UNIVERSITY OF COLORADO, BOULDER

Department of Computer Science

Enhancing Incremental Learning Processes with Knowledge-Based Systems

Gerhard Fischer
Department of Computer Science and Institute of Cognitive Science
University of Colorado
Boulder, Colorado 80309

To be published in: Heinz Mandl and Alan Lesgold (eds): "Learning Issues for Intelligent Tutoring Systems", Springer Verlag, New York, 1986 (i.P.)

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

1. Introduction

In the past, computer systems limited the user to modes of communication that made the machine's job easier. But now, as computer cycles become plentiful, our focus can shift to the users and how to make it easier, more productive and less frustrating for them to cope with complex systems. Empirical investigations show that on the average only a small fraction of the functionality of complex systems is used. Figure 1-1 summarizes data based on careful observations of persons using systems like UNIX, EMACS, SCRIBE, LISP etc. in our environment. It also describes different levels of system usage which typically can be found within many complex systems. The different domains correspond to the following:

D_1: the subset of concepts (and their associated commands) that the users know and use without any problems.

D_2: the subset of concepts which they use only occasionally. Users do not know details about them and they are not too sure about their effects. Descriptions of commands (e.g. in the form of property sheets), explanations, illustrations (see section 6.1) and safeguards (e.g. UNDOs) are important so that the user can gradually master this domain.

D_3: the mental model [Norman 82; Fischer 84] of the user, i.e. the set of concepts which he/she thinks exist in the system. A *passive help system* (see section 6.2) is necessary for the user to communicate his/her plans and intentions to the system.

D_4: represents the actual system. Passive help systems are of little use for the subset of D_4 which is not contained in D_3, because the user does not know about the existence of these system features. *Active help systems* (see section 6.2) and *Critics* (see 6.3) which advise and guide a user similar to a knowledgeable colleague or assistant are required so that the user can incrementally extend his/her knowledge to cover D_4.

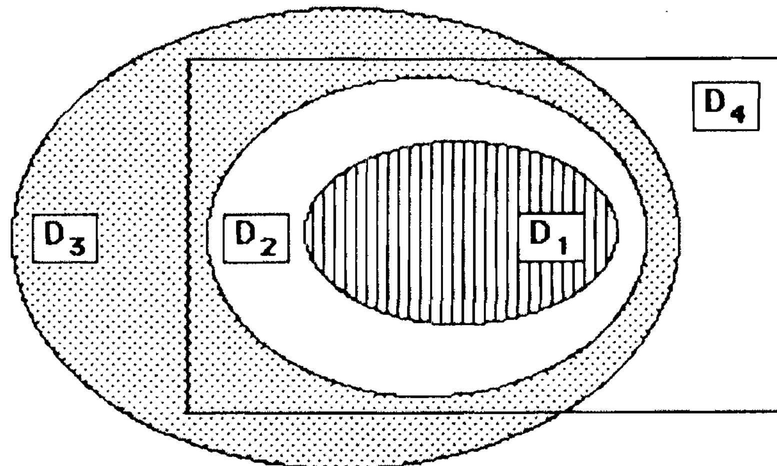

Figure 1-1: Levels of System Usage

As far as instructional strategies are concerned it is important to note that only D_3 can be learned by using the methods of free exploration, whereas D_4 requires some guiding and coaching. The system's model of the user (see section 5.2) will be used to determine the domains D_1 - D_4 for an individual user.

2. General Principles for Enhancing Incremental Learning Processes with Knowledge-Based Systems

2.1 The Paradigm of "Increasingly Complex Microworlds"

Over the last several years we have developed a general paradigm for instruction which is best described as a sequence of "Increasingly Complex Microworlds (ICM)" [Fischer, Burton, Brown 78; Fischer 81; Burton, Brown, Fischer 83].

The ICM paradigm was developed to capture instructional processes for complex skills which are difficult to learn because the starting state and goal state are too far apart. The student is exposed to a sequence of increasingly complex microworlds, which provide stepping stones and intermediate levels of expertise so that within each level the student can see a challenging but attainable goal. Increasingly complex microworlds can also be used to provide protective shields for the novice that prevent him/her from being dumped into unfamiliar system areas (see Figure 2-1). The paradigm requires a precise representation of the knowledge that is learned in a specific microworld and how to choose the next microworld. It serves well as a model to capture the essence of incremental learning processes.

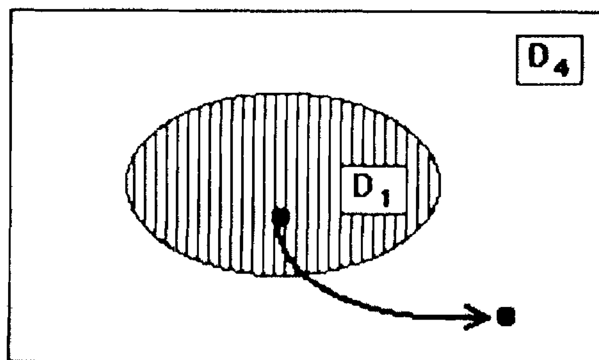

Figure 2-1: Protective Shields

In our application (the incremental learning of *Lisp* skills; see section 4) the following microworlds can be defined:

- use of an interactive environments with multiple windows, icons, pointing devices;
- dynamic and associative memory structures (e.g. lists and property lists);
- applicative programming styles (e.g. little use of the assignment operators);
- destructive versus non-destructive functions;
- macros (which can be used to create more problem specific representations; they reduce the cognitive complexity, because the goal structure of the problem domain can be more directly mapped onto the program);
- use of existing tools (e.g. reader with real-time indenting, inspector, browser, trace and step package, *Kaestle*, *Code-Improver*, compiler etc.) ;
- use of existing building blocks in the construction of new programs (e.g. pattern matcher, window systems, packages to support an object-oriented style of programming, expert system shells).

In our future research we will extend the power of the *Code-Improver* (see section 6.3) so it can decide which microworlds are familiar to and have been mastered by individual learners. We will also develop a system component which will make suggestions to a student to move on to the next microworld.

The richness of powerful *Lisp* systems (similar to the great variety of different slopes in skiing; [Fischer, Burton, Brown 78]) will allow people to learn those parts of the system first that are of immediate relevance to their tasks. Higher level programming formalisms and technological improvements (e.g. a pattern matcher, an expert system shell, a first rate programming environment) have eliminated certain prerequisites to use computer systems successfully.

2.2 The Critical Issues

The three major goals of our research are:

1. to enhance incremental learning processes with knowledge-based systems and to get a deeper understanding of how people understand, learn and operate complex systems.

2. to apply and test our general framework by implementing system components which will support the instructional process which the normal computer science student or software engineer has to go through when *he/she learns to cope with complex Lisp systems*.

3. to empirically evaluate the effectiveness of the user support systems constructed both to assess their effectiveness and to discover possibilities for improvement.

To achieve these goals we are investigating:

1. How can complex systems be constructed so that they have *no threshold and no ceiling*? It should be easy to get started (i.e. microworlds should provide *entry points*), but these systems should also offer a rich functionality for experienced users.

2. How can our theoretical paradigm of constructing *increasingly complex microworlds* be exploited to build complex systems that support incremental learning strategies?

3. What are the general principles that determine the right mixture of *free exploration* and

coaching? How can we guarantee that systems take the initiative when necessary and at the same time are non-intrusive?

4. How can we turn "non-constructive" bugs into "constructive" ones and develop a broad collection of *self-checking methods*?

5. How can we use *models of the user* to make systems more responsive to the needs of *different individual users* and how does the system behavior reflect the *transition of a user from a novice to an expert*?

6. How can *explanations* be tailored to the user's conceptualization of the task?

7. What is the role and relative importance of *verbal* and *nonverbal* (e.g. graphical) explanatory material? When should one be used in favor of the other?

8. How can we *evaluate* these systems?

We claim that knowledge-based systems with qualitatively new human-computer communication capabilities (see section 5) are one of the most promising ways to achieve our stated goal. We propose to extend the comprehensibility of systems by using a large fraction of the computational power of the machine to support sophisticated user support systems (see Figure 2-2).

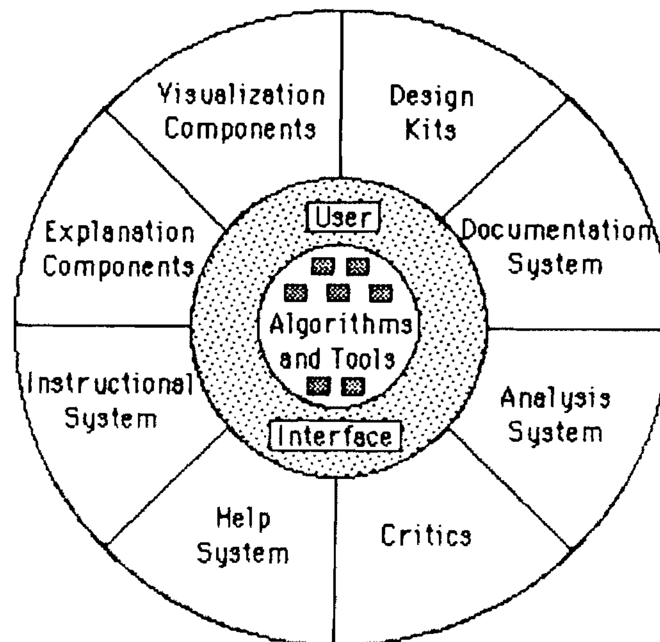

Figure 2-2: From Tools to Communication Partners

In the past our efforts have been concentrated on the incremental learning of *computer systems*, because they allow us to exploit the information structures already present in the machine and they do not require maintaining consistency between an outside world and a model of this world in the computer. In addition this problem domain is ideally suited for our purposes: it occurs in our daily work; a student pool for testing and evaluation is readily available; there is a growing interest in *Lisp*-based systems and a serious shortage of appropriately trained persons.

However, we would like to emphasize that our concerns are of more general nature. Computer systems are used as *vehicles* only to carry out the work; the more general goals of our research are to develop theories and construct experimental systems which make a true contribution towards an improvement of instructional theories and processes in general.

3. Instructional Strategies

Our instructional strategies are oriented towards the *intermediate* user who has mastered a system to some extent (see Figure 1-1). This implies that it is not sufficient to rely solely on tutorial strategies but we have to develop several ways to support a user:

1. a critic should "look" over his/her work and make suggestions for improvements;

2. a tutoring component (using some of our visualization techniques) can illustrate the advice given;

3. an explanation component can provide the rationale that leads to the improvement.

In our research we want to determine the balance between supporting an exploratory learning style of *learning-by-doing* (the basic philosophy behind the interest worlds in LOGO environments; [Papert 80]) and a guided learning experience through *coaching assistance* (the primary instructional strategy supported by systems in intelligent computer-assisted instruction; [Sleeman, Brown 82]).

Learning-by-doing is tightly coupled with *learning-on-demand*. Users are often unwilling to learn more about a system or a tool than is necessary for the immediate solution of their current problem. To be able to successfully cope with new problems as they arise requires a consultant which generates advice tailored to the user's specific need. This approach provides information only in relevant situations and eliminates the burden to learn a lot of things in advance (i.e. at times when it is unknown if the information ever will be used or when it is difficult for the learner to imagine an application).

To get a deeper understanding of how to support *learning-on-demand* our work on *knowledge-based help systems* (see section 6.2) serves as a starting point. We have to be able not only to respond to errors but

to notice -- based on a model of the task and a model of the individual user -- **suboptimal actions** of the user which serve as a basis for individual help. It is not always clear when a solution is suboptimal and hence should trigger an activity of the help system. A **metric** is necessary to judge how adequate a user's action is. Except for narrow problem domains (e.g. simple games [Burton, Brown 82]), optimal behavior cannot be uniquely defined.

We will use the identification of a user with a specific microworld to prevent the system from using inadequate conceptualizations and knowledge structures which may lead to the following difficulties:

- the help offered may not be understood by the learner because it refers to unknown concepts;
- the help offered may put too heavy a load on the learners short term memory;
- the help system "forces" the user to do something which he/she does not want to do. A possible solution to this problem might be to make the metric visible and to allow the user to change it; but we must be aware that this increases the control of the user as well as the complexity of the system. This increase in functionality will be of little use if we do not find adequate communication structures for it.

Our approach towards the construction of instructional systems can be contrasted with some other approaches:

1. **The LISP Tutor:** Anderson and his research group [Anderson et al. 84] address similar issues but their work is oriented towards *tutoring* which allows them to pursue a *predefined* course of action, whereas our actions are triggered by the behavior and the incremental learning process of individual users. Their system deals with the early phase of a skill acquisition process in which it is important to provide guidance and correct the acquisition of "wrong" behavior as early as possible.

2. **The PROUST system:** Johnson and Soloway [Johnson, Soloway 84] provide systems with a *deep* understanding of programs and misconceptions of users about them. They look in great detail at very specific programs which they use as case studies in their work. Concentrating on a *very small* number of examples they are able to create a very elaborate ("deep") representation of these objects. Their system is not able to take a basically arbitrary *Lisp* program and criticize it analogous to our *Lisp-Critic.*

3. **The West system:** Brown's and Burton's work on the *WEST* system [Burton, Brown 82] as well as our joint work [Fischer, Burton, Brown 78; Fischer 81; Burton, Brown, Fischer 83] have been very influential to the research described in this paper. Contrary to their approach in "West", where one can rely on the evaluation of an arithmetic expression to determine a metric for "optimal" and "suboptimal" behavior, we have to model much more complex skills (e.g. how to write "good" *Lisp* code) which requires hundreds of rules (see section 6.3).

The following requirements can be derived from our desire to support the instructional strategies described:

- the system must be able to support users on *all* levels of expertise (novices to experts);

- it is *not restricted to a tutoring system* although tutoring aspects can be found.
- it supports *learning on demand*; interesting new topics are introduced into the instructional process when there is a need for them; it should *take the initiative* when weaknesses of the user become obvious; not every recognized suboptimal action should lead to an intervention.
- it ought to have *explanation capabilities*, because even the best advice is sometimes not understood; it should *give additional information* which was not explicitly asked for but which is likely to be needed in the near future.
- it needs to have a model of its communication partner in order to be able to tailor its advice and explanations to the level of expertise of the user; *knowing what the user knows* the system may be able to make predictions on the kind of problems the user is likely to encounter when tackling specific tasks.
- it will be able to provide advice relating to the domains D_3 and D_4 in Figure 1-1; it will assist the user in the *stepwise extension* of his/her view of the system by making sure that basic concepts are well understood and by not introducing too many new features at once.
- it should be *be non-intrusive*; only frequent suboptimal behavior without the user being aware of it should trigger an action of the system.

4. The Application Domain: Enhancement of LISP Skills

Our system building efforts have the goal to build a *Lisp-Critic*. This system may be best thought of as a knowledgeable colleague, a consultant or an advisor of the programmer. It should be relevant in the standard situation that occurs among humans: some person is working on a program, somebody else enters the room and both persons start a dialogue on the program that is under development.

4.1 Rationale for the Selection of the Problem Domain

The rationale for the selection of this problem domain is based on the following:

1. **Relevance.** There is a *great need* to train people to use complex *Lisp* systems which form the basis for the new tools of the 5th generation computer systems; there are currently very few people around who can successfully exploit the computational power and broad functionality of high-performance personal work stations.

2. **Complexity.** The systems used are sufficiently *complex that nobody masters them completely*; incremental learning processes are a must.

3. **Expertise.** To train students and computer scientists has been our task for several years; we have a *good understanding of the problems* which they encounter and we have a large group of people around us to test our ideas and systems in a natural setting.

4. **Appropriateness.** We have enough *control* over these systems (because we built most of them ourselves) to augment them with knowledge structures (e.g. models of the task, models of the users, visualization techniques) which are required to enhance incremental learning.

4.2 A Scenario

The following dialogue exemplifies some of the features that the system will have. It shows the functionality only, it does *not* show the user interface. We use a natural language dialogue to illustrate the main ideas; this form of interaction will be replaced or enhanced in the future by an interaction style which is more oriented towards direct manipulation (using windows, menus, icons, mouse, etc.). The system will not necessarily be used exclusively interactively; it may also be used in a batch mode, writing comments and suggestions for improvement into a separate file. One alternative scenario is that before users log off the system in the evening, they ask the system to look over the work during the night.

The following dialogue assumes that the user is not using the system the first time; from earlier sessions the system has incrementally built up a model of the user that reflects his/her knowledge state, preferred style of programming etc.. We will use **this font** to print the **input of the user**, *another font* for the *output of the computer*, and a third one to comment on both. Code will be printed in a smaller **bold** or *italic* font.

Dialogue	Comments
.	User types in some piece of new code.

```
.
.
(prog (line)
    (setq line (append (explode word) chars))
 loop
    (cond ((null line) (return (terpri))))
    (princ (car line))
    (princ " ")
    (setq line (cdr line))
    (go loop))
.
.
```

Dialogue	Comments
I have some suggestions on how to improve your code, want to see them?	System steps in actively; ACTIVIST aspect.

Yes

The code

```
(prog (line)
    (setq line (append (explode word) chars))
 loop
    (cond ((null line) (return (terpri))))
    (princ (car line))
    (princ " ")
    (setq line (cdr line))
    (go loop))
```

could be rewritten by

- *using the* ***msg****-macro and*
- *replacing the* ***prog****-form by a* ***do****-form*

as:

```
(do ((line (append (explode word) chars)
        (cdr line)))
    ((null line) (terpri))
    (msg (car line) B))
```

System suggests using a macro and a higher level looping control structure than prog by using transformation rules contained in its domain knowledge base.

What does *msg* do?

User asks for explanation of a function. PASSIVIST aspect.

msg is a macro that expands into more primitive print functions; "B" will print a blank; other arguments will be printed by 'princ'.

System generates explanation of "msg". The underlined words may be explained further. Note, that the system does not print the whole manual entry for "msg" but instead gives the information on the "B" keyword only, which is relevant to the specific suggested improvement.

What is a macro?

User asks for the explanation of a concept.

A macro is ...

System prints out explanation tailored to the knowledge state of the user.

Give me another example of a macro

User asks for more examples of macros.

The Code

```
(setq line (cdr line))
```

could be rewritten as:

```
(unpush line)
```

System prints out another example generated from other portions of the user's code if possible.

4.3 A Conceptual Hierarchy of Lisp Knowledge

The main question to be answered in order to provide intelligent help to a programmer is: what does the system need to know to fulfill its job? What is the knowledge that the system can draw upon and how is it structured?

We are currently restricting ourselves to build a program that can assist the programmer up to the level of *application-independent knowledge* about programming; we do not try to model the knowledge that is specific to certain algorithms. The system also does not have an understanding of higher-level programming constructs like loops, hashtables etc. at the level of Rich's and Shrobe's *Programmer's Apprentice* [Rich, Shrobe 78]. Our approach can be compared with the *computer aids to writers* in the UNIX world [Cherry 81].

We formalize the programming knowledge using ***Concepts***, ***Goals***, and ***Functions*** as the basic representational entities. We are using *ObjTalk* (an object-oriented knowledge representation formalism; [Lemke 85; Rathke, Lemke 85; Rathke 86]) to represent this knowledge because it offers the right kind of descriptive power. Some basic knowledge types can be defined as classes in the following way:

Concepts. A *Concept* is an entity that is primitive in the sense that no action is associated with it; they are used to tie the procedural knowledge together that is embodied in the *Goals*.

```
(ask class new: Concept
     (superc class)
     (descr (SuperConcept (default 'Lisp-Knowledge))
            (ContraryConcept)
            (RelatedConcept)))
```

Goals. *Goals* specify normative categories for the Lisp programmer. *Goals* embody procedural knowledge that can be applied to produce program code. *Goals* formulate the heuristics and higher order principles that may be followed to produce good software. There may be several layers of *goals*, i.e. goals may have subgoals. *Goals* will be used to represent individual *rules* that can be used to classify and transform program code.

```
(ask class new: Goal
     (superc class)
     (descr (RelatedConcept)
            (SuperGoal)
            (RelatedGoal)
            (ConflictingGoal)))
```

Functions. *Functions* capture the knowledge that is closely associated with the actual writing of Lisp code.

If the system's model of the user had been different, e.g. if the user were a more advanced person already knowing about macros and do-forms, the interaction might have been different:

I have some suggestions on how to improve your code, want to see them?

System steps in actively; ACTIVIST aspect.

Yes

The code

```
(prog (line)
    (setq line (append (explode word) chars))
 loop
    (cond ((null line) (return (terpri))))
    (princ (car line))
    (princ " ")
    (setq line (cdr line))
    (go loop))
```

could be rewritten by

- *using the* ***msg****-macro,*
- *replacing the* ***prog****-form by a* ***mapc****-form and*
- *using a destructive function (nconc) instead a copying one (append).*

as:

```
(mapc '(lambda (char) (msg char B))
        (nconc (explode word) chars))
```

Note, that the system was able to generate a reasonable new name for the parameter of the lambda-expression. This will be done through reference to other parts of the code or by having some knowledge about the users naming style.

Why replace *append* by *nconc*?

nconc *is more efficient, because it does not copy list structures.*

System justifies its suggestion by referring to higher level concepts.

Please explain!

You may start the Kaestle system and execute

```
(append (explode 'word) '(c h a r s))
```

and

```
(nconc (explode 'word) '(c h a r s))
```

within the Kaestle-window or run the "nconc-append demo".

System suggests two approaches; the first involving more activity on the side of the user, the second being a prefabricated demonstration (that may use the current bindings of "chars" and "word"). The system could have also chosen a textual explanation by presenting the relevant manual information.

Show the demo

The system runs the demo which shows on the level of the internal representation of list structures how these are copied and modified by the two functions append and nconc, respectively (see Figure 4-1).

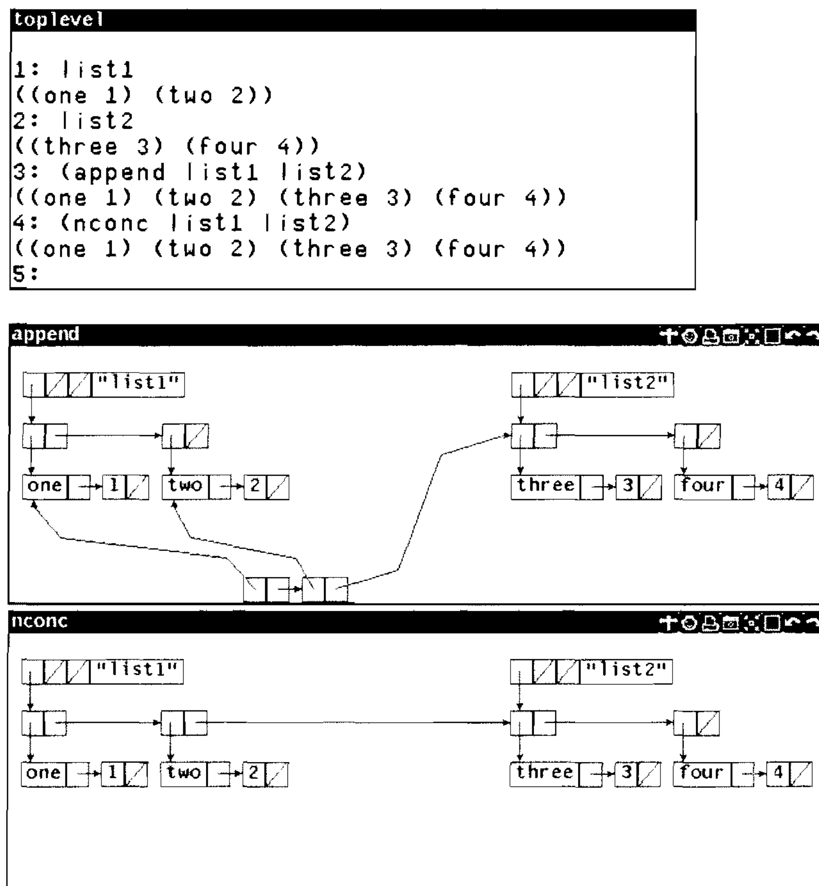

Figure 4-1: Illustration of append versus nconc

Our visualization tool *Kaestle* (see Section 6.1) illustrates the difference between the copying operation append (new cons-cells are used) and the destructive operation nconc (the list 'list1' is modified). Both function seem to produce the same result (as seen in the window 'toplevel'), but nconc causes a side effect which is undesirable in many situations.

```
(ask class new: Function
     (superc class)
     (descr (SuperConcepts)
            (Pattern)
            (Subparts)
            (RelatedFunctions)
            (SubConcepts)
            (RequiredConcepts)
            (Critics)
            (Specializers)
            (Generalizers)
            (Optimizers)))
```

The knowledge base will consist of a large number of instances of these primitive representational units. Some examples of instantiations of the classes are:

- *Concepts*

```
(ask Concept new: Lisp-Knowledge (SuperConcept Programming-Knowledge))
(ask Concept new: Control-Structures (SuperConcept Programming-Knowledge))
(ask Concept new: Data-Structures (SuperConcept Programming-Knowledge))
(ask Concept new: Readability)
(ask Concept new: Macros (SuperConcept Readability Speed-Efficiency))
```

- *Goals*

```
(ask Goal new: Avoid-Creating-Garbage
     (RelatedConcept Garbage-Collection))
(ask Goal new: Avoid-Multiple-Evaluation-Of-Identical-Expression
     (RelatedConcept Speed-Efficiency))
(ask Goal new: Do-Not-Copy-Repeatedly
     (RelatedGoal Avoid-Creating-Intermediate-ConsCells)
     (descr
       (Pattern) (Consequence) (Modifier) (Type)))
```

- Figure 4-2 shows a typical *ObjTalk* representation of expert knowledge concerning the *Lisp* function *do*.

The slots in this structure can be described as follows:

- *SuperConcepts:* a set of more general concepts which are instantiated through this function.
- *Pattern:* the *Lisp* syntax of the function. It can be used to recognize it from some piece of program code and conversely to generate code from an abstract description.
- *SubParts:* lists of primitive blocks needed to build up a complete function.
- *RelatedFunctions:* functions that can be used to achieve similar functionality. They may be more special or more general.
- *SubConcepts:* a list of more special uses of the function exploiting a certain part of the whole functionality. The example `DoWithMultipleLoopVars` addresses the issue that more than one loop variable can be used.
- *RequiredConcepts:* a link to concepts that are needed to understand how the function is working (i.e. links to previous microworlds).
- *Critics:* patterns of program code which are clues to missing or incorrect concepts. Often these patterns are undetected by the *Lisp* runtime system and the compiler because they are

```
(ask Function renew: Do        ;;; expert knowledge about the do function
     (SuperConcepts            ;;; links to higher level concepts
       Iteration ParallelEvaluation LambdaBinding SpecialForms
       SequentialProgramming)
     (Pattern                  ;;; syntax of the function
       (do (?*DoDeclaration) ?DoExitClause ?*ProgBody))
     (SubParts
       DoDeclaration DoExitClause ProgBody)
     (RelatedFunctions         ;;; do can be transformed to these functions
       SimpleDo Prog MapFunctions)
     (SubConcepts
       DoWithMultipleLoopVars DoWithLocalVars DoWithUninitializedVars
       DoWithMoreThan1Var DoWithNoDoVar DoWithCondExitIdentical
       DoWithCondExit DoWithCondSomethingExit DoWithNilExitClause)
     (RequiredConcepts
       ProgBody UninitializedVarsAreNil ParallelAssignment
       LeftToRightEvaluationSequence)
     (Critics                  ;;; recognize incorrect syntax
       MultipleDeclaredVariables IgnoredParallelAssignment MalformedExitClause
       MalformedDeclaration)
     (Specializers
       Do2Mapcar Do2Mapc DoWithTconc2Mapcar-1 DoWithTconc2Mapcar-2 Do2SimpleDo Do2Let)
     (Generalizers
       Prog2Do-1)
     (Simplifiers)
     (Optimizers))

;;; This special form of do reflects the usage of more than one loop variable
;;; It is used to recognize the use of this functionality.
(ask Function renew: DoWithMultipleLoopVars
     (SuperConcepts Do)
     (Pattern
       (do (?*vars1 (?v1 ?i1 ?r1) ?*vars2 (?v2 ?i2 ?r3) ?*vars3)
             ?test ?*body))
     (UsedIn
       Do2Mapcar DoWithTconc2Mapcar-1 DoWithTconc2Mapcar-2)
     (Examples ...)
     (Rating))

;;; Transformation rule from prog to do and vice versa.
(ask CriticRule renew: Prog2Do-1
     (FromFunction Prog)
     (ToFunction Do)
     (FromPattern (prog (?var) (setq ?var ?init) ?label:symbolp
                    (cond (?test (return ?result)))
                    ?*body (setq ?var ?rep) (go ?label)))
     (ToPattern (do ((?var ?init ?rep)) (?test ?result) ?*body)))

;;; Transformation rule: do <--> mapc
(ask CriticRule renew: Do2Mapc
     (FromFunction Do)
     (ToFunction Mapc)
     (FromPattern (do ((?var ?init (cdr ?var))) ((null ?var) ?*result) ?*body))
     (ToPattern (mapc '(lambda (?elem) ?*(replace: ?*body (car ?var) ?elem))
                   ?init) ?*result)
```

Figure 4-2: Representation of the "do" Function

not explicitly erroneous. Examples are unreached pieces of code, code which computes a constant value, code which runs only interpretively etc..

- *Specializers*, *Generalizers*, *Simplifiers* and *Optimizers:* transformation rules which make the knowledge base operational. They allow the transition from a concept to a related concept and can also be used to generate explanations for this relation. The Prog2Do-1 generalizer connects this function to the related prog function and the Do2Mapc specializer provides the link to one of the higher level "map"-functions.

The network of *concepts*, *goals*, and *functions* serves a variety of purposes; it may be used

- to make the relations between different parts of the programming knowledge explicit;
- to justify and thereby explain the improvements suggested by the rules (see section 6.3);
- to enable the student to actively browse through the conceptual space (e.g. to learn about similar concepts) of the application domain;
- to restrict the domain of tutoring to parts of the conceptual knowledge; i.e. criticize and advice relative to a subset of concepts and goals.
- to derive a model of the user (see section 5.2).

5. Architecture of a System to Support Incremental Learning Processes

5.1 Knowledge-Based Human-Computer Communication

Knowledge-Based systems are one promising approach to equip machines with some human communication capabilities. Based on an analysis of human communication processes we have developed the model shown in Figure 5-1.

The system architecture in Figure 5-1 contains two major improvements over traditional approaches:

- the **explicit** communication channel is widened (e.g. we use windows, menus, pointing devices, etc.) and
- information can be exchanged over the **implicit** communication channel.

The four domains of knowledge shown in Figure 5-1 have the following relevance:

1. **Knowledge of the problem domain** (see section 4): Intelligent behavior builds upon large amounts of knowledge about specific domains. This knowledge imposes constraints on the number of possible actions and describes reasonable goals and operations. If, for example, in UNIX a user needs more disk space it is in general not an adequate help to advise him/her to use the command **rm** *[1] [Wilensky et al. 84] although it would perfectly serve his/her explicitly stated goal. The user's goals and intentions can be inferred if we understand the correspondence between the system's primitive operations and the concepts of the task domain.

[1] The command deletes all files in the current directory.

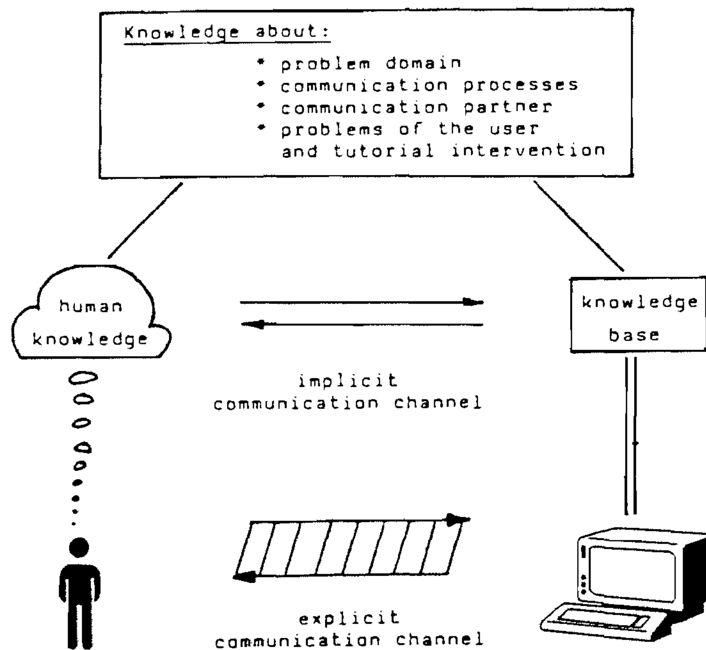

Figure 5-1: Architecture for Knowledge-Based Human-Computer Communication

2. **Knowledge about communication processes:** The information structures which control the communication should be made explicit, so the user can manipulate them.

3. **Knowledge about the communication partner** (see section 5.2): *The* user of a system does not exist; there are many different kinds of users, and the requirements of an individual user grow with experience.

4. **Knowledge about the most common problems which users have in using a system and about instructional strategies** (see section 3): This kind of knowledge is required if someone wants to be a good coach or teacher and not only an expert; a user support system should know when to interrupt a user. It must incorporate instructional strategies which are based on pedagogical theories, exploiting the knowledge contained in the system's model of the user.

5.2 The Use of Models to Support Incremental Learning Processes

To support incremental learning processes and learning-on-demand the system must be able to represent for specific users information about the user's conceptual understanding of a system, the user's individual set of tasks for which he/she uses the system, the user's way of accomplishing domain specific tasks, the pieces of advice given and whether the user remembered and accepted them and the situations in which the user asked for help.

The domain chosen for this research is much more complex than the simple editing tasks considered in the *Activist* and *Passivist* systems (see section 6.2) or a simple game like the WEST system [Burton, Brown 82]. While parametric models (i.e. ratings of the user in a small set of concepts) or overlay models, in the form of a one-to-one association of tasks and (expert) solutions, are sufficient for simple tasks, a network consisting of *programming concepts*, *Lisp functions*, *data types*, and *transformation rules* must be developed to represent both the expert knowledge and the incomplete, suboptimal, and partly erroneous knowledge of the user.

The representation of the user's knowledge and skills is done as a modified subset of the system's expert knowledge. Compared to the system's expert knowledge, some of the concepts and transformation rules may be missing. Others may be present but incomplete or incorrect. In addition, there may be erroneous subconcepts which were detected in code written by the user. An example of a user's representation of the *DO*-function (as hypothesized by the system) is given in Figure 5-2.

```
(ask Function renew: Do
     (SuperConcepts          ;;; fewer known superconcepts
       Iteration SequentialProgramming)
     (Pattern                ;;; less general form
       (do (?*DoDeclaration) (?test ?result) ?*LambdaBody))
     (SubParts               ;;; there is a lambda body instead of a prog body
       DoDeclaration DoExitClause LambdaBody)
     (RelatedFunctions       ;;; many related functions are unknown or
       Prog)                 ;;;  not recognized as being related
     (SubConcepts
       DoWithMultipleLoopVars DoWithMoreThan1Var
       DoWithCondExit DoWithCondSomethingExit)
     (RequiredConcepts
       LeftToRightEvaluationSequence)
     (Critics
       MultipleDeclaredVariables IgnoredParallelAssignment MalformedExitClause
       MalformedDeclaration)
     (Specializers)          ;;; no transformation rules available
     (Generalizers)
     (Simplifiers)
     (Optimizers))
```

Figure 5-2: A User's Representation of the "do" Function

Given this detailed model of the user and the system's understanding of *Lisp* knowledge, the following actions of the system become possible:

- *Select appropriate actions with respect to the user:* If the comparison of the user model and the system's expert knowledge reveals weaknesses in a specific area, the system should only become active if this area is adjacent (in terms of the ICM-paradigm) to already known areas and does not require too many other areas unknown to the user.

- *Select examples from the domain the user is familiar with:* By using an executable form of

representation it is possible to generate illustrations out of areas which the user already understands and thus reduce the cognitive distance that has to be bridged.

- *Present only the missing pieces of knowledge:* In dialogues a large amount of time is spent to find out what each communication partner knows and does not know about the subject area. Provided a detailed user model the system can concentrate on the very points where the user needs help.

- *Better understanding of the user:* Using knowledge about the user's understanding of a problem domain makes it much easier to find out about his/her real problem. We encountered many cases where a user had a problem which originated in a wrong decomposition of a higher level problem. Using knowledge about the user it is possible to trace a problem back to its real roots.

The problem of knowledge acquisition for the user model will be solved primarily using program code written by the user. Techniques as described in section 6.3 will be extended from recognizing pieces of code which can be improved to both recognizing the goals of a piece of code as well as existing and missing concepts which led to its generation. Since only in very few cases a definitive assumption about the knowledge of the user can be made, it is important to have many clues which allow to make uncertain inferences when no specific evidence is available. Concepts grouped into a set of microworlds are such a clue. If, for example, the user has shown to know a certain concept and there is no information about any of the prerequisite concepts then it is very likely that he/she also knows these concepts to some extent.

6. Prototypical System Components to Enhance Incremental Learning Processes

In this section visualization techniques, knowledge-based help systems and the *Code-Improver* system will be briefly described. These system components were developed over the last several years, have been in active use for some time and will serve as important building blocks towards our goal to construct a *Critic* for *Lisp*.

6.1 Visualization Techniques

Our visualization tools were developed as extensions to the *FranzLisp* programming environment. All these tools display and visualize relationships among data and control structures that are otherwise invisible. They all together build a *software oscilloscope*, that is used in a similar way as the oscilloscope of the electrical engineer.

The most important data structure of *Lisp* is the *list*. With *Kaestle* [Boecker, Nieper 85; Boecker, Fis-

cher, Nieper 86] the graphic representation of a list structure is generated automatically and can be edited directly with a pointing device. By editing we do not only mean changing the structure itself but changing the graphic representation, the layout, of the structure. *Kaestle* is integrated into a window system and multiple *Kaestle*-windows may be used at the same time. The *user interface* is menu-based (see Figure 6-1) and the *program interface* is realized through *ObjTalk* methods which can be triggered by sending messages to a *Kaestle*-window.

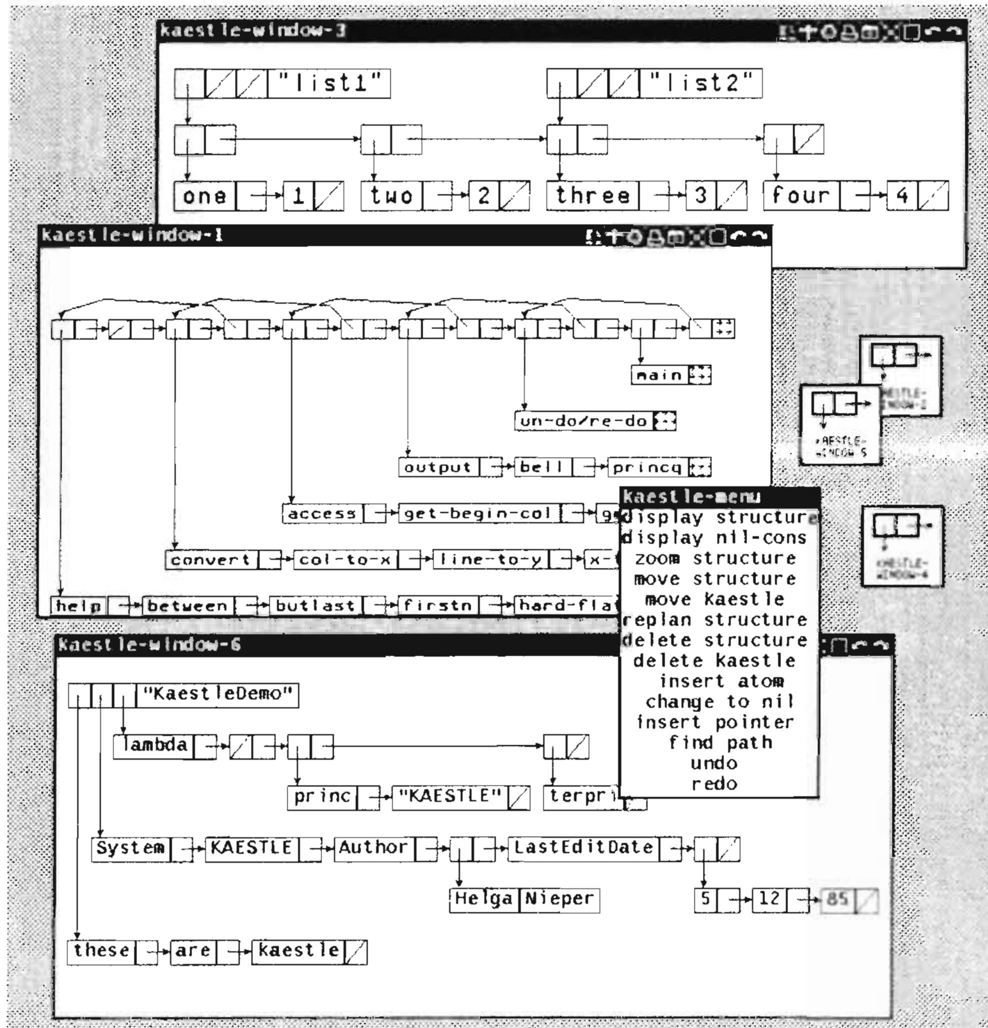

Figure 6-1: *Kaestle*: Visualization of Data Structures

The user of *Kaestle* may take one of the following roles:

1. An *active* role: A graphic representation can be generated from whatever the user types in and the user is encouraged to an exploratory style of learning.

2. A *passive* role: An inexperienced user does not know which structures and which operations on

them lead to interesting effects. To display prestored examples (or even examples taken from the actual context) *Kaestle* can be used through a program interface, i.e. programs can be written which generate graphic representations in a movie-like manner.

Our visualization tools can be used to support different instructional strategies. *Kaestle* supports the graphic display of ***data structures*** to answer questions like: What are the (list) structures that the system is currently working on? How do they change through the execution of programs?

Kaestle is a tool which can be used by the *Lisp-Critic* to illustrate explanations given by the system to answer questions like:

- What is the difference between several list creation functions (e.g. `cons` and `list`)?
- What is the difference between `equal` and `eq`?
- What is the difference between non-destructive and destructive functions (e.g. `append` and `nconc`[2])?
- Why is it possible to transform `(append (explode word) chars)` to `(nconc (explode word) chars)`?
- Why is it wrong to transform `(append chars (explode word))` to `(nconc chars (explode word))`?
- Why does `nconc` not work if the first argument is `nil`?
- How is a stack implemented in *Lisp*? What are `push` and `pop` doing?

We will adapt these and similar tools to help and explanation systems (see Figure 4-1), augmenting natural language by graphic- and movie-like capabilities [Boecker, Fischer, Nieper 86].

6.2 Knowledge-Based Help Systems

Our knowledge-based help systems (for details see [Fischer, Lemke, Schwab 84; Fischer, Lemke, Schwab 85]) have created some of the basic ideas towards our goal to support learning by demand. **PASSIVIST**, a passive, natural language based help system, is implemented in OPS5 [Brownston et al. 85]. Flexible parsing using OPS5 is achieved by a rule-based bottom-up method. The consistent structure of the system as a set of productions and a common working memory allows the use of the same knowledge in several stages of the solution process. It uses a help strategy in which each step of the solution is presented and explained to the user who then executes this step and immediately sees the resulting effects. Help is given as text generated from sentence patterns according to the goal structure of the problem solving process and key sequences and subgoals are displayed graphically.

ACTIVIST, an active help system for an EMACS-like editor, is implemented in *FranzLisp* and *ObjTalk*. *Activist* deals with two different kinds of suboptimal behavior:

1. the user does not know a complex command and uses suboptimal commands to reach a goal (e.g. he/she deletes a string character by character instead of word by word).

[2] see the figure in the scenario in section 4.2

2. the user knows the complex command but does not use the minimal key sequence to issue the command (e.g. he/she types the command name instead of hitting the corresponding function key).

Similar to a human observer, *Activist* handles the following tasks:

- to *recognize* what the user is doing or wants to do.
- to *evaluate* how the user tries to achieve his/her goal.
- to *construct a model of the user* based on the results of the evaluation task.
- to decide (dependent on the information in the model) *when* and *how* to interrupt (tutorial intervention).

In *Activist* the recognition and evaluation task is delegated to 20 different *plan specialists* (Figure 6-2).

```
DELETE left part of word
U S E R   M O D E L

plan executed:                      2
well done:                          1
wrong command used:                 1
with unnescessary keys:             4
command with wrong keys used:       0
with unnescessary keys:             0
messages sent to user:              0

I N T E R N A L   I N F O R M A T I O N

proposed commands:  rubout-word-left
optimal keys:       ESC h

commands:
keys:
automaton in state: Start
```

Figure 6-2: A Detailed View of one Plan Specialist

Each plan specialist recognizes and evaluates one possible plan of the problem domain. Such plans are for example *"deletion of the next word"*, *"positioning to the end of line"*, etc.. A plan specialist consists of:

1. A transition network, which matches all the different ways to achieve the plan using the functionality of the editor. Each transition network in the system is independent. The results of a match are the *used editor commands* and the *used keys* to trigger these commands.
2. An expert which knows the optimal plan including the *best editor commands* and the *minimal key sequence* for these commands.

Figure 6-3 displays the user model (consisting of all plan specialists) that *Activist* has built up. For each

plan there is a pane which shows the performance of a specific user concerning this plan. Panes with black background indicate that the corresponding plan is currently not monitored by the active help system. A set of heuristics is used to focus the attention of the system on the critical issues.

The dialogue window at the bottom displays a help message given to the user. He/she has executed the command *set-cursor-to-beginning-of-line* by typing in the command name. *Activist* gives the hint, that this command is also bound to the key *CTRL-A*.

```
A:Wo>AnfWo       B:Wo>EndWo       C:Leer>AnfLiWo   D:Leer>EndReWo   E:Leer>EndLiWo
D: 0   G: 0      D: 1   G: 1      D: 5   G: 2      D: 2   G: 2      D: 3   G: 0
Com:  0 -> 0     Com:  0 -> 0     Com:  2 -> 5     Com:  0 -> 0     Com:  0 -> 0
Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0
F:Leer>AnfReWo   G:Ze>AnfZe       H:Ze>EndZe       I:EndZe>AnfReZe  K:AnfZe>EndLiZe
D: 1   G: 0      D: 3   G: 0      D: 0   G: 0      D: 1   G: 1      D: 1   G: 1
Com:  0 -> 0     Com:  3 -> 6     Com:  0 -> 0     Com:  0 -> 0     Com:  0 -> 0
Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0
L:Bel>EndBuf     M:Bel>AnfBuf     O:Wo*AnfWo       P:Wo*EndWo       Q:Leer*AnfLiWo
D: 5   G: 0      D: 3   G: 0      D: 2   G: 1      D: 1   G: 1      D: 0   G: 0
Com:  1 -> 1     Com:  1 -> 3     Com:  1 -> 4     Com:  0 -> 0     Com:  0 -> 0
Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0
R:Leer*EndReWo   S:Ze*AnfZe       T:Ze*EndZe       U:Bel*AnfBuf     V:Bel*EndBuf
D: 1   G: 1      D: 1   G: 0      D: 3   G: 1      D: 1   G: 0      D: 1   G: 0
Com:  0 -> 0     Com:  1 -> 4     Com:  2 -> 2     Com:  1 -> 4     Com:  0 -> 0
Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0     Key:  0 -> 0

bisy-dialog-window

give COMMAND: set-cursor-to-beginning-of-line
(set-cursor-to-beginning-of-line) is bound to ^A
```

Figure 6-3: The User Model of *Activist*

6.3 The Code-Improver System

The *Code-Improver* system [Boecker 84] can be used to get ideas on how to improve *Lisp* code. The direction of improvements may be either of the following:

- improvements that make the code more *cognitively* efficient (e.g. more readable or concise);
- improvements that make the code more *machine* efficient (e.g. smaller or faster); these improvements include those that can be found in optimizing compilers.

The user must choose which kind of suggestions he/she is interested in.

The system is used by two different user groups for two slightly different reasons:

- by intermediates who want to *learn* how to produce better *Lisp* code; we have tested the usefulness of the tool by gathering empirical, statistical data using the students of an introductory *Lisp*-course as subjects;
- by experienced users who want their code to be "straightened out"; instead of doing that by hand (which these users in principle would be able to) they use a system to carefully reconsider

the code they have written. The system is used to detect optimizations and simplifications and it has proven especially useful with code that is under development and gets changed and modified continuously.

The system operates by using a large set of transformation rules (for examples see Figure 6-4) that describe how to improve code. The user's code is matched against these rules and the transformations suggested by the rules are given to the user; the code is not modified automatically.

Saving Cons Cells

```
(rule append/.1-new.cons.cells-to-nconc/.1...          ;;; the name of the rule
   (?foo:{append append1}                              ;;;
      (restrict ?expr                                  ;;;    the condition
                (cons-cell-generating-expr expr))      ;;;
      ?b)                                              ;;;
   ==>
   ((compute-it:                                       ;;;
      (cdr (assq (get-binding foo)                     ;;;
                 '((append . nconc)                    ;;;    the action
                   (append1 . nconc1)))))              ;;;
    ?expr ?b)                                          ;;;
   safe (machine))                                     ;;; purpose and validity of the rule
```

Example:

```
(append (cdr (reverse a)) b) ---> (nconc  (cdr (reverse a)) b)
```

Avoiding Unnecessary Comparisons

```
(rule eq/equal-predicate-t
   (?foo:{eq = equal}
      (restrict ?expr (predicate-expr expr))
      ?result:{nil t})
   ==>
   ?expr safe (people machine))
```

Example:

```
(eq (numberp a) t) ---> (numberp a)
```

An Unknown Function

```
(rule length.explode/n-to-flatsize
      (length (?foo:{explode exploden} ?a))
      ==>
       (flatsize ?a) safe (machine people))
```

Example:

```
(length (explode a)) ---> (flatsize a)
```

Figure 6-4: Some Rules of the *Code-Improver* System

It is important to note, that the system is *not* restricted to a specific class of *Lisp* functions or application domain. It accepts whatever *Lisp*-code is given to it. However, there is a trade-off: since the system does not have any knowledge of specific application areas or algorithms it is naturally limited in the kind of improvements that derive from its more general knowledge about programming. The improvements suggested by the system are of the following kind:

- suggesting the use of macros (e.g. `(setq a (cons b a))` may be replaced by `(push b a)`);
- replacing compound calls of *Lisp* functions by simple calls to more powerful functions (e.g. `(not (evenp a))` may be replaced by `(oddp a)`);
- specializing functions (e.g. replacing `equal` by `eq`); using integer instead of floating point arithmetic wherever possible;
- finding alternative (simpler or faster) forms of conditional or arithmetic expressions;
- eliminating common subexpressions;
- replacing 'garbage' generating expressions by non-copying expressions (e.g. `(append (explode word) chars)` may be replaced by `(nconc (explode word) chars)`);
- finding and eliminating 'dead' code (as in `(cond (...) (t ...) (dead code))`);
- (partial) evaluation of expressions (e.g. `(sum a 3 b 4)` may be simplified to `(sum a b 7)`).

The current version of the *Code-Improver* system runs in batch mode. Like the "writers-workbench" UNIX tools, *diction* and *explain* [Cherry 81], it is given a file containing *Lisp* code and produces suggestions how to improve it.

7. Conclusions

The scenario given in section 4.2 characterizes our goals towards the construction of a *Lisp-Critic*; the systems described in section 6 serve as important stepping stones towards this goal.

Our approach towards the enhancement of incremental learning processes using knowledge-based systems does the following:

1. it applies the paradigm of *increasingly complex microworlds* to give people ideas and hints to improve their *Lisp* skills;

2. it supports people in "real" working situations by using and *combining different system components for assistance*;

3. it is oriented towards the *intermediate user* who is already involved in his/her own doing and should not be restricted to a particular tutorial sequence or to a very small number of specific case studies;

4. it builds a bridge between *learning-by-doing* and *guided tutoring and coaching* by trying to combine the best of both worlds;

5. it supports a large variety of instructional strategies and represents a *substantial amount* of knowledge about *Lisp* programming;

6. it uses our tools as object and medium (providing us with a large methodological advantage); *Lisp* and *ObjTalk* provide a universal framework for representation and are the object of the incremental learning process;

7. it *exploits graphical, aesthetically pleasing interfaces* to illustrate structures and concepts and animates the dynamics of procedures;

Acknowledgements

This paper is based on a joint research effort with my colleagues Heinz-Dieter Boecker, Andreas Lemke, Helga Nieper and Clayton Lewis who have made major contributions to the ideas and system components described in this paper. This research was supported by grants from the Office of Naval Research (contract number: N00014-85-K-0842) and the University of Colorado, Boulder.

References

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

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

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

[Boecker, Nieper 85]
H.-D. Boecker, H. Nieper, *Making the Invisible Visible: Tools for Exploratory Programming*, Proceedings of the First Pan Pacific Computer Conference, The Australian Computer Society, Melbourne, Australia, September 1985.

[Brownston et al. 85]
L. Brownston, R. Farrell, E. Kant, N. Martin, *Programming Expert Systems in OPS5: An Intoduction to Rule-Based Programming*, Addison-Wesley, Reading, MA, 1985.

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

[Burton, Brown, Fischer 83]
R.R. Burton, J.S. Brown, G. Fischer, *Analysis of Skiing as a Success Model of Instruction: Manipulating the Learning Environment to Enhance Skill Acquisition*, in Rogoff (ed.), *Everyday Cognition: its Development in Social Context*, Harvard University Press, Cambridge, MA, 1983.

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

[Fischer 81]
G. Fischer, *Computational Models of Skill Acquisition Processes*, Computers in Education, 3rd World Conference on Computers and Education, Lausanne, Switzerland, July 1981, pp. 477-481.

[Fischer 84]
G. Fischer, *Formen und Funktionen von Modellen in der Mensch-Computer Kommunikation*,

in H. Schauer, M.J. Tauber (eds.), *Psychologie der Computerbenutzung*, Oldenbourg Verlag, Wien - Muenchen, Schriftenreihe der Oesterreichischen Computer Gesellschaft, Vol. 22, 1984, pp. 328-343.

[Fischer, Burton, Brown 78]
G. Fischer, R. Burton, J.S. Brown, *Analysis of Skiing as a Success Model of Instruction: Manipulating the Learning Environment to Enhance Skill Acquisition*, Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence, Conference of the Canadian Society for Computational Studies of Intelligence, 1978.

[Fischer, Lemke, Schwab 84]
G. Fischer, A. Lemke, T. Schwab, *Active Help Systems*, Proceedings of Second European Conference on Cognitive Ergonomics - Mind and Computers, Gmunden, Austria, Springer Verlag, Heidelberg - Berlin - New York, September 1984.

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

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

[Lemke 85]
A.C. Lemke, *ObjTalk84 Reference Manual*, Technical Report CU-CS-291-85, University of Colorado, Boulder, 1985.

[Norman 82]
D. Norman, *Some Observations on Mental Models*, in D. Gentner, A. Stevens (eds.), *Mental Models*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1982.

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

[Rathke 86]
C. Rathke, *ObjTalk: Repraesentation von Wissen in einer objektorientierten Sprache*, PhD Dissertation, Universitaet Stuttgart, Fakultaet fuer Mathematik und Informatik, 1986, (forthcoming).

[Rathke, Lemke 85]
C. Rathke, A.C. Lemke, *ObjTalk Primer*, Technical Report CU-CS-290-85, University of Colorado, Boulder, February 1985.

[Rich, Shrobe 78]
C. Rich, H.E. Shrobe, *Initial Report on a Lisp Programmer's Apprentice*, IEEE Transactions on Software Engineering, Vol. SE-4, No. 6, 1978, pp. 456-467.

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

[Wilensky et al. 84]
R. Wilensky, Y. Arens, D. Chin, *Talking to UNIX in English: An Overview of UC*, Communications of the ACM, Vol. 27, No. 6, June 1984, pp. 574-593.