Massachusetts Institute of Technology
Artificial Intelligence Laboratory

Working Paper 244 February 1983

# Interfacing to the Programmer's Apprentice

Kent Pitman

*Abstract*

In this paper, we discuss the design of a user interface to the Knowledge-Based Editor (KBE), a prototype implementation of the Programmer's Apprentice. Although internally quite sophisticated, the KBE hides most of its internal mechanisms from the user, presenting a simplified model of its behavior which is flexible and easy to use. Examples are presented to illustrate the decisions which have led from high-level design principles such as "integration with existing tools" and "simplicity of user model" to a working implementation which is true to those principles.

This paper has been submitted to SoftFair, an IEEE/NBS/SIGSOFT co-sponsored conference on software development tools, techniques, and alternatives, which will be held at the Hyatt Regency Crystal City, Arlington, VA., July 26-28, 1983.

## I. Introduction

The ultimate goal of the Programmer's Apprentice (PA) project is to show how programmer productivity and reliability can be enhanced by software tools based on theories of how today's expert programmers analyze, synthesize, modify, explain, and verify their programs.

The Knowledge-Based Editor (KBE) is a first step toward the realization of this goal. Based on the theories of program analysis presented in [Waters 78], it has served as a vehicle for experimenting with possible modes of interaction in an apprentice-based system.

In previous papers (*e.g.*, [Rich,Waters 81], [Waters 82]), various scenarios have been presented to illustrate the viability of the KBE as a program development tool. Those scenarios, however, have focused almost exclusively on the underlying capabilities of the system, leaving open the question of how the interface to the system might be implemented. This paper will focus on what the actual interface to the KBE has come to look like and why.

## II. Foundations of the KBE (Review)

This section provides a brief review of the theory behind the KBE. Readers who are already familiar with this theory may wish to skip directly to section III.

The premise upon which the idea of a Programmer's Apprentice is based is that most of the time spent by programmers (even expert ones) in their work is wasted on actions of a repetitive nature. It is our hope that, through the mechanization of the more mundane, often nearly rote tasks of programming, we can leave more of the programmer's time free to work on the aspects of each problem which are truly novel and which warrant more attention than they are frequently given in today's programming environments.

The KBE derives most of its power from a formalism called the **plan calculus,** in which programs are represented by an abstract representation called a **plan** [Rich,Shrobe 76] [Waters 78]. In the current implementation, a plan is a hierarchical structure of operation nodes, connected at each level of the hierarchy by data and control flow **links.**

Complex programs are viewed as being built out of simpler *cliches*, or stereotyped program fragments. These cliches, though often textually intertwined and hard to treat independently at the surface level of code, can be modeled by modular components at a deeper level using the plan calculus.

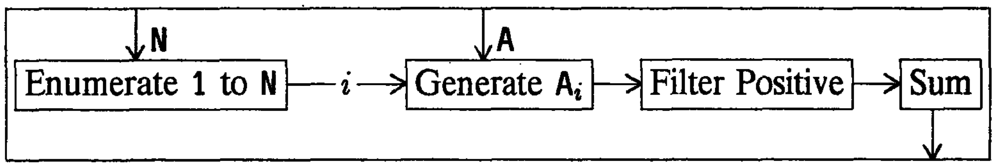
For example, consider the program fragment:

```
Z = 0;
DO I = 1 TO N;
   IF A(I) > 0 THEN Z = Z + A(I);
   END;
```

At the surface level, this program consists of an assignment statement followed by a DO loop. The DO loop is further composed of an IF statement, which is in turn composed of a test and assignment. Such a description, however, offers little support for intelligent reasoning about programs. It does not, for example, explain why the above program is essentially equivalent to the program:

```
   S = 0;
   J = 1;
L: IF A(J) > 0 THEN S = S + A(J);
   J = J+1
   IF NOT(J > N) THEN GOTO L;
```

The surface (syntactic) structure of these program fragments are different and the second program uses a different name for the index variable. However, they share the same plan, the general shape of which is:



- The range of integers from 1 to N are enumerated.
- Using this sequence as indices, a range of an array A is enumerated.
- The positive elements of the resulting sequence are filtered.
- The resulting series is accumulated as a sum.

**A Plan Description**

It is worth noting here that individual elements of the deep (plan) structure may manifest themselves at multiple points in the surface structure. For example, the enumeration of the array index, which in the first program is achieved by a single DO construct, is achieved in the second program by an initialization of the variable J early in the program and a later statement incrementing J.

These two programs can share the same plan representation because the data flow information in the plan is represented directly as paths from producers of values to their consumers rather than indirectly through the use of named variables. Even the variables Z and S, which are used free in these examples to express data flow from the sum, are only artifacts of the surface language. If the programs were expressed using recursion rather than iteration, the sum might be accumulated as recursive invocations returned and those variables would be missing altogether.

When comparing equivalent programs written in different languages, the variation in syntactic form can be even more striking. In APL, for example, the iterative operations in the above example would be replaced by vector operations and the variable names would not be present, but the underlying plan would still be the same.

The plan language serves as a canonical form in that it smoothes over many kinds of superficial syntactic distinctions. As such, the plan representation provides a useful common representation for the KBE's internal storage and manipulation of both cliches and user programs.

## Plan Library

The cliches known by the KBE are stored in a **plan library**. Some cliches in the library are very general, such as "search." Others correspond to terms which occur in the domain-specific vocabulary of a particular application, such as "equality within epsilon" in numerical programming. The KBE needs both of these kinds of plans in order to converse with the user in the same terminology that programmers would use in discussions amongst themselves.

When the user makes an editing request that uses a cliche, a copy of the cliche's plan is made from the prototype in the library and merged into the plan for the user's program.

## KBE Architecture

The KBE consists of several relatively independent modules working together. An **analyzer** is used to translate surface code into plan structure. Programs are edited at the plan level. Then a **coder** is used to translate code back into Lisp.

In previous papers, all communication with the KBE has been carried on using a formal language which superficially resembles English. Here is an example of such an interaction:

```
>Define a program SQRT with a parameter NUM.
>Implement it as a successive approximation.
>Implement the approximation as an average of 'RESULT
  and '(// NUM RESULT).

  ;;; KBE-supplied definition of SQRT

  (DEFUN SQRT (NUM &AUX RESULT)
    (PROG NIL
          (SETQ RESULT initial-value)
       LP (COND ((test RESULT) (RETURN RESULT)))
          (SETQ RESULT (// (+ (// NUM RESULT) RESULT) 2))
          (GO LP)))
```

The KBE codes what it can of the program, marking the rest to be filled in by later commands. For example:

```
>Implement the test as an equality within epsilon
  of 'NUM and '(* RESULT RESULT).

  ;;; KBE-supplied definition of SQRT

  (DEFUN SQRT (NUM &AUX RESULT)
    (PROG NIL
          (SETQ RESULT initial-value)
       LP (COND ((< (ABS (- (* RESULT RESULT) NUM)) epsilon)
                 (RETURN RESULT)))
          (SETQ RESULT (// (+ (// NUM RESULT) RESULT) 2))
          (GO LP)))
```

## III. Philosophy of Interface Design

Although there has been much research in the area of program editors that deal at the syntactic level, relatively little work has been done in the area of editors that manipulate code at a semantic level.

In this section, we will identify some of the principles that we feel should guide the design of an interface to an apprentice. Then, in the following section, we will make these ideas concrete with examples from our implementation.

### An Integrated Environment

It is important that the apprentice system present itself as an *extension* to its resident programming environment, not as a replacement thereof. Many text editors (*e.g.*, EMACS [Stallman 80]) offer the capability of viewing programs at many different levels, such as the character level, the token level, and the expression level. The existence of the apprentice in the environment will not obviate the need to manipulate programs at the level of their syntax; such complete automatic programming is beyond the scope of our current research. Rather, we hope that this system will broaden the expressive capability of the user of such an editor so that he can manipulate code not only at the syntactic level, but also at a semantic level if that will better suit his purpose.

This extension must also be *well-integrated* into the programming environment so that the decision of whether or not to put the apprentice to work on some problem involves very little overhead. If invoking the apprentice means suspending one editor and entering another or typing a large number of keystrokes, the programmer may frequently find that using the apprentice is not worth the cost in terms of time or disruption to his sense of focus.

The apprentice should also be *non-invasive*; when not in use, it should not present the user with constant reminders of its presence. At any point, the user should be able to ignore the presence of the apprentice in the environment and revert to editing at the syntactic level, if he sees this as the easiest way to accomplish his goal.

As with a conventional text editor, the apprentice should be *capable of maintaining partial state* in order to accomodate a shifting sense of focus. The user may wish to be moving between several projects at once; it would be inappropriate for the apprentice to force the user to finish one subproblem before proceeding to another.

The apprentice should use a *familiar style of interaction.* Its command language should be like something the programmer is used to. Rather than present him with a completely new command language, it should attempt to exploit any working vocabulary he already has, whether natural or formal.

In matters of programming style, the apprentice should be *non-prescriptive.* It is up to the apprentice to conform to the user's sense of style, and not vice versa. The apprentice may wish to make occasional helpful suggestions, but should do so cautiously and should always be willing to defer to the user.

The apprentice must be *sensitive to its role as an interactive tool.* Commands to the apprentice, because they involve more than simple text manipulation, may take longer to execute than normal editing commands. If the wait is noticeable, the programmer time that should be saved by invoking the apprentice may be lost again by forcing the user to sit idle awaiting the result. Hence, the apprentice must be able to respond quickly to all requests, or it should allow batch and interactive modes to minimize the time lost by the user to supervisory duties.

## Simple User Model

The user needs to be kept aware of the relevant aspects of the internal state of his tools, but should not be burdened with unnecessary levels of detail. The interface should have an *insulating* effect on the user, shielding him from irrelevant implementation details.

It is not effective to reason directly about program text; programs must be manipulated at the plan structure level. Yet, we would like it very much if the user were required to know as little as possible about the plan representation and how it is maintained. It is sufficient that when commands are given to the apprentice, the observed effect is a change in the textual representation of that program. If the changes are always realized in a predictable and reliable fashion, the user will be more than happy to adopt the simpler model that he is simply manipulating program text.

In a system which presents a simplified model of itself to the user, some internally generated error conditions may have no meaning to that user. It is important not only to minimize the occurrence of such errors, but also to handle them when they do occur in a way which will not violate the user's model of the system. A simple metaphor is only desirable if it can be maintained consistently. If it breaks down frequently, the the illusion of simplicity vanishes and the user becomes frustrated by his inability to predict the system's response to requests. The user's model of the system must be supported not only by intelligible behavior when it works, but also when it doesn't.

## IV. Implementation

Our implementation builds on ZWEI, the text editor used on the Lisp Machine [Weinreb,Moon 81]. ZWEI is a real-time, display-oriented text editor descended from Emacs. It has a rich command set which can be easily customized and extended using ordinary Lisp code because ZWEI is actually resident in Lisp.

In this section, in order to illustrate how our design principles have come to be realized in practical situations, we survey some of the some of the ways ZWEI has been extended to interface to the KBE. The examples in this section are taken directly from sessions with the current implementation.

### A Natural Extension to ZWEI

The Lisp Machine's character set is much larger than conventional ASCII. In addition to to the usual `SHIFT` and `CONTROL` keys, it offers special shift keys called `META`, `SUPER`, and `HYPER`. Each editor command is a single (possibly shifted) character. Commands tend to be arranged so that commands involving higher 'levels of shift' do increasingly more sophisticated things. For example, many `CONTROL` commands act at the character level, `META` commands at the word level, and `CONTROL-META` commands at the Lisp S-expression level. Since normal ZWEI does not define many commands that use the `SUPER` and `HYPER` shift keys, the KBE system reserves commands in that namespace for its own purposes.

The standard ZWEI command `CONTROL-=` can be typed at any point in the buffer and will type out information about the position of the cursor textually in the buffer and visually on the screen. Playing on this theme, our extended ZWEI environment offers the command `SUPER-=`, which will type out any available information about the cursor location not at the character level, but at the plan level. So, at the appropriate time, typing `CONTROL-=` might produce output such as as

```
X=[5.  chars|38.  pixels|6 columns] Y=5 Char=40
```

while typing `SUPER-=` might produce output such as

```
The cursor is at the approximation of the program SQRT.
```

To illustrate with a second example, the normal ZWEI command `META-X` prompts the user for the name of an "extended command" to be invoked. For example, when the user types `META-X`, ZWEI prompts him with:

```
Extended Command:
```

The user then types the name of some extended command to be executed. For example, to compile the definitions in his editor buffer, he responds:

```
Extended Command:
Compile Buffer
```

In the KBE extension, SUPER-X uses a similar style, prompting for a textual request in the KBE's pseudo-English input language. When the user types SUPER-X, he sees:

```
Execute KBE Commands:
```

To instruct the KBE to write a program which sums a list of numbers, he might respond:

```
Execute KBE Commands:
Define a program SUMLIST with a parameter L. Implement the
program as a sum of a list enumeration of 'L.
```

Of course, the important fact here is not that we have chosen any particular command character but rather that we have observed the style rules used by the designers of ZWEI and attempted to interface our command set in a way that will be natural and predictable to prospective users of our extension.

**Mixed Mode Editing**

When interacting with the apprentice, it is frequently faster to simply edit the text of a program in the buffer than to explain to the apprentice how to edit it. It is for this purpose that we insist that the apprentice not try to replace the normal editor environment. The user can, for example, type:

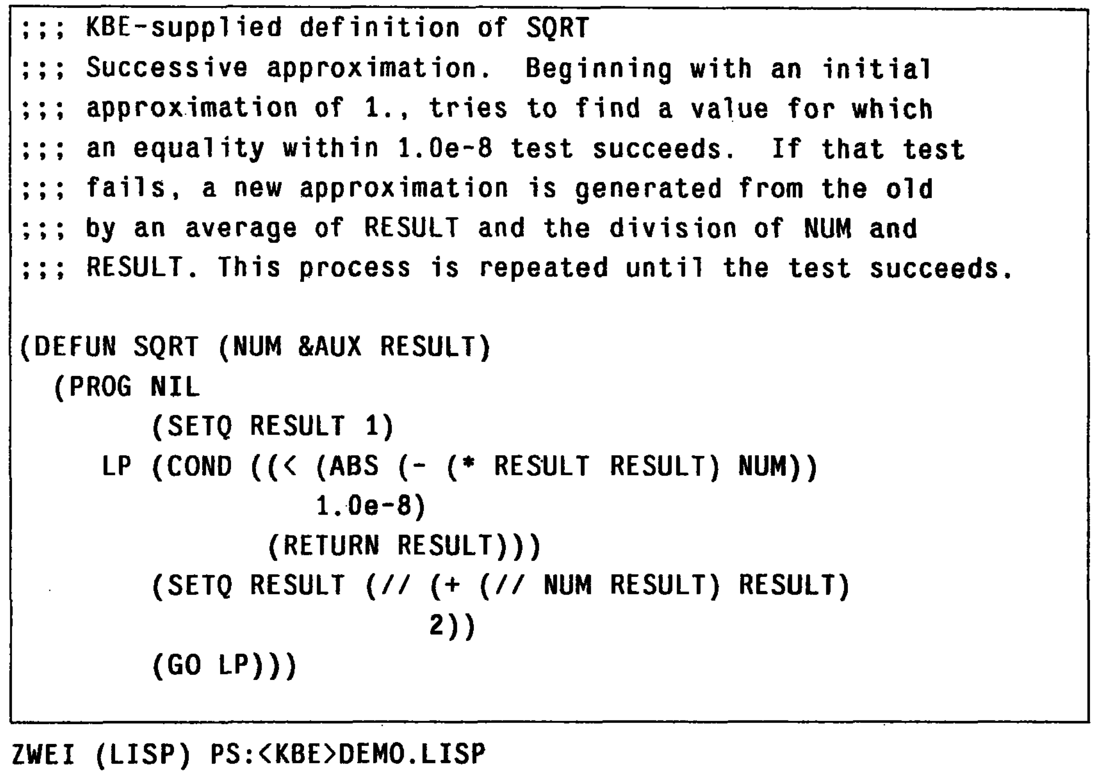
```
Execute KBE Commands:
Implement the initial value of the program SQRT as '1 .
```

Frequently, however, it simpler to just edit a 1 into the appropriate place in the program.

If the user modifies the text of a program which the KBE has coded and never asks the KBE to further manipulate that program, no conflict arises. On the other hand, if the user decides after editing some program that he must ask the apprentice to deal further with that program's code, the apprentice must (and does) have mechanisms built into it which recognize that the code in the user's editor buffer is not the same as the code that it last produced. At this point, it will generally invalidate the information that it has stored about the program and analyze the program from scratch producing a fresh model of the program.[1]

## Program Explanation Facility

In normal ZWEI, the command META-; is used to enter a comment area.[2] In the KBE implementation, SUPER-; is like META-; except that it not only enters a comment field, but also writes a comment for the code at the cursor. The comments, though not as polished as those that would be generated by a person, are still considerably better than what one could expect from a program that was writing comments using only syntactic information.



```
;;; KBE-supplied definition of SQRT
;;; Successive approximation.  Beginning with an initial
;;; approximation of 1., tries to find a value for which
;;; an equality within 1.0e-8 test succeeds.  If that test
;;; fails, a new approximation is generated from the old
;;; by an average of RESULT and the division of NUM and
;;; RESULT. This process is repeated until the test succeeds.

(DEFUN SQRT (NUM &AUX RESULT)
   (PROG NIL
         (SETQ RESULT 1)
      LP (COND ((< (ABS (- (* RESULT RESULT) NUM))
                   1.0e-8)
                (RETURN RESULT)))
         (SETQ RESULT (// (+ (// NUM RESULT) RESULT)
                          2))
         (GO LP)))
```

```
ZWEI (LISP) PS:<KBE>DEMO.LISP
```

When the user asks that a comment be produced for a piece of code, the area of the buffer at the cursor is inspected to determine what program text is being referred to. The textual information from the buffer is matched against an annotated copy of the program that was made when the program was coded by the apprentice.[3] The relevant plan information is recovered from correspondence information stored in the annotated program, and a comment is then generated from that plan structure, using a technique not unlike coding [Cyphers 82].

[1]In the current implementation, the plan structure produced by re-analysis will be inferior to the plan structure produced as the KBE constructed the program. This is because cliche recognition is not possible in the current analyzer. Our next implementation will attempt to correct this deficiency using techniques described in [Brotsky 81].

[2]The semicolon is intended to be mnemonic for "comment" because it is the comment character for Lisp and many assembly languages.

[3]If the annotated copy does not match, the KBE must first re-analyze the the buffered text, since the model of the user's code is now obsolete.

The user refers to the surface level of the program in requesting that a comment be generated. He does not worry about how the KBE's knowledge is actually maintained internally. This is an example of how the interface insulates the user from unnecessary detail.

Once generated, the comment is inserted into the buffer and the programmer is completely free to inspect, modify, or remove it using normal ZWEI editing primitives. In this way, the user is always in control of the end product. The apprentice does not insist that the code be commented in this way; it simply offers the comment in response to a user request and then lets the user decide if he is happy with the result.

**Two-Way Buffer Correspondences**

The above example of comment generation illustrates that the KBE maintains a mapping from the text to the plan structure. It is possible to invert this relation to achieve other goals. By giving a deep description of code in the buffer, the user can, for example, cause the cursor to be moved to the corresponding point in the buffer. In a conventional editor, this would be difficult to do efficiently in the general case since the user might shuffle functions around in his editor buffers in ways that are hard for the apprentice to track. Fortunately, however, the ZWEI editor tracks the movement of functions from place to place in an incremental fashion, making it possible to locate a part of one program in a file containing many programs without actually searching the entire buffer for that program. We have found this capability to be an invaluable aid in the construction of our extended environment.

The ZWEI command `META-.` prompts for the name of a function and then moves the cursor to that function's definition. The KBE offers a command `SUPER-.` which prompts:

```
Move cursor to:
```

In reply, the programmer types a definite reference as in:

```
Move cursor to:
the splice out action of the program DELETE
```

## Reference Language

Although our implementation principally supports Lisp, we have constructed a mock-up of a PL/I interface which has allowed us to get a feel for what it would be like to have the capability of multi-lingual interactions. The user simply types SUPER-X and selects the desired reference language. For example:

```
Execute KBE commands:
Use language PL/I.
```

Part of the reason that the transition from PL/I to Lisp can occur smoothly is that, in its reasoning, the apprentice uses high-level, abstract terms. Only a small fraction of its representation of the program is actually language dependent. Consider the following modification to our earlier SQRT scenario:

```
Execute KBE Commands:
Implement the approximation as an average of 'RESULT and '(//
NUM RESULT).  Use language PL/I. Implement the test as an
equality within epsilon of "NUM" and "RESULT*RESULT".
```

In response, the KBE replies:

```
 /* KBE-supplied definition of SQRT */

 SQRT: PROCEDURE(NUM) RETURNS (FLOAT);
           DCL NUM FLOAT,
               RESULT FLOAT;
           RESULT = initial-value;
       LP: IF ABS(RESULT*RESULT-NUM) < epsilon
              THEN RETURN (RESULT);
           RESULT = (NUM/RESULT+RESULT)/2.;
           GOTO LP;
         END SQRT;

ZWEI (PL1) PS:<KBE>DEMO.PL1
```

## Background Processing

Because the Lisp Machine allows convenient control of multi-processing, we have chosen to implement the apprentice as a separate process.

If the user invokes a command that requires an immediate response from the apprentice, the user's process enters a wait state and (if there are no unrelated tasks competing for the processor) the apprentice is free to use the full resources of the machine. When the apprentice has finished handling the request, the user's process awakens to handle the reply.

On the other hand, by using HYPER-X instead of SUPER-X, the user has the option of requesting things of the apprentice without waiting for the results. When this happens, the apprentice process proceeds in background at a lower priority than the user's process, so that response to other commands by the user are minimally interfered with. This is useful because, in the current implementation, each command to the apprentice takes 3 to 20 seconds to execute, which can be slow enough to cause irritation.

The apprentice maintains a queue of requests to be handled, so many requests can be queued up in advance by the user. When the apprentice finishes with each request, it notifies the user of this fact in a way that does not force him to respond. The user can query the status of the results and the requests still pending, and he can choose (by typing HYPER-RESUME) when he wants to have the request results merged into the editor buffer. The apprentice will also notify the user when it has completed all of its assigned tasks.

The apprentice can be placed in a mode where it asynchronously updates program text in the editor buffer as soon as it becomes available. In most cases, however, the effects are highly disconcerting, breaking the user's sense of focus and sometimes displacing the cursor in unexpected ways. Because this violates the principle that the apprentice should be non-invasive, synchronous buffer update using HYPER-RESUME is the default.

## V. Conclusions

We have surveyed the design criteria which have helped to shape our interface to the KBE, a complex tool for program understanding and manipulation.

Although internally quite sophisticated, the KBE hides most of its internal mechanisms from the user, presenting a simplified model of its behavior which is flexible and easy to use.

Examples have been given of how an existing editor command set has been extended in a natural way to accomodate interaction with our system. We have shown how the use of ordinary program text as a reference language for a deeper underlying structure can be used to shield the user from unnecessary levels of detail. Finally, we have shown how the user can still exploit this hidden information about the deep structure of his programs to modify and document his code.

Through such discussion, we have shown how our KBE implementation satisfies its design principles and cooperates with existing tools for program development to provide a more productive editing environment.

## *Acknowledgments*

The design of the interface was influenced by **EMACS**, and the **Lisp Machine**, and **ZWEI** (the Lisp Machine editor).

Much useful creative support during the design of the KBE interface was provided by Howard Cannon, David Chapman, Mike McMahon, David Moon, Charles Rich, Richard Stallman, Richard Waters, and Daniel Weinreb.

Thanks to Daniel Brotsky, David Chapman, Henry Lieberman, Charles Rich, Richard Waters, and Daniel Weise for reading drafts of this paper.

## *References*

[Brotsky 81] D. **Brotsky**, *Program Understanding Through Cliche Recognition*, (M.S. Proposal), MIT AI WP-224, December, 1981.

[Cyphers 82] D.S. **Cyphers**, *Automated Program Explanation*, MIT AI WP-237, August, 1982.

[Rich,Shrobe 76] C. **Rich** and H.E. **Shrobe**, "Initial Report on A Lisp Programmer's Apprentice", *IEEE Transactions*, Vol. SE-4, No. 5, November, 1978.

[Rich,Waters 81] C. **Rich** and R.C. **Waters**, *Abstraction, Inspection, and Debugging in Programming*, MIT AI Memo 634, June, 1981.

[Stallman 80] R.C. **Stallman**, *Emacs Manual For Twenex Users*, MIT AI Memo 555, September, 1980.

[Waters 78] R.C. **Waters**, "A Method for Analyzing Loop Programs", *IEEE Transactions*, Vol. SE-5, No. 3, May 1979, pp. 237-247.

[Waters 82] R.C. **Waters**, "The Programmer's Apprentice: Knowledge Based Program Editing", *IEEE Transactions*, Vol. SE-8, No. 1, January, 1982.

[Weinreb,Moon 81] D. **Weinreb** and D. **Moon**, *Lisp Machine Manual*, 4th ed., MIT Artificial Intelligence Laboratory, July 1981.