MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 255

February 3, 1972

WHY CONNIVING IS BETTER THAN PLANNING

Gerald Jay Sussman

Work reported herein was conducted at the Artificial Intelligence Laboratory, a Massachusetts Institute of Technology research program supported in part by the Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under Contract Number N00014-70-A-0362-0002.

Acknowledgement:

I must deeply acknowledge the profound influence of Joel Moses on this paper. Some of the ideas here are directly due to him; others were independently arrived at by him. Most of my ideas were arrived at by _observation_ of _real_ users of MICRO-PLANNER as part of my duties in its maintenance. Drew McDermott, Tom Knight, Terry Winograd, Greenblatt, Eastlake, and Gosper provided valuable sounding boards for these ideas.

The Problem with PLANNER

A higher level language derives its great power from the fact that it tends to impose structure on the problem solving behavior for the user. Besides providing a library of useful subroutines with a uniform calling sequence, the author of a higher level language imposes his theory of problem solving on the user. By choosing what primitive data structures, control structures, and operators he presents to the user, he makes the implementation of some algorithms more difficult than others, thus discouraging some techniques and encouraging others. So, to be "good", a higher level language must not only simplify the job of programming, by providing features which package programming structures commonly found in the domain for which the language was designed, it must also do its best to discourage the use of structures which lead to "bad" algorithms.

For example, consider the problem of calculating the n<u>th</u> element in the Fibonacci sequence. In LISP, the most natural algorithm for this computation is the exponentially exploding, doubly recursive:

```
                (DEFUN FIB (N)
                     (COND ((ZEROP N) 1)
                           ((ONEP N) 1)
                           (T (+ (FIB (- N 1))
                                 (FIB (- N 2))))))
```

A much better (linear in N rather than 2**N) but less natural (in LISP) algorithm is:

```
                (DEFUN FIB (N)
                     (COND((ZEROP N) 1)
                          (T (PROG (NM1 NM2 TEM)
                                   (SETQ NM1 1)
                                   (SETQ NM2 1)
                                LP (COND ((ONEP N) (RETURN NM1)))
                                   (SETQ TEM (+ NM1 NM2))
                                   (SETQ NM2 NM1)
                                   (SETQ NM1 TEM)
                                   (SETQ N (- N 1))
                                   (GO LP)))))
```

Thus LISP has led us down the wrong path. Is LISP a bad language because recursion is easy and automatic? I think not. The mechanism of recursive control structure, though the wrong one to use in this admittedly somewhat pathological case, is often both the most natural and most efficient control structure, especially in problems of symbolic manipulation for which LISP was designed.

With this in mind, let us now consider backtrack control structure, which occupies a place in PLANNER analogous to that of recursion in LISP. I contend that <u>automatic</u> backtracking is the wrong structure for the domain for which PLANNER was intended, that is, Artificial Intelligence. I will argue that:

1. Those cases in which automatic backtrack control is natural and appropriate are <u>always</u> the <u>worst</u> algorithms for solving a problem.

2. The most commonly used case of automatic backtracking can almost always be replaced by a purely recursive structure which is not only more efficient but also clearer, both semantically and syntactically.

3. The availability of automatic backtracking encourages superficial analysis of problems and poor programming practice; much worse, the pervasiveness of automatic backtracking in the PLANNER language discourages deep analysis of problems.

4. Attempts to fix 3 by the introduction of the artifice of failure messages are unnatural and cumbersome.

Thus I contend that the problem with PLANNER is automatic backtrack control structure. I must stress, however, that PLANNER has introduced many valuable constructs into our way of thinking, the most important of which are pattern-directed data base search and pattern-directed procedural invocation which tend to promote easy interfacing between programs, a great boon to our lab. Note that I am also not contending that good programs cannot be implemented in PLANNER; that would be absurd. I am only claiming that PLANNER does not encourage such behavior.

We now consider the points in detail.

1. We will readily admit that the "best" programs do no backtracking; they know where they are going at each step and never need to undo a bad decision. Good programs that know the structure of the problem domain (such as Moses' SIN) have no need for an ability to thrash about, searching for a good approach (as in SAINT). Pure backtracking (without failure messages) is essentially a mechanism for easily undoing a bad decision in the hope that a better alternative will be found. Thus it is only appropriate to algorithms which make such bad decisions either because of lack of sufficient guiding structure in the problem space or of sufficient knowledge of that structure in the program. At this point you may complain that in most interesting spaces not enough may be known about the space a priori to guide a program absolutely; that a good program may have to probe the space with experiments which then yield information which guides the rest of the program. Indeed, this is true; but it is just these cases, in which we want to be able to return not only information about why a particular method failed to achieve its

stated goal, but information about the structure of the space discovered along the way, in which we must resort to the highly unsatisfactory mechanism of failure messages.

2. Observation of Pat Winston's group's use of MICRO-PLANNER tends to indicate that one of the more important uses of backtracking, in programs which are not searching because they know exactly where they are going, is in information retrieval. These programs maintain rather massive data bases of information about a visual scene. Such programs often must be able to search out relevant goodies from a mass of irrelevancies. For example:

```
(GOAL (?X IS BIG))
(GOAL (?X IS GREEN))
(GOAL (?X ON $?Y))
(GOAL (?Y IS BLUE))
(stuff ?X ?Y)
```

This means "do the stuff" on objects X and Y such that "the big green X is on the blue Y." Note that what is going on here is sequential filtering of the possible assignments of X and Y by pattern directed search of the data base and theorems. We see that backtracking is used here because any choice of a particular big X may be bad because that particular X may not be green. The stack frame of each goal statement thus maintains a list of the hitherto untried possibilities and if a failure reaches it, it tries the next one and proceeds down. A much simpler and more straightforward approach would be to use ordinary recursive and iterative control structure to filter the possibilities directly. Thus, for example, it is easy to write a LISP function FOR-EACH with which one might write:

```
(FOR-EACH (?X IS BIG)
     (FOR-EACH (?X IS GREEN)
          (FOR-EACH (?X ON ?Y)
               (FOR-EACH (?Y IS BLUE)
                    (stuff ?X ?Y)))))
```

(of course a macro could be provided which expanded, say the following into the above)

```
(FILTERS ((?X IS BIG)
          (?X IS GREEN)
          (?X ON ?Y)
          (?Y IS BLUE)
         (stuff ?X ?Y))
```

Here, FOR-EACH is just a standard LISP function which, upon entry, looks up all of the assertions and theorems matching the pattern given as its first argument (with values substituted for variables which are assigned). It then assumes the first possibility, assigning variables appropriately, and evals its second argument. If the expression ever returns, rather than leaving the loop, the list of possibilities is CDRed and the process repeats. Notice that by appropriately nesting our loops no backtracking is required in the data retrieval. Here stuff is done on each X and Y which satisfies the criteria unless stuff decides it has had enough. This good nesting of loops has decided advantages. Besides being more efficient than backtracking (a marginal advantage), good nesting makes the scope of the action clear. There is no chance an unexpected failure will propagate back into this mess and chug along without our explicit programming of a failure catcher. I want to emphasize that this is not a made up problem. I am not sitting on my butt contemplating my navel. This problem is observed in real users of MICRO-PLANNER who complain that they just can't control their programs because they don't know what the programs are doing. The problem is really quite insidious. Usually any choice made in the offending piece of code eventually fails for the same reason that the first one did; then the only symptom that the program is running amok is that it takes forever to tell you it can't win. Let's consider the options available to the user to prevent this problem. You might say that he should finalize the program from just before the first filter to just after the code which he doesn't want reentered upon failure, but this is unsatisfactory because the code in question probably has side effects which should be undone upon failure from outside the intended code, but we want that failure not to be reversed by this block of code. All in all I think that it is essentially clearer if any loops or nesting structure is desired, that it should be made explicit rather than implicit so that the user is forced to think about what he is doing when he writes it down.

3. As PLANNER is currently organized, the easiest program to write in PLANNER is an exponential depth-first search. Other program organizations, though certainly possible in PLANNER, are clearly more complex. This is because PLANNER is trying to be both general and automatic. The defaults are chosen throughout the system so that backtracking happens unless you explicitly prevent it from happening. It is easy to say that people should write their programs to avoid backtracking except when absolutely necessary, but it is much harder to actually do it when the language gives you every opportunity to write bad programs.

4. In order to give the user a modicum of control over the dangerously uncontrolled backtrack mechanism, failure messages were incorporated. The basic idea of failure messages is that the user should be able to program in the ability to fail to a specific point which he sets up to catch the failure by matching its message. This does not give the user the ability to perform even the simplest of control functions. Suppose, for example, we have a goal which invokes a theorem. This theorem, in probing the search space, discovers structure in the space. It would like to get at the list of theorems which are pending in the goal which called it (the alternatives which will be tried if the current theorem fails) and edit it. It would perhaps like to filter it, deleting some entries and inserting others. It might even wish to sort the list of alternatives according to some general criterion. It has not yet,

however, failed, and thus cannot return a failure message. Furthermore, it cannot get at the list of alternatives pending on its failure. This is not a minor problem; it comes up in Greenblatt's chess program very often. For example, an analysis of a move may discover that we are in danger of being forked. This changes the whole set of criteria by which we want to judge alternatives. We must try to make a move which meets the discovered threat, if possible.

CONNIVER - A Different Approach

For some time I have been studying PLANNER and the uses to which it has been put, hoping to learn just what modifications would be desirable to the user community. As we have seen, these investigations have led me to decide that the basic structure of PLANNER was wrong, though its manifest success indicated that it contained many good and powerful ideas. This observation, that we were faced with a structure of good ideas glued together haphazardly with the poor glue of automatic backtracking, led to the design of a new language, CONNIVER, which incorporates those good ideas in a cleaner structure. CONNIVER is designed to satisfy the following desiderata: It must be automatic enough to be easy to use without being so automatic as to relieve the user of the responsibility for the behavior of his program. Thus, it must provide the user with the equipment to use powerful tools like backtracking and multiprocessing, relieving him of the low level bookkeeping needed to implement such mechanisms, without automating the invocation of these massive mechanisms. CONNIVER must walk this tightrope while maintaining simplicity by keeping the number and complexity of the primitives small. This path has been abandoned by PLANNER which started out as a simple and elegant (but not necessarily right) theory of problem solving, but which recently, in its effort to become an all inclusive panacea, gobbled up every idea ever proposed for the implementation of algorithms, regardless of considerations of either merit of the idea or consistency and parsimony of the theory. The last major consideration is that there must be no invisible control structure, such as the implicit loops with undefined scopes that were objected to in the last section. In CONNIVER the user is forced, often to his inconvenience, to explicitly outline his control structure, keeping him honest and his program clear. With this in mind I now proceed with a description of CONNIVER.

We first consider that part of CONNIVER which introduces no control structure more exotic than recursion. This is done mainly to show just how much of the "normal" programming people do in PLANNER via backtracking (the "natural" method in PLANNER) can be done more clearly, easily, and efficiently via the more conventional structures of recursion and iteration. We assume that CONNIVER has available to it a PLANNER-esque pattern matcher and is embedded, as is PLANNER, in a LISP-like language (such as MUDDLE). This section will deal primarily with the information retrieval aspect of the kind of programming people do in PLANNER, including the construction and maintenance of a pattern-directed data base and the use of pattern-invoked procedures. This, I have observed, comprises the largest proportion of legitimate MICRO-PLANNER programming.

In order to avoid confusion with PLANNER, similar constructs in CONNIVER will be given different names from those used in PLANNER. The objects in the data base which correspond to assertions in PLANNER will be called items, pattern-invoked functions, which correspond to theorems, will be called <u>methods</u> and also appear in the data base. In CONNIVER the data base is a tree structure of <u>contexts</u> and all additions, deletions, and searches are done relative to some context, just as variable bindings are searched and set relative to an environment in LISP. As we shall see later, variable bindings and flow of control in CONNIVER are also context dependent. For now, however, we need only know that the context is pushed (bound) at some functional (or methodological — ha!) invocations and popped (unbound) at some returns. As in LISP, every time we do a bind we obtain an environment which contains the previous environment, so in CONNIVER, each time we rebind the context, we obtain a new context which <u>contains</u> the previous context as a subset. It is important that the direction of containment be understood as it is critical and contrary to the direction of lexical scoping.

The data base construction primitives are as follows: (Syntactic variables are lower case, optional arguments are delimited by dashes, segment syntactic variables are delimited by stars, and procedural invocations by angle brackets.)

1) <ADD <u>item skeleton or method</u> -context->

2) <DELETE <u>item skeleton or method</u> -context->

These add (or delete) the item or method indicated to the context indicated. If no context is given, the current one is assumed. Objects in a context are accessible to all containing contexts. If the object to be added (or deleted) is already accessible (or inaccessible) in the context given ADD (DELETE) has no effect. Often, rather than actually deleting an object from a subcontext of a given context one really would rather hide it from all contexts containing the given context so that it will reappear when the given context is popped. Thus we have:

3) <HIDE <u>item skeleton or method</u> -context->

4) <REVEAL <u>item skeleton or method</u> -context->

REVEAL undoes a HIDE just as a DELETE undoes an ADD. Corresponding to the antecedent and erasing theorems in PLANNER, CONNIVER has context monitoring methods which monitor all contexts which contain them (except contexts from which they are hidden). The various monitor methods are:

5) <IF-ADDED declaration pattern *body*>

6) <IF-DELETED declaration pattern *body*>

If an item is added (or deleted) to a context the context is searched for monitors whose pattern matches the item affected and all such monitors are run in the order they are found. These monitors are not to be confused with the daemons which watch subcontexts from supercontexts and are described in the section on fancy control structures.

Now that we have our data base built up, how do we use it? CONNIVER has one basic primitive for accessing the data base:

7) <FETCH pattern -context->

FETCH searches the context indicated (if none is given the current one is assumed) for items and IF-NEEDED methods (analogous to consequent theorems) matching the pattern given. It returns a list of <u>possibilities</u>, the format of which will be discussed, but which logically consists of the items found followed by the methods applicable in the context. The user may, for example, examine the list, sort it according to some criteria, or edit it as he chooses. The main way he will use this list, however, is as follows:

8) <TRY-NEXT <u>list of possibilities</u> <u>no more</u> <u>return filter</u>>

Executing this primitive causes the following: If there are any possibilities, the first one is "tried" and removed from the list. If there are none left, the expression passed in <u>no more</u> is evaluated. Now what do I mean by tried? Now I must describe the format of the list. Suppose the current context has items:

(SUSSMAN IS-A CROCK), and (MOSES IS-A LOSER) and we execute: <FETCH (?X is-A ?Y)>

It will return as a value the list: (((X SUSSMAN) (Y CROCK)) ((X MOSES) (Y LOSER))).

That is, each possibility is a possible set of assignments for the variables in the pattern. Trying the first of these means setting X to SUSSMAN and Y to CROCK. These are <u>item possibilities</u>. Suppose one has exhausted the item possibilities and that the next possibility is a <u>methodological possibility</u>. The indicated method is executed. The value it returns to the TRY-NEXT is then interpreted as a set of item possibilities with which it is to be replaced. These, after being examined by the return filter, are then appended to the beginning of the list and the first one is tried. If none are returned, the first one is, of course, the next method. A method may thus return as a suggested possibility another method, not necessarily applicable in this context, thus providing a powerful mechanism for recommendations. Let us now look at an IF-NEEDED method:

9) <IF-NEEDED declaration pattern *body*>

As such a method runs, every so often it decides that the current assignments to the variables in the pattern are valuable and should be noted, to do this we execute:

10) <NOTE -variable->

If no variable is specified the note is made by adding the current substitution instance of the method pattern to a list stored in the value of the variable PROPOSALS, which is automatically bound on entering an IF-NEEDED. If a variable is specified it is the receptor of the noted proposal. The proposals are returned by:

11) <ADIEU -list of proposals->

ADIEU returns the list of proposals. If no such list is given, it returns from the method with the value of PROPOSALS. Similarly, if a method runs off its end, it returns the value of PROPOSALS.

By now you are probably complaining "So what, it looks like PLANNER to me, with the names changed and the data base slightly hairier." Note, however, that I have made no mention of any fancy exotic control structure; there is not yet any backtracking or failure (the same thing) or multiprocessing. These structures have their place in CONNIVER, as we shall see, but almost all of the programs currently written in MICRO-PLANNER can be written in the given subset of CONNIVER with only a net increase in clarity, efficiency, and without backtracking. I will illustrate this soon with examples, but first let me summarize that this recursive subset of CONNIVER is just the PLANNER data base searcher, pattern matcher, and pattern directed procedure invoker lifted from the encumbrance of all-pervasive backtracking and placed on its own to be used separately.

We shall now do the infamous (HUMAN TURING) example — often used to demonstrate backtracking. For convenience, before we start I wish to define a macro which packages a common loop. I want to make it clear that I do not condone the general use of FOR-EACH; it is just CONNIVER's way of simulating PLANNER's GOAL for illustration. It will <u>not</u> be provided as a primitive of the system, as it is too easy to use carelessly for searching without direction. In the following let <FOR-EACH pattern *body*> expand into:

```
<REPEAT ((POSSIBILITIES <FETCH pattern>))
     <TRY-NEXT. POSSIBILITIES <RETURN ()>>
    *body*>
```

We are using MUDDLE syntax. Note that when we run out of possibilities we just have the loop terminate by recursive return; there is no FAIL. We now compare CONNIVER with PLANNER. We first ask for a fallible object:

```
<PROG (X)                                   <PROG (X)
     <FOR-EACH (FALLIBLE ?X)                     <GOAL (FALLIBLE ?X)>
          <RETURN .X>>>                          <RETURN .X>>
```

So far, there is no difference; now we need the method (theorem) "humans are fallible".

```
        <IF-NEEDED (X) (FALLIBLE ?X)           <CONSEQUENT (X) (FALLIBLE ?X)
             <FOR-EACH (HUMAN ?X) <NOTE>>>          <GOAL (HUMAN ?X)>>
```

Again, no real difference. It comes, however, when we "need" backtracking to further restrict the answer to Greeks:

```
        <PROG (X)                              <PROG (X)
            <FOR-EACH (FALLIBLE ?X)                 <GOAL (FALLIBLE ?X)>
               <FOR-EACH (GREEK ?X)                 <GOAL (GREEK ?X)>
                 <RETURN .X>>>>                     <RETURN .X>>
```

Our first observation should be that we didn't really need backtracking since CONNIVER came up with the answer without backtracking or other control structure kludgery! I furthermore wish to argue that the CONNIVER answer is clearer than the PLANNER answer. My reasoning is that in the PLANNER case one <u>must</u> think about the implicit loop (find a fallible X, is he Greek? if not get the next one, if so return) in order to really understand the program. That loop, however, is not properly nested; its scope is not clear since you can fall back into it from the return below it. This form of badly nested loop, which is the hallmark of PLANNER's form of backtracking, is a very powerful but dangerous feature. Any such feature which disrupts the local control structure of a program (such as jumping into the scope of a DO loop) should be avoided like the plague, because of the resulting lack of clarity and modularity of code, unless its absence poses a serious hardship. As will be discussed in the next section, CONNIVER supplies, for the cases where it is needed, a very different form of backtracking whose presence is much less disruptive, and which easily replaces PLANNER type backtracking in those cases where it is justified.

HAIRY CONTROL STRUCTURES
the Truth about CONNIVER

By now you should be raising the serious question "What was that magic in the last section? How is Sussman cheating? Where has he paid for his heresy?" One possible objection to what I did is that I consed up a giant list of possibilities (or proposals) which I had to carry around with me. That objection, however, is invalid. If you have a program which has huge numbers of possibilities at each turn, your chances of getting a PLANNER program or a CONNIVER program to terminate before the sun blows up are equally infinitesmal. In either case, you had better come up with a better theory of the problem which narrows the search space to a more manageable proportion. Furthermore, what CONNIVER conses up as a list, PLANNER pushes on the stack since all untried possibilities must be eventually tackled. Then what is backtracking really about? Is there a case where backtracking is really relevant? The answer is <u>yes</u>. Consider the fact that whenever TRY-NEXT invokes a method, the poor method must come up with a proposal and then say "does this satisfy you?" expecting to work very hard to get the next one if the answer is no. Thus if there are only a few proposals but coming up with one is much harder than trying it out, we have a real need for backtracking. CONNIVER does, in fact, give you this feature if you really need it, but you must carefully think about it before you use it. The great disadvantage of backtracking is that the possibilities come only one at a time, thus we have no list to sort and filter and thus we find it harder to bring knowledge of the problem space to bear on the solution. CONNIVER provides the following backtracking primitive:

12) <AU-REVOIR -variable->

AU-REVOIR, like ADIEU, causes a method to return to the TRY-NEXT which invoked it with a list of proposals, either the value of the variable specified, or the value of PROPOSALS if none is specified. The difference is that AU-REVOIR specifies that if the proposals returned are exhausted and you still need more, the method which called AU-REVOIR will be resumed in the context it returned from as if AU-REVOIR had returned to it. The value of AU-REVOIR is a failure message which can be set by the user as the value of an optional fourth argument to TRY-NEXT. The user can then try to figure out how to use the message in the method failed back to. The example method of the last section can be written using backtracking as follows; no modifications are necessary to the caller:

```
<IF-NEEDED (X) (FALLIBLE ?X)
          <FOR-EACH (HUMAN ?X)
                    <NOTE>
                    <AU-REVOIR>>>
```

Note that backtracking in CONNIVER hardly interferes with the proper nesting of the (propose a goodie — try it out) loop structure of either the method or of the caller (unmodified). In contrast with PLANNER's chronological backtracking, the recursive backtracking of CONNIVER does not disrupt a program's recursive structure. Not only does this lead to benefits of clarity, legibility, and ease of debugging, (as well as ease of implementation), it also means that failure messages are much more useable in CONNIVER than in PLANNER because the guy who is rejecting a goodie sent to him complains directly to the guy who sent it, not one of his henchmen who hangs later down the backtrack stack.

You must realize, of course, that what is meant by backtracking in CONNIVER is not the same is what is meant by backtracking in PLANNER. Note that in backtracking to an AU-REVOIR no decision has been unmade. In general, CONNIVER backtracking does not mean undoing something which has been done. No items are ever removed from the data base except by an explicit call to DELETE or by popping the context in which the item resided by returning from it in the standard recursive way. The distinction becomes clearer if we consider the way one would handle a choice point in each language. In PLANNER you choose one of the possible choices and proceed, deducing the consequences of the choice made until either the program terminates, indicating that the choice was a correct one, or a failure occurs, undoing the deduced consequences until the choice point is reached, where a different decision is made. In CONNIVER, on the other hand, choices are made by calling one of several possible subroutines. A new context is then set up in which the consequences of the choice are stored. In any case, success or failure of the choice is indicated by the return of the function call which made the choice, unbinding the context of the choice. It is then up to the calling program to decide how to proceed. This decision may be made on the basis of examining the value returned by the subroutine and whatever changes it made to the more global data base and variable bindings. In this scheme a choice does not just succeed or fail. It very naturally has the power to probe the structure of the problem domain, adding whatever it learns to the contexts of its calling procedures, thus modifying their behavior.

In CONNIVER, therefore, what I called backtracking is really just a method of saving the current context (of the AU-REVOIR as the last proposal returned) so that it is not lost when popped (it is saved in POSSIBILITIES) and can be re-entered and continued where it left off (by TRY-NEXT, when it hits this new third kind of possibility, a context). This is really a very general form of mulitprocessing control. For such a control structure to be possible, the underlying applicative language must have (logically) a list structured control stack.

Besides giving us this different form of backtracking, CONNIVER also provides us with daemons. But what is a daemon? Daemon is one of those nice words that everybody "knows" the meaning of, until they try to formalize it. Perhaps first an example is in order. Many times, when attacking a problem in mathematics the first approach taken fails because of a specific unsolved subproblem. Another approach is then tried which, in itself, does not solve the problem, but which discovers information which unblocks the first approach (an impossible situation if it really failed in the PLANNER sense) which then proceeds to solve the problem. Ira Goldstein has given me a specific instance of this in geometry. In his case, the first approach to a proof, (and also the eventual winner) which was chosen first because of its extreme plausibility, becomes hung up on a subproblem that depends upon a very implausible construction. The second approach then finds the construction required extremely plausible and does it. The consequences of this construction should then wake up the first approach. CONNIVER provides the following primitives to implement these ideas:

13) <HANG <u>release condition</u> context>

If HANG is executed, the program ceases execution in the current context. Control is passed to the context given. If the release condition is ever satisfied by any computation, the running context is stopped and control reverts to the HANG which then returns as its value the interrupted context. Some possible release conditions are:

a) <ADD pattern -context->

b) <DELETE pattern -context->

c) <EXTERNAL-CONDITION type> where type might be CLOCK-TICK, TTY, etc.

d) <INTERNAL-CONDITION type> where type might be: FLOATING-

OVERFLOW, etc.

e) NEVER - means only if explicitly resumed.

The current context can always be gotten as the value of:

14) <CONTEXT>