Enhancing Incremental Learning Processes with Knowledge-Based Systems

Gerhard Fischer
Andreas Lemke
Helga Nieper-Lemke

CU-CS-392-88 March 1988

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

This research was supported by the Office of Naval Research under Contract No. N00014-85-K-0842.

University of Colorado at Boulder

Department of Computer Science

ECOT 7-7 Engineering Center
Campus Box 430
Boulder, Colorado 80309-0430
(303) 492-1502

March, 1988

Enhancing Incremental Learning Processes with Knowledge-Based Systems

Final Project Report

Grant No. N00014-85-K-0842 from the Office of Naval Research

July 1985 - December 1986

Gerhard Fischer, Andreas C. Lemke, and Helga Nieper-Lemke
Department of Computer Science and Institute of Cognitive Science
University of Colorado
Boulder, CO 80309-0430

Abstract

The purpose of this project was to investigate the enhancement of incremental learning processes with knowledge-based support systems. The research was carried out in the context of learning to use high-functionality computer systems. As a basis for the design of support systems, the structure of learning spaces was studied. The model of increasingly complex microworlds was found to adequately describe the relationships between the units of expertise.

The usability of systems can be increased by reducing and by enhancing learning processes. Construction sets and design environments are support systems which reduce learning processes. Two prototypical systems (WIDES and TRIKIT) demonstrate techniques for achieving this goal. Critics are task-oriented intelligent systems to enhance learning processes. As an instance of a critic, the LISPCRITIC was designed and implemented. It contains knowledge about FRANZLISP which is utilized to make programs either more cognitively efficient or more machine efficient. The LISPCRITIC supports learning by suggesting improvements and explaining relevant concepts.

Keywords

learning spaces, units of expertise, microworlds, incremental learning, reducing learning effort, learning on demand

Table of Contents

List of Figures

1. Introduction

Many current systems that are based on high technology cannot be understood completely by any person (There are no experts any more [Draper 84]). Users are no longer getting a complete training before they use the system. Rather, learning is incremental and intermixed with phases of knowledge and skill application.

In the context of this project, these effects were studied in the domain of using *high-functionality computer systems*. Modern computer systems are best thought of in terms of their capacity to serve as knowledge stores rather than in their capacity to compute. The needs of users whose tasks involve complex knowledge can only be met by systems that are large and complex. We should not expect that these systems will be small and simple. In order to develop computer systems that are able to support complex knowledge use, a high degree of function must be built into these systems. To illustrate the dimensions of this functionality, an analysis of some typical systems is given in Figure 1-1.

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:

- FRANZLISP: 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 1-1: Quantitative Analysis of Some Systems

LISP and WLISP programming were the primary example domains which we have studied in the project (WLISP [Fabian, Lemke 85] is an object-oriented user interface toolkit and programming environment).

Chapter 2 introduces the concepts of "microworld" and "learning space" that are central to this research. Chapter 3 discusses learning processes seen as transitions in a space of microworlds. The concept of incremental learning is discussed in Chapter 3.1. Learning processes can be supported in two ways. The need to learn can be reduced, and learning processes can be enhanced by knowledge-based support systems. Principles and prototypical systems to reduce learning are discussed in Chapter 4, methods of enhancing learning processes are described in Chapter 5.

2. Learning Spaces

This chapter introduces some terminology and concepts which are fundamental to the rest of the report.

In order to support incremental learning, we have to find out about the knowledge units and skills (expertise) that have to be acquired. We view them as being organized in a space of interrelated *units of expertise* such as facts and concepts. The units are ordered by a prerequisite relation. Some units may need to be learned before others can be understood and applied. An example in the domain of LISP is:

> In order to use the **mapcar** function which applies a function to a list of values, one must know that a function can be specified as a function name or as a lambda expression.

Learning spaces provide information about the relationship between knowledge units and capabilities of the learner. They allow us to answer the questions which units are required for doing certain tasks and which tasks can be accomplished by somebody with certain knowledge and skills. In addition, the efficiency of doing a task is heavily influenced by the particular units known to the learner.

Information about learning spaces can be used to make learning processes more efficient and to tailor them to specific learners and their tasks. This is especially true when the learning path has to be determined by the individual needs of the learner and therefore cannot be adequately prescribed by a tutorial.

2.1 Microworlds

It is useful to group those units of expertise that are collectively required to learn other units or to do certain tasks. We call these groups "microworlds." A microworld is a sufficiently consistent submodel of the real world. It sets limits on what and how things can be done. Within this simpler world many things become possible which, if based on the full complexity of the world, would be restricted to highly trained experts. Within a microworld the *objective computability* (i.e., what someone can do in principle) may be reduced, but the *subjective computability* (i.e., what someone really can do) will be increased.

As we will see in the following example, creating a microworld can mean to modify the original system so that it is amenable to a microworld structure. Artifacts and concepts may be introduced that explicitly hide the complex structure of the real system. These artifacts may get in the way when the learner tackles more sophisticated problems.

The WLISPRC SHEET (Figure 2-1) restructures the WLISP system. Like many large systems, WLISP has several configuration parameters that make it adaptable to various uses like programming and text processing. Many of these parameters (e.g., process window font, screen background, automatic break window) are not easy to access; their names are hard to remember; the user may not even know of their existence.

The WLISPRC SHEET is a system configuration sheet showing a subset of those parameters. Having this sheet, it is no longer necessary to remember their names, access functions, and possible values. The user must only learn how to get the sheet on the screen (by clicking on its icon) and how to edit the parameter values. For both capabilities, however, previous knowledge can be reused. The sheet's functionality is limited to the parameters shown, but this limited functionality enables almost every user of WLISP to customize their system along some of the most important dimensions. The WLISPRC SHEET has the advantage that the other, lower-level methods of setting parameters are not hidden and can still be used to access the other parameters by more knowledgeable users.

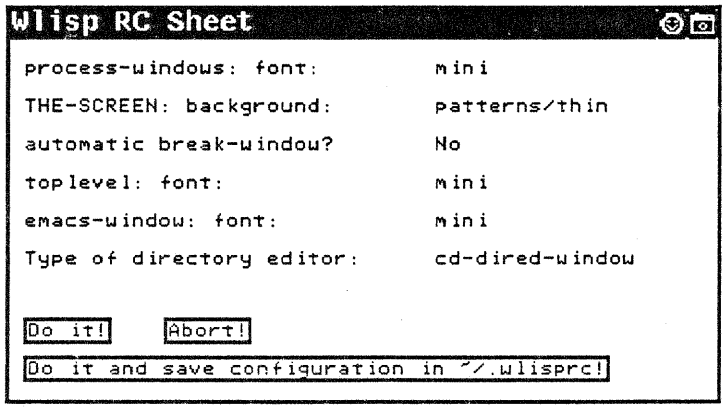

Figure 2-1: The WLISPRC SHEET

Before the WLISPRC SHEET existed, a few expert users customized the WLISP system using low-level methods. Less sophisticated users either did not customize at all or made temporary modifications that had to be reexecuted manually every time the system was started. The WLISPRC SHEET introduced a microworld for users of WLISP. After the implementation of the sheet, most of the users from the second group made customizations using the new facility.

This microworld for the task of customizing WLISP required the introduction of the WLISPRC SHEET, which represents a modification of the WLISP system. Customization of the version of WLISP without the WLISPRC SHEET required mastery of a larger microworld that inhibited learning progress.

2.2 The Topology of Learning Spaces

A simple model of the structure of a learning space is the *onion model* (Figure 2-2). Microworld A represents the knowledge needed for the simplest set of tasks. Microworld B is the knowledge for the next more complex set of tasks, etc. The microworlds form a series of *increasingly complex microworlds* [Fischer 81]. Each microworld is included in the next higher one because it contains units which are necessary for all the following ones. The boundaries between microworlds are determined by the sets of tasks. Whether a particular unit belongs into a microworld depends on whether it is needed to execute the tasks.

A desirable property of a system is a small initial microworld (A) which represents a *low learning threshold.* If the subsequent microworlds are small and high in number, then it will be easy to proceed and to gradually become an expert. If the difference between one microworld and the next is large, progress will tend to be inhibited, and the learner will settle on a suboptimal plateau. See Section 3.2 for more details.

In practice, there is more than one way to proceed from a microworld. This refinement leads to the *lettuce model* of learning spaces (Figure 2-3). Microworld A is the initial microworld. Microworlds B and C are extensions to A, but are independent of each other, and the learner can choose in which direction

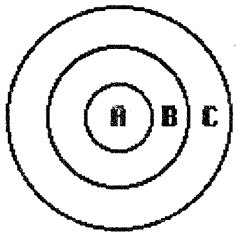

Figure 2-2: The Onion Model of Learning Spaces

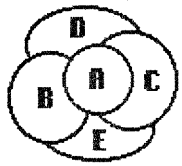

Figure 2-3: The Lettuce Model of Learning Spaces

to proceed first. Likewise, D and E are independent extensions to the union of A, B, and C. Every area includes those other areas it has a concave border to. Obviously, if a microworld X allows to do a task, then also every microworld Y that contains X allows to do it, possibly in a more efficient way.

If a task can be achieved in different ways, then there may exist several alternative (minimal) microworlds which would allow to do the task. A simple example is the situation that there are multiple commands that essentially achieve the same effect (e.g., see [Ehrlich, Walker 87], Section 5.1).

The lettuce model can be generalized further. If a system can be used to do completely different sets of tasks, or if a set of tasks can be accomplished in different ways, then there need not be a single kernel (*multikernel lettuce model*; Figure 2-4).

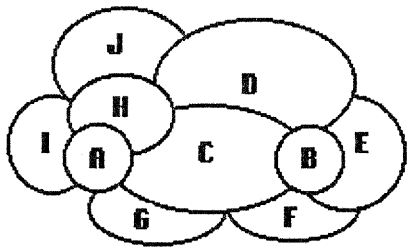

Figure 2-4: The Multikernel Lettuce Model of Learning Spaces

Microworlds are not closed with respect to intersection, i.e., the units of expertise common to two microworlds may not, by themselves, be sufficient for accomplishing useful tasks.

2.3 Analysis of the LISP Learning Space

We have analyzed the domain of learning the FRANZLISP programming language to determine which one of the models described in the previous section fits best the topology of this learning space.

We asked local programmers to provide us with programs typical of their work. Six programs were written by different experienced LISP programmers, and eleven programs were written by different beginners. Figure 2-5 shows a list of the thirty most frequently used functions for the expert and beginner categories (numbers indicate by how many programmers of the group they were used).

The list of expert functions contains all the functions that were used by at least five of the six programmers. The beginner functions were used by at least three programmers.

At the bottom of the figure, we give a comparison of the two columns. The functions are classified into three categories. The first category contains those functions that were among the thirty most frequently used functions for both beginners and experts. The second and third categories list functions that were among the thirty most frequently used functions for experts but not for beginners and vice versa.

All three categories are approximately the same size. Experts use more efficient functions (`eq` versus `equal`, + versus `plus`). Beginners use safer functions (`equal`, `plus`).[1] Experts compile (`declare`). Experts use more specialized i/o functions such as `tyi`, `readc`, and `patom`. These functions, however, do not show up in the figure because there are many of them and each single one is used infrequently.

There is a small kernel of functions that are frequently used by both experts and beginners. There are no functions that experts no longer use,[2] i.e., this supports the lettuce model of microworlds where initial microworlds are a subset of later microworlds. There is however a shift to more machine efficiency, and experts have a greater repertoire of specialized functions.

---

[1] See the LISPCRITIC described in Section 5.2, a system that supports the transition from beginner to expert by introducing the beginner, e.g., to more efficient functions.

[2] This can be seen by looking at the complete lists of functions used.

The thirty most frequently used functions in expert and beginner programs:

experts (6)		beginners (11)	
setq	6	terpri	11
return	6	read	11
quote	6	quote	11
or	6	null	11
not	6	equal	11
eq	6	defun	11
do	6	cond	11
def	6	or	10
declare	6	do	10
cons	6	cons	10
cond	6	cdr	10
cdr	6	car	10
car	6	and	10
cadr	6	print	9
caddr	6	princ	9
and	6	let	9
-	6	list	8
+	6	setq	7
tyi	5	plus	6
rplaca	5	not	6
numberp	5	member	6
null	5	random	5
list	5	numberp	4
let	5	lessp	4
cddr	5	cadr	4
cadddr	5	caddr	4
atom	5	append	4
1-	5	sub1	3
1+	5	reverse	3

Functions among the thirty most frequently used functions for both beginners and experts:

and, caddr, cadr, car, cdr, cond, cons, do, let, list, not, null, numberp, or, quote, setq

Functions frequently used by experts but infrequently used by beginners:

+, -, 1+, 1-, atom, cddr, declare, def, eq, return, rplaca, tyi

Functions frequently used by beginners but infrequently used by experts:

append, defun, equal, lessp, member, plus, princ, print, random, read, reverse, sub1, terpri

Figure 2-5: Usage of LISP Functions

3. Learning Processes

The transition to another microworld is a step forward in a two dimensional space as shown in Figure 3-1. Proceeding to a more complex microworld can mean an increase in functionality (e.g., learning LISP functions for file operations makes it possible to write LISP programs that access and manipulate files). In the figure, user A made such a transition.

Proceeding to a more complex microworld can also mean a gain in efficiency, which may not only mean machine efficiency, but also cognitive efficiency (e.g., learning a high-level function that directly solves the problem instead of combining a set of primitives). User B proceeded to a microworld that allows him or her to operate more quickly and more efficiently. But user B did not add to the range of tasks that he or she is able to do.

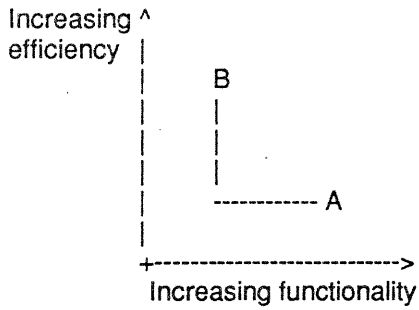

Figure 3-1: Progress to Other Microworlds

An example from the UNIX operating system illustrates this relationship: If a user switches frequently between two working directories with rather long path names using the command `cd <pathname>`, then this is rather inefficient because much typing is required, which may lead to typing mistakes and a high cognitive load for remembering the correct pathnames. The efficiency of this task can be improved by learning a new command: `pushd`. `pushd <pathname>` has the same effect as the `cd` command, but, in addition, pushes the previous working directory on a stack. Subsequent uses of `pushd` without an argument, then, switch between the current and the previous working directory involving little typing and memory load. The `pushd` command, however, does not introduce any new functionality that was not covered by the `cd` command.

3.1 Incremental Learning

Incremental learning eliminates the separation between learning and applying what has been learned. Learning something incrementally allows doing useful work before knowing everything. Intertwining these two phases (Figure 3-2) has several advantages:

- The learning phase is not a big unproductive block (this overcomes the "production paradox" [Carroll, Rosson 86] to some extent).
- The decision on what exactly will be learned can be deferred until it is needed ("learning on

demand" [Fischer 87a]). This requires, however, that it is a small enough step that can be accomplished within the available time.

- A real need is a good motivation.
- Being in a situation when knowledge can be applied, provides a context for the learning process. The knowledge acquired can be linked with the situation (at least one applicability condition for the knowledge is known) and therefore better understood.

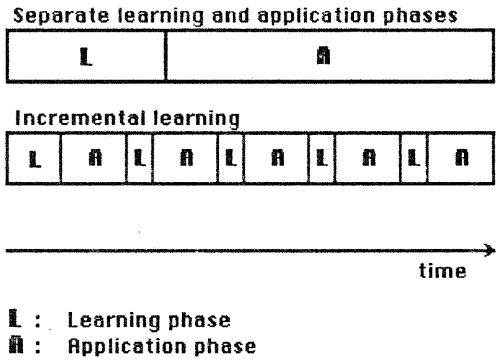

Figure 3-2: Different Relationships Between Learning and Application Phases

Many computer systems offer a great functionality, which is not intended to be used by every single user. Rather, different users will choose to learn only those parts that are needed to do their jobs. When new tasks have to be done, new functionality may have to be used and learned.

The theory of learning spaces (see Chapter 2) can tell in which way learning and work phases can be intertwined. The initial microworld describes the minimal knowledge necessary. Learning is only useful in increments of microworlds. The dependency relation of a learning space determines what the microworlds are that have to be mastered in order to do a certain set of tasks.

3.2 Microeconomics of Learning Processes

In this section we will analyze in more detail the impact of microworld size as briefly mentioned in Section 2.2. The interesting aspect of size is the effort that a learner has to spend to learn a microworld. Each learning effort results in a benefit such as extending what can be achieved or doing something more efficiently.

Sometimes effort and benefit can be directly compared. If learning a particular function takes one hour and the use of this function saves five minutes a day and the function will be used considerably longer than twelve days, then this is a good investment.

When learning is incremental, effort and benefit of proceeding to another microworld can be considered. If the benefit does not exceed the effort, then the borderline between the microworlds is a barrier, and learning progress will tend to stop. A sequence of many small efforts is, in general, preferable to a few large ones even if they are matched with equally large benefits (see Section 2.2).

4. Reducing Learning Processes

4.1 General Principles

The usability of systems can be increased by *enhancing* learning processes and by *reducing* them. The amount of expertise necessary to do a task is not only determined by its own nature, but also by the design of the system being used to accomplish it. There is no conservation law of complexity [Simon 81]. Many existing computer systems show a great potential of being improved in this respect through the application of modern human-computer communication technology. These chances should be explored before considering sophisticated learning support tools.

Learning processes can be reduced in several ways:

1. *Exploiting the basic human information processing capabilities.* The human visual system, for example, is very powerful. Instead of using the `pushd` command to manipulate an invisible stack, it is much easier to change working directories by pointing into a display of one's directory structure [Fischer 87b].

2. *Exploiting problem domain knowledge.* Many systems provide abstractions which are derived from hardware and software characteristics of the implementation. Abstractions should instead come from the learner's conceptual world, i.e., from the problem domain (human problem-domain communication; [Fischer, Lemke 88a]). The Music Construction Set (from Electronic Arts) provides high-level concepts such as measures and notes of different pitch and duration displayed in the common music notation. The musician does not need to learn about the clock frequency of the computer and how to program time delay loops. A good computer system may come to the point where a computer expert can make less use of it than a domain expert.

3. *Hiding redundant or rarely used detail.* A system should be easy to use for simple things. It should show a reasonable behavior when only a small part of its functionality is being used. Even if the available abstractions are on the task domain level, they may be very complex. Parameters affecting specific details should have defaults so that a user need not be aware of them. Printing a document, for example, should be doable with a simple command. The user should not be required to know of all the options like specifying a different printer, different page formats, etc.

4. *Consistency and uniformity.* This is a problem from which many systems have suffered, especially when they have evolved over a long period of time. Consistent systems allow to draw upon previous knowledge. One instantiation of this is building computer systems according to a real world metaphor. One of the most prominent is the desk top metaphor for user interfaces as pioneered by the XEROX STAR [Smith et al. 83].

5. *Learning versus retrieving information.* Learning can be avoided by easy and fast information retrieval. Information retrieval systems (like help systems, documentation systems, or "cheat sheets") can be viewed as an extension of the human memory. User interface techniques like menus are based on this principle. Many existing information retrieval systems (like manuals) suffer from the fact that access is by implementation unit (e.g., LISP function, UNIX command) rather than by application goal (e.g., command input methods, object display methods).[3] Systems in the style of *cook books* are perceived as very helpful exactly because they provide this type of access. For example:

[3]This issue is being further explored in a current project [Fischer, Kintsch 86; Fischer, Nieper 87].

```
If you want to:
        quickly switch back and forth between two directories in UNIX,
then:
        use the pushd command.
```

6. *Reducing functionality.* By tailoring the functionality and complexity of a system to the needs of the user, learning can be further reduced. The design of a special purpose system for a particular class of users may be very effective even if there is already a more general system which would do the job. This reduction may lead to more complicated solutions to some problems, but this is acceptable if outweighed by the advantages. The WLISPRC SHEET makes use of this principle in that it shows only a subset of the parameters.

4.2 Construction Sets and Design Environments

In this section, we will describe a class of systems that are based on the microworld model and the principle of reduced learning processes.

We distinguish between two types of support systems: *construction sets* (e.g., the Music Construction Set) and *design environments* (e.g., WIDES, see below). Construction sets provide domain level building blocks (notes, rests, clef signs, time signatures) to build artifacts of the domain (musical pieces). Although they eliminate programming errors below the domain level, they do not prevent musical errors such as disharmonies. Design environments assist the user in designing artifacts that are "reasonable" on the domain level. In the domain of music, a design environment might have knowledge about instrumentation, interesting rhythms, etc.

Both kinds of systems reduce learning processes by exploiting the user's domain knowledge. Users can interact with the system in terms familiar to them. They need not learn abstractions peculiar to a certain system.

The following two sections describe two design environments for specific areas of the WLISP construction kit. WIDES is a design environment for basic characteristics of window types, and TRIKIT is a design environment for graph display and edit tools.

4.2.1 WIDES

Because almost all modern user interfaces are window-based, one of the major tasks of user interface design is the definition of a suitable combination of window types. Many current window systems and user interface tool kits offer a wide variety of components such as text, graphic, and network windows and editors or controls like menus and push buttons. The goals of WIDES [Fischer, Lemke 88b; Fischer, Lemke 88a] are:

1. to provide a level of abstraction above the object-oriented implementation of these components,
2. to reduce the knowledge required to use the components,
3. to make their use more effective by preventing errors and suggesting the "right" components to use,
4. to support the acquisition of expertise in using these tools.

WIDES provides a safe learning environment in which no fatal errors are possible and in which enough

information is provided in each situation to ensure that there is always a way to proceed. The design environment allows its users to create specific window types for their applications.

In the following we will give an example of how WIDES employs techniques like menu selection and an adaptive, dynamic suggestion list to greatly simplify window design. Merits and shortcomings of WIDES will be discussed.

Description of WIDES. The initial state of the system is shown in Figure 4-1. It is a window with four panes:

- a `Code` pane that displays the current definition of the window type,
- a menu of `suggestions` for enhancements of the window type,
- a `history` list,
- a menu of general `operations`.

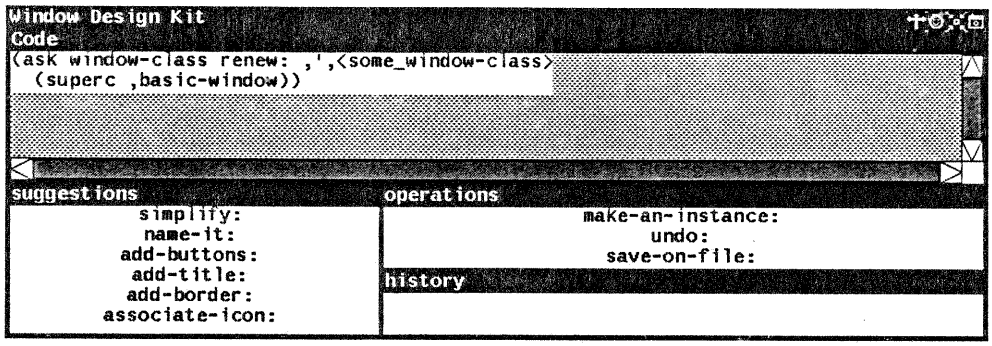

Figure 4-1: Initial State of WIDES

Figure 4-2 shows a window and an icon of the selected type.

An even more complex modification is demonstrated in Figure 4-3. Windows can be associated with push buttons such as those in the upper right corner of the WIDES window. Clicking the button with the mouse causes a message to be sent to the window. As an extension of the push buttons in the title bar supplied by default (the two right-most ones), a button for burying the window is to be added. After selecting `add-more-buttons-to-title-bar:` from the suggestions menu, the user is asked to choose a button icon and a message from two menus. The bury button appears as the leftmost button in the instance of `test-window` in Figure 4-3.

The `save-on-file:` operation may be used to save the final definition for later use. Although not much code is being generated by the system because it can use many high-level building blocks (see the `Code` panes in the various stages of the design process), having WIDES represents a significant advantage for the user. In order to construct a new window type, it is no longer necessary to know what building blocks (e.g., `title-mixin`) exist, what their names are, and how they are applied. It is no longer necessary to know that new superclasses have to be added to the `superc` description of a class. Also, WIDES determines their correct order. The system knows what types of icons are available, how an icon is associated with a window, etc.

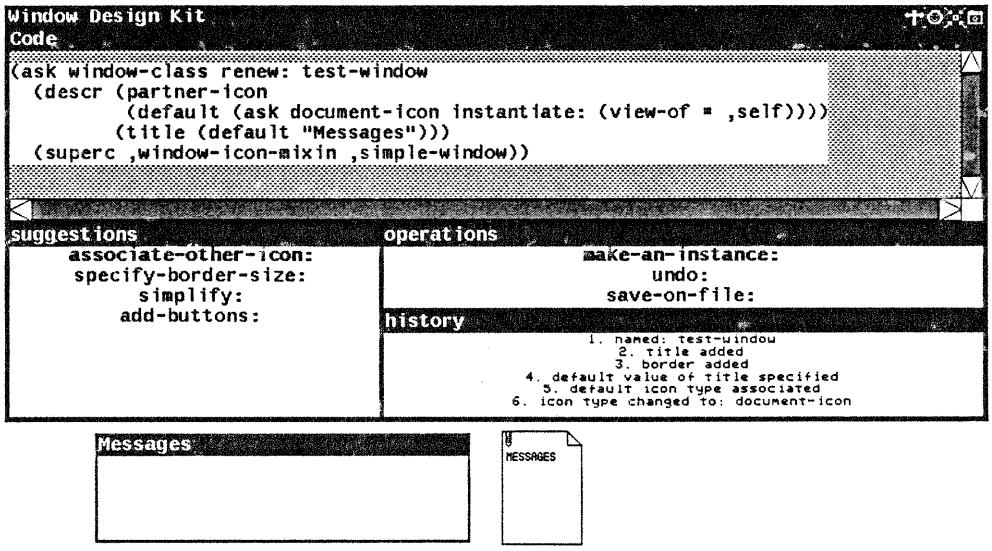

Figure 4-2: The Window and its Associated Icon

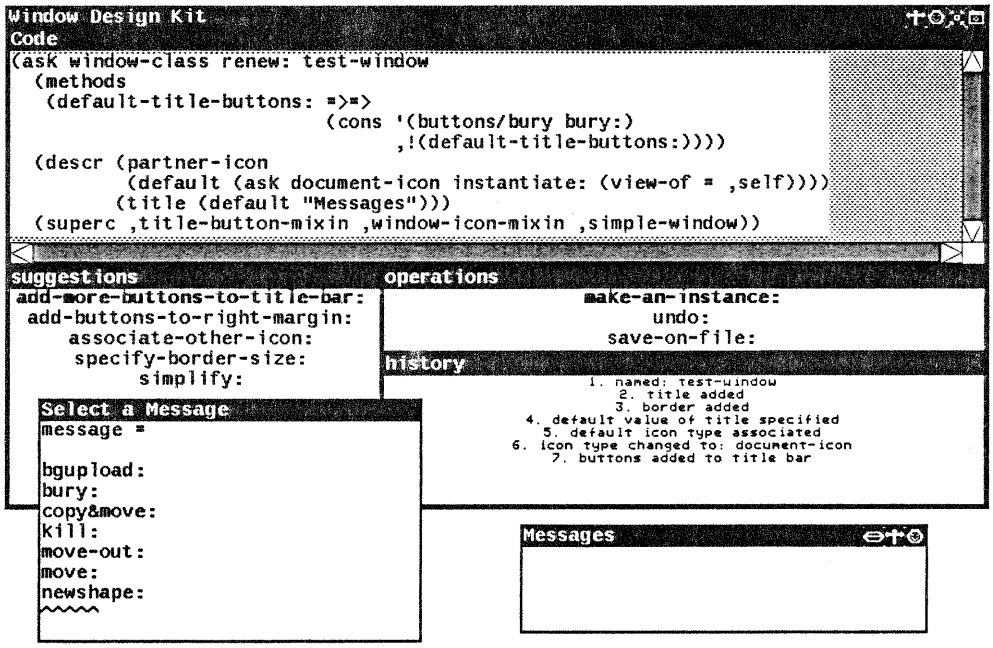

Figure 4-3: Adding a Button to the Title Bar

User interface techniques like prompting and menus make it easy to experiment in the domain of window construction. The system makes sure that errors are "impossible." This does not mean that these techniques make sure that users always *understand* what they are doing.

In WIDES, the task of window design is decomposed into a hierarchy of semantically meaningful subtasks. The critical decisions form the highest levels, and the dependent tasks can be found at the lower levels and the leaves. The suggestions menu is dynamically updated according to this model of window design. Suggestions, such as adding a button to the title bar, are given when appropriate within the overall design process. On the other hand, the dialogue is not completely system-driven, and the user can direct the design process towards desired aspects of the design.

Methods like this can quite easily be applied in well-structured domains like the present one. There are two problems, however, that need to be addressed. The first one is the understanding problem. Seeing an option in a menu does not imply that its significance is obvious. What does `associate-icon:` mean? What is the function of a window's icon? Another problem may be the sheer number of options. We did not look into this problem because it does not occur in this relatively small system, but future systems may offer hundreds of choice points. For these design environments, a system of reasonable defaults may provide some help if it is combined with a set of predefined samples that are already rather specific starting points.

4.2.2 TRIKIT

A very common user interface problem is the display and modification of hierarchical and network structures. Application systems that deal with rule dependency graphs, concepts of a domain of expertise (for explanation purposes), goal trees, or inheritance hierarchies in object-oriented languages are examples in which this problem occurs.

Our response to this problem was to build a design environment for TRISTAN, a generic display and editing tool for directed graphs [Nieper 85]. Figure 4-4 illustrates its usage. The application programmer, who is an expert for the application system but not for building user interfaces, sets and adjusts parameters of a generic tool, TRISTAN, and specifies the links between it and the application. The result of the design process is a new, application-specific tool for displaying and editing a graph structure.

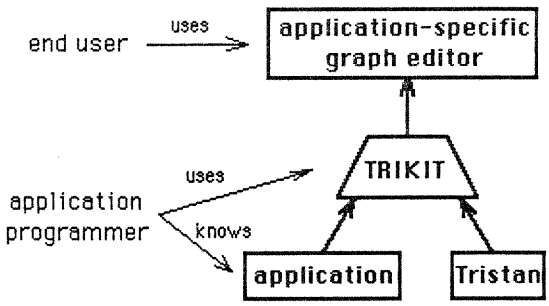

Figure 4-4: Usage of TRIKIT

Application Domain of TRIKIT. In this section, we will use the example of a hierarchical file system, which may be displayed as in Figure 4-5. Here the nodes are directories and files which are the leaf nodes. The lines represent the membership relation between files and directories, which can themselves be members of other directories. Each of the nodes is a data structure with properties like name, creation

time, owner, protection, and size. There are operations to retrieve pieces of the graph (e.g., `list-directory`) and to create and delete nodes and lines.

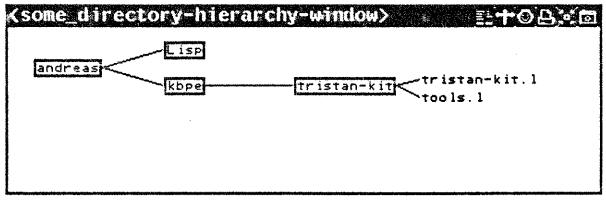

Figure 4-5: A Hierarchical File System Display

TRIKIT presents itself to the user as an interaction sheet as shown in Figure 4-6. In this window, the user specifies the interface to the application, chooses a graphical representation for the nodes, and controls the creation of the user interface.

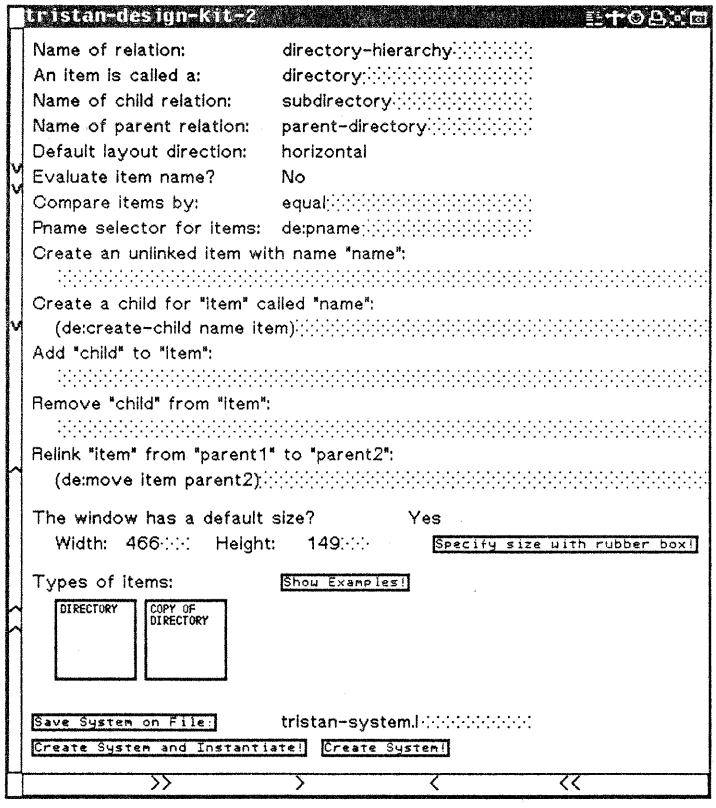

Figure 4-6: Description of the Directory Hierarchy Window

The following types of fields may be found in the interaction sheet:

edit fields	indicated by their dotted background; for entry and modification of names, numbers, program code, etc.; a mouse click on the field moves the cursor into it and allows editing of its contents.
choice fields	if the number of possible values of a field is very small, this type of field is being used; mouse clicks circle through the set of values.
menu fields	for a larger number of choices; a mouse click produces a pop up menu.
push buttons	low and long rectangles with a black frame; a mouse click activates their associated action.
subform icons	large squares; a mouse click produces a subform.

Clicking the square representing the **DIRECTORY** subform produces the form of Figure 4-7. While the main form is associated with the graph in general, the subforms describe the properties of its nodes.

Let us examine the use of the system through the example of building a directory editor like the one shown in Figure 4-5. A directory editor is a tool for viewing a hierarchical file system and for doing operations on it such as creating or removing a directory, moving a file into another directory, and renaming files.

In Figure 4-6, the first four fields have been filled in to reflect the terms of the file system domain. They establish a common vocabulary for the user and the system. They describe the names of the relation to be displayed, the names of the items that are elements of the relation, and those of the links to superordinate and subordinate nodes in the relation. The **Default layout direction** specifies whether the layout should be oriented horizontally (as in Figure 4-5) or vertically. The next field, **Evaluate item name?**, says that a user-entered name of a file or directory represents itself as opposed to being the name of a variable holding the actual item. **equal** is used as a comparison function for directory names. If the system is to be a true editor, it should be possible to create new nodes and to alter the graph structure. For this purpose, the meaning of creating a child and of relinking a node from one parent to another have been specified using the application functions **de:create-child**[4] and **de:move**.

Figures 4-7 and 4-8 show the forms for a directory and a file node. The two most important fields are the fields that determine how the parents and the children of a node are computed in the new application. The **de:parents** function computes the list of superdirectories and the **de:children** function computes the subdirectories, that is, the contents of the directory. The **Item representation** field determines how a node is displayed. For directories, it has been set to **string-region** which displays a string surrounded by a border; files use **label-region**, which has no border (see Figure 4-5). The **de:pname** function in the **Label** field computes a "print name," or a label, for the items; that is, it strips off the leading pathname component and leaves the file name, which is unique only locally within its directory. There has also been an action associated with the nodes: Clicking a directory node will make the directory it is representing the current working directory; clicking a file node will load the file into an editor.

---

[4]Functions with a **de:** prefix belong to the application domain (a directory editor). They are application-specific and must be supplied by the application programmer.

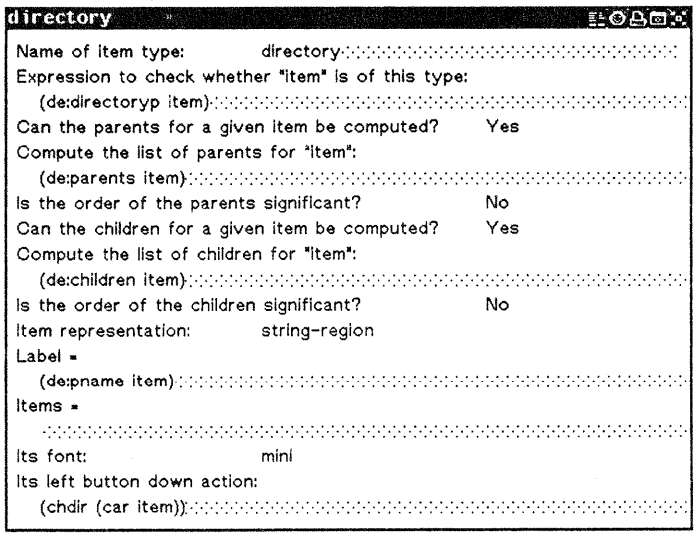

Figure 4-7: The Directory Node Form

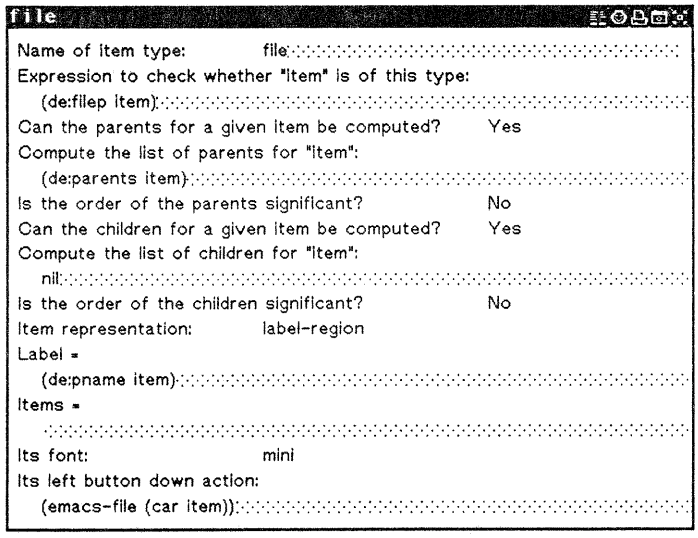

Figure 4-8: The File Node Form

Discussion. With TRIKIT, the user can construct useful systems without knowing the details of the selected building blocks. This design process happens on the level of abstract properties of graphs (e.g., horizontal versus vertical layout of the graph), not on the implementation level. Sometimes it is necessary to use code-level specifications, for instance, code that computes the list of parents for an item. This form of specification does not pertain to the graph as such, but is required for the interface between application and TRIKIT. TRIKIT does not make any assumption about how graphs might be implemented in application systems.

5. Enhancing Incremental Learning Processes

Reducing the amount of expertise to acquire can only be part of a complete strategy. Methodologies have to be developed which support incremental learning of new knowledge and skills. Figure 5-1 shows a classification schema and some systems falling into its categories.

Specificity	Initiative of system	Increases efficiency	Increases functionality
specific	active	---	---
	passive	Manual, Passivist	Manual, Passivist
task-oriented	active	ACTIVIST	---
	passive	---	LISPCRITIC
unspecific	active	---	---
	passive	DYK	DYK

Figure 5-1: Types of Systems to Enhance Incremental Learning

There are three dimensions: specificity, initiative, and type of improvements:

- *Specificity.* Support systems for *specific* information help in cases when the user needs an answer to a well defined question such as which command to use for changing the working directory. *Task-oriented* information pertains to what the learner is doing in general. Critics (like the LISPCRITIC (see below) or ACTIVIST, an active help system [Fischer, Lemke, Schwab 85]), are support systems providing this kind of information. Suggestions may be based on a rather crude understanding such as of syntax only. *Unspecific* information systems involve no active search by the learner. Frequently, helpful information comes from "unexpected" sources. Examples are: watching other people, reading electronic bulletin boards, using a DYK system (Did You Know; [Owen 86]), etc. This style of learning is important for learning about new facilities and their applicability conditions for new and unexpected purposes.
- *Initiative.* Traditional support systems have mostly been *passive*, i.e., the learner had to ask and search for information. Manuals, whether online or printed, and the PASSIVIST system, a natural-language-based help system, [Fischer, Lemke, Schwab 85] are examples. These systems can enhance learning by making the search for information effective using well structured presentations. If, however, the system volunteers information when it can infer a need by the user, then it is called an *active* system. Active systems need *metrics* to decide when to display information. If a system becomes active too often, it will be perceived as intrusive or will be ignored. The right point of time for displaying a suggestion and which information to offer are crucial issues.
- *Type of improvements.* Support systems can increase the efficiency with which to do tasks, and they can increase the range of tasks that can be done (see Chapter 3).

5.1 Critics

Critics can become active spontaneously or on request. *Passive critics* are systems that react to unspecific requests for help. The critic tries to understand what the learner is doing and suggests improvements when the learner asks for advice. *Active critics* automatically present information when they can infer that the user can benefit from it. Active critics may be based on the same mechanisms of understanding as critics. However, they are "real time" systems.

5.2 The LISPCRITIC

The LISPCRITIC is a passive critic for FRANZLISP (Figure 5-2). It suggests improvements to program code. The critic works in one of two modes. 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.

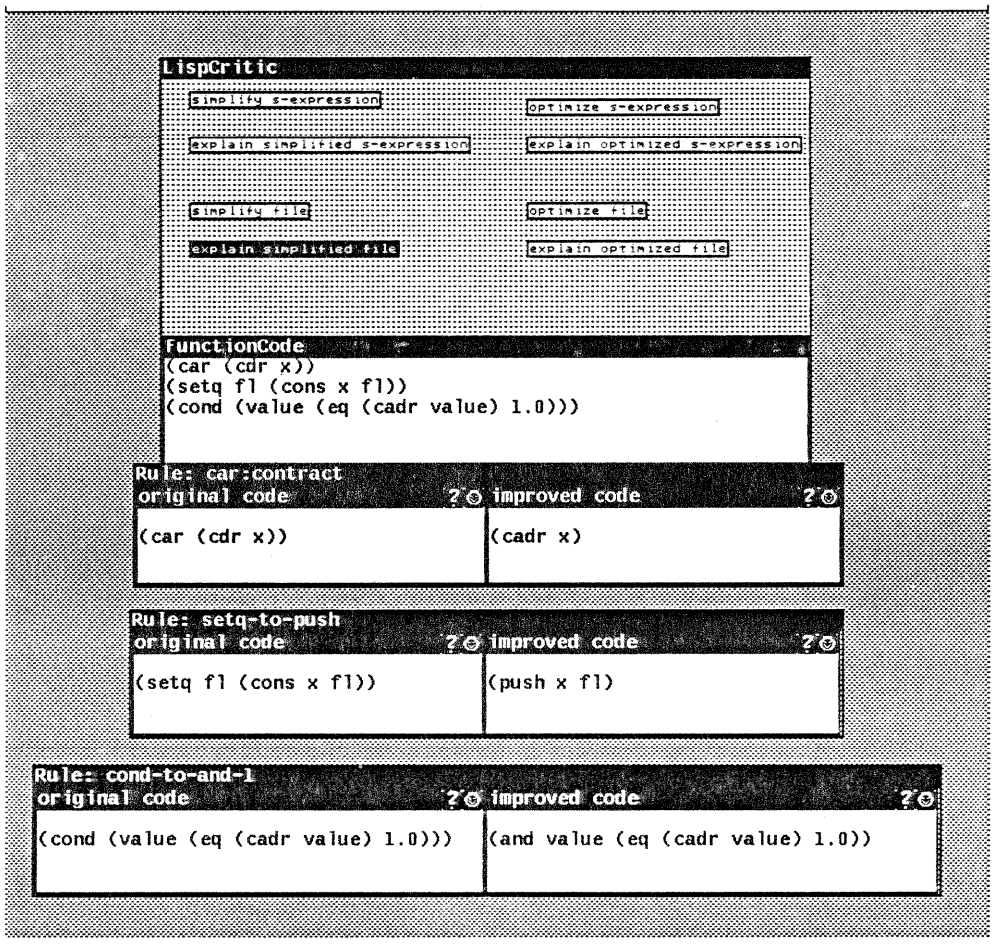

Figure 5-2: The LISPCRITIC

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 LISPCRITIC works on. The other three windows show some of the transformations carried out on the program. The "?" 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.

The system is used by two user groups who have different purposes. One group consists of intermediate users who want to learn how to produce better LISP code. We have tested the usefulness of the LISPCRITIC for this purpose by gathering statistical data on the programs written by students in an intro-

ductory 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 LISPCRITIC 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.

The LISPCRITIC 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 5-3);
- replace garbage-generating expressions by non-copying expressions (e.g., `(append (explode word) chars)` may be replaced by `(nconc (explode word) chars)`; see Figure 5-5);
- 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)`).

The Architecture of the LISPCRITIC. 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 LISPCRITIC operates by applying a large set of transformation rules that describe how to improve code. Figure 5-3 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 5-4. 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 5-2) that is used together with conceptual knowledge structures about LISP to generate explanations (see Figure 5-6). The user model (for a more detailed discussion see [Fischer 88]) 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.

Support for Understanding the Criticism. Our experience with the LISPCRITIC in our LISP courses has shown that the given criticism is often not understood. Therefore we use additional system components to illustrate and explain the LISPCRITIC'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 5-5, 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 second rule in Figure 5-3), whereas the transformation

Transform a `cond` into an `and`

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

Example (see Figures 5-2 and 5-6):

```
(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 append1}                            ;;; the original code
      (restrict ?expr                                ;;; condition
         (cons-cell-generating-expr expr))           ;;; (rule can only be applied
                                                     ;;; 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
```

Example (see Figure 5-5):

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

Figure 5-3: Examples of Rules in the LISPCRITIC

`(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 5-6). 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 LISPCRITIC, and they should also serve as a starting point for a goal-directed "Did you know (DYK)" mode of learning.

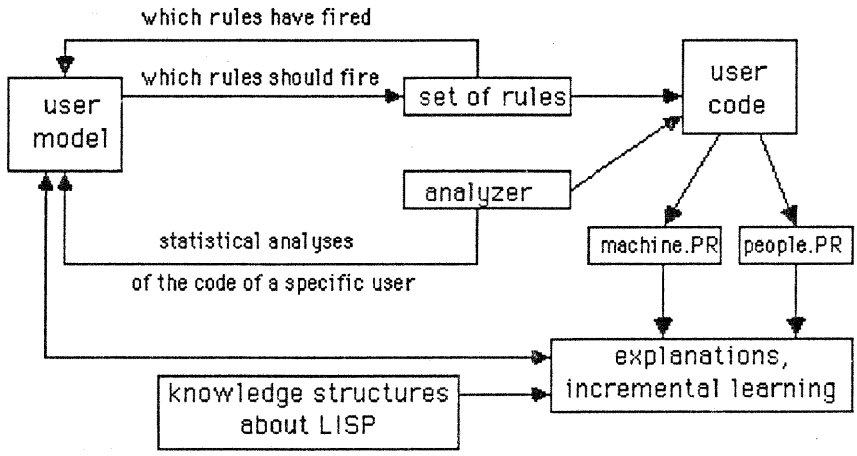

Figure 5-4: The Architecture of the LISPCRITIC

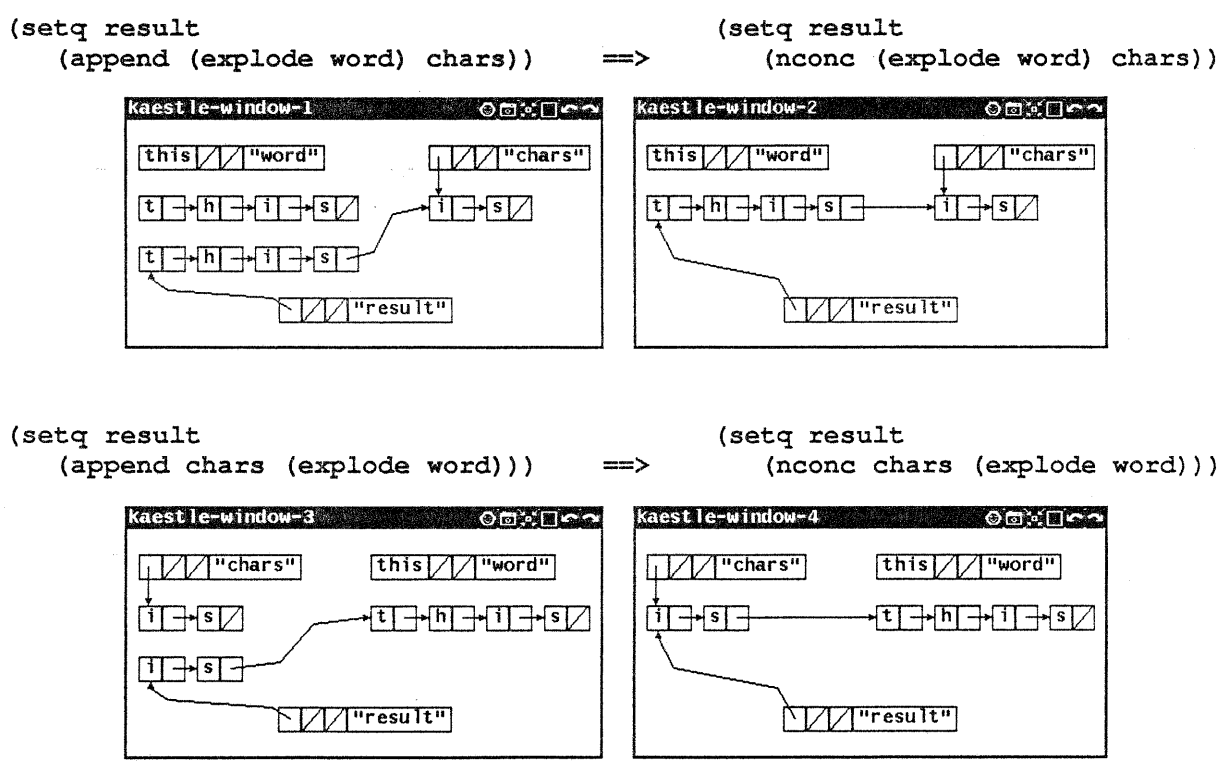

Figure 5-5: Illustration of the Validity of a Rule Using KAESTLE

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)**.

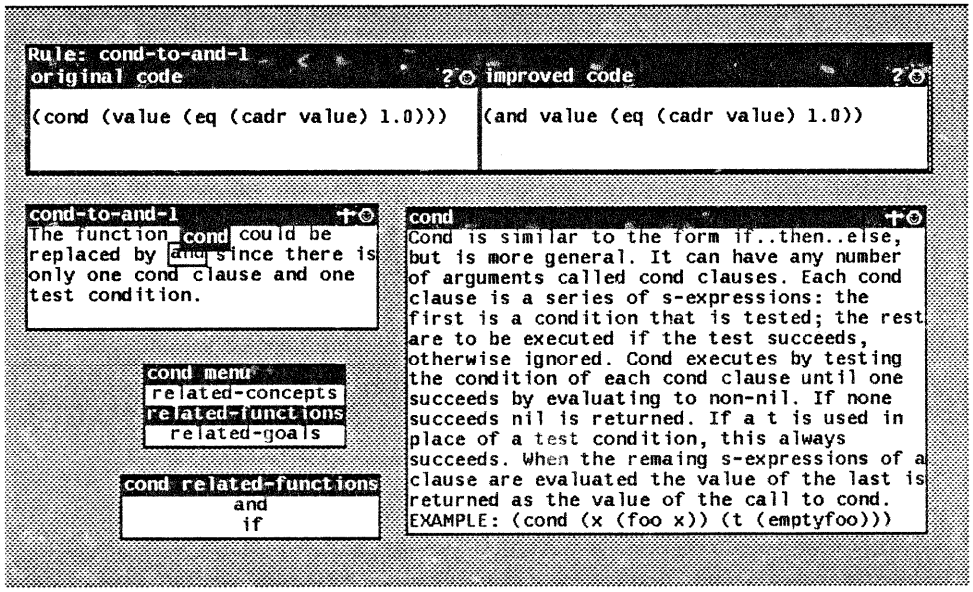

Figure 5-6: The User Browses Through the Knowledge Base

6. Conclusions

Our research effort has led to both a better understanding of the concept of incremental learning processes and to the design, implementation, and analysis of knowledge-based support systems for them.

The concept of the learning space of a system describes the structure of the knowledge and skills relevant to using the system. Analyzing several existing computer systems we found diverse learning spaces. Some complex systems, such as WLISP without the WLISPRC SHEET, showed by themselves relatively unstructured spaces not amenable to partitioning into microworlds. One possible explanation for this fact is that these systems are not based on abstractions well suited to the particular problem domains. By restructuring WLISP using the WLISPRC SHEET, a microworld structure could be achieved that provides a low threshold for beginners who could easily customize their systems. At the same time, the generality ("high ceiling") of the system could be preserved, and more knowledgable users can continue to adapt the system in more sophisticated ways.

The learning spaces of other systems (e.g., WLISP with its enhancements and FRANZLISP), can be structured according to the lettuce model of microworlds, which is able to explain the prerequisite relations existing between areas of a learning space.

In the LISP domain, we have found a relatively small kernel microworld which is being used by everyone. Even large programs written in FRANZLISP use only a relatively small percentage of all available functions. This suggests that a focus on this kernel is possible and will give the user a considerable expressive power which occasionally must be supplemented with special purpose knowledge.

Reducing and enhancing learning processes are two equally important goals. We have developed a classification of support systems for these goals. Our critics and design environments are such systems. They can bring knowledge about the structure of learning spaces to bear on a broad class of users including experienced users. Comparing WIDES with the LISPCRITIC, it is evident that the more semantically constrained a domain the more support a system can give. LISP functions, the basic elements of the domain of LISP programming, are very general and can be combined freely to form a program. This is not true for the higher-level building blocks of the WIDES design environment. Based on their more constrained semantics, there are more chances to advise the user in their application.

We believe that the microworld model can lead to the design of improved computer systems. Our prototypical implementations can serve as models for future systems that are based on these ideas. In addition to the uses studied in this research, microworld models of learning spaces can serve as a base for modeling users. They give valuable information for inferring the user's knowledge and for supporting goal directed learning processes (learning on demand).

Although informational systems, such as WLISP, can be shaped much easier than other technical devices to fit the presented requirements to increase learnability, the microworld model is certainly valid in more general areas of design as well.

Appendix I. List of Publications in the Context of this Project

In the context of this project, the following papers were published:

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. 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.
3. G. Fischer, A.C. Lemke, C. Rathke: **From Design to Redesign**, Proceedings of the 9th International Conference on Software Engineering (Monterey, CA), IEEE Computer Society, Washington, D.C., March 1987, pp. 369-376.
4. 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, 1988.
5. G. Fischer, A.C. Lemke: **Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication**, Human-Computer Interaction, Vol. 3, No. 3, 1988.
6. G. Fischer: **Making Computers more Useful and more Usable**, Proceedings of the 2nd International Conference on Human-Computer Interaction (Honolulu, Hawaii), Elsevier Science Publishers, New York, August 1987.
7. G. Fischer: **Learning on Demand: Ways to Master Systems Incrementally**, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1987.

Acknowledgements

The authors would like to thank Heinz-Dieter Boecker, who developed many of the original ideas, the set of rules and the interpreter for the LISPCRITIC; Christopher Morel, who implemented the explanation capabilities; Bart Burns, who implemented the statistical analysis package; and Catherine Cormack, who analyzed user programs with the statistical package.

References

[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.

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

[Draper 84]
S.W. Draper, *The Nature of Expertise in UNIX*, Proceedings of INTERACT'84, IFIP Conference on Human-Computer Interaction, Elsevier Science Publishers, Amsterdam, September 1984, pp. 182-186.

[Ehrlich, Walker 87]
K. Ehrlich, J.H. Walker, *High Functionality, Information Retrieval, and the Document Examiner*, in G. Fischer, H. Nieper (eds.), *Personalized Intelligent Information Systems, Report on a Workshop (Breckenridge, CO)*, Institute of Cognitive Science, University of Colorado, Boulder, CO, Technical Report No. 87-9, 1987, Ch. 5.

[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.

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

[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, *Making Computers more Useful and more Usable*, Proceedings of the 2nd International Conference on Human-Computer Interaction (Honolulu, Hawaii), Elsevier Science Publishers, New York, August 1987.

[Fischer 88]
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.

[Fischer, Kintsch 86]
G. Fischer, W. Kintsch, *Theories, Methods and Tools for the Design of User-Centered Systems*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1986.

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

[Fischer, Lemke 88b]
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, 1988.

[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, Nieper 87]
G. Fischer, H. Nieper (eds.), *Personalized Intelligent Information Systems, Report on a Workshop (Breckenridge, CO)*, Institute of Cognitive Science, University of Colorado, Boulder, CO, Technical Report, No. 87-9, 1987.

[Nieper 85]
H. Nieper, *TRISTAN: A Generic Display and Editing System for Hierarchical Structures*, Technical Report, Department of Computer Science, University of Colorado, Boulder, CO, 1985.

[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.

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

[Smith et al. 83]
D.C. Smith, C. Irby, R. Kimball, B. Verplank, E. Harslem, *Designing the Star User Interface*, in P. Degano, E. Sandewall (eds.), *Integrated Interacitve Computing Systems*, North-Holland, Amsterdam - New York - Oxford, 1983, pp. 297-313.

[Wilensky 84]
R. Wilensky, *LISPcraft*, W.W. Norton & Company, New York - London, 1984.