas, especially Andreas Girgensohn and Scott Henninger. The research was partially supported by NSF grant CDA 8420944 and by a grant from the Colorado Institute for Artificial Intelligence. The CIAI is sponsored in part by the Colorado Advanced Technology Institute (CATI), an agency of the State of Colorado. CATI promotes advanced technology education and research at universities in Colorado for the purpose of economic development.

References

1. 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.

2. Bruce K. Britton, John B. Black (Ed.). *Understanding Expository Text.* Lawrence Erlbaum Associates, London, 1985.

3. B. Chandrasekaran, C. Tanner, J.R. Josephson. "Explaining Control Strategies in Problem Solving". *IEEE Expert 4*, 1 (Spring 1989), 9-23.

4. W.G. Chase, H.A. Simon. "Perception in Chess". *Cognitive Psychology 4* (1973), 55-81.

5. L. Danlos. *The Linguistic Basis of Text Generation.* University of Cambridge Press, Cambridge, 1987.

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

7. R. Fincher-Kiefer, T.A. Post, T.R. Greene, J.F. Voss. "On the role of prior knowledge and task demands in the processing of text". *Journal of Memory and Language 27* (1988), 416-428.

8. G. Fischer. A Critic for LISP. Proceedings of the 10th International Joint Conference on Artificial Intelligence (Milan, Italy), Los Altos, CA, August, 1987, pp. 177-184.

9. G. Fischer. Cooperative Problem Solving Systems. First International Symposium on Artificial Intelligence, Monterry, Mexico, October 1988, 1988. to be published.

10. G. Fischer, T. Mastaglio. Computer-Based Critics. Proceedings of the Twenty-Second Annual Hawaii Conference on System Sciences, Vol. III: Decision Support and Knowledge Based Systems Track, IEEE Computer Society, January, 1989, pp. 427-436.

11. 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.

12. B.A. Fox. Robust learning environments -- the issue of canned text. Institute of Cognitive Science, University of Colorado, Boulder, Colorado, 1988.

13. W.J. Hansen, C. Haas. "Reading and writing with computers: a framework for explaining differences in performance". *Communications ACM 31* (September 1988), 1081-1089.

14. R. Kass. Modelling User Beliefs for Good Explanations. MIS-CIS-87-77, LINC LAB 82, University of Pennsylvania, 1987.

15. A. Kennedy, A. Wildes, L. Elder, W.S. Murray. "Dialogue with machines". *Cognition 30* (1988), 37-72.

16. W. Kintsch. The Representation of Knowledge and the Use of Knowledge in Discourse Comprehension. In *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.

17. George R. Klare. *The Measurement of Readability.* Iowa State Press, Ames, Iowa, 1963.

18. T. Mastaglio. User Modelling in Computer-Based Critics. Proceedings of the 23rd Hawaii International Conference on the System Sciences, IEEE Computer Society, 1990. to be published.

19. K. McKeown, M. Wish, K. Matthews. Tailoring Explanations for the User. Proceedings of the Ninth International Joint Conference on Artificial Intelligence (18-23 August 1985, Los Angeles, CA), August, 1985, pp. 794-798.

20. J. Moore. Explanations in Expert Systems. USC/Information Sciences Institute, 9 December 1987.

21. J. Moore. Responding to 'Huh': Answering Vaguely Articulated Follow-Up Questions. Proceedings CHI '89 Human Factors in Computing Systems, New York, May, 1989, pp. 91-96.

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

23. B. Reeves. Finding and Choosing the Right Object in a Large Hardware Store -- An Empirical Study of Cooperative Problem Solving among Humans. Department of Computer Science, University of Colorado, Boulder, CO, 1989. forthcoming.

24. W. Strunk, E.B. White. *The Elements of Style, 2nd ed.* Macmillan, New York, 1972.

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

26. W. R. Swartout. "XPLAIN: A system for creating and explaining expert consulting programs". *Artificial Intelligence 21*, 3 (1983), 285-325.

27. Randy L. Teach, Edward H. Shortliffe. An Analysis of Physicians' Attitudes. In *Rule-Based Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1984, pp. 635-652.

28. 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.

29. E.H. Weiss. "Breaking the grip of user manuals". *Asterisk -- Journal of ACM SIGDOC 14* (Summer 1988), 4-11.