MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

Working Paper 202 July 1980

A Proposal for Sniffer: a System that Understands Bugs

by Daniel G. Shapiro

Abstract:

This paper proposes an interactive debugging aid that exhibits a deep understanding of a narrow class of bugs. This system, called Sniffer, will be able to find and identify errors, and explain them in terms which are relevant to the programmer. Sniffer is knowledgeable about side-effects. It is capable of citing the data which was in effect at the time an error became manifest.

The debugging knowledge in Sniffer is organized as a collection of independent experts which know about particular errors. The experts (sniffers) perform their function by applying a feature recognition process to the text for the program, and to the events which took place during the execution of the code. No deductive machinery is involved. The experts are supported by two systems; the cliche finder which identifies small portions of algorithms from a plan for the code, and the time rover which provides complete access to all program states that ever existed.

Sniffer is embedded in a run-time debugging aid. The user of the system interacts with the debugger to focus attention onto a manageable subset of the code, and then submits a complaint to the sniffer system that describes the behavior which was desired. Sniffer outputs a detailed report about any error which is discovered.

CONTENTS

FIGURES

1. Introduction

This proposal outlines an investigation into the area of program understanding within artificial intelligence. The focus is the topic of debugging, by which I mean the recognition and analysis of errors which occur during the execution of code. This work is concerned with bugs in arbitrary programs, meaning programs not restricted by size, complexity, or domain of application. The expertise which the system contains is about errors in typical coding tasks. It does not contain background knowledge about the domain of the particular program being examined.

The end result of my work will be an automated assistant for locating bugs, called Sniffer, that displays a deep understanding of a narrow class of errors. Starting from a complaint supplied by the user, this system is able to trace a bug to its source, and describe the error in terms of the intended purpose of the responsible code. Sniffer is knowledgeable about side-effects; it is capable of explaining how and why a bug occurred by examining the data which existed at that time in the program's execution, and by examining the execution sequence which was actually followed.

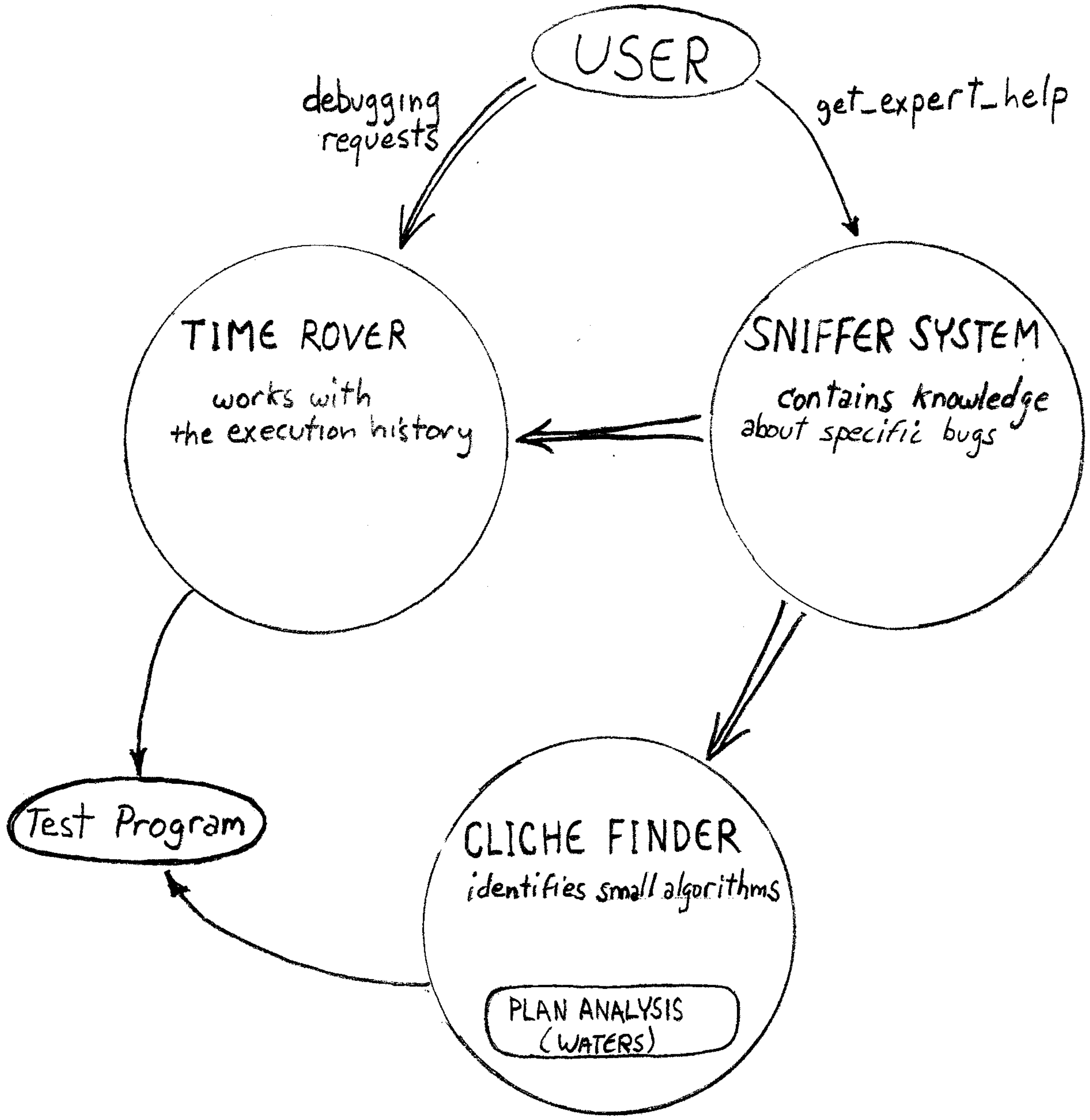
Sniffer has three major components; the sniffer system, which contains all the information relevant to recognizing specific bugs, the time rover, which supports queries about a program's history, and the cliche finder, which identifies fragments of algorithms in programs that are later used as a basis for recognizing errors. (See figure 1). Sniffer is embedded in a run-time debugging aid. The user of the system is called on to perform the initial stages of debugging. He/she employs the facilities of the time rover to roughly locate the error, and then invokes the sniffer system with a complaint that describes the behavior which was desired.

The sniffer system is organized as a collection of experts that know about specific bugs. Each one can examine the suspect code, the user-supplied complaint, and the execution history of the program to determine if the bug it knows about is present. No generalized deduction method is involved in this process; the sniffers locate errors by applying a pattern recognition style analysis to the sequence of events which caused the bug. This analysis is supported by the cliche finder, which in turn is supported by a system, written by Richard Waters [Waters 1978], that transforms code into a regular, and language independent representation called a PLAN. All the examination performed by the sniffers uses this format. As a result, the sniffers can analyze code written in any source language.

The sniffer system will be implemented with a demon invocation control structure. In this format, each sniffer waits for a particular set of features to be identified. Whenever that set is present, the associated bug expert is executed. The feature detectors (called triggers) discover facts by examining the program text, the complaint and the recent portions of the execution history. The triggers perform inexpensive tests. The sniffers perform whatever hard tests are required.

The cliche finder is constructed as a pattern matcher that recognizes common subtasks in the PLAN language representation of programs. The objective of the system is to raise the level of

Fig. 1. The Design of Sniffer



discourse about a program. Rather than talk about "car" and "cdr" operations, the cliche finder makes it possible to speak about aggregates the size of variable interchanges or list enumerations. The cliche finder operates on the primitive structures of the PLAN language, which include an explicit representation for the data and control flow within a program, and a taxonomy for the building blocks of all recursive and iterative routines.

The time rover monitors the execution of the test program (the program undergoing analysis). It records both control information, and the succession of values acquired by all data objects in the code. At every instance of a side-effect operation, the system deposits a record, preserving the old data. The result is a complete picture of the program's state as it evolves through time. The information in this trace is sufficient to rewind the program to an earlier point, or to run it backwards if that is desired. The time rover also provides some sophisticated methods for accessing this data, which both the user and the automated aides can employ. In this system, which will be implemented in lisp, the time rover can apply any lisp expression as if it occurred at an arbitrary moment during the test program's execution.

A general scenario for use of Sniffer is as follows: the user is sitting at a terminal, watching a program run. At some point, he becomes aware that the output is incorrect, although the program is still functioning. He stops the execution and investigates the problem using the facilities of the time rover. He might examine the order of function calls on the stack, the values of several parameters, or events and data in procedures *which were invoked and which successfully returned* some time ago. Eventually, the user finds a particular execution of a region of code which seems to contain a problem. He makes a complaint to the sniffer system, of the form

(get_expert_help *expected-result time-t this-code-region*)

The sniffer system analyzes the code for *expected-result* and for *this-code-region* to obtain a quick understanding of the type of the error. All the relevant sniffers are then invoked.

A sniffer might look at a specific execution of a nested conditional, or compare the values in a list before and after a function was called, or ask the user for further information. If the bug the sniffer knows about is present, it outputs a detailed error report. This report includes a description of the problem in high level terms, and a detailed analysis of the events which caused the error. I will present a detailed scenario after each of the components of this research have been discussed in more depth.

Sniffer will be implemented in Lisp on the MIT lisp machine. I have chosen the lisp machine because it has the high speed and large memory capacity required by Sniffer. The programs submitted to the the system will also be written in lisp. This decision simplifies implementation considerations, although it restricts the programs which can be analyzed to the world of lisp. The focus of this research remains in language independent techniques.

1.1 The Domain of Errors

The first set of sniffers that I will construct will be concerned with errors involving header cells of list structures. There are a number of bugs in this category. For example, consider the following scenario:

Suppose there is a list insertion routine

```
(insert item the-list)
```

which supposedly performs its function by side effect, that is, by splicing *item* into the list. However, in the case where *item* is to become the first member of *the-list*, the routine returns a flag, or a new list instead. If the caller of insert expects it to function by side effect, every invocation with data destined for the top of the list will be unsuccessful.

In more general terms, this class of bugs concerns conflicting assumptions over what constitutes data objects, and who maintains them. In this case, the caller of insert expected the function to be responsible for the list's integrity, but this was not true. Insert was designed without a blank first record that could be side effected in the event that the data to be inserted belonged at the top of the list.

Sniffer as a whole is not limited to bugs of this type. Its mechanisms allow a wide range of errors to be detected. I chose this category because the errors it involves are an excellent vehicle for demonstrating machine understanding. The area includes bugs which are hard for people to analyze and correct, and they are generally subtle due to the presence of side effects. This shows off the abilities of Sniffer to the best advantage.

2. The Time Rover

The purpose of the time roving facility is to allow the user, and the bug experts, to query the history of the program undergoing analysis. Some typical questions might be: When was the function *Foo* last evaluated? Relative to now, when will the variable *C* cease being 6? What would be the result of the following test, if applied to the array *A* at the time when *sort-key* was 16? To support these kinds of questions, the time rover makes it possible to execute arbitrary expressions in the program environments when previous states were in effect. In this system, the entire expressive power of lisp is available for forming these requests.

The best way to explain the issues involved in time roving is to discuss its implementation. For the purposes of explanation, I will describe a particularly simple, but straightforward mechanism. The mechanism which I actually implement will be more efficient.

2.1 Terminology

The <u>execution</u> <u>history</u> of a program refers to the sum total of events which occurred while it was running; the flow of control, the sequence of side effect operations, etc. The <u>execution</u> <u>trace</u> refers to the physical structures which I use to represent the execution history.

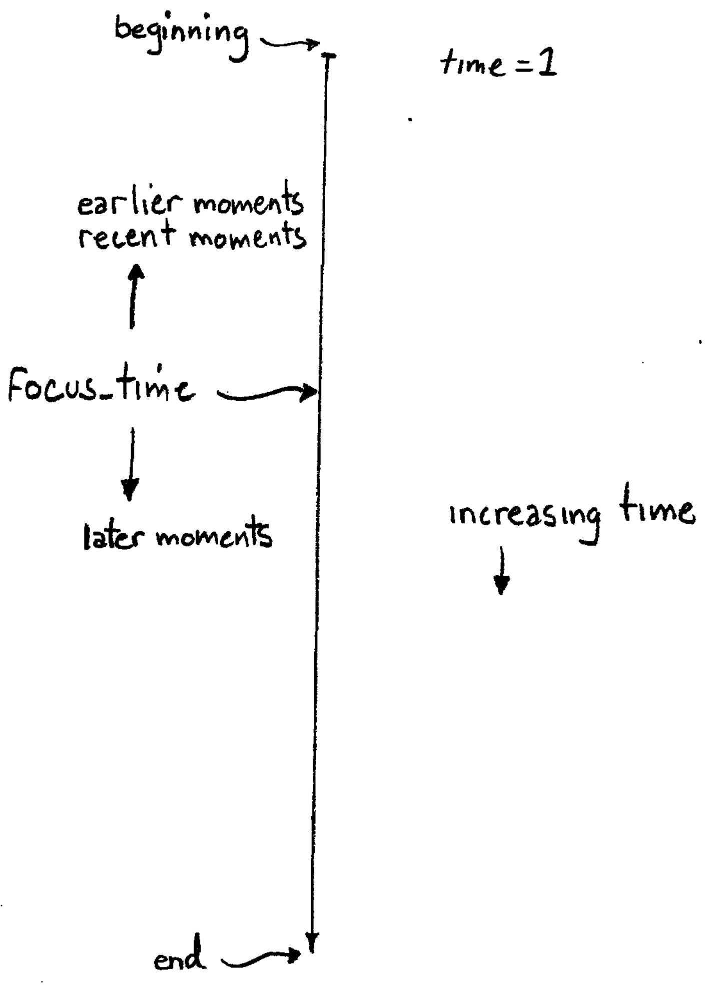
Within an execution history there are various named times, or <u>moments</u>. <u>Time</u> can ordinarily be thought of as an integer. It starts at 1 and increases monotonically as execution progresses. The *beginning* and the *end* refer to the first and the last moments during the execution of the user's program. The *focus_time* is the focus of attention in the execution history. There is also a convention for naming directions. Figure 2 illustrates these ideas. The <u>time-environment</u> is an abstract object in which one can look up the variable bindings, properties, etc., that are in effect at some moment. For lack of a better method, all moments will be referred to in the present tense.

2.2 Implementation

The time rover is divided into two components, called the *keeper* and the *seer*. The keeper is responsible for developing the execution trace for the test program, while the seer listens to the user's debugging requests and answers his questions concerning the history of the code. Both the keeper and the seer are constructed as evaluators for lisp which have been modified to accomplish specific tasks.

The keeper is designed to make it possible to use lisp to query the history of the test program. It is implemented in terms of the data structures of the execution trace. This evaluator provides the

Fig. 2. Vocabulary for discussing time travel



normal capabilities of lisp, but deals with a wider class of objects that include the previous versions of data. The keeper is used in two distinct ways. First, it runs the test program and generates the execution trace. Second, it is employed by the seer (during the debugging session) to execute a class of requests that augment the data in the execution trace.

The seer provides the user with a workspace for writing functions and for executing requests that access the history of the code. This evaluator has been augmented to operate on time-stamped data which refers to information from the execution trace, as well as normal lisp lists which are maintained in the environment of the seer. The use of time-stamped objects makes it possible to execute lisp expressions as if they occurred at arbitrary moments during the initial evaluation of the test program.

2.3 The keeper

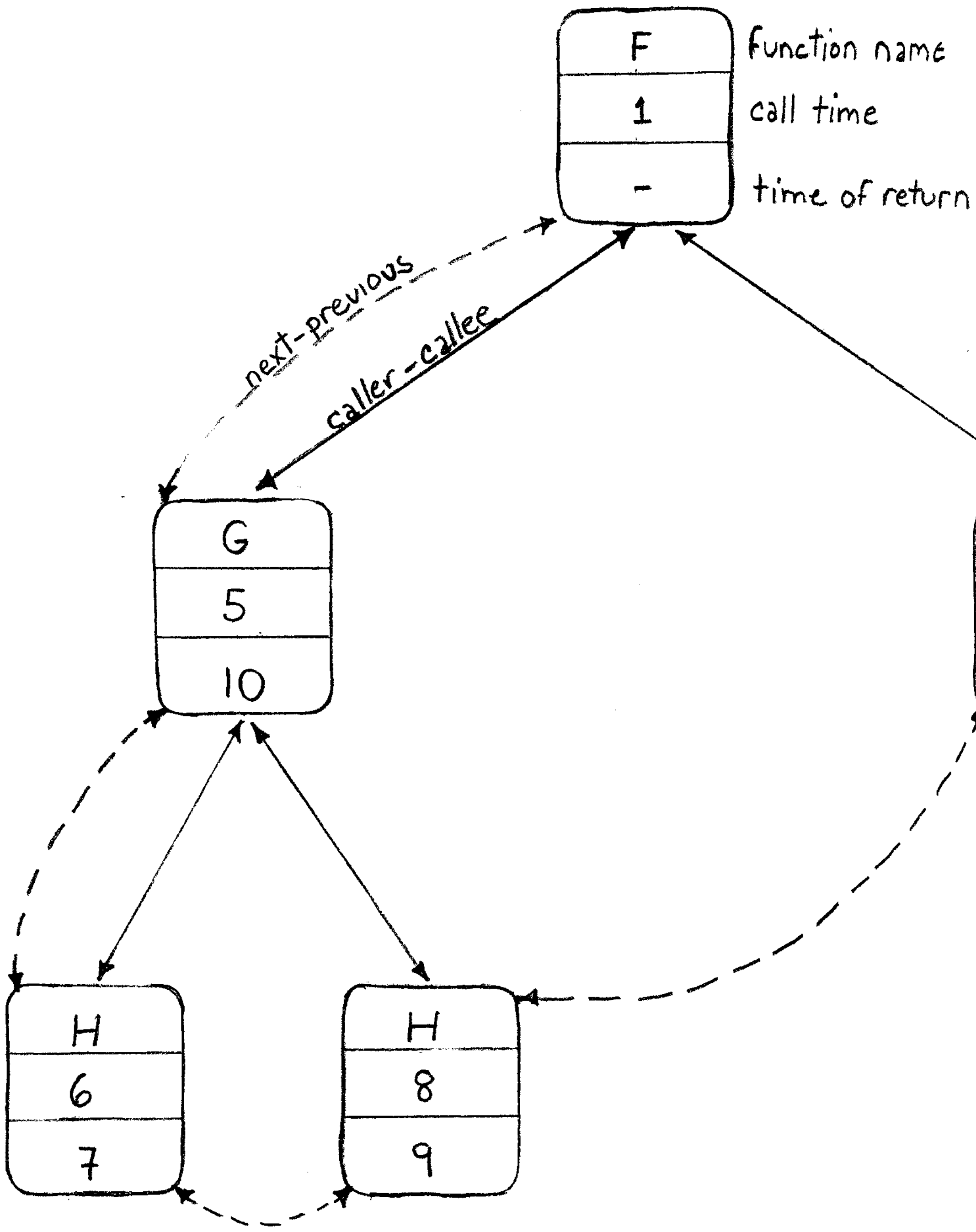
The keeper remembers the previous versions of the test program's state with two data structures; the execution tree, and the incarnation array. The execution tree records information about the flow of control, and the incarnation array stores the history of the data objects used in the test program.

The execution tree records function invocations (both the ones which have returned and the invocations which are still active). The tree is a straightforward extension of the lisp stack. In typical stack mechanisms, one stack frame is written for each lambda expression that is executed. In the execution tree, only the top level functions in the code are recorded (the internals of primitives are ignored), but none of these frames are thrown away. If a series of snapshots were taken during execution, they would show the tree grow with nested invocations, return to an earlier spot when a function returns, and then grow a new branch as more functions are called. The result is a tree structure showing the history of function applications in the program. (See figure 3.) The frames will be threaded with bi-directional links to allow attention to move backwards and forwards across caller-callee relationships. They will also be threaded by order of their invocation time, to allow linear motions in time. (Functions which are adjacent in time are not necessarily adjacent in the tree. Again, see figure 3.)

The incarnation array records the sequence of values which each data cell acquires during the execution of the test program. This information is stored in terms of [name, value] pairs, called trace-cells. A trace-cell is an immutable object that records the contents of a cons (or an atom value cell) at a particular time. This approach is necessary because side effect operations destroy the previous contents of cells. In order to be able to recreate the states of a program, the keeper cannot forget this information. Name-value pairs are required because in the presence of many versions of a cell, a single address is insufficient to distinguish one copy from another. To identify a trace-cell, both a cell-id and a time (at which to look it up) are required.

Trace-cells are not values for lisp objects. They are an implementation device for storing the information which is required to tell one version of cons from another. (Trace-cells are invisible to the programmer.) In the keeper, a value is a cell-id (just as an address is a value in normal lisp). Since a cell-id does not completely identify a trace-cell, the data associated with a given id (at a given time) is found by scanning the incarnation array for the most recent trace-cell with the appropriate name. (The incarnation array is a time-ordered collection of trace-cells.) This search fulfills a role which is exactly analogous to looking up an address in normal lisp. During the initial execution of

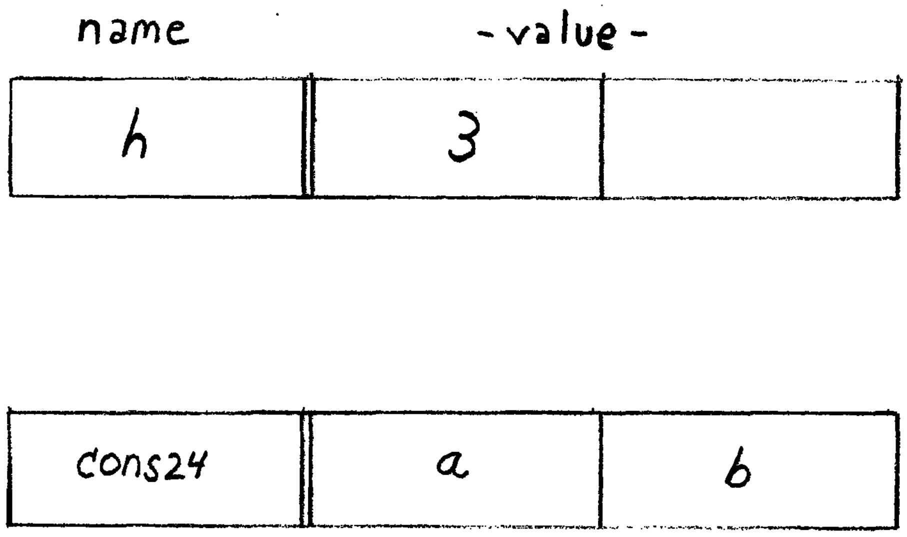
Fig. 3. An execution tree
The solid lines indicate call and return relationships between functions. The dotted lines show the execution order of these functions. Execution trees tend to be very bushy in iterative programs and very deep in recursive code.



the test program, the current execution time is used as the starting point for scanning the incarnation array. During debugging, that time is supplied by the seer.

The primitive operations of lisp are modified to accommodate trace-cells. The functions which produce side effects cause trace-cells to be deposited, and the implementations for the information obtaining operations, *car,cdr,* and *symeval* are modified to access these structures. (I will discuss the new versions of *eq* and *equal* in a later section.) For example, (see figure 4)

Fig. 4. Some example trace-cells



the function *setq* in the statement

```
(setq h 3)
```

produces a trace-cell which contains a name field equal to the atom name, *h,* and a value field composed of the number 3. (Atom names function as cell-ids, and the name for a number *is* that number. Numbers do not require trace-cells because their values cannot change.) The function *cons* in the statement

```
        (cons 'a 'b)
```

results in the trace-cell [cons24, a.b], which indicates that the value associated with the cell-id, *cons24* at the current time is the (traditional) cons of the cell-ids *a* and *b*. (The *cons* function is a side-effect operation because it allocates storage where none was required before.) The functions *rplaca* and *rplacd* create similar trace-cells, except that the name field is the same as the one from the cell which is being updated.

The operations *car*, *cdr*, and *symeval* each map a cell-id to another cell-id. They all involve an identical scan through the trace-cells in the incarnation array. The *car* of a cell-id is the *car* of the value part of the corresponding trace-cell (the one in effect at the current time). The *cdr* of a cell-id is the *cdr* of the corresponding value part. The function *symeval* takes in a cell-id (which should be an atom name), scans the incarnation array for the most recent trace-cell with that name, and outputs the value part of the trace-cell which is discovered.

The function "@" is a useful utility for evaluating an atom at a specific time. The effect of the statement

(@ *time* '*atom-name*)

is to invoke the eval function of the keeper on the atom-name provided, in the context of the time specified.

2.3.1 An example of the evaluation process

Figure 5 shows a collection of snapshots of the incarnation array as each of the statements

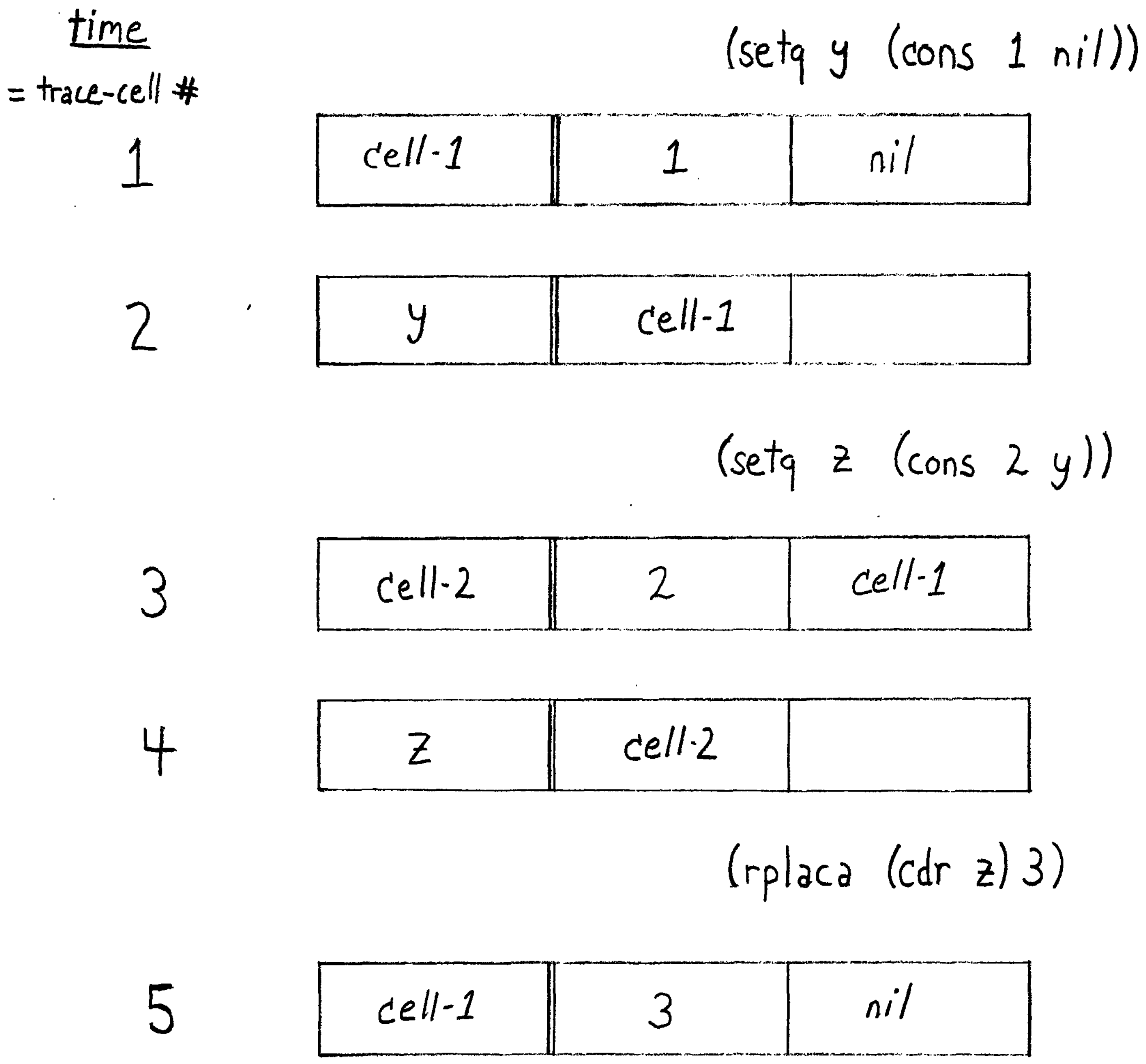
```
(setq y (cons 1 nil))
(setq z (cons 2 y))
(rplaca (cdr z) 3)
```

are executed.

The first event is the creation of the trace-cell for "(cons 1 nil)". The cell-id field is arbitrarily set to *cell-1*, and the trace-cell is deposited at time 1 as well. The data part is identical to the normal lisp representation for a cons. The setq operation deposits a trace-cell with the name field *y*, and a data part consisting of the cell-id, *cell-1*.[1] No pointers are involved. Similarly, in the trace-cell which is

1. I have simplified the picture a bit by discarding some of the details involved in representing atoms. Lisp normally requires storage for an atom's print name as well as its value, function definition and property list. There are also separate pointer types that identify atom definitions, numbers, and atoms. The representation used in the implementation of the keeper will take account of these facts.

Fig. 5. The development of the incarnation array during execution



deposited by "(cons 2 y)", the value of y is represented by *cell-1* again. This process continues until "(rplaca (cdr z) 3)" is evaluated. In normal lisp, this side-effect would have changed the value of an existing cell. In the keeper, a new trace-cell is deposited with the same cell-id, *cell-1.*

In order to evaluate lisp expressions, the keeper has to find the appropriate trace-cell every time a cell-id is referenced (there may be many with the same name). This is accomplished by searching the incarnation array for the most recent trace-cell which has the desired cell-id in the name field. For a simple example, in figure 6, the value of y at time-2 is found from trace-cell #2 to be the cell-id, *cell-1.* To print out the value of y, the contents of *cell-1* at time-2 have to be printed. Since there were no side effects between time 1 and time 2, trace-cell #1 is the correct result. The list "(1)" is printed.

In order to evaluate the predicate "(@ time-5 'y)" the keeper has to discover that y was changed by an indirect side effect through z. This process is accomplished as follows. Starting from time-5, the keeper looks for the most recent setq record for the atom y. This shows that *cell-1* was the value of y at <u>time-2</u>. Starting again at time-5, the keeper finds the most recent version of *cell-1.* This turns out to be in trace-cell #5 that contains the list "(3)". In order to print out the elements of a list, each of the cell-ids involved has to be interpreted in the same fashion. For example, "(@ time-4 'z)" is a list built from trace-cells #3 and #1 (the list "(2 1)"), and from trace-cells #3 and #5 at time-5, corresponding to the list "(2 3)".

A useful way of thinking about the incarnation array is to treat it as though it contained discreet time-environments. In this model, each cell-id may have a "binding" in several time-environments, and given a time, the proper trace-cell can be found.

The *end* time-environment is an environment of special interest. When the keeper is used to evaluate a normal lisp program (as opposed to running debugging requests), it only references data in the *end* time-environment. The keeper's speed can be enhanced by maintaining a separate table which always contains this environment. This table, called the <u>now-array</u>, is a mapping of cell-ids to their most recently defined trace-cells. In different words, the keeper represents the state of a running program as the shallow bindings of all cell-ids to their current trace-cells. (Furthermore, since cell-ids can be chosen at random, they can be set up as indices into successive memory locations of the now-array. This means that no searching is required to access a trace-cell in this environment.) The keeper updates the now-array and the incarnation array every time a side effect occurs.

The time-environment which the now-array represents can be changed by recalculating the shallow bindings of all the cell-ids in the program. This is likely to be an expensive process but it opens up several possibilities. First, if many debugging requests are going to concentrate on a particular time, the overall speed of the system can be increased. The second gain is that this makes it easy to restart a program from an arbitrary moment in its past, after its data structures have been examined or patched.

The memory requirement for the keeper is very large (it is proportional to the duration of execution, not the size of a program). If this threatens to exceed the capacity of the lisp machine, it would be possible to "forget" about certain portions of the execution history. These regions would become opaque to the time rover.

2.3.2 Equality and coreference

The concepts of equality and coreference have to be extended to fit an environment where many versions of data cells are available simultaneously. In normal lisp, there are only two ways to compare objects. One can ask if they are *eq*, meaning that they have the same name or address (which is equivalent to asking if they are coreferent), or if they are *equal*, meaning that they contain isomorphic data structures.

In the keeper, more distinctions are available. One can ask if two trace-cells are identical (I call this test *unmodified*), or if two cell-ids are the same (*eq*). See figure 5. These questions arise when objects are compared across times. For example,

```
(eq (@ time-2 'y) (@ time-5 'y))
```

is true. Here, the list contained in y is different at the two times although the top level cell-id which is the *value* of y is *cell-1* in both cases. The statement

```
(unmodified (@ time-2 'y)(@ time-5 'y))
```

is false. This test shows that y was modified between the two times.

When these predicates are extended to lists, one can ask if two lists contain the same cell-ids at every level (called *eq**), or if they involve the same trace-cells at every node (*unmodified**). *Unmodified** is the coreference test in the time roving environment. *Eq** is a weaker function. For example, suppose that the initial contents of the variable y in figure 5 are reinstated by having the test program execute the statement

```
(rplaca (cdr z) 1)
```

at time-6. In this case, the expression

```
(eq* (@ time-6 'z)(@ time-4 'z))
```

is true, but

```
(unmodified* (@ time-6 'z)(@ time-4 'z))
```

is false.

Note that two lists are not necessarily identical if their top level trace-cells are the same. There is always the possibility that some internal cell has changed across the two times. From figure 5,

```
        (unmodified (@ time-5 'z)(@ time-4 'z))
```

is true, but

```
        (unmodified* (@ time-5 'z)(@ time-4 'z))
```

is false.

The function *equal* remains essentially unchanged in the context of the keeper. It still tests for isomorphism of structure. There is no requirement that the lists share the same trace-cells or even that the same cell-ids are involved. The atoms at the leaf nodes of the tree must be identical.

The relationship between these functions is summarized in figure 6.

Fig. 6. The heirarchy of equality tests
The equality tests for lists are stronger than the analogous tests on cell-ids; *eq** implies *eq* and *unmodified** implies *unmodified.* The converse is not true. Among the list functions, *unmodified** implies *eq**, and *eq** implies *equal* (since lists with corresponding cell-ids have the same atoms). The function *equal* does not imply any other form of equality; two lists which contain the same integers do not have to be related in any other way.

$$\begin{tikzcd}
 & \text{equal} \\
\text{eq} & \text{eq}* \arrow[l] \arrow[u] \\
\text{unmodified} \arrow[u] & \text{unmodified}* \arrow[l] \arrow[u]
\end{tikzcd}$$

implications not shown are not present

2.4 The seer

The function of the seer is to provide the user with a uniform mechanism for operating on data from the execution trace, and for manipulating objects which are defined in his own local debugging environment. The seer is constructed as an evaluator for lisp that is extended to contain time-stamped objects, called *t-pairs*, which refer to data from the incarnation array.

A *t-pair* contains two parts, a *reference time* and a *cell-id* where the reference time specifies the time-environment to use for interpreting the cell name. Reference times are sticky, in the sense that the *car* of a t-pair is another t-pair with the same reference part. This approach allows the user to change the perspective used to view an entire lisp object by altering the reference time attached to its topmost cell-id. A t-pair is represented by a bracketed pair of the form {*time id*}.

The primitive operations of the seer are modified to accommodate this new data type. If a primitive is called on a normal lisp object, then it is evaluated in the normal way (this might yield a t-pair). When a primitive is applied to a t-pair, it is evaluated with the aid of the corresponding operation of the keeper. For example, from figure 5, *symeval* of {time-4 z} is the t-pair {time-4 cell-2} where cell-2 was obtained by applying the keeper's *symeval* function to z at time-4.

The function "@" (defined earlier as a utility which invokes the keeper's evaluator on a lisp form) can be used to state the effect of these primitives in a more concise form.

```
(symeval {t id}) => {t (@ t (symeval id))}
(car {t id}) => {t (@ t (car id))}
(cdr {t id}) => {t (@ t (cdr id))}
```

This notation is intended to express the fact that each of the information obtaining primitives of lisp maps a t-pair into another t-pair with the same reference time. Since the keeper's evaluator is the only system that can examine the incarnation array, the keeper is used to obtain the cell-id which is the result of the given primitive applied at the reference time.

An additional primitive of the seer, called *cts* (for change time stamp), alters the time environment used to interpret a cell-id. It operates by simply substituting the existing reference time for a new one. For example,

```
(cts time-5 {time-2 y}) => {time-5 y}
```

which evaluates (in figure 5) to "(3)" instead of "(1)".

It is not immediately clear how to interpret the application of a side effecting primitive to a time-stamped object. The issue is that a t-pair refers to an object from the history of the test program which was never subjected to the side effect that the user is requesting. (Information obtaining operations are benign in this sense. They have no potential for altering the data in the trace.) If the execution trace is intended to record the actual history of the program, how can side effects created by the debugger be factored in?[1]

The approach I take is to interpret all debugging requests that access the history of the code as explorations into alternate time-tracks for the test program's development. These debugging requests are processed as if the test program executed them at the specified time. For example, in the context of figure 5, the effect of the statement

```
(@ time-4 '(rplacd (cdr z) 1))
```

is to grow a branch off of the incarnation array at time-4 (forming an incarnation tree) and to deposit a trace-cell for *cell-1* at that time. The side effects created by the functions *setq, cons,* and *rplaca* are handled in a similar way. (See figure 7.)

This approach implies a small redefinition of the function "@". I have described @ as a utility for invoking the evaluator of the keeper. To be more specific, @, in the statement

(@ *time* '*expression*)

instructs the keeper to form a branch in the incarnation tree, and then hands the *expression* to the keeper to be evaluated in the context of the time-environment defined by *time.* (The seer evaluates the parameters to @.) @ returns a t-pair which packages together the cell-id returned by the keeper and the time at which the keeper finishes its evaluation. Time is now interpreted as a pointer into the incarnation tree, which links trace-cells in an upwards chaining fashion.

On a more subtle note, if the keeper is to be able to process the list which defines the function submitted by @, the list has to be copied into the trace-cell format which the keeper manipulates. (In fact, the list is copied into the beginning of the alternate time track where execution will take place.) Note that this makes it impossible for the user to assign a variable in the keeper to data from the debugging environment. The statements (in the seer)

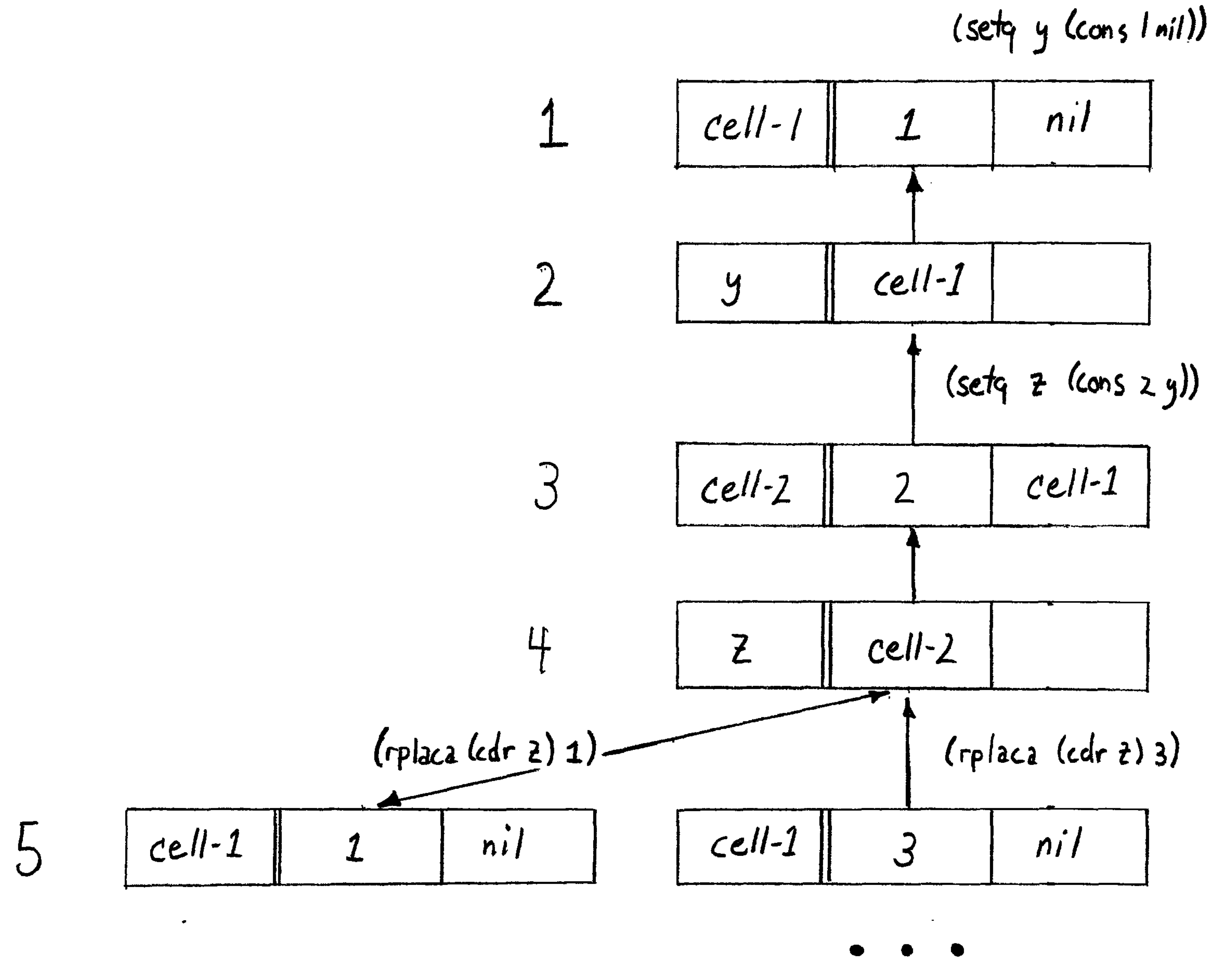
```
(setq D '(a b c))
(@ time-2 '(setq y D))
```

will result in an error when the keeper attempts to *symeval* D at time-2, assuming D is not defined in the context of the test program at that time. Although it represents an error in this example, in some cases it is desirable to pass a free variable into a time-environment within the test program.

This definition of @ makes it possible to express the action of the seer's primitives on t-pairs by

1. There are many very confusing ways to resolve this question. If the debugging session is considered to occur after the test program is executed then a side-effect to a variable, say at time-10, would actually occur at a moment which is later than any moment in the execution history. This implies that a request which accesses the supposedly side-effected data at time-11 finds that nothing has changed.

Fig. 7. An example of an alternate time-track
This figure shows the creation of a branch in the execution history in response to the code statements shown.



the following rewriting rules.

```
(symeval {time id}) => (@ time '(symeval id))
(car {time id}) => (@ time '(car id))
(cdr {time id}) => (@ time '(cdr id))
(setq {time id} x) => (@ time '(setq id x))
(rplaca {time id} x) => (@ time '(rplaca id x))
(rplacd {time id} x) => (@ time '(rplacd id x))
```

The information obtaining operations create degenerate branches of the incarnation array (the time does not increase), and the side effecting operations augment the data in the trace. Note that the *cons* of two t-pairs within the seer is not implemented in terms of the keeper's primitives. The statement

```
(cons (@ time-4 'z)(@ time-2 'y))
```

simply creates a cons cell in the environment of the seer which contains the resulting t-pairs.

2.5 A summary of the keeper and the seer

The keeper and the seer define a mechanism that allows the user to execute and then examine the history of a test program. The keeper creates the execution history, and evaluates any requests submitted by the seer which access that data. The seer provides the user with a lisp environment for executing debugging requests. It answers questions about the execution history by employing the facilities of the keeper. Figure 8 shows the relationship of these systems.

The overall picture which the system presents has the user's debugging requests occurring in a kind of a super-time which is not ordered with respect to the execution history. From the user's perspective, all of the information in the trace is equally accessible.

The use of alternate time tracks makes it possible to move to moments in the test program's past and evaluate arbitrary lisp expressions in that context. The user can define and execute functions in any time-environment, or explore hypotheses about the test program's behavior by re-executing portions of the code on modified data. The alternate histories which these actions create can themselves be investigated in the same manner.

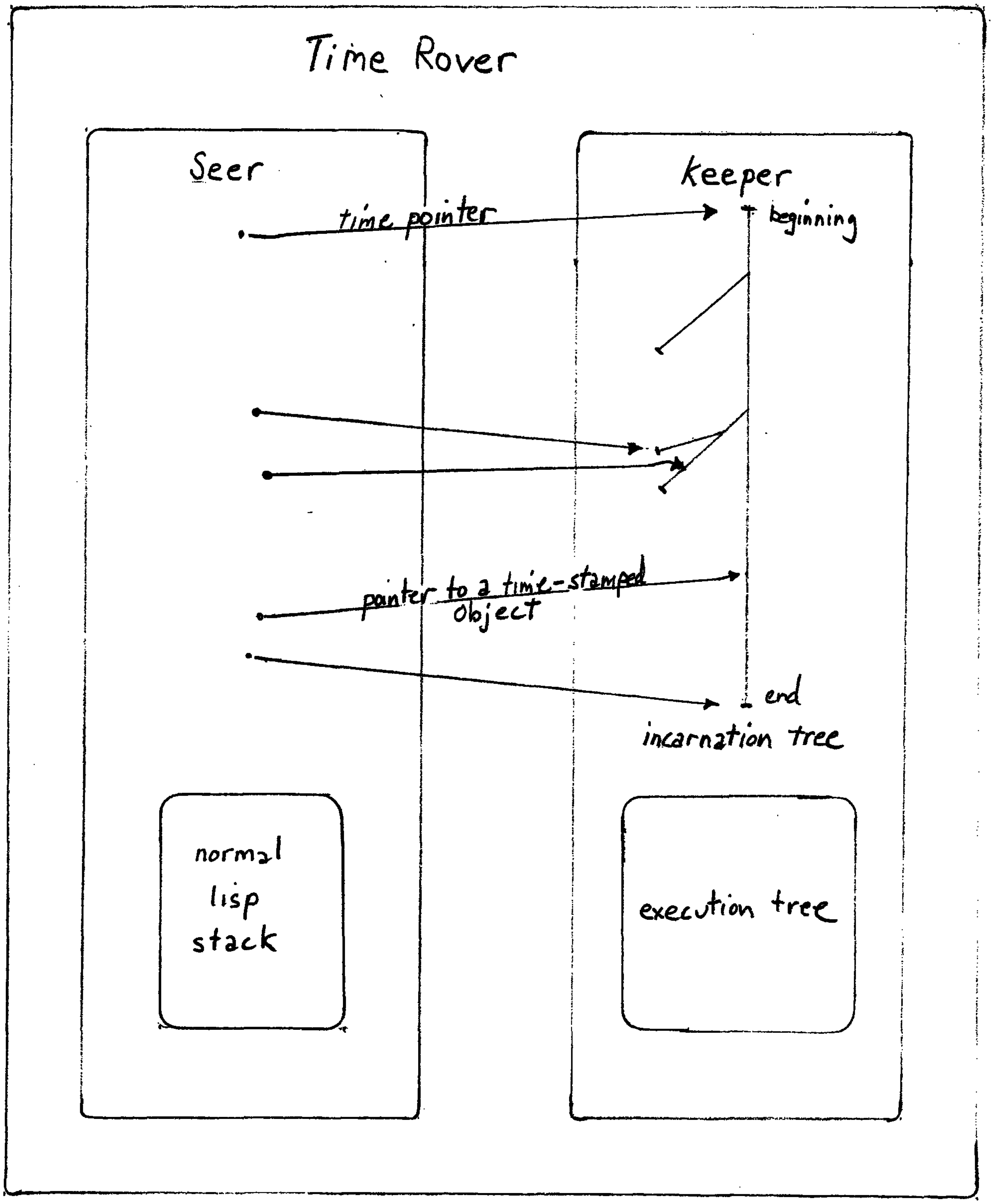
The functions of the keeper and the seer could conceivably get combined into a single evaluator that would have an extra degree of freedom, namely time. In this system, called the *time-probe*, it would be possible to write programs that routinely call procedures which will be defined in the future to modify data which was current at some time in the distant past. The difference between the time rover and this hypothetical system is that the time-probe can travel in its own history. Neither the seer nor the keeper has this ability (and it is not clear that they require it).

The creation of the time-probe is left for future research.

2.6 Methods for specifying times

The primitives for locating times are cast in the framework of search through the incarnation array. There is a notion of the focus of attention, called the *focus_time*, which can be moved throughout the execution history. The searches for other moments move either forward or

Fig. 8. An overview of the time rover



backwards from that time.

Time is a data type recognized by the keeper. There are two functions which yield times; **future_when** and **past_when**. The syntax is

(future_when *form*)

where *form* is an arbitrary predicate evaluated by the keeper. The function future_when scans forward in time from *focus_time* and returns the first moment when *form* yields a non-nil (or non-error) result. Past_when performs the analogous function for moving towards earlier moments in the history.

The implementation for these functions is fairly intricate. It would be prohibitive to attempt to apply *form* at every moment in the history which is scanned, so the search functions first compute the reference set of cell-ids accessed by form, and then move attention to the nearest moment when one of those cell-ids is attached to a different trace-cell. At this time, *form* is reevaluated and the reference set computed once again. The process repeats until *form* returns a non-nil value (success), or until the search passes beyond the boundaries of the incarnation array (failure).

2.6.1 Geographical positioning

The object of geographical positioning is to identify a time, or a range of times in an execution history by physically pointing at the code which was being executed. (This turns out to be a very intuitive method for supporting time roving.) The user interface for this system consists of a standard text editor for locating sections of code, and the lisp machine mouse[1] for highlighting particular areas of the program text.

There are two types of geographical positions. A *geo-position* is an s-expression with its first or last token highlighted, and a *geo-region* is an s-expression with both endpoints selected. *Geo-positions* identify either the beginning or the end of the execution of an s-expression. *Geo-regions* correspond to the duration of an evaluation. Since any given s-expression may have been executed many times, each geographical position potentially identifies a set of times in the history. Geographical positioning is used in conjunction with the search mechanisms already described to uniquely identify moments.

Geo-positions will be represented by underlining an s-expression and identifying its first or last token with a superscript asterisk (a token is an atom-name or a parenthesis). For example,

1. The mouse is a hand held input device that controls a cursor on the lisp machine screen. The cursor moves as the mouse is rolled across a table, and by pressing a control button on the mouse, the user can select the object which is under the cursor.

```
*
(append alpha the-list)
```

represents the first moment in the execution of the append function. This moment is before append has been entered, and before the parameters have been evaluated, but after it is known that execution reached this s-expression. The statement

```
                      *
(append alpha the-list)
```

identifies the last moment in the execution of append, after the function's result has been calculated, but before the next s-expression has been entered. The *geo-position*

```
          *
(append alpha the-list)
```

represents the evaluation of the atom alpha. The evaluation of an atom is a primitive unit of time, its beginning is also its end.

The difference between the last moment of one expression and the first moment of the next shows up in the following example;

```
            *
(cond (n m) (x y))
```

The clause (x y) is physically adjacent to the clause (n m) but the first moment in the execution of (x y) may be radically separated in time from the execution of (n m).

Highlighting the initial portions of a function definition identifies the first moment in that function's execution. e.g.,

```
*
(defun foo (w) ... )
```

Geo-regions will be displayed with both endpoints highlighted. The region in between will be underlined. For example,

```
*                     *
(append alpha the-list)
```

represents the entire execution of the append function.

The search routines **future_when** and **past_when** interact with geographical positions to identify times. For example, if *focus_time* is at the end of the program, and the current geographical position (held in a system variable called *here*) is set to

```
(do ((i 1 (+ 1 i)))
    ((> i 10))
            *
    (print i))
```

Then the expression

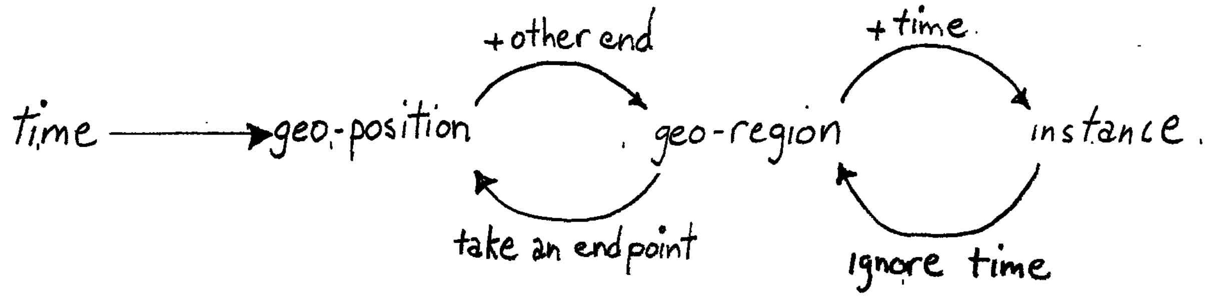
```
        (past_when (and (equal i 6) (at here)))
```

returns the moment in the do loop just after 6 was printed. (The function "at" returns true if it is applied at a moment within the execution of the *geo-region* or *geo-position* provided as its argument.)

As a further refinement, it is important to be able to handle a single execution of a piece of code in addition to the sets of executions which are provided by *geo-regions* or *geo-positions*. The sniffer system requires such an *instance* as its input. An *instance* is defined to be a specific execution of a *geo-region*. It is uniquely determined by a *geo-region* and a time, or two times (which are used to identify the endpoints of the *instance*). Either form can be employed.

Geo-positions, *geo-regions*, *instances* and *times* are closely related. An *instance* becomes a *geo-region* if the execution time is ignored. A *geo-region* can be converted into *geo-position* by the use of the operations **beginning** and **end**. Similarly, a *geo-position* identifies a *geo-region* if its other endpoint is included. Each *time* identifies a *geo-position* in the sense that every *time* is at the beginning or end of some s-expression's evaluation. Note that the same time may be contained by many *geo-regions*. Figure 9 summarizes these relations.

Fig. 9. Conversions between temporal and geographical data types



3. The sniffer system

The sniffer system is responsible for identifying and describing bugs which occur during the execution of a user's program. The input to the system is the suspect portion of the user's code (more exactly, a particular execution of that portion) together with a lisp predicate that functions as a partial specification for that code. The output is an explanation of the bug that relates the data which was in effect at the time of the error to the error, and describes the bug in terms of the intended purpose for the code.

The sniffer system is organized as a collection of experts (sniffers) each of which identifies a particular error. The sniffers do not employ a uniform reasoning mechanism. They locate errors by applying a bag of tricks to the events surrounding a bug. This design is built on the observation that debugging knowledge does not factor well. The necessary information apparently takes the form of a large number of independent facts about specific errors. At the current level of understanding, an expert system model seems appropriate. (In the future, a more hierarchical approach might be successful. However, it is entirely possible that debugging knowledge truly is (and always will be) a disparate collection of ideas.)

The sniffers identify errors by applying a recognition process to the code, and to the execution history of the program. There is no general reasoning involved. Each sniffer assumes its error has occurred if it discovers that a particular collection of features are present. This process requires two basic capabilities: the ability to analyze code for specific features, and the ability to examine events which occur during the execution of the code in question.

The feature analysis is supported by a system (designed and implemented by Richard Waters [Waters 1978] that translates programs into a regular, language independent representation called PLANS. Given a PLAN representation, it is possible to build feature detectors (using a pattern matching paradigm) that recognize fairly complex programming actions, such as an enumeration of all the leaf nodes of a tree, or the splice-in of a list element to preserve an ordering property. This can be scaled up to recognize larger actions, for example, that a function performs a membership test, as in the scenario which follows. All of the analysis performed by the sniffer system uses the PLAN representation. This gives Sniffer a strong element of language independence in its operation.

The process of recognizing features is not as difficult as it might seem. The PLAN language does an excellent job of regularizing the representation for small algorithms. Conceptually related segments of code that are widely separated in the text are closely associated in the PLAN diagrams. This grouping is based on the presence of data flow between the segments. Iterative and recursive behavior is categorized in such a way that initialization steps, the independent actions in loop bodies, and termination tests are all separated into clearly defined sections.

There are only a few distinct PLANS that represent actions the size of list enumerations. The difficulty of recognizing larger fragments (such as a membership test) grows with the size of the algorithm, but it is mitigated by the fact that the sniffer system only needs to be reasonably certain that a particular feature has been recognized. Taken together, a series of good guesses builds up a complete enough picture to reliably identify a bug. In the context of a debugging aid, the few remaining misdirections can be tolerated.

The ability to investigate events from the execution history of the code is provided by the time rover, which I have discussed. The experts are free to use any of the time-rover's facilities. They can investigate nearby events in the environment of the caller of the suspect function, or examine the code for subordinate routines. It is also possible for the sniffers to scan over large areas of time and text, for example, to find an error at an unknown time within the processing of a particular object. With these abilities, bugs can be traced to radically different places and times from the locations in which they were detected. The example in the scenario is a simple usage; the sniffer that succeeds stays very near the time and place of the complaint that is provided.

The implementation of the sniffer system will contain two portions; a collection of <u>triggers</u>, and a collection of individual bug experts, or <u>sniffers</u>. The control structure is demon invocation, where the triggers act as feature detectors that control the invocation of the sniffers (demons). The triggers are responsible for performing inexpensive tests on the input. The sniffers perform all the computationally expensive tasks which are required. This cost/ease distinction is the primary basis for the division of labor; both the triggers and the sniffers can examine the execution history of the code, the text of the code, and the user-supplied predicate.

I chose demon invocation as the control structure for the sniffer system for a simple reason; in the absence of a detailed theory of debugging, there is no motivation for a non-uniform problem solving mechanism. I have limited the scope of each expert so that sharing of information is not required, and their total number is small enough so that a demon mechanism is sufficient for focusing the system's attention. If need dictates, the control strategy can be changed.

4. A scenario using Sniffer

This chapter contains a hypothetical scenario for the use of Sniffer. However, before I can investigate a class of errors, I need an example program to spike with bugs. This program, called the test program has to be complex enough to illustrate subtle errors, but also simple enough to avoid becoming a distraction from the main part of the research.

4.1 The test program

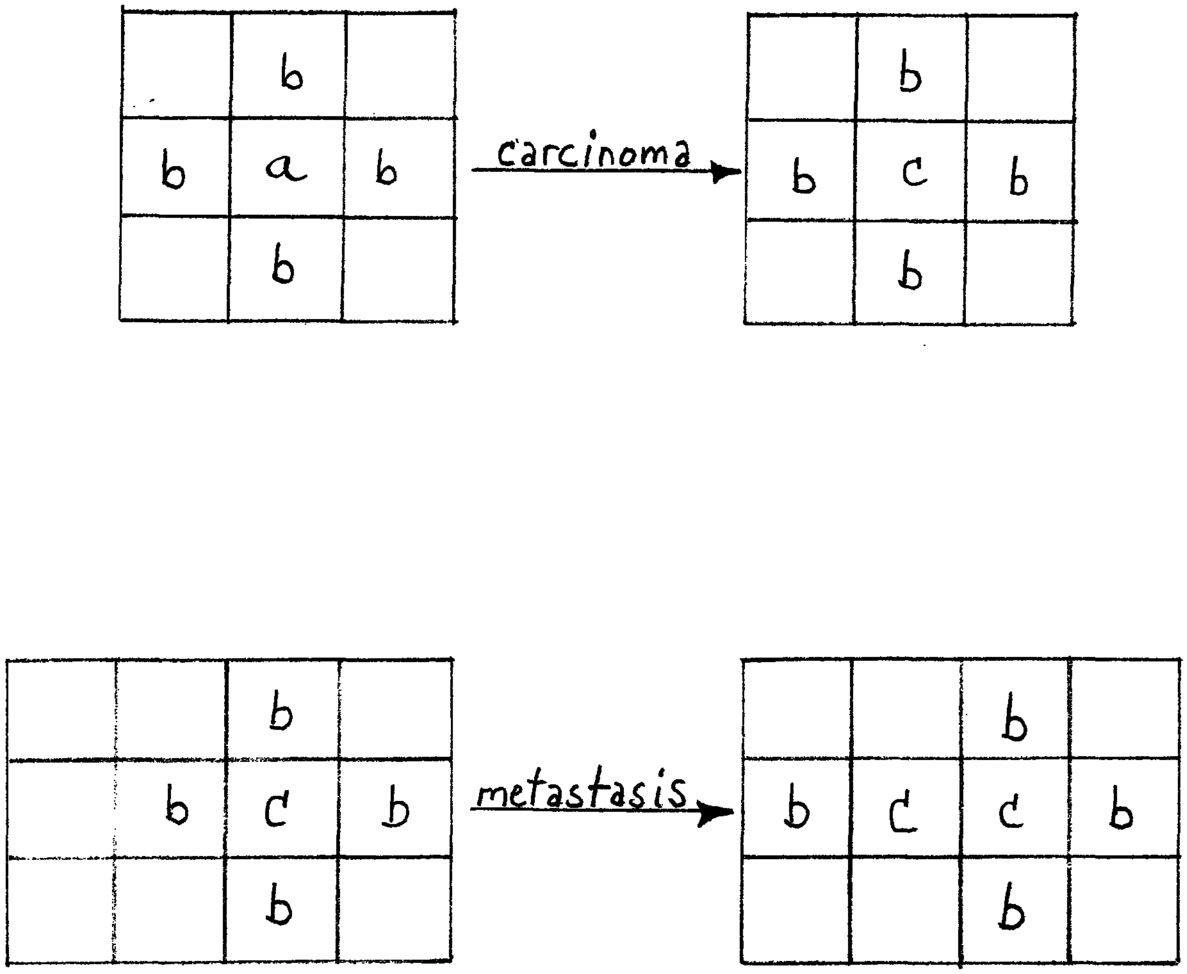
I have chosen to implement a morphogenesis simulation, called *prosper*, which loosely models the growth of a colony of bacteria. In *prosper*, the user provides an initial pattern of cells and a collection of production rules which govern their division. The simulation outputs a trace of the shapes the bacteria colony assumes through time.

The cells live on a rectilinear array called the grid. Each cell occupies one square of the grid and may have up to four neighbors, corresponding to the top, right, bottom and left positions of the array. Every cell has three basic properties, a type, an age, and a division time (which is the next time at which it is expected to divide). The productions cause cell division. They are local transformations that apply to one cell in the context of its immediate neighbors. Productions can access any of the properties of the adjacent cells. For example, a typical transformation (see figure 10) might map a cell of type "c" surrounded by "b" cells into two "c" units. In order to make the necessary room, the neighbors are pushed out of the way.

Prosper is implemented as a production rule system that operates on data kept in a prioritized queue. This queue, called the events-queue, orders the cells according to their division time. The cell with the next (or lowest) division-time has the highest priority. (See figure 11 for the top level code.) The flow of control is as follows: the grid is initialized with some pattern of cells, and those cells are assigned division times and placed on the events-queue. The central loop removes the first member of the queue, and finds the set of productions which can affect cells of that type. One of these candidates is selected and applied. The transforms are responsible for requeueing any second-generation cells which they produce. *Prosper* terminates when the events-queue is empty.

The grid is implemented as a hash table keyed on the location of cells. (This allows incidental connectivity to be discovered, when separate formations grow together.) The transformations are stored in a library, also in the form of a hash table keyed on the type of the cell affected. The events-queue is implemented as a sorted list, with division-time used as the index.

Fig. 10. Some sample transformations



4.2 The scenario

The dialogue starts after the program, *prosper*, has been running for some time, and has started to generate incorrect output at the terminal. The problem is that the user expected a collection of productions to cause an explosive growth of cancer cells (cells of type "c"), and nothing happened. (These productions are shown in the previous figure.)

The user's input will be indicated by a ">" prompt. A protocol for his thoughts during the debugging process is enclosed in brackets. All symbols of geographical and temporal data types will

Fig. 11. The code for *prosper*

```
(defun prosper (&aux transform-lib grid)
    (setq grid (create-grid))
    (setq transform-lib (create-transform-lib)) ;read in the transformations
    (do ((events-queue (events-queue-init grid)
         (matches) (cell)))
        ((null events-queue) nil) ;process the events-queue until it is empty
        (display-grid grid)
        (setq cell (events-queue-first events-queue))
        (setq matches (find-transforms (cell-type cell) transform-lib))
        (apply-transform (select-transform matches grid) grid)))
```

be displayed in italic font. System defined variables and keywords will be shown in boldface.

[The program is outputting bad data. It is time to interrupt it and find the bug.]
The user stops execution during the function aref (array reference).

```
>break
aref: active
```

The designation active indicates that the function never returned. Similarly, inactive states that the function completed execution.

[The right place to start the debugging process is by looking at the top level function, prosper. Move the focus of attention to the most recent point in time at which prosper was being executed.]

```
> (move (past_when (eq function_name 'prosper)))
focus_time = 17247
```

```
> (print_frame)
prosper: active
executing at:

(defun prosper ( &aux transform-lib grid)
    (setq grid (create-grid))
    (setq transform-lib (create-transform-lib))
    (do ((events-queue (events-queue-init grid)
         (matches) (cell)))
        ((null events-queue) nil)
        (display-grid grid)
        (setq cell (events-queue-first events-queue))
                                                              *
        (setq matches (find-transforms (cell-type cell) transform-lib))
        (apply-transform (select-transform matches grid) grid)))
```

This notation was developed in the section on geographical positions. It indicates that *focus_time* is at top level during the execution of prosper, just after the atom, transform-lib, was evaluated. At *focus_time*, all the bindings within prosper are in effect. After this moment, the execution enters find-transforms, and the flow of control eventually leads to the interrupted execution of aref.

[Since the problem is that cancer cells are not dividing, are there any scheduled for processing? What are the contents of the events-queue?]

```
> events-queue
((250 (b (5 6) 2)) (255 (a (5 7) 3)) ...)
```

The events-queue is a represented as a list of pairs, with the division-time as the first entry, and a cell as the second. The top member of the queue has the lowest division-time and therefore the highest priority. Cells have a type, an x-y location and a division-count, in that order.

[Print out just the types of the cells in the queue.]

```
>(mapcar (function cadr) events-queue)
(b a a b b a c c a a b b b b b b b b a ...)
```

[The queue contains cancer cells (cells of type "c"), but they are not clustered near the top as expected. Is the cell currently being processed a cancer cell?]

```
>cell
(a (3 4) 3)
```

[It is not. When was the most recent time when a cancer cell was being processed? Its division should have instigated explosive growth.]

The user finds the desired moment by using the geographical positioning method. He selects a position in the code with the mouse. The *geo-position* chosen is placed in the variable *here.*

```
>
(defun prosper ( &aux transform-lib grid)
   (setq grid (create-grid))
   (setq transform-lib (create-transform-lib))
   (do ((events-queue (events-queue-init grid)
        (matches) (cell)))
        ((null events-queue) nil)
        (display-grid grid)
        (setq cell (events-queue-first events-queue))*
        (setq matches (find-transforms (cell-type cell) transform-lib))
        (apply-transform (select-transform matches grid) grid)))

        here = (setq cell (events-queue-first events-queue))*
```

[Find the time in the past when execution was at *here* and the cell being processed was of type "c".]

```
> (move (past_when (and (at here) (equal (car cell) 'c))))
focus_time = 7016

> cell
(c (2 1) 2)
```

[This cell should have metastasized, and yet it didn't. The bug is probably involved with the processing of this cell. Step through the next section of the execution history to find out.]

[What transforms were chosen as candidates?]
The user selects a position with the mouse.

```
 ...
(setq matches (find-transforms (cell-type cell) transform-lib))*
(apply-transform (select-transform matches grid) grid)))

here = (setq matches (find-transforms (cell-type cell) transform-lib))*
```

The next expression looks forward in time to the instant when *here* was being executed and evaluates matches in that environment.

```
> (@ (future_when (at here)) 'matches)
((c 'surrounded-by-A-cells 'metastasize)
 (c 'old-age-cell 'die))
```

Transforms have three parts, the type of the cell they affect, a function which determines whether the transform can apply, and a function which contains the manipulations of the transform itself.
[There were two transforms listed as potentials. The first one is the one which causes explosive growth, the second one applies to old age cells. Which one was chosen?]

```
> (move (future_when (at here)))
```

```
focus_time = 7100
```

The user finds out which transform was selected by using @ to evaluate the select-transform function in the current time-environment. The result is necessarily identical to the one returned by the original invocation.

```
> (@ focus_time '(select-transform matches grid))
(c 'surrounded-by-A-cells 'metastasize)
```

[move to the time when "metastasize" was being evaluated]

```
> (move (future_when (eq function_name 'metastasize)))
focus_time = 7330

> (print_frame)

metastasize: inactive
focus_time = 7330
executing at:
(defun metastasize (top-cell right-cell bottom-cell left-cell key-cell
                    &aux location new-cell)
          *
          (setq new-cell (create-cancer-cell))
          (increment-division-count key-cell)
          (setq location (cell-location right-cell))
          (make-room-between key-cell right-cell grid)
             ;push away all the cells to the right of the key-cell
          (grid-insert new-cell location grid)
          (events-queue-insert new-cell (+ div-time 2) events-queue)
          (events-queue-insert key-cell (+ div-time 2) events-queue))
             ;requeue the cancer cells almost immediately
```

[The calls to events-queue-insert should have placed the cancer cells, new-cell and key-cell, on the events-queue with a high priority division time. The events-queue-insert function is looking suspect. Check to see if it actually inserted the items.]

The user identifies a region of interest with the mouse.

```
>

          ...
(grid-insert new-cell location grid)
*                                                        *
(events-queue-insert new-cell (+ div-time 2) events-queue)
(events-queue-insert key-cell (+ div-time 2) events-queue)

           *                                                        *
region1 = (events-queue-insert new-cell (+ div-time 2) events-queue)
```

The following expression tests to see if the call to the events-queue-insert had any effect at all upon the events-queue. This is the strongest kind of equality test possible. See the section on equality and coreference for a detailed discussion. Beginning and end are functions that return the endpoints of *instances* of *geo-regions.*

```
> (unmodified*  (@ (future_when (at (beginning region1))) 'events-queue)
                (@ (future_when (at (end region1))) 'events-queue))
t
```

[All suspicions are confirmed. The insert function was called, but the queue never received the data. This is a suitable point to ask the debugging expert for its opinion.]

```
> (get_expert_help '(events-queue-member events-queue new-cell) focus_time region1)
```

The get_expert_help function invokes the sniffers. The first argument is a lisp predicate that is expected to apply (to be non-nil) after the code for *region1* is executed at *focus_time.* The sniffers use the predicate as a partial specification for the code in the region. The sniffers examine the code for the predicate and for *region1.* They also investigate the state of the events-queue at *focus_time* and the execution path which events-queue-insert actually followed. The following error report would be the result of the sniffer which recognized the bug.

Sniffer report: **Cons-bug for sorted lists**

Statement of the error: *From analysis of the predicate*

```
        (events-queue-member events-queue new-cell)
```

the error is that the function, events-queue-insert, failed to add the item

```
        (cons (+ div-time 2) new-cell)
```

to the events-queue, when called by

```
        (events-queue-insert new-cell (+ div-time 2) events-queue)
```

The cause of the bug: *The insertion did not occur because the caller of events-queue-insert expected the insertion to be accomplished by side effect but events-queue-insert did not do so on this invocation.*

Analysis: *The function, events-queue-insert, is intended to insert an item into a sorted list. The function was invoked at a time when*

```
        events-queue = ((110 (a (3 4) 2)) (115 (b (5 6) 2))(118 (a (4 7) 2)) ...)
```

and the item to be inserted equaled

```
        (105 (c (2 1) 2))
```

There are two possible data paths through the function, one of which passes through a splice in operation in events-queue-insert. In this particular execution, the item was sorted to the top of events-queue, and the function returned a new list containing

```
        (cons '(105 (c (2 1) 2)) events-queue)
```

instead. No side effect to the list was performed. The new list returned was not used to side effect events-queue. The caller did not independently modify events-queue.

The data to be inserted was forgotten.

The mechanisms which support this analysis are described in the following chapter.

5. The cliche finder

The obvious question at this point is to ask what mechanism allows the cons-bug sniffer to explain the error in such high level terms. The sniffer employs terminology which indicates that it understands the concepts of an ordered list, a membership test, and the notions of data and control flow through a program. (The ability to wield this kind of vocabulary is a critical step towards the goal of program understanding; it is important to be able to converse with a programmer in terms of the building blocks he uses to construct programs.)

The cliche finder is responsible for building the vocabulary used in the bug report. It identifies fragments of algorithms in the code in order to provide the sniffer system with a context for identifying errors. The cliche finder is primarily used by the triggers of the sniffer system, which are concerned with finding quick tests that determine the topic of an error. In the implementation I am designing, these tests correspond to textual analyses of the code. Questions which access the execution history are performed by the bug experts themselves.

The cliche finder recognizes commonly used program elements by applying a pattern matching process to the PLAN language representation of the code under analysis [Waters 1978]. There are many small algorithms which the system is capable of finding.[1] For example, the cliche finder identifies the following features in the code for events-queue-insert (see below).

```
(defun events-queue-insert (item time evq)
   (let ((entry (cons time item)))
        (cond ((or (null evq) (before? entry (car evq))) (cons entry evq))
              ((do (1(new (cdr evq) (cdr new))
                    1(old evq new))
                   ((or 2(null new) 3(before? entry (car new)))
                    4(rplacd old (cons entry new))))))))

(defun before? (item1 item2)
   (< (car item1) (car item2)))
```

The statements subscripted with a "1" are recognized as a *trailing pointer enumeration*, which is a specific method for enumerating all the elements of a list. The rplacd expression (4) is identified as a *splice in* operation affecting the output of the trailing pointer part. Each of the statements 2 and 3 are recognized as termination tests of the do loop. The function before? used in this context is identified as an *ordering predicate* for the elements of the list, evq.

The cliche finder is capable of correlating the features it discovers to identify larger behavioral

1. See [Rich, 1980] for a library of such programs. His dissertation is largely concerned with identifying typical programming techniques.

units. For example, the do loop in the code for events-queue-insert is recognized as a *trailing pointer insertion.* The membership test in events-queue-member is recognized as a whole in a similar way. (This pyramiding of results could also be handled by the sniffer system. this division of labor is for convenience, not necessity.)

5.1 An overview of PLAN structures

The cliche finder derives all of its abilities from the expressive power of PLANs. PLANs impose several critical restrictions on the representation of programs which makes the recognition of algorithms a feasible task.

PLANs ignore the way in which control and data flow is implemented. The result is that many textual representations for the same algorithm are mapped into identical PLAN structures. For example, it makes no difference if a program uses conditionals or goto statements to implement the flow of control. All the possible ways of using variables to remember values or hold partial results are judged equivalent. PLAN diagrams extract only the essential interconnections between operations from the text of a program.

PLANs associate related segments of code which may have been widely separated in the original text. The PLAN structures are compound objects composed of data flow related segments. The fact that one segment outputs data which another consumes is a simple proof that both are working towards some unified purpose. The consequence of this organization is that feature detection in PLAN space involves far less search than it would require in the original text for the code.

The PLAN representation is partitioned into fragments which have stereotyped behaviors. This allows complex programs to be understood in terms of simple purposeful parts. For example, iterative and recursive routines are represented by a single PLAN structure (a PLAN Building Method, or PBM in Waters' terminology) which contains five types of components; *initializations, generators, filters, accumulators* and *terminators.* (The output of his analysis system labels the segments which fulfill each of the five roles.) An initialization is a segment that is executed once before a loop is entered. A generator produces a sequence of values that are used in later calculations (a list enumerator is an example of a generator). Filters restrict the sequence of values which are visible beyond their location in the code. Accumulators perform calculations, they remember results. Terminators are like filters in that they restrict sequences of values, however, they also have the potential to stop the execution of a loop. The remaining plan building methods categorize the program actions in straight line code. Taken together, the PBMs provide a complete parse of a program into these purposeful parts. (The mechanisms which perform this analysis are too lengthy to describe here. See [Waters 1978] for a full explanation.)

PLANs are quasi-canonical representations for small algorithms: The logical structure analysis which creates PLANs isolates independent actions from one another. A small program such as events-queue-insert contains 5 or 6 such segments which are trivial to identify. The claim is that all reasonable insertion algorithms have PLANs of similar complexity, and furthermore, these PLANs are composed of almost the exact same parts. The result is that the recognition of small algorithms from PLANs does not appear to be a difficult task. I present a detailed example in the following section.

5.2 The PLAN for events-queue-insert

The following three figures present the PLAN for events-queue-insert in its entirety. The diagrams contain a considerable amount of information. (PLAN structures are alternate representations for programs, many details which are hidden in the code are explicitly represented in a PLAN.) There is also a fair amount of special notation which I will explain. I present this PLAN in order to put the discussion of the cliche finder on a concrete foundation. In this section I will show exactly what structures the cliche finder has to recognize to identify small algorithms.

5.2.1 Notation

PLAN diagrams contain three kinds of entities; boxes, solid lines and dashed lines. Boxes represent actions which may be either primitive or compound. A primitive action corresponds to a black box in the code, such as a cons statement in lisp. There are eleven types of compound actions, these include *conjunctions, predicates,* and *conditionals* for representing straight line code, and *filters, accumulations* and *terminations* for representing looping behavior. Dashed lines represent control flow, solid lines represent data flow. For example, the diagram of figure 12 represents the top level PLAN for events-queue-insert as the PBM *exclusive or*, where the predicate

```
(or (null evq) (before? entry (car evq)))
```

determines whether the function returns through a cons, or enters the expression containing the body of the loop. (See page 37 for the code of events-queue-insert.) There is data flow from the inputs *item* and *time* to the cons function

```
(cons time item)
```

which produces the data value *entry* that is tested by the predicate above. The diagram contains branched control flow to show that there are two possible outcomes of the test. The box at the bottom labeled *join* is there for syntactic purposes, it preserves the one-in one-out property of

Fig. 12. The top level PLAN for events-queue-insert

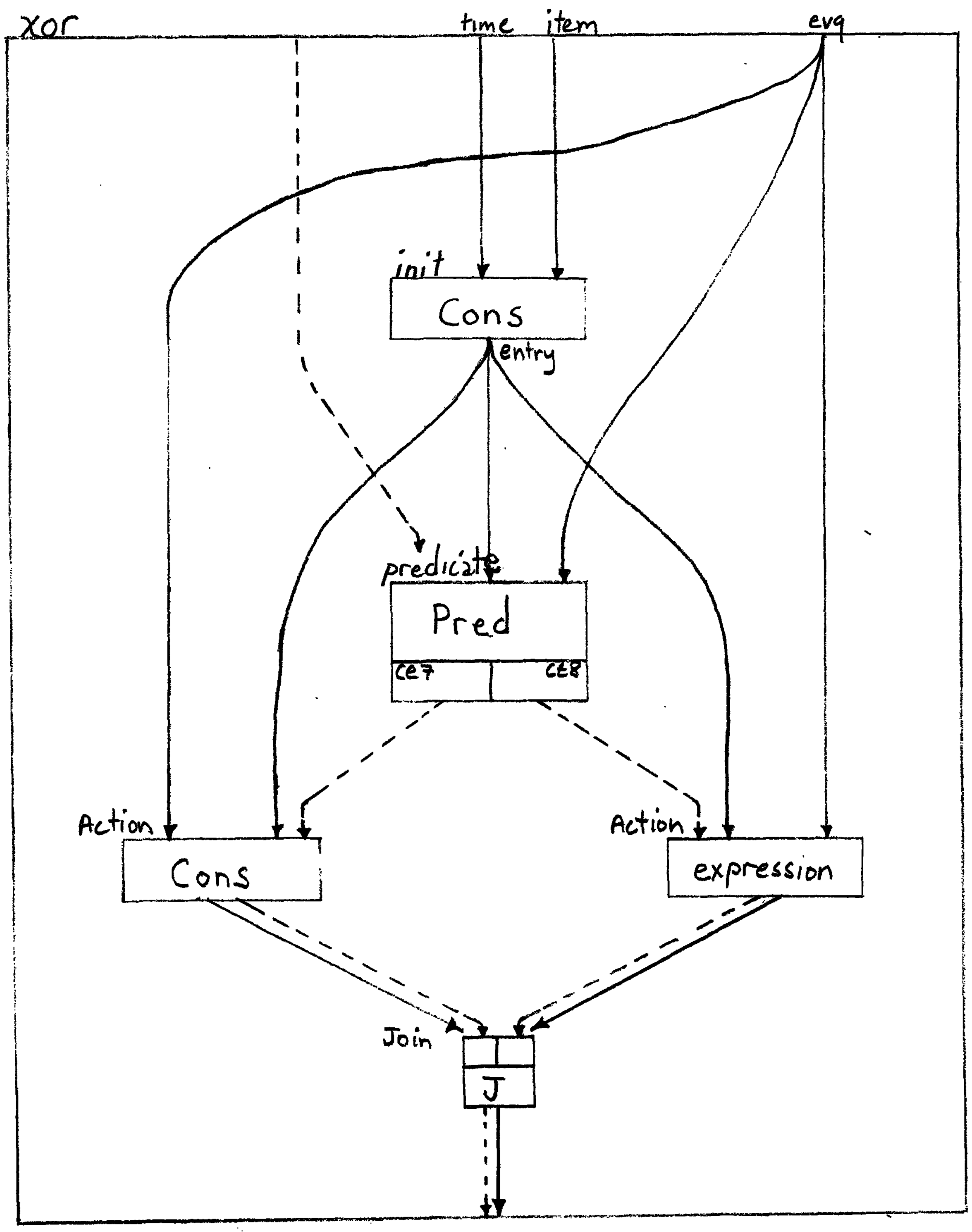


Fig. 13. The predicate for testing list elements

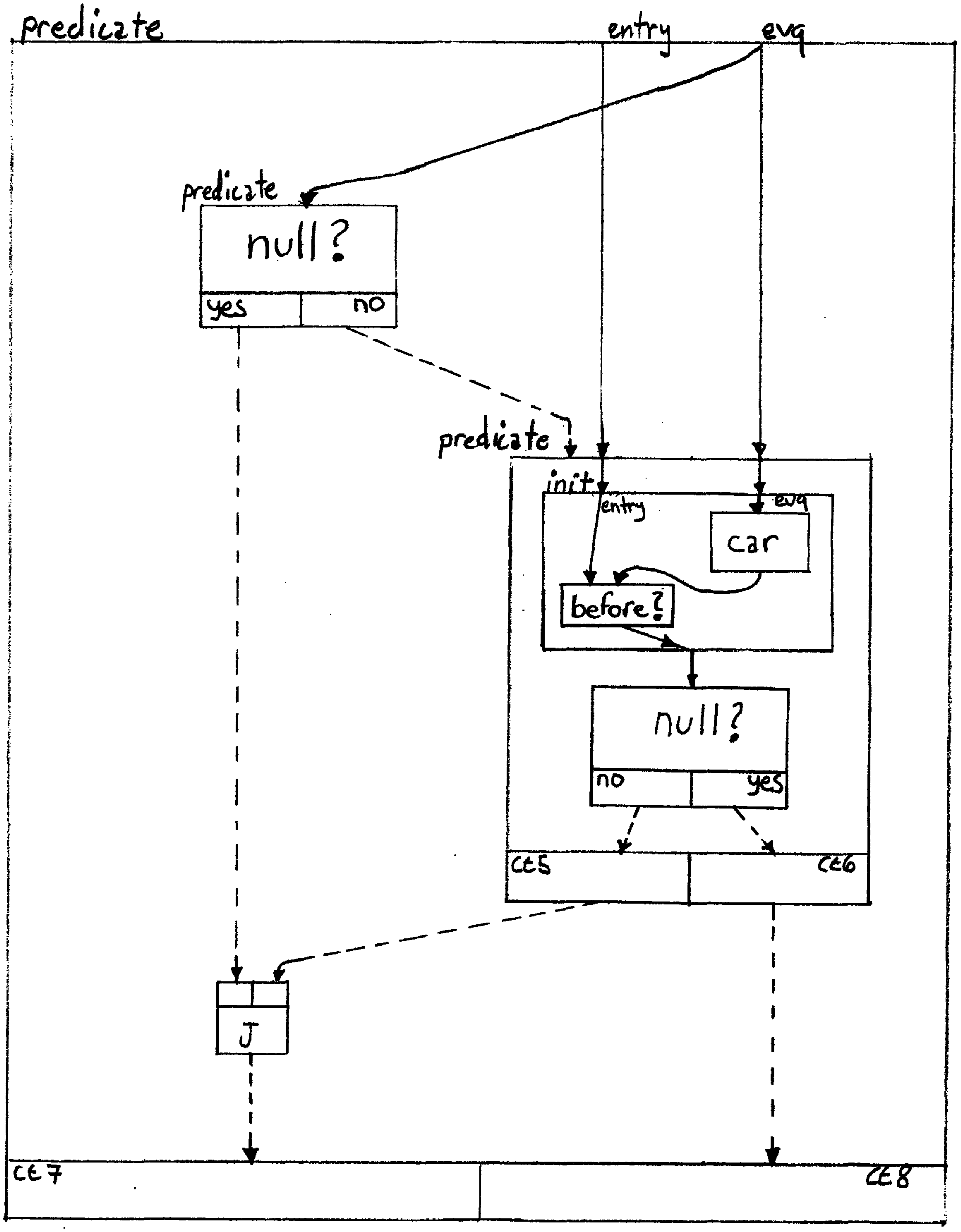
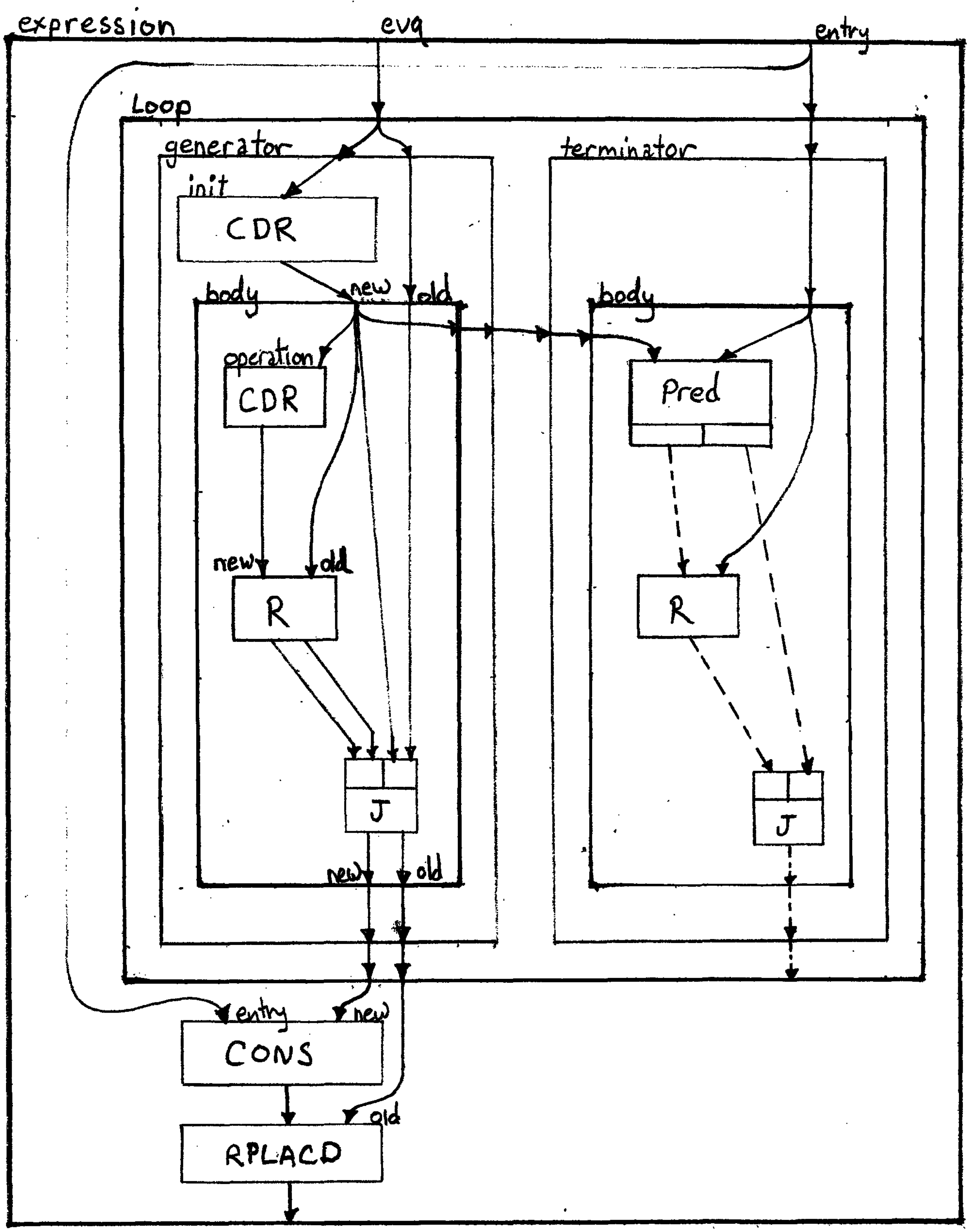


Fig. 14. The PLAN for inserting an element in a list



compound actions.

Each compound action has certain allowable components, called *roles*. There is a grammar (which I will not present here) that restricts the number, and contents of these permissible parts. In the figures, the roll a component fulfills is printed on its upper left-hand corner.

5.2.2 An overview of the sections of the PLAN

The PLAN for events-queue-insert is broken up into a conditional that determines whether the loop is to be entered (figure 12), a compound predicate (figure 13) and the expression containing the loop (figure 14). The loop is further decomposed into a *generator*, which enumerates the elements of the events-queue (evq in the diagram), and a *terminator* which controls the execution of the loop body.

The generator represents the code segment

```
((do ((new (cdr evq) (cdr new))
      (old evq new))
      ...))
```

Generators are composed of an optional *initialization* and a *body* which is the portion that is executed many times. The body can contain an *operation*, a *recursion* and a *join*, which I explain in a moment.

The sole input to the generator is the variable named evq. This data passes through the initialization operation

```
(cdr evq)
```

which outputs the data (labeled new) that is operated on by the body of the generator. The body receives two inputs, new and old, where *old* starts as the unmodified events-queue. The *operation* of the generator body is the function cdr, from the code

```
(cdr new)
```

above. At each successive iteration, this operation causes *new* to become successive sublists of the events-queue. The data values *new* and *old* become the output of the generator, emerging from the data join box in the diagram. The join indicates that the output can come from one of two places; it can be the input to the generator body (in case the generator terminates), shown by the data lines that pass straight through the diagram, or it can come from the box labeled "R" which stands for a recursive instance of the enumerator. The cross over of data, where *new* becomes *old* at the next iteration, can be seen from the change of labels on the data flow lines at the input ports of the R segment.

The terminator for the loop is conceptually executed in parallel with the generator. At each iteration, the predicate compares the entry with the value of *new* that is obtained from the top of the body portion of the generator segment. If the predicate returns through its right hand branch, control passes out of the terminator segment, and iteration of the generator body is stopped as well.

5.2.3 The details of finding cliches

Using the PLAN representation, the task of identifying programming cliches becomes a process of matching known PLAN diagrams against the structures in an analyzed program. The simplest segments are easy to identify. For example, there is only one reasonable way to express a list enumeration that returns pointers to two successive subsets of a list. This PLAN is called a *trailing pointer enumeration* and it has the exact same picture as the generator in figure 14. The PLAN for a trailing pointer enumeration requires that a cdr operation be the *initialization* of the generator and that the data flow line which enters the initialization also becomes the second input of the generator body. These restrictions ensure that successive elements are returned no matter how many times the body is executed.

There are other ways to enumerate the elements of a list which do not follow the trailing pointer enumeration PLAN. For example a *simple enumerator* corresponds to "cdring" down a list. Only one pointer is involved. A simple enumerator can be used to construct a different insertion operation if the pointer is always "one behind" the place where the new entry might be inserted (see figure 15).

Events-queue-insert also contains a *splice-in* operation which is trivially recognized from the PLAN. The PLAN for a splice-in is shown in figure 16. This operation is composed of a cons and a rplacd function where the cons creates an augmented list, and the rplacd attaches it to the end of the immediately preceding portion of the list. The algorithm is represented in a PLAN diagram by requiring that one input to the cons, and one input to the rplacd function be derived from a single data path. This path must be split by a cdr operation just prior to the cons and rplacd statements involved. This pattern is present in both of the list insertion methods shown in figures 14 and 15.

The recognition process becomes more difficult as the size of the function to be identified increases. The problem is that PLANs do not completely canonicalize representations of the same algorithm. This implies that a matcher would have to cope with a significant number of variations in order to identify a function the size of a list insertion. (The fact that there are several distinct algorithms for implementing such functions is not important. Each method could be recognized by a match against a separate pattern.)

It turns out that Sniffer does not require a clever matcher that can account for these differences. There are two reasons why this is true. First, the feature detection process is only used to narrow the context of an error enough to identify the bug experts which should be run. A reasonable suspicion

Fig. 15. A PLAN for a second list insertion algorithm
This figure contains the PLAN for the code segment

```
        (do ((new evq (cdr new)))
            ((or (null (cdr new))(before? entry (cadr new)))
             (rplacd new (cons entry (cdr new)))))
```

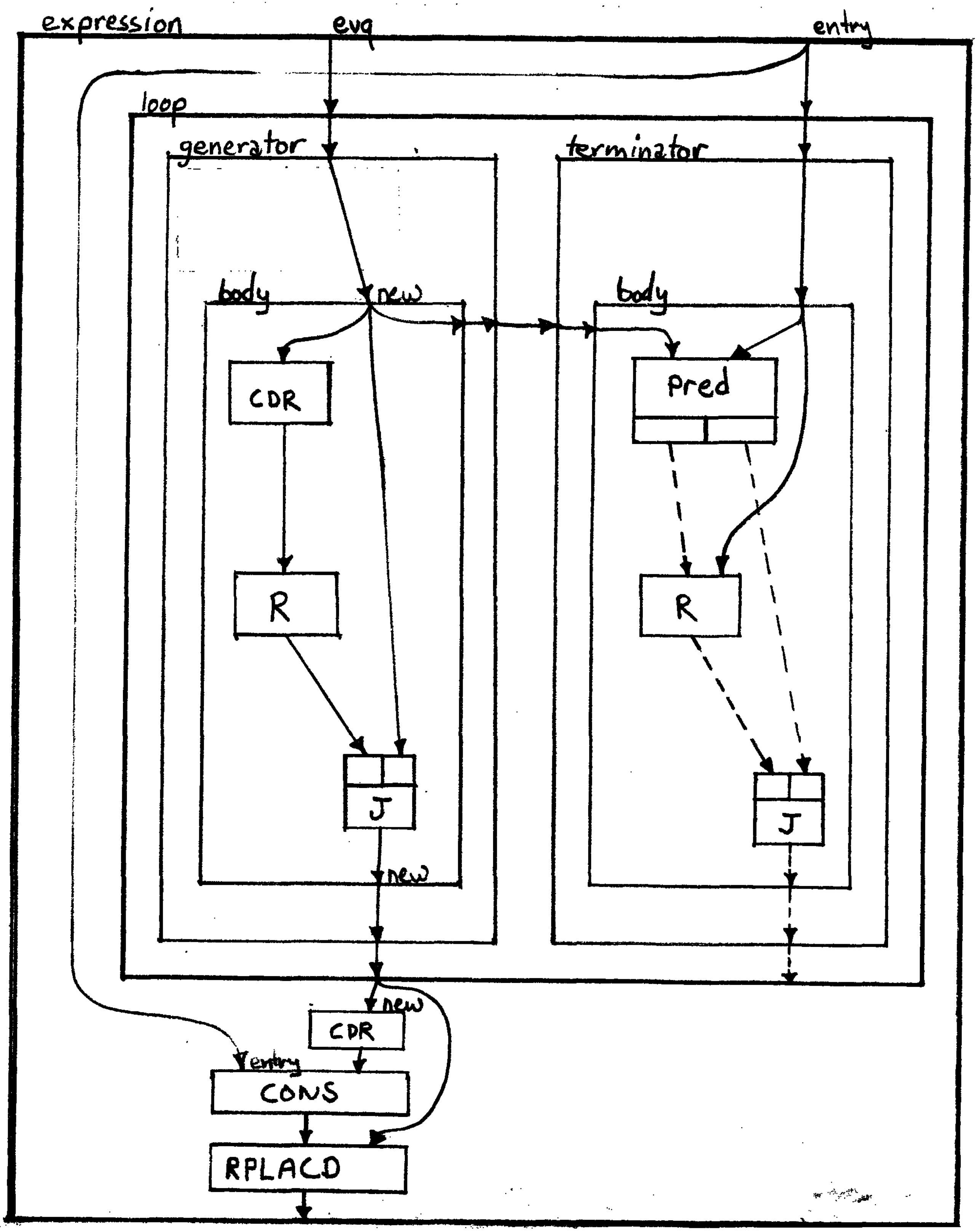
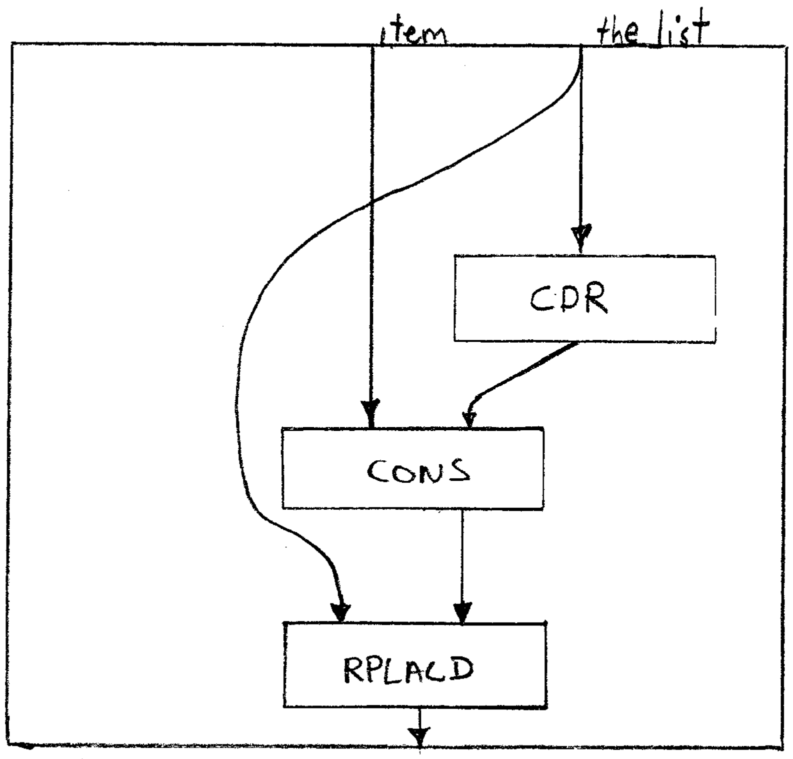


Fig. 16. The PLAN for the splice-in operation



that a feature is present is sufficient for this task. Second, there is reason to believe that an iron-clad recognition process is not possible in the context of a debugging aid. In this environment, the algorithms which have to be recognized are already expected to differ from the pure examples.

The cliche finder identifies a segment of code as a *trailing pointer insertion* by matching its PLAN against the PLAN in figure 14. This match is not expected to be completely successful, only a subset of the features in the diagram must be present for the algorithm to be identified. (Note that this gives the results of the recognition process the flavor of a confidence estimate as opposed to a binary determination.) The following features are required at the minimum:

A *trailing pointer enumeration* must be present.

There must be a *splice-in* operation.

Both of the outputs of the enumerator must be inputs to the splice in.

There must be data flow line which is an input of a loop segment, a terminator for the loop, and the splice-in operation. This line is identified as the item to be inserted.

There must be a data flow line which is an input of the loop, and the sole input of the trailing pointer enumerator. This line is identified as the list which is being side effected.

5.3 An explanation of the bug report

The cons-bug sniffer's report is a template based reply that summarizes the data collected in recognizing the error. The template mechanism is employed purely to finesse the need for any natural language generation facilities. The specificity of the report, and the high level vocabulary that it employs are justified by the mechanisms which comprise Sniffer. The bug experts understand what they say.

The cons-bug sniffer generates its description of the error by relating events in the execution history of the program to the intent for the code which is evaluated. In order to do this, the sniffer puts questions to the time rover, and uses the facilities of the cliche finder to investigate the PLAN for the code. In addition, the sniffer has to be able to identify the portion of a PLAN structure that corresponds to the execution of a particular region of a program. With this ability, the cons-bug sniffer can recognize the cliche which is being pursued at any given instant. (The information required to do this comes from two sources, first, all s-expressions in the user's code are annotated by their execution times. Second, the PLAN analysis system indexes the PLAN structures by the code expressions that generate them.)[1]

5.3.1 The mechanism of the cons-bug sniffer

In the scenario, the sniffer system is invoked by the expression

`(get_expert_help '(events-queue-member events-queue new-cell)` *focus-time region1*`)`

where *focus-time* and *region1* identify an instance of the execution of events-queue-insert, and events-queue-member is a predicate that details the behavior which is desired.

1. Strictly speaking, this indexing is not always possible, since the integrity of code statements is not completely maintained during the translation to PLANs. The correspondence *is* available at the level of the primitives of the code, meaning the built in functions of lisp and the unexpanded functions in the user's code.

The first step in the investigation is to generate the PLAN for the relevant code, namely the user-supplied predicate, and the function events-queue-insert. Next, the sniffer system runs the triggers, which perform simple tests to establish the context of the error. Once the context is known, the relevant bug-experts are executed.

The triggers are implemented by the pattern recognizers of the cliche finder, as well as some simple tests for keywords in the program text. They discover the following facts:

> The word "member" is contained in the name of the predicate function, and the word "insert" is contained in the function that forms the region under investigation. Both functions contain the word "queue" in their names.
>
> The function events-queue-insert contains a *trailing pointer enumeration* and a *splice-in* operation. It is tentatively identified as a *trailing pointer insertion*. (See the section on the details of finding cliches.)
>
> The predicate events-queue-member contains a *simple-enumerator.* It is further identified as a membership predicate which tests for the presence of its second parameter in the list formed by its first parameter.

This information is sufficient to invoke the cons-bug sniffer (and possibly other sniffers which key on the same facts).

The cons-bug sniffer proceeds in the following fashion: it attempts to make sure that it applies in the current situation, it identifies the minimum criteria for its bug to be present, and then it finds out as much as possible about the events surrounding the bug.

In order for the cons-bug sniffer to be relevant in the current situation, the desired behavior expressed by the user-supplied predicate has to be investigated further. The sniffer uses the time rover to run the predicate both before and after the region of code is executed. If the predicate becomes true across that region, the user has misstated the problem. The PLAN for events-queue-insert is also analyzed to make sure that it contains no additional elements which make its total function different from an insert operation (the cliche finder only determines that events-queue-insert *contains* a trailing pointer insertion). This question is answered by an analysis of the PLAN in figure 12. This analysis shows that there *is* data flow from the input parameters of events-queue-insert to the trailing pointer insertion, and also that the item to be inserted is generated by a cons operation applied to two of the inputs of the routine. Note that this fact gives rise to the comment in the bug-report that identifies the item to be inserted is a cons of the input parameters (see page 36).

The cons-bug sniffer determines that the cons-bug is present if the following facts can be established.

The insertion function returned without side-effecting the list.

The function returned after forming a cons of the item to be inserted and the list.

The calling function failed to remember the result of the cons.

No function near the time and place of the region under investigation independently modified the list to include the item to be inserted.

There are several techniques involved in answering these questions.

The first question is answered by using the time rover. If the list was not side effected during this execution of events-queue-insert, then the function *unmodified** will return true when applied to the list at the endpoints of the execution. The variable containing the list has already been identified by some simple analyses of the PLAN structure (see above). (To be very careful, the *unmodified** operation should be applied to the cell-id which corresponded to the head of the list at the beginning of the insertion operation. The variable which contained the list could have been reused while the list itself remained unchanged.)

The fact that the events-queue-insert returned through a cons operation can be established by looking in the execution tree, which monitors the invocations of the s-expressions in the user's code. This information will be accessed by examining the PLAN for events-queue-insert, and asking the cons function which is on the exit branch of figure 12 when it was last executed. (Questions of this form can be answered because primitives in PLANs are cross indexed with the s-expressions they correspond to. The s-expressions are annotated by their execution times.) The sniffer knows to look at this particular cons function because it is encountered during a backtrace through the control flow from the exit point of the PLAN. If more than one cons function with the required operands is present, all of them which are not on the same control path as the splice in operation are asked the same question. This investigation gives rise to the discussion of possible data paths which appears in the analysis section of the bug report.

It is trivial to determine that the function *metastasize* (see page metastasize_code) fails to remember the data returned by events-queue-insert. The cons-bug sniffer invokes the PLAN analyzer on the transform and examines the data flow out of events-queue-insert. In this case, there is none, so the result cannot be used. In the absence of any complications, this information generates the statement in the bug report that the caller of events-queue-insert expected the function to perform its action by side effect.

In a different implementation, it would be possible for metastasize to remember the results of the insert operation by executing the statement

```
(setq events-queue (events-queue-insert time item events-queue))
```

Since PLAN diagrams do not directly show the presence of setqs, the cons-bug sniffer would have to detect this occurrence by consulting the execution history instead.

The function metastasize might independently modify the queue if the data returned by events-queue-insert was treated as a flag indicating that the item was not inserted by side effect. This intent can be recognized by the presence (in the PLAN) of a conditional test on the results of the insert function, where the output branch taken leads to a cons operation binding the item to be inserted onto the list.

Once the cons-bug sniffer has identified that its bug is present, it establishes further context by using the time rover to examine the data values which were involved in the execution of the code. (The value of the events-queue quoted in the bug-report was obtained in this way.)

The primitives for investigating PLANs make it possible to determine the value which flowed across a particular line during a given execution. This ability is implemented through the time rover, and the indexing of PLAN elements with execution times I have spoken of. (Note that in the case where a line is not the direct result of a side effect operation which would be remembered in the execution history, it is derivable by duplicating the execution from an earlier point in the code.)

Given this ability, the inputs and outputs of functions which do not perform side-effects can also be reported. In addition, data values can be associated with their roles in recognized cliches by identifying the data flow lines they correspond to in the PLAN. When the bug report says that "the item was sorted to the top of the events-queue" it has identified that the predicate in figure 12 returned true, and that the predicate matches the PLAN for an *ordering predicate* which is a recognized cliche. The reference to the top of the queue is made possible by an examination of the values input to the predicate in this particular execution. The cons-bug sniffer has ample suspicion to pursue this line of investigation.

The cons-bug sniffer is also capable of recognizing that events-queue-insert does not maintain a header cell (a blank record at the top of the queue). This recognition comes from the observation that the function has a path for returning a value that does not require a side effect, and that this path is followed on a null list input. This last conclusion is not arrived at by inference; the sniffer is perfectly capable of testing code routines by executing them via the time rover. The sniffers can also scan the execution history of the code for empirical evidence. For example, the absence of a header cell might be detected by the fact that the top cell of the list changes some time after its creation. These kinds of abilities should make the sniffers capable of detecting a variety of errors.

6. Related work

To my knowledge, no previous work has had the primary goal of generating a deep understanding of bugs in programs. The most closely related efforts are Hacker (Sussman 1973) and Ruth's thesis entitled "The analysis of algorithm implementations" [Ruth 1973].

Hacker is a system that designs and modifies programs to solve problems in the blocks-world domain. It employs an iterative approach. The system proposes a possibly buggy solution for a problem, runs the code, and analyzes any error which is produced. Hacker then applies a method for modifying the code that is believed to correct the error of the type discovered. If the new solution does not work, the process is repeated.

Hacker is primarily an effort in learning and automatic programming. It is not a thesis about debugging. (The title of the system emphasizes this. Hacker's complete name is "A Computational Model of Skill Acquisition".) One of the system's major focus points is that it explicitly represents knowledge about coding. There is no doubt that Hacker demonstrates a deep understanding of the programs it writes. It can notice when a program violates one of the subgoals of a blocks-world task, and it can use the information associated with this error to generate a complex program that avoids the error. However, the process of bug classification is the least well-defined portion of the system.

Hacker gains a considerable amount of its leverage from the use of a toy domain which allows only a limited set of well understood operations. The errors which Hacker can discover are intimately involved with the blocks-world environment. In contrast, Sniffer recognizes bugs in arbitrary programs, and it is deeply concerned about the organization of knowledge required to locate a variety of errors.

Greg Ruth's dissertation describes a system that can recognize algorithms of a given class, and can also recognize buggy versions of those algorithms. The system is based on a grammar which identifies correct programs. It inputs a program, and a grammar for a class of programs, and attempts to parse the function with that grammar. If it succeeds, then the code is recognized as a member of the set of correct programs. Ruth extends the numbers of correct programs which can be found by applying a collection of behavior preserving transformations to the code being analyzed. Much of the knowledge in the system is involved with these rewriting rules.

Ruth's analyzer can also recognize programs which have errors. It does this by applying corrective transformations to the input code and then attempting to recognize the resulting routine. If the new function is found to be a member of the set defined by the grammar, then the error is analyzed as the inverse of the corrective transformation which was applied.

The kinds of bugs which Ruth's system can discover have a very syntactic feel. It treats programs as textual objects, without any detailed representation for their composition or the purpose of their parts. Sniffer, on the other hand, generates its power from an in depth analysis of the building blocks involved.

The Programmer's Apprentice Project at MIT has performed a good deal of work in the domain of program understanding. (My research is a part of this effort.) Rich and Shrobe [1976] laid down the basics for the decomposition of lisp programs into purposeful parts. Rich's dissertation [Rich 1980] develops a mathematical foundation for the representation of programs as PLANs and develops a library of PLANs for typical programming techniques. The complexity of these PLANs ranges from the level of a variable interchange to the PLAN for the queue and process strategy which is used in *prosper*. Rich also makes concrete suggestions for the construction of PLAN recognition systems which are directly relevant to the procedures used by the cliche finder (the cliche finder is designed for a much more constrained task). The PLANs recognized by the cliche finder can be found in this library. Sniffer is built on top of the system created by Richard Waters for his dissertation [Waters 1978] which has been discussed at length in this proposal.

There has been some work towards an abstract theory of bugs in programs (Miller and Goldstein 1977) which goes beyond the domain dependent classifications developed in Hacker. The authors propose a categorization for bugs that relates to a planning grammar used for designing programs. Syntactic bugs involve violations of the grammar, semantic bugs are concerned with a violation of the problem specification as expressed by a well formed statement in the grammar. This grammar does not have the conceptual richness of the PLAN representations used in the Apprentice Project, and the relation of the bug types to errors in more detailed programs is unclear. (For an excellent review article on the topic of program understanding and debugging, see [Lukey 1978].)

There has been a considerable amount of work in the creation of expert systems which perform complex tasks. The unifying characteristic of these efforts is that they often rely on a number of independent methods for gathering information and reference a large number of independent facts in the process of generating solutions. The methods used for organizing these tasks is directly relevant to my work on Sniffer.

The Simulation and Evaluation of Chemical Synthesis project [Wipke 1969] is an example (I would also place Macsyma, much of the work in AI and medicine, and the Dendral project in this class). SECS is an expert in the design of organic synthesis. The information relevant to this task includes empirical facts about reaction conditions and the sensitivities of functional groups, the 3-dimensional shape of the molecule, the composition of the molecule, and electronic energy levels of both the product and the reactants. To coordinate these different sources of information, the system confines a great deal of its expertise to a set of productions which examine these facts and determine if a given chemical reaction (applied to a particular molecule) will succeed or fail.

Work in the general area of tools for supporting the programming process is also relevant to my work in Sniffer. There are two different approaches to these tasks. First, there are systems that simplify the process of tracking down bugs, and second, there are methods that prevent bugs from happening in the first place. The first category includes debugging environments similar to the one implemented on the Lisp Machine, which provides a single step evaluator and predicates for examining the data in the function stack. Every major programming installation has some facility for monitoring the execution of assembly language code. The time rover is an extension of these techniques.

Bug prevention methods are primarily in the domain of software engineering. Many of the ideas included under this term relate more to the process of coding than to the structure of the code which is produced. However, data abstraction techniques [Liskov 1977] are particularly relevant to the kinds of errors which Sniffer detects.

Data abstractions occupy the borderline between program understanding methods and programming language techniques, since abstraction mechanisms build the level of vocabulary used to discuss a program. This is often done in very concrete terms; research in verification tends to rely heavily on data abstractions as a place to attach restrictions about the properties of code segments.

Data abstractions also imply a very strong form of type checking which makes certain kinds of errors much harder to commit. For example, the cons-bug error (which concerns the integrity of an object and the division of responsibility for maintaining its properties) can only be committed within the confines of a particular abstraction. These kinds of errors can not be totally avoided, but their frequency can be diminished with these techniques.

7. Summary

In this proposal I have described a system which has the competence to recognize, and deeply understand bugs in programs. It displays its understanding by describing errors in terms relevant to the purpose for the code, and by identifying the specific events which lead up to the manifestation of the error. The system is knowledgeable about side effects.

Sniffer organizes the knowledge required to find bugs into a collection of experts that understand specific errors. This approach is based on the observation that debugging is essentially an arcane science, with little theory available that can provide a systematic approach for locating an error. The expert system methodology excels in this type of task environment. It provides a simple, and modular organization for the quantity of details which are involved.

Sniffer implements its competence as a recognition process based on a structural analysis of the code which is involved. The use of recognition builds an understanding of programs in terms of their component parts. The resultant vocabulary is invaluable for discussions of the causes and solutions of errors. Recognition is shown to be a powerful, and conceptually simple alternative to the creation of deductive engines for elucidating facts about code.

Appendix I - Time table

I expect to complete the work outlined in this proposal by the end of the Fall semester, on January 15, 1981. Roughly a third of the time (2 out of 6 months) has been set aside for producing the final document, and the initial four months are reserved for the necessary coding.

The implementation can begin immediately, as this proposal provides a complete design for the system. My plan is to make an initial pass through each of the portions of Sniffer in order to develop a minimal system that duplicates the behavior of the scenario. This process should identify any problems that remain in the design.

The support functions have to be implemented first. This includes the time rover, the cliche finder and the primitives for investigating PLANs. The demon invocation mechanism that underlies the sniffer system is a standard technique in AI programming. The first version of the sniffer system will contain one bug expert, namely the cons bug discussed in the scenario. When the first pass has been completed, Sniffer will be extended to include additional bug-experts and each of the supporting facilities will be enhanced in turn.

Appendix II - Bibliography

Corey, E.J. and Wipke, W.T. Computer Assisted Design of Complex Organic Syntheses, Science v.166 no.10, pp 178-192, (October 1969)

Liskov,B., Snyder,A., Atkinson,R. and Schaffert, C. Abstraction Mechanisms in Clu, MIT/LCS Computation Structures Group Memo 144-1, (January 1977)

Lukey,F., Features, AISB pp 10-14 (December 1978)

Miller,M.L., and Goldstein,I.P., Structured Planning and Debugging, IJCAI5, MIT,pp 773-779, (1977)

Rich,C., and Shrobe, H.E., An Initial Report on a LISP Programmer's Apprentice, MIT/AI/TR-354 (1976)

Rich, C., A Library of Plans with Application to Automated Analysis, Synthesis and Verification of Programs, MIT Ph.D. thesis, (1980)

Ruth,G., Analysis of Algorithm Implementations, MAC/TR-130, (1973)

Sussman,G.J., A Computational Model of Skill Acquisition, MIT/AI/TR-297, (1973)

Waters,R.C., Automatic Analysis of the Logical Structure of Programs, MIT/AI/TR-492 (1978)