AI Magazine Volume 7 Number 1 (1986) (© AAAI)

KBEmacs: Where's the AI?

Richard C. Waters

MIT Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, Massachusetts 02139

The Programmer's Apprentice project uses the domain of programming as a vehicle for studying (and attempting to duplicate) human problem solving behavior. Recognizing that it will be a long time before it is possible to fully duplicate an expert programmer's abilities, the project seeks to develop an intelligent assistant system, the Programmer's Apprentice (PA), which will help a programmer in various phases of the programming task. The Knowledge-Based Editor in Emacs (KBEmacs) is an initial step in the direction of the PA.

A question that has been asked about KBEmacs is, "Where's the AI?" This article answers this question by describing the key AI ideas that underly the system. Going beyond this, the article uses the development of KBEmacs as an example that illustrates a number of general features of the process of developing an applied AI system. As part of this, the article compares the way AI ideas are used in KBEmacs with the way they were used in the initial proposal for the PA.

This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grants MCS-7912179 and MCS-8117633, and in part by the International Business Machines Corporation

I would particularly like to acknowledge the assistance of my partner Charles Rich, who has made important contributions to every aspect of my work for a decade

KBEmacs is the result of a group effort that began with the original Programmer's Apprentice proposal of Charles Rich and Howard Shrobe Over the years, many other people have contributed to this effort Gerald Sussman has been an inspiration and a mentor for us all Kent Pitman implemented the user interface for KBEmacs and assisted with many other aspects of the system Daniel Brotsky, David Chapman, David Cyphers, Roger Duffey, Gregory Faust, Daniel Shapiro, Peter Sterpe, and Linda Zelinka contributed ideas to KBEmacs, while working on related parts of the Programmer's Apprentice project. Special thanks are due to Crisse Ciro, for her assistance with the illustrations, and to Roger Racine (of the C S. Draper Laboratory), for his assistance with regard to Ada

An Example of Using KBEmacs

In order to give a feeling for the capabilities of KBEmacs, this section presents a condensed summary of the scenario in Waters (1985). In that scenario, a programmer uses KBEmacs to construct an Ada program in the domain of business data processing. It is assumed that there is a data base which contains information about various machines (referred to as units) sold by a company and about the repairs performed on each of these units. In the scenario, the programmer constructs a program called UNIT_REPAIR_REPORT, which prints out a report of all of the repairs performed on a given unit. The directions in Figure 1 might be given to a human assistant who was asked to write this program.

A key feature of these directions is that they refer to a significant amount of knowledge that the assistant is assumed to possess. First, they refer to a number of standard programming algorithms: "simple report," "enumerating the records in a chain," and "querying the user for a key."

Abstract

The Knowledge-Based Editor in Emacs (KBEmacs) is the current demonstration system implemented as part of the Programmer's Apprentice project KBEmacs is capable of acting as a semiexpert assistant to a person who is writing a program, taking over some parts of the programming task. The abilities of KBEmacs stem directly from a few key AI ideas. However, in many ways KBEmacs does not appear to be an AI system, because its abilities are limited and because (like many applied AI systems) the AI ideas are buried in a large volume of code that has little relevance to AI. The primary goal of this article is to present the AI ideas behind KBEmacs. In addition, the construction of applied AI systems is discussed, in general, using the development of KBEmacs as a case history.

Define a simple report program UNIT_REPAIR_REPORT.
Enumerate the chain of repairs associated with a unit record, printing each one.
Query the user for the key (UNIT_KEY) of the unit record to start from.
Print the title ("Report of Repairs on Unit" & UNIT_KEY).
Do not print a summary.

Hypothetical Directions for a Human Assistant.
Figure 1.

Second, the directions assume that the assistant understands the structure of the database of units and repairs. Another feature of the directions is that, given that the assistant has a precise understanding of the algorithms to be used and of the database, little is left to the assistant's imagination. Essentially every detail of the algorithm is spelled out, including the exact Ada code to use when printing the title.

The commands shown in Figure 2 can be used to construct the program UNIT_REPAIR_REPORT using KBEmacs. The Ada program that results from these commands is shown in Figure 3.

```
Define a simple-report procedure
   UNIT-REPAIR-REPORT.
Fill the enumerator with a chain-enumeration of
   UNITS and REPAIRS.
Fill the main-file-key with a query-user-for-key
   of UNITS.
Fill the title with (''Report of Repairs on
   Unit'' & UNIT-KEY).
Remove the summary.
```

KBEmacs Commands.
Figure 2.

A key feature of the commands in Figure 2 is that they refer to a number of standard algorithms known to KBEmacs: "simple_report," "chain_enumeration," and "query_user_for_key." In addition, the commands assume an understanding of the structure of the database. The "Fill" commands specify how to fill in the parts of the simple_report algorithm.

Without discussing in any detail either the commands or the program produced, two important observations can be made. First, the commands used are similar to the hypothetical directions for a human assistant. Second, a set of five commands produces a 56-line program. (The program would be even longer if it did not make extensive use of data declarations and functions defined in the packages FUNCTIONS and MAINTENANCE_FILES)

The KBEmacs commands and the hypothetical directions differ in grammatical form but not in semantic content. This is not surprising in light of the fact that the hypothetical directions were, in actuality, created by restating the knowledge-based commands in more free-flowing English.

The purpose of this translation was to demonstrate that although the KBEmacs commands may be syntactically awkward, they are not semantically awkward. The commands are neither redundant nor overly detailed. They specify only the basic design decisions that underly the program. There is no reason to believe that any automatic system (or, for that matter, a person) could be told how to construct the program UNIT_REPAIR_REPORT without being told at least most of the information in the commands shown.

The leverage that KBEmacs applies to the program-construction task is illustrated by the order of magnitude difference between the size of the set of commands and the size of the program. A given programmer seems to be able to produce more or less a constant number of lines of code each day, independent of the programming language being used. As a result, there is reason to believe that the order of magnitude size reduction provided by the KBEmacs commands would translate into an order of magnitude reduction in the time required to construct the program. It should be noted that since program construction is only a small part (around 10%) of the programming life cycle, this does not translate into an order of magnitude savings in the life cycle as a whole.

Another important advantage of KBEmacs is that using standard algorithms (such as simple_report and chain_enumeration) enhances the reliability of the programs produced. Because the standard algorithms known to KBEmacs are intended to be used many times, it is economically justifiable to lavish a great deal of time on them in order to ensure that they are general purpose and bug free. This reliability is inherited by the programs that use the standard algorithms.

When using an ordinary program editor, programmers typically make two kinds of errors: picking the wrong algorithms to use and incorrectly instantiating these algorithms (that is, combining the algorithms together and rendering them as appropriate program code). KBEmacs eliminates the second kind of error.

Four Key AI Ideas

Three basic AI ideas—the assistant approach, clichés, and plans—underlie the PA project as a whole and KBEmacs in particular. These ideas define the approach taken and are the basis for the capabilities of the system. A fourth idea—general-purpose automated deduction—is an important aspect of the project as a whole but is not used by KBEmacs.

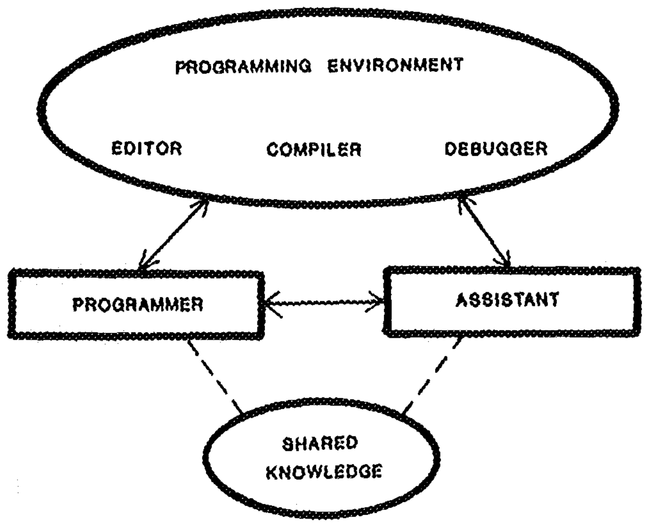
```
with CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
use CALENDAR, FUNCTIONS, MAINTENANCE_FILES, TEXT_IO;
procedure UNIT_REPAIR_REPORT is
   use DEFECT_IO, REPAIR_IO, UNIT_IO, INT_IO;
   CURRENT_DATE: constant STRING := FORMAT_DATE(CLOCK);
   DEFECT: DEFECT_TYPE;
   REPAIR: REPAIR_TYPE;
   REPAIR_INDEX: REPAIR_INDEX_TYPE;
   REPORT: TEXT_IO.FILE_TYPE;
   TITLE: STRING(1..33);
   UNIT: UNIT_TYPE;
   UNIT_KEY: UNIT_KEY_TYPE;
   procedure CLEAN_UP is
      begin
         SET_OUTPUT(STANDARD_OUTPUT);
         CLOSE(DEFECTS); CLOSE(REPAIRS); CLOSE(UNITS); CLOSE(REPORT);
      exception
         when STATUS_ERROR => return;
      end CLEAN_UP;
begin
   OPEN(DEFECTS, IN_FILE, DEFECTS_NAME); OPEN(REPAIRS, IN_FILE, REPAIRS_NAME);
   OPEN(UNITS, IN_FILE, UNITS_NAME); CREATE(REPORT, OUT_FILE, ''report.txt'');
   loop
      begin
         NEW_LINE; PUT(''Enter UNIT Key:  ''); GET(UNIT_KEY);
         READ(UNITS, UNIT, UNIT_KEY);
         exit;
      exception
         when END_ERROR => PUT(''Invalid UNIT Key''); NEW_LINE;
      end;
   end loop;
   TITLE := ''Report of Repairs on Unit '' & UNIT_KEY;
   SET_OUTPUT(REPORT);
   NEW_LINE(4); SET_COL(20); PUT(CURRENT_DATE);
   NEW_LINE(2); SET_COL(13); PUT(TITLE); NEW_LINE(60);
   READ(UNITS, UNIT, UNIT_KEY);
   REPAIR_INDEX := UNIT.REPAIR;
   while not NULL_INDEX(REPAIR_INDEX) loop
      READ(REPAIRS, REPAIR, REPAIR_INDEX);
      if LINE > 64 then
         NEW_PAGE; NEW_LINE; PUT(''Page:  ''); PUT(INTEGER(PAGE-1), 3);
         SET_COL(13); PUT(TITLE); SET_COL(61); PUT(CURRENT_DATE); NEW_LINE(2);
         PUT(''  Date   Defect   Description/Comment''); NEW_LINE(2);
      end if;
      READ(DEFECTS, DEFECT, REPAIR.DEFECT);
      PUT(FORMAT_DATE(REPAIR.DATE)); SET_COL(13); PUT(REPAIR.DEFECT);
      SET_COL(20); PUT(DEFECT.NAME); NEW_LINE;
      SET_COL(22); PUT(REPAIR.COMMENT); NEW_LINE;
      REPAIR_INDEX := REPAIR.NEXT;
   end loop;
   CLEAN_UP;
exception
   when DEVICE_ERROR | END_ERROR | NAME_ERROR | STATUS_ERROR =>
      CLEAN_UP; PUT(''Data Base Inconsistent'');
   when others => CLEAN_UP; raise;
end UNIT_REPAIR_REPORT;
```

The Ada program UNIT_REPAIR_REPORT.
Figure 3.

The Assistant Approach

When it is not possible to construct a fully automatic system for a task, it is, nevertheless, often possible to construct a system that can assist an expert in the task. In addition to being pragmatically useful, the assistant approach can lead to important insights into how to construct a fully automatic system.

Figure 4 shows a programmer and an assistant interacting with a programming environment. Though presumably less knowledgeable, the assistant interacts with the tools in the environment (for example, editors, compilers, and debuggers) in the same way as the programmer and is capable of helping the programmer do what needs to be done. It is assumed that the programmer will not be able to delegate to the assistant all of the work which needs to be done and therefore will have to interact directly with the programming environment from time to time in order to do things that the assistant is not capable of doing.



A Programming Assistant.

Figure 4.

The key issue in using an assistant effectively is division of labor. Because the programmer is more capable, the programmer will have to make the hard decisions about what should be done and what algorithms should be used. However, much of programming is quite mundane and can easily be done by an assistant. The key to cooperation between the programmer and the assistant is effective two-way communication, whose key, in turn, is shared knowledge. It would be unbearably tedious for the programmer to explain each decision to the assistant from first principles. Rather, the programmer needs to be able to rely on a body of intermediate-level shared knowledge in order to communicate decisions easily.

The previous discussion applies equally well to human assistants and automated assistants. KBEmacs is intended to interact with a programmer in the same way that a human assistant might. The long-range goal of the PA is to create a "chief programmer team," wherein the programmer is the chief, and the PA is the team.

An important benefit of the assistant approach is that it is nonintrusive in nature. The assistant is available for the programmer to use, but the programmer is not forced to use it. Note that this contrasts sharply, for example, with program generators, which completely take over large parts of the programming task and do not allow the programmer to have any control over them. A key goal of KBEmacs is to provide assistance to the programmer without preventing the programmer from doing simple things in the ordinary way. The intent is for the programmer to use standard programming tools whenever that makes things easy and to use KBEmacs only when doing so delivers real benefit.

A key part of the assistant approach as described here is the assumption that the assistant is significantly less knowledgeable than the programmer. There are situations where one might want an assistant system that was more knowledgeable than the programmer (for example, a system that assists end users or neophyte programmers). However, KBEmacs does not attack these kinds of problems. The goal of KBEmacs is to make expert programmers super-productive, rather than to make bad programmers good.

Clichés

The term *cliché* is used in this article to refer to a standard method for dealing with a task, for example, a lemma or a partial solution. In normal usage, the word cliché has a pejorative sound that connotes overuse and a lack of creativity. However, it is not practical to be creative all of the time. For example, when constructing a program, it is usually better to construct a reasonable program rapidly than to construct a perfect program slowly.

A cliché consists of a set of roles embedded in an underlying matrix. The roles represent parts of the cliché which vary from one use of the cliché to the next but which have well-defined purposes. The matrix specifies how the roles interact in order to achieve the goal of the cliché as a whole.

As an example of a cliché in the domain of programming, consider the clichéd algorithm simple_report used in the example. This cliché enumerates a sequence of items and prints them out.

The cliché simple_report has five main roles. The *title* is printed on a title page and, along with the page number, at the top of each succeeding page of the report. The *enumerator* enumerates some sequence of items. The *print_item* prints out information about each of the enumerated items. The *column_headings* are printed at the top of each page of the report in order to explain the output of the print_item. The *summary* prints out some summary information at the end of the report.

The matrix of the cliché specifies several different kinds of information. First, it specifies pieces of fixed computation that do not vary from one use of the cliché to the next, for example, how to print out a title page including the title, date, and time.

Second, the matrix specifies the control flow and data flow that connect the roles with each other and with the fixed computation. For example, data flow connects the output of the enumerator with the input of the print_item, and control flow specifies that the summary will not be printed until all of the enumerated items have been printed.

Third, the matrix specifies various constraints on the roles. For example, the print_item is constrained to contain a computation that is appropriate for printing out the type of item which is enumerated by the enumerator. Similarly, the column_headings are constrained to correspond to the print_item.

When a cliché is used, it is instantiated by filling in the roles with computations that are appropriate to the task at hand. This creates an instance of the cliché that is specialized to the task. In Figure 2 we see that in order to construct the program UNIT_REPAIR_REPORT, the enumerator of the cliché simple_report is filled in with a chain_enumeration, the title is filled in with the specified title, and the summary is removed. The constraints described earlier operate to fill in the print_item and column_headings with computation appropriate for printing out repair records. (The role main_file_key is part of the cliché chain_enumeration.)

Given a particular domain, clichés provide a vocabulary of relevant intermediate- and high-level concepts. Having such a vocabulary is essential for effective reasoning and communication in the context of the domain. It is important to note that this is just as important for human thought as it is for machine-based thought.

Both men and machines are limited in the complexity of the lines of reasoning they can develop and understand. In order to deal with more complex lines of reasoning, intermediate-level vocabulary that summarizes parts of the line of reasoning must be introduced. Once this intermediate vocabulary is fully understood, it can be used to express the full line of reasoning in a sufficiently straightforward way.

Men and machines are similarly limited in the complexity of the descriptions they can communicate. Just as it is, in general, never practical to reason about something from first principles, it is in general never practical to describe something in full detail from first principles. Effective communication depends on the shared knowledge of an appropriate vocabulary between speaker and hearer.

An essential part of the cliché concept is reuse. Once something has been thought out (or communicated) and given a name, it can then be reused as a component in future thinking (communication). There is an overhead in that something must be thought out very carefully in order for it to serve as a truly reusable component. However, if successful, this effort can be amortized over many instances of reuse.

A corollary of the cliché idea is that a library of clichés is often the most important part of an AI system. In KBEmacs, a large portion of the knowledge that is shared between man and machine is in the form of a library of algorithmic clichés. This library can be viewed as being a machine-understandable definition of the vocabulary programmers use when talking about programs.

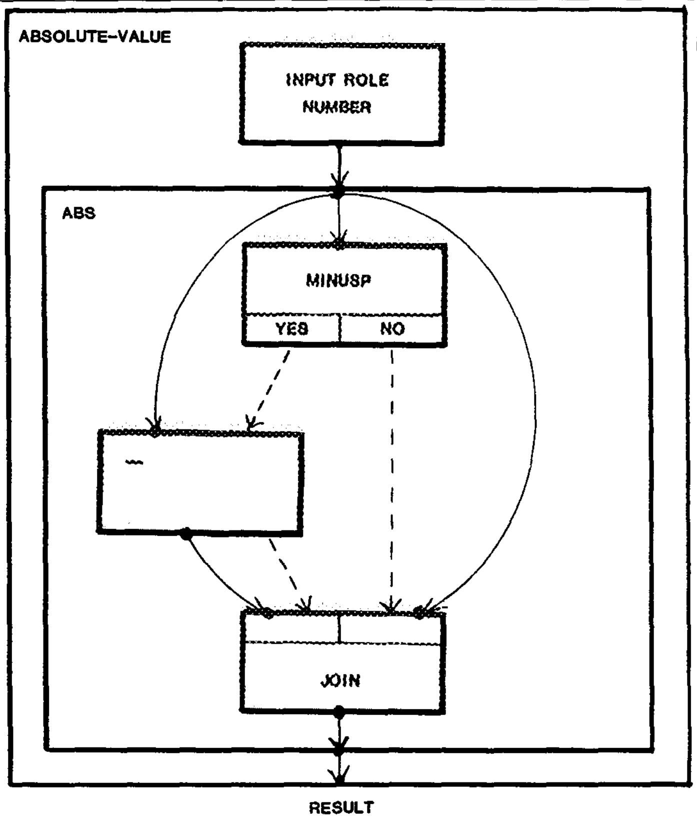
Plans

Selecting an appropriate knowledge representation is the key to applying AI to any task. As a practical matter, the only way to perform a complex (as opposed to merely large) operation is to find a knowledge representation in which the operation can be performed in a relatively straightforward way. To this end, many AI systems make use of the idea of a plan—a representation which is abstract in that it deliberately ignores some aspects of a problem in order to make it easier to reason about the remaining aspects of the problem.

To be useful, a knowledge representation must express all of the information relevant to the problem at hand. The plan formalism used by KBEmacs is designed to represent two basic kinds of information: the structure of particular programs and knowledge about clichés. The structure of a program is expressed essentially as a hierarchical flowchart where data flow, as well as control flow, is represented by explicit arcs. In order to represent clichés, added support is provided for representing roles and constraints.

Equally important, a knowledge representation must facilitate the operations to be performed. The two key operations performed by KBEmacs are simple reasoning about programs (for example, determining the source of a data flow) and combining clichés together to create programs. The plan formalism is specifically designed to support these operations. For example, the fact that data flow is expressed by explicit arcs makes it easy to determine the source of a given data flow.

Figure 5 is a diagram of a simple example plan, the plan for the cliché *absolute-value*. The basic unit of a plan is a segment (drawn as a box in a plan diagram). A segment corresponds to a unit of computation. It has a number of input ports and output ports that specify the input values it receives and the output values it produces. It has a name that indicates the operation performed. A segment can either correspond to a primitive computation (for example, the segment "−") or contain a subplan that describes the computation performed by the segment (for example, the segment ABS). All of the computation corresponding to a single program or cliché is grouped together into one outermost segment. The roles of a cliché are represented as specially-marked segments (for example, the segment NUMBER).



A Plan for the Cliché Absolute-Value.

Figure 5.

As in a flowchart, control flow from one segment to another is represented by an explicit arc from the first segment to the second (drawn as a dashed arrow). Similarly, data flow is represented by an explicit arc from the appropriate output port of the source segment to the appropriate input port of the destination segment (drawn as a solid arrow). It should be noted that, like a data flow diagram and unlike an ordinary flowchart, data flow is the dominant concept in a plan. Control flow arcs are only used where they are absolutely necessary. In Figure 5 control flow arcs are necessary in order to specify that the operation "−" is performed only when the input number is less than zero.

A key feature of the plan formalism is that it abstracts away from the syntactic features of programming languages and directly represents the semantic features of a program. Whenever possible, it eliminates features that stem from the way things must be expressed in a particular programming language, keeping only those features which are essential to the actual algorithm. For example, a plan does not represent data flow in terms of the way it could be implemented in any particular programming language, for example, with variables, nesting of expressions, or parameter passing. Rather, it just records what the net data flow is. Similarly, no information is represented about how control flow is implemented.

Abstracting away from the syntactic features of a program has several advantages. One advantage is that it makes the internal operations of KBEmacs substantially programming language independent. Another advantage is that plans are much more canonical than program text. Programs (even in different languages) that differ only in the way their data flow and control flow are implemented correspond to the same plan.

A second important feature of the plan formalism is that it tries to make information as local as possible. For example, each data flow arc represents a specific communication of data from one place to another, and, by the definition of what a data flow arc is, the other data flow arcs in the plan cannot have any effect on this. The same is true for control flow arcs. This locality makes it possible to determine what the data flow or control flow is in a particular situation by simply querying a small local portion of the plan.

The key benefit of the locality of data flow and control flow is that it gives plans the property of additivity. It is always permissible to put two plans side by side without there being any interference between them. This makes it easy for KBEmacs to create a program by combining the plans for clichés. All KBEmacs has to do is paste the pieces together. It does not have to worry about issues such as variable name conflicts, because there are no variables.

A third important feature of plans is that the intermediate segmentation breaks a plan up into regions which can be manipulated separately. In order to ensure this separability, the plan formalism is designed so that nothing outside a segment can depend on anything inside that segment. For example, all of the data flow between segments outside an intermediate segment and segments inside an intermediate segment is channeled through input and output ports attached to the intermediate segment. As a result of this and other restrictions, when modifying the plan inside a segment, one can be secure in the knowledge that these changes cannot effect any of the plan outside the segment.

One of the most powerful ideas underlying AI systems is the idea of a *representation shift*—shifting from a representation where a problem is easy to state but hard to solve to a representation that may be less obvious but in which the problem is easy to solve. Much of the power of KBEmacs is derived directly from the representation shift from program text to the plan formalism.

General-Purpose Automated Deduction

General-purpose automated deduction is best understood in contrast to reasoning performed by special-purpose procedures. In a general-purpose automated deduction system, not only the facts being reasoned about but also various theorems and other reasoning methods are represented as data objects. Only a few basic reasoning methods (for example, reasoning about equality) are built into the system. This makes it possible for a general-purpose automated deduction system to reason about a wide range of problems and to flexibly use a wide range of knowledge when doing so. In addition, such a system can be straightforwardly extended by adding new theorems and new kinds of knowledge.

In contrast, special-purpose reasoning systems typically embed theorems in procedures. Such procedures are fundamentally restricted in that each one solves a narrowly defined problem using a limited amount of knowledge. In order to attack a new problem or use additional knowledge, a new procedure has to be written.

The Programmer's Apprentice

Before looking at how these AI ideas are used in KBEmacs, it is instructive to look at how these ideas are used in the design for the PA initially proposed by Rich and Shrobe. Figure 6, reproduced from Rich and Shrobe (1978), shows the architecture initially proposed for the PA. The diagram shows four knowledge representations (in squares) and three processing modules (in ellipses) that mediate between them.

In addition to program code, the PA maintains two kinds of plans for a program. *Surface plans* represent the primitive operations in a program along with the data flow and the control flow. (They are somewhat simpler than the plans described in the last section that are used by KBEmacs.) *Deep plans* add information about specifications, the logical relationships between the parts of a program, and the design of a program. (They contain somewhat more information than the plans used by KBEmacs.)

The surface analysis module translates between program text and surface plans. The programming knowledge base contains a library of algorithmic clichés represented as plans. The recognition module analyzes a surface plan in terms of the clichés in this library and constructs a corresponding deep plan. The verification module verifies that a given deep plan satisfies its specifications. In doing so, it relies on preproven lemmas about the correctness of various clichés in the programming knowledge base. Both the recognition and verification modules are intended to be based on a general-purpose automated deduction subsystem operating in the domain of plans.

In Rich and Shrobe, the diagram in Figure 6 is accompanied by a scenario showing what the character of the interaction between the PA and a programmer might be like. This scenario uses free-form English dialog between the PA and the programmer in order to illustrate the assistantlike nature of the interaction and to suggest what kind of assistance might be possible.

The initial PA proposal focuses on describing the basic AI ideas behind the system (the assistant approach, clichés, plans, and general-purpose automated deduction in the domain of plans) and explaining why they provide important leverage on various programming tasks. However, the proposal is weak when it comes to specifics.

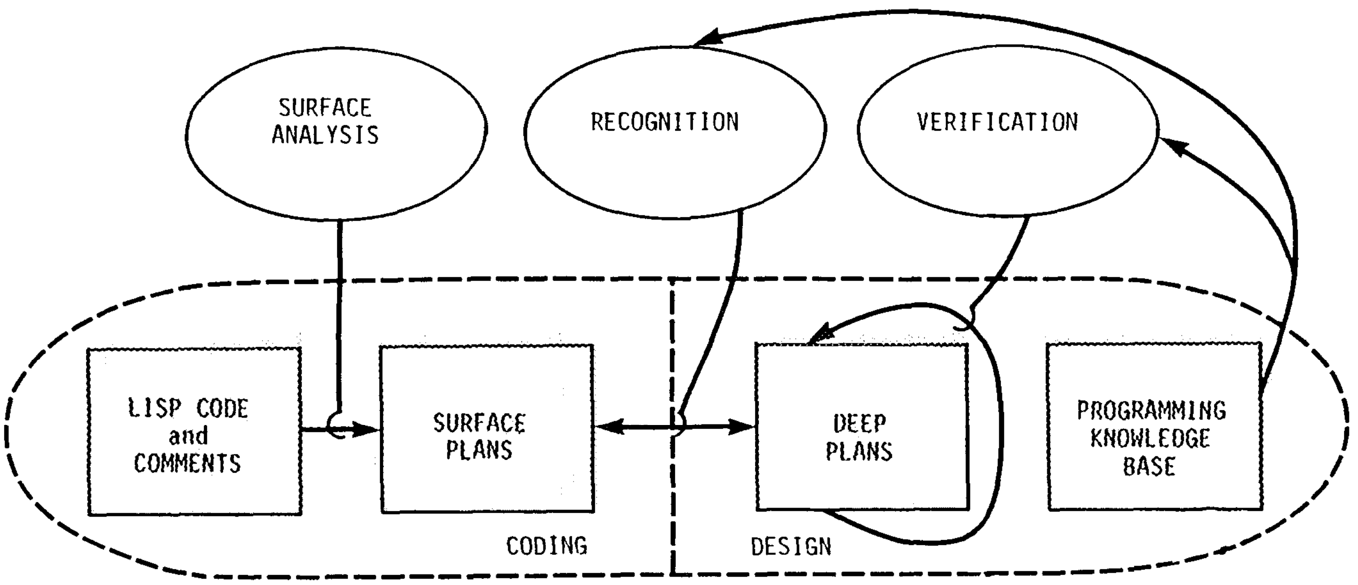
The lack of specificity in the initial PA proposal is probably due to the fact that, at the time the proposal was written, little attempt had yet been made to actually implement the PA. The only aspect of the system that was at all well developed was the plan representation. Initial exploratory efforts had been made to implement the surface analysis and verification modules. However, no attempt had been made to implement either the recognition module or the programming knowledge base.

In particular, no attempt had been made to implement a system that actually interacted with a programmer. In consequence, the proposal contained very little indication of what a practical user interface might be like or what exactly the PA would do for a programmer. (It was always clear that the interaction between the PA and a programmer would have to be more restricted in both form and content than what was shown in the scenario.)

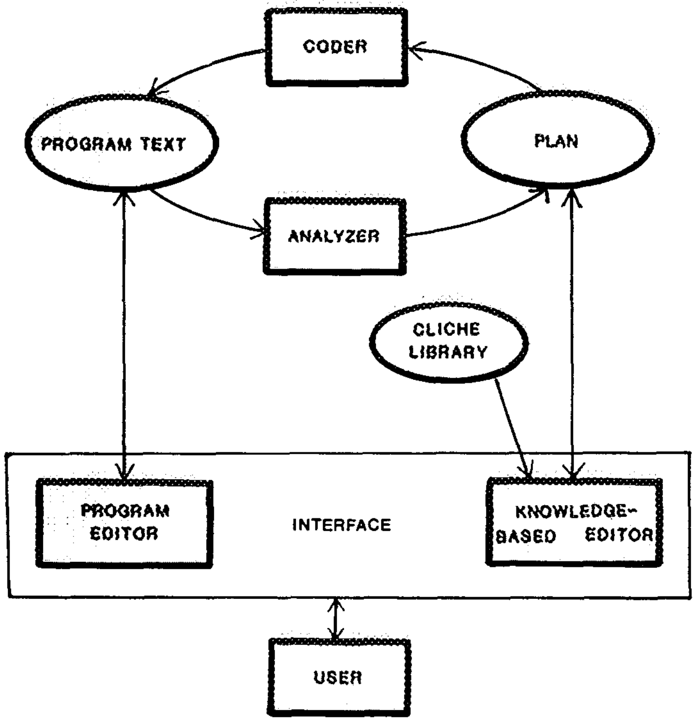
KBEmacs

The KBEmacs system is the culmination of a multiyear effort to produce a running system that exhibits some of the capabilities of the PA. KBEmacs is written in Zetalisp (1984) on the Symbolics Lisp Machine. Figure 7 shows an architectural diagram for the system.

KBEmacs maintains two representations for a program: program text and a plan. At any moment, the programmer can either directly modify the program text with a text editor or request that KBEmacs make a change to the plan by issuing a command to the knowledge-based editor phrased in terms of algorithmic clichés. An interface unifies these two modification modes so that they can both be conveniently accessed through a standard Emacs-style text editor. The analyzer is used to create a new plan whenever the program text is changed. The coder module



The Initial Architecture Proposed for the PA.
Figure 6.



The Architecture of KBEmacs.
Figure 7.

is used to create new program text whenever the plan is changed.

The major value of KBEmacs stems from the fact that it has a knowledge base of algorithmic clichés (the cliché library) and a significant amount of knowledge (procedurally embedded in the knowledge-based editor) about how to combine them. A user can build up a program rapidly and reliably by selecting various algorithms to use and delegating to the system the task of combining them together to construct a program. However, the system is nonintrusive because the user can fall back on ordinary text editing at any time.

The first thing to notice about KBEmacs is that it shares a great deal with the initial PA proposal. The overall nature of the system is based on the assistant approach. A modified version of the plan representation initially designed for the PA forms the backbone of the system. Most of the power of the system comes from the ability to shift back and forth at will between the textual representation for a program (which makes some operations easy) and the plan representation (which makes other operations easy). The major source of knowledge in the system is the library of algorithmic clichés.

The second thing to notice about KBEmacs is that it omits several features of the initial PA proposal. Except for a few isolated capabilities that are intended to be suggestive of the wider range of abilities intended for the full PA, KBEmacs focuses on the narrow task of program construction (constructing a program once the algorithms to use have been chosen). Given the narrowing of focus, several other simplifications follow. Because neither design nor verification is supported, no verification module is needed, and information about specifications and logical dependencies is omitted from the plan representation. In addition, KBEmacs does not use general-purpose automated deduction. Rather, the reasoning that is required is implemented by special-purpose procedures.

The third thing to notice about KBEmacs is that it contains several components not present in the initial PA design, that is, the coder, the knowledge-based editor, the text editor, and the interface. These components fill gaps in the initial design of the PA, providing a user interface that supports program construction.

The simplifications made during the implementation of KBEmacs were not made because the ideas involved were judged to be unnecessary for the PA as a whole but rather in the interest of getting an initial system running as soon as possible. The intention was to take a few of the most important ideas behind the PA and wring as much capability as possible out of them. This approach proved quite successful both in demonstrating the power of these ideas and in discovering more about what a usable PA should be like.

Current Status

KBEmacs is a research experiment. Rapid prototyping and rapid evolution have been the only goals of the current implementation. As a result, it is hardly surprising that KBEmacs is not fast enough, robust enough, or complete enough to be used as a practical tool.

Knowledge-based operations on large programs can take longer than 5 minutes. (A processing time of less than 2 seconds would be desirable.) KBEmacs has not been extensively tested, and there has been no visible diminution in the rate at which bugs have been discovered during this testing. This suggests that many bugs remain to be found. KBEmacs is incomplete in two primary ways. First, the system handles only about 50% of Ada. (It handles 90% of Lisp.) Second, KBEmacs knows only a few dozen clichés. (At the very least, a practical tool would need to know many hundreds of clichés.)

Although these problems are quite serious, it appears that they could be overcome by reimplementing KBEmacs from scratch, with efficiency, robustness, and completeness as primary goals. However, because KBEmacs is quite large (some 40,000 lines of Lisp code), reimplementation would be an arduous task. As a result, it has been decided that within the Programmer's Apprentice project, no attempt will be made to turn KBEmacs into a practical tool. Rather, the project will continue to focus on the fundamental research issues associated with the PA. It is hoped that some other group will eventually produce a practical tool based on the concepts behind KBEmacs.

From a research perspective, work on KBEmacs has reached a point of diminishing returns, where the restricted set of fundamental ideas it is based on has been used for essentially all it is worth. As a result, KBEmacs has been mothballed. Work has already begun on a new system that will combine the features of KBEmacs and the initial PA proposal. The principal improvements in this new system will be an extended plan representation, Rich (1981), and a general-purpose automated deduction module, Rich (1985).

Where's the AI?

Returning to the original question, KBEmacs is based on three AI ideas: the assistant approach, clichés, and plans. These ideas are the source of essentially all its power.

In addition, several of the individual modules in KBEmacs use AI techniques. The knowledge-based editor performs a considerable amount of reasoning about plans (albeit procedurally embedded). A simple constraint system is used in conjunction with the algorithmic clichés in the cliché library in order to propagate some of the effects of design decisions. The coder uses simple planning and constraint propagation in order to balance competing suggestions of what variables to use in a program and other aesthetic considerations.

Building an Applied AI System

It is interesting to look at the development of KBEmacs as a case study in the building of applied AI systems. The work on KBEmacs can be divided into three (somewhat intermixed) phases: thought experiments, implementation, and concrete experiments. The thought experiment phase investigated the basic AI ideas that were needed and laid out a tentative design for the system. This phase culminated in the production of the initial PA proposal.

Once the decision was made to implement a system that actually did something, a shift of focus occurred away from pursuing particular AI ideas to achieving at least part of the goal by any means. Throughout the implementation phase, pragmatic non-AI problems tended to swamp all other considerations. For example, well over half of all the effort expended on KBEmacs went into implementing an analyzer and coder that would make it possible for KBEmacs to operate on programs of realistic size and complexity. The implementation phase was typified by the deliberate suppression of details in the interest of trying to get directly at the heart of the problem.

Once parts of KBEmacs started to work and concrete experiments began, it was again possible to focus on larger issues. As an example of the kind of changes that occurred during the experimentation phase, it is interesting to note that there was a significant evolution in the way the system interacted with the programmer. Probably due to their importance to the internal operation of the system, plans were initially assigned a prominent position in the user interface. However, experimentation revealed that programmers were much happier thinking in terms of program text and were somewhat confused by plans. As a result, the interface was modified so that, as much as possible, everything appears to be happening in terms of program text.

In retrospect, the development of KBEmacs was a valuable research experiment. However, mundane implementation absorbed much more effort than anyone would have liked. Looking at other systems that have successfully applied AI techniques, the development of KBEmacs does not appear atypical with regard to either point. In particular, it seems to be an unfortunate fact of life that most of the effort expended on almost any full-scale system is directed toward the solution of relatively uninteresting pragmatic problems.

References

Rich, C (1981) *A formal representation for plans in the programmer's apprentice.* IJCAI 7, 1044-1052

Rich, C (1985) *The layered architecture of a system for reasoning about programs.* IJCAI 9, 540-546.

Rich, C. and Shrobe, H (1978) Initial report on a lisp programmer's apprentice *IEEE Transactions on Software Engineering* 4:456-466

Waters, R C. (1985) The programmer's apprentice: A session with KBEmacs *IEEE, Transactions on Software Engineering* 11:1296-1320.

Zetalisp (1984) Lisp machine documentation (release 4) Cambridge, Mass.: Symbolics, Inc.

REQUEST FOR PROPOSALS

Future AAAI Conference Sites

The AAAI's Conference Committee (Jay M. Tenenbaum, Chair; Ronald Brachman, and Michael Genesereth) requests proposals from the membership for conference sites for 1988, 1990, and 1991.

The proposal should be structured around the new five day format described elsewhere in this issue of the AI Magazine. Based on a predictive attendance of 6,500, the proposals should include the following information:

1. Description of the local AI community and its willingness to support the conference.
2. Description of the variety of available housing ranging from first class hotel rooms to dormitories.
3. Description of the University and/or Convention Center's large meeting rooms (ranging from 300 to 3,500 theater seating) for a minimum of three parallel sessions. Description of another set of three, parallel meeting rooms (used for tutorials) that can accomodate from 200 to 500 schoolroom seating each.
4. Description of available exhibit space (minimum requirement of 80,000 net square feet) and local service contractors.
5. Description of local regulations (*e.g.*, labor union laws, liquor licenses, and local tax structure).
6. Description of local housing and convention support services from the city's Convention and Visitors Bureau. Description of procedures for processing university housing reservations.
7. Description of site's accessibility by air and ground transportation and local ground support transportation.

Ideally, the Conference Committee would prefer to hold the science sessions on a university campus and the engineering sessions at the larger convention facility.

For further details about this RFP, please contact:

Ms. Lorraine Cooper
AAAI
445 Burgess Drive
Menlo Park, CA 94025-3496

Submit all proposals to:

Jay M. Tenenbaum, Chair
AAAI Conference Committee
445 Burgess Drive
Menlo Park, CA 94025-3496