MASSACHUSETTS INSTITUTE OF TECHNOLOGY
A. I. LABORATORY

Artificial Intelligence
Memo No. 259

May 1972

THE CONNIVER REFERENCE MANUAL

Drew V. McDermott and 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-0362-0003.

265

Introduction

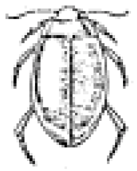
This manual is intended to be a guide to the philosophy and use of the programming language Conniver, which is "complete," and running at the AI Lab now. It assumes good knowledge of Lisp, but no knowledge of Micro-Planner, in whose implementation many design decisions were made that are not to be expected to have consequences in Conniver. Those not familiar with Lisp should consult Weissman's (1967) Primer, the Lisp 1.5 Programmer's Manual (McCarthy et. al., 1962), or Jon L. White's (1970) and others' (PDP-6, 1967) excellent memos here at our own lab.

The first chapter of this manual, Basic Conniver, is intended to show the Lisp user what Conniver programs look like, and what data structure the system provides for the user to interact with. By the time he is through with it, any Lisp writer should be able to write Conniver programs. Chapter the second, on Hairy Control Structure, explains the frame structure underlying the Conniver programs and encourages you to use it. Here the miracle of generator functions is revealed; here, too, is demonstrated the ease with which Conniver programs can do what your old-fashioned Micro-Planner programs do. The third section of the Overview, on Hairy Data Structure, explains the context-layered data base in somewhat more detail than in chapter one, and introduces some new data types and data structure-manipulating functions. The section On Pattern-Directed Invocation explains the operation of the Conniver matcher, used in the invoking of procedures associated with the data base. The remaining sections are on implementation details, such as using the language, and using it in conjunction with Lisp. Finally, the Appendix and Index provide a detailed guide to the user who has grasped the basic principles of the language.

Conniver embodies few original ideas, but is hopefuly an original combination of the good ideas of others. We must acknowledge Carl Hewitt's Planner language (Hewitt, 1971) for giving us most of our ideas about data structure, although Conniver looks at its world differently from Planner. The control structure, including the concepts "access" and "control," was enormously influenced by Daniel G. Bobrow. (Bobrow and Wegbreit, 1972). The variable declaration syntax is closely related to the MUDDLE syntax developed by Christopher Reeve. The way these ideas have been used is in large measure the way Joel Moses has thought they ought to be; if there is a "Conniver philosophy," a lot of it is his.

Several people read the first draft of this overview and influenced this one, especially David McDonald, Terry Winograd, Sidney Markowitz, and Jeff Rubin. The current semantics of data-property functions is partly due to a suggestion by Michael Genesreth. Last in this category, but not least in one respect, is Michael Levin; his confusion at the terseness of some of my explanations has gone one step in avenging the confusion of an entire generation of programmers at his Lisp 1.5 manual.

DVM



Water beetle;
Captotomus interrogatus
(Length ½ in.)

Basic Conniver

Conniver is a programming language designed to make easy the definition of processes cooperating to solve problems in the realm of Artificial Intelligence. It looks a lot like Lisp, with two additions:

(1) A system-maintained data base

(2) The ability to manipulate arbitrary control environments.

The data base structure gives the user a generalized way of storing most of the data in the world model he is creating. In addition, the data pool is hierarchical, so the user can run processes in independent, possibly conflicting contexts.

An important type of datum in the data base is the item, an arbitrary (but printable) list structure, such as

```
               (GOO1 (ITEM GOO1 IS FALSE))
or             (JOHN HATES (SHY GIRLS)).
but not        ((((((((((...    or (NIL NIL NIL NIL NIL NIL...
```

with the restrictions that at any given point it be present or absent from the data base. Items are slipped in and out with the functions ADD and REMOVE (which, like almost all Conniver primitives, evaluate their arguments). For example,

```
               (ADD '(JACK LIKES LEAN))
               (REMOVE '(JACK LIKES FAT))
```

make the item (JACK LIKES LEAN) present and the item (JACK LIKES FAT) absent. The arguments to ADD and REMOVE are skeletons which, upon substitution of the values of their variables, specify items. Commas indicate variable values, as in

```
          (ADD '(,X ON ,Y))
```

where the commas specify that the item ("value of X" ON "value of Y") is to be added to the current context. For example, if X = LAWRENCE and Y = CHATTERLY, then the skeleton specifies the item (LAWRENCE ON CHATTERLY).

The existence of contexts makes it easy to create and manipulate hypothetical models. A context can be regarded as a seperate world model or data base of its own. ADD and REMOVE always apply, implicitly or explicitly, to the data base represented by a particular context. These models are not entirely independent; contexts stand in certain logical relations to each other; namely, one may be a sub-context of another. This is meant in the sense that a stack frame of a language processor is a sub-frame of the frame beneath it on the stack. There is no subset relation between a context and its super-context; rather, items may be present or absent in one independent of their status in the other, much as some variables are rebound in a stack frame and others retain their old status. In the Conniver data base, all items have the same status (present or absent) in a context when it is created as they did in its super-context; each item retains that status until it is overriden by the user. For now, the reader may assume that any such re-specification of status within a context is not done in its super-context; hence, resetting a context to its super-context (the analogy with returning from a stack frame is overwhelming) makes everything done at the lower level invisible. We shall soon see examples of this.

A typical Conniver program generates a tree of <u>context-frames</u>, each branch of which constitutes a separate context. A user is started with a <u>global</u> <u>context</u>, a list of the one <u>global</u> <u>context</u> <u>frame</u>, with the global variable CONTEXT pointing to it. ADD and REMOVE work on the data base which is the value of CONTEXT (but can be given an optional second argument to indicate their context explicitly). The context can be "pushed down" with the function PUSH-CONTEXT, which sprouts a node from any context, adding a new context-frame. The value returned is a fresh sub-context of PUSH-CONTEXT's argument. (This structure is presented with more rigor and detail in the first paragraphs of Chapter 3.)

As a first example, let us consider a function FORCEWIN, of two arguments, PLAYER and SQUARE, which has a non-NIL value only if PLAYER can force a win on the next move of a game of tic-tac-toe by playing in square number SQUARE. (FORCEWIN is obviously only a fragment of a complete tic-tac-toe program.) To represent situations in this game, I choose the presence of an item of the form (HAS x y) to mean player x (= naught (O) or cross (X)) has put his mark in square y; if the item (FREE x) is present, it means square number x has no mark in it yet.

FORCEWIN, a Conniver CEXPR analogous to a Lisp EXPR, is defined with the function CDEFUN as EXPR's are by DEFUN. CDEFUN allows more sophisticated variable declarations than Lisp (see "Using Conniver," below); in particular, every CEXPR has a <u>body</u> in which auxiliary variables (called PROG variables in Lisp) may be bound, and statements labeled with atoms. Equivalent bodies are given to every COND clause and PROG. There are no tags in the example which follows, but there are auxiliaries to be bound, as signalled by the keyword atom "AUX".

```
(CDEFUN FORCEWIN (PLAYER SQUARE)
                 "AUX" ((CONTEXT (PUSH-CONTEXT CONTEXT)))
     (ADD ´(HAS ,PLAYER ,SQUARE))
     (REMOVE ´(FREE ,SQUARE))
     (MAKEMOVE (OTHER PLAYER))
     (RETURN (TRY-NEXT (WINMOVES PLAYER) NIL))   )
```

Here, "AUX" specifies that the variable CONTEXT is to be rebound in the frame of FORCEWIN to a new sub-context of its old value (Fig. 1).

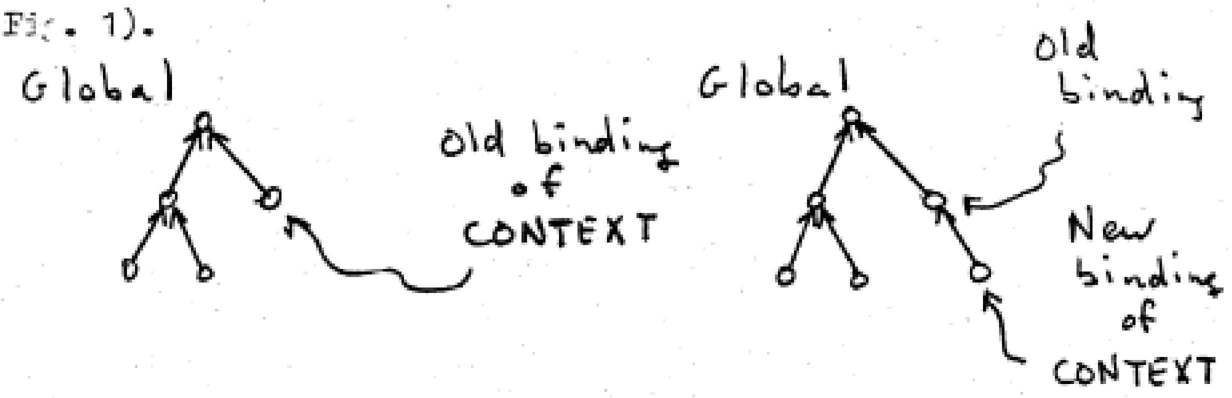


Figure 1.

Every item in the old context retains its present or absent status in the new binding, but new ADDitions and REMOVals apply only to the new context-frame. When this function is exited, CONTEXT´s old value will be restored, and the new sprout and all its items will eventually be garbage-collected (if none of the functions called by FORCEWIN saved a pointer to one of them, of course).

FORCEWIN operates by "imagining" that PLAYER has square SQUARE, and that the square is no longer free. It then imagines the other player (OTHER (X) = O; OTHER(O) = X), making a move. It does it with the function (MAKEMOVE player), which finds and makes player's best move in a situation. (Presumably FORCEWIN is itself a subroutine of MAKEMOVE.) The last line of the program then checks if PLAYER has a winning move, in spite of the other player, returning NIL if not; the system function TRY-NEXT is expected to return a winning move from the list proposed by the user function WINMOVES; it returns its second argument if the list is empty. FORCEWIN RETURNs its value, and rebinds CONTEXT, freeing the square SQUARE. (The RETURN could have been omitted; functions, like COND clauses and PROG's, return whatever value their last lines produce.)

WINMOVES is a generator-function, which interacts with a <u>possibilities</u> <u>list</u>, a very important data type in Conniver. Conniver is designed to ease the burden of writing problem-solving programs, many of which search through a problem-space network defined by a list of possibilities (solution plans, sub-goals, etc.) at each node. Game-playing programs come to mind immediately, but the same type of search occurs in programs designed to solve problems in language, vision, and theorem-proving.

Since the Conniver philosophy is that program design is the user's responsibility, it allows anything to be a possibility in such lists, and encourages the user to play with them. However, there are system possibility types (such as those generated by data-base searches), and system functions for inserting and extracting possibilities from possibilities lists. TRY-NEXT is the major function for extracting them; in the simple case of winning moves (here, just the numbers of the squares PLAYER can play in to make three marks in a row), it pops off the first possibility and returns it. (For a detailed account of the format of system possibility types, and what TRY-NEXT does with each, see the next chapter.)

The system provides the function CDEFGEN for defining generator-functions, routines which add possibilities to a list being considered by other routines. A generator-function, or generator, is like an ordinary program, except that it is expected to return a list of the possibilities it has discovered, which TRY-NEXT uses as described; in this way the generator communicates the values it wishes to return. In the simplest case, the possibilities are actual values; TRY-NEXT pops off and returns the next one, removing it from the possibilities list. When there are no more values, TRY-NEXT evaluates its second argument, and returns it. Hence the construction (TRY-NEXT (WINMOVES PLAYER) NIL) means "the first winning move, if any, else NIL."

The generator WINMOVES itself might be defined with CDEFGEN as follows:

```
(CDEFGEN WINMOVES (PLAYER)
                   "AUX" (SQUARE1 P1 SQUARE2 P2 X)
   (CSETQ P1 (FETCH '(HAS ,PLAYER ?SQUARE1)))
:OUTERLOOP
   (TRY-NEXT P1 '(ADIEU))
   (CSETQ P2 (FETCH '(HAS ,PLAYER ?SQUARE2)))
:INNERLOOP
   (TRY-NEXT P2 '(GO 'OUTERLOOP))
   (COND ((LESSP SQUARE1 SQUARE2)
          (COND ((CSETQ X (THIRD-IN-ROW SQUARE1 SQUARE2))
                 (COND ((PRESENT '(FREE ,X))
                        (NOTE X))   ))   ))   )
   (GO 'INNERLOOP)  )
```

(Notice that "AUX" variables not assigned an initial value are left <u>unassigned</u>, not made NIL as in Lisp; referencing such a variable before assigning it is an error.)

Since WINMOVES is a generator, special data structures will be set up when it is called. These include a binding of a variable called PROPOSALS that WINMOVES expands from its initial NIL value using the function NOTE. NOTE makes its argument a proposal by pushing it onto the PROPOSALS list. When WINMOVES is through with the list, it uses the function ADIEU to put it into the proper format (see next chapter), and return it. A later call to TRY-NEXT which inspects it will use it as a possibilities list. Since ADIEU reverses PROPOSALS before returning it, the proposals will be in the order NOTEd.

To understand WINMOVES, other details must be explained:

(1) (FETCH pattern) returns a possibilities list whose elements are <u>item possibilities</u>, derived from all the items

present that match pattern. (This is oversimplified; what such possibilities look like, how matching works and what other possibilities can be FETCHed will be explained later.) WINMOVES generates a possibilities list P1 specifying all the squares PLAYER owns, and regenerates it as P2 each time around OUTERLOOP. (No prizes are offered for a more efficient implementation of this generator.)

(2) When TRY-NEXT finds such an item possibility in a FETCH-generated possibilities list, its action is to assign the question-marked variables in the FETCH's pattern to the values that they matched in the item. Hence the statements tagged INNERLOOP and OUTERLOOP in WINMOVES set SQUARE2 and SQUARE1 respectively. Notice that atoms used as labels must be flagged with ":" (which turns them into expressions of the form (CTAG atom)).

(3) THIRD-IN-ROW returns the third square in the row, column, or diagonal of its two arguments, or NIL if they are not collinear.

(4) For each distinct pair of collinear squares owned by PLAYER, if the third square is free, then it is a winning move, and is NOTED. (The function (PRESENT pattern) returns T if there is an item matching pattern present in the current context.)

Thus the possibilities list returned by this version of WINMOVES includes <u>all</u> winning moves. (The (LESSP SQUARE1 SQUARE2) insures that each is found only once.)

Such a way of generating objects is not always the best. It may be, for example, much more expensive to generate each one than to try it out. Alternatively, the generation of successive possibilities may depend on what the calling function did with the previous ones. Or it may simply be that the generator has no idea how many possibilities its superior wants. (Note that FORCEWIN is interested only in the first.)

What is needed is a way of returning <u>some</u> of the possibilities while maintaining the generator in existence for further duty if required. The way this is done is with the primitive AU-REVOIR, which behaves something like ADIEU,but pushes one further datum onto PROPOSALS before turning it into a possibilities list. The new datum is a <u>tag</u> to the point just inside the AU-REVOIR in the generator. The POSSIBILITIES returned, therefore, includes a pointer to the generator's activation. When TRY-NEXT encounters such a thing in a possibilities list, it GOes to the tag, reawakening, in effect, the generator's process, including its bindings and context. The generator can then note new values and repeat the AU-REVOIR, or do an ADIEU, either of which this time returns its PROPOSALS to TRY-NEXT after converting it to a possibilities list; TRY-NEXT then splices it into the front of the possibilities list it already has.

A version of WINMOVES more congenial to use by FORCEWIN, and other functions that do not necessarily want all the winning moves at once, differs little from the old in appearance, but much in effect:

```
(C[illegible]FGEN WINMOVES (PLAYER)
                 "AUX" (SQUARE1 P1 SQUARE2 P2 X)
   (CSETQ P1 (FETCH '(HAS ,PLAYER ?SQUARE1)))
:OUTERLOOP
   (TRY-NEXT P1 '(ADIEU))
   (CSETQ P2 (FETCH '(HAS ,PLAYER ?SQUARE2)))
:INNERLOOP
   (TRY-NEXT P2 '(GO 'OUTERLOOP))
   (COND ((LESSP SQUARE1 SQUARE2)
          (COND ((CSETQ X (THIRD-IN-ROW SQUARE1 SQUARE2))
                 (COND ((PRESENT '(FREE ,X))
                        (NOTE X)
                        (AU-REVOIR))   ))   ))   )
   (GO 'INNERLOOP)  )
```

The only difference is the introduction of (AU-REVOIR) following (NOTE X). (This could have been abbreviated (AU-REVOIR X).) However, now a call to WINMOVES returns a PROPOSALS with just two elements: a winning move and a tag to the end of the AU-REVOIR.

If the tag is ever GOne to (by TRY-NEXT the second time it tries to pop off a winning move), AU-REVOIR will do a return <u>in WINMOVES</u> and execution will proceed with (GO 'INNERLOOP). The effect on TRY-NEXT will be that it will magically come up with yet another winning move and tag. Only when all winning moves have been generated can WINMOVES do an ADIEU, which leaves the possibilities list empty and causes TRY-NEXT to return its second argument.

These two examples do not exhaust the ways in which a generator may interact with a possibilities list. For sophisticated problems, it will almost certainly be necessary for generators to inspect the POSSIBILITIES bound in the frame of TRY-NEXT, filter some of them out, add properties to them that the program looking at them should know about, or even take control of their generation by setting empty the POSSIBILITIES bound in the frame of the upper TRY-NEXT and itself calling TRY-NEXT on each of the possibilities, in order to accomplish some particularly complicated filtering. The functions GET-POSSIBILITIES and SET-POSSIBILITIES enable a generator to access this binding of the list. Clearly, in order for a user's program to edit a possibilities list, he must know the formats of the various types of possibilities; these are given in the next chapter. Communication the other way, from the <u>user</u> of the generated possibilities, is made possible by an optional message argument to TRY-NEXT that it sends to the generator, which is returned, in the generator's activation, as the value of AU-REVOIR. All of these features are described in detail in the appendix.

Hairy Control Structure

Hopefully, the example I have been pursuing has made it clear that Conniver is just Lisp with primitives for dealing with its own control structures in one way or another. Conniver, lenient as it is about tags, function closures (FUNARGs in Lisp), and environments, allows processes to interact in many ways.

Conniver treats control environments as data types called frames. Each carries with it variable bindings, a control environment stack, a saved state of the Conniver interpreter, and, if CONTEXT is rebound in it, a data base context. An internal pointer to such a frame is an unprintable object which we refer to as a "fr," or internal frame. A user-manipulated frame is a structure of the form (*FRAME fr). A tag is a frame and a "program counter," of the form (*TAG body fr). Tags of a sort are produced and used implicitly with AU-REVOIR and TRY-NEXT; the user can generate them explicitly with the function TAG of one atomic argument, that returns a tag to the piece of PROG, COND, CEXPR, or GENERATOR labelled with that atom. For example, the following toy program prints out FOO BAR:

```
(CDEFUN PRINTFOOBAR () "AUX" (PLACE)
   (COND ((CSETQ PLACE (ZOWIE))
          (GO PLACE))   ))

(CDEFUN ZOWIE ()
   (PRINT 'FOO)
   (RETURN (TAG 'PRINTBAR))
:PRINTBAR
   (PRINT 'BAR)
   NIL)

(PRINTFOOBAR)
```

and returns NIL. (Note that GO always evaluates its argument, and expects an atom or a tag.)

Tags and frames are useful for many purposes. Relative evaluation, using the Conniver function (CEVAL expression environment) can take a frame or tag as its second argument. Functional arguments can be generated with the function (CLOSURE function) which generates the <u>closure</u> of function in the current frame, an object that behaves like function, but evaluates its free variables in the frame now associated with it. A closure is of the form (*CLOSURE function fr).

This flexible control structure can be used to provide an intimate association between a tree of problem-investigating Conniver processes and a tree of contexts. In particular, as in Planner, procedures can be invoked by the addition or removal of an item to a context, by virtue of being linked to a pattern that matches the item. Such <u>data base-sensitive</u> procedures are called <u>methods</u>, of type <u>if-added</u> or <u>if-removed</u> (or <u>if-needed</u>, discussed below). When an item is added (removed), any if-addeds (if-removeds) whose patterns match the item are <u>invoked</u>. When a method is invoked, a new frame is created for it, its variables are bound, and its pattern is matched against the item. If the match succeeds, execution begins at the front of the if-added's (if-removed's) body. For example, the (anonymous) method

```
(IF-ADDED NIL (HAS ?WHO ?SQUARE)
              ("AUX" (WHO SQUARE) (REMOVE ´(FREE ,SQUARE))))
```

with name NIL, pattern (HAS ?WHO ?SQUARE), and body ("AUX"...), automatically erases (FREE square) when it is asserted that (HAS someone square). Its use as a bookkeeper could save a line in the function FORCEWIN.

A method is itself a data-type stored in the context-structured data base, so it may be present only in the contexts the user specifies. Methods are ADDed and REMOVEd just like items, and like items, indexed in the data base by their patterns. The function IF-ADDED (IF-REMOVED) creates an if-added (if-removed) method with the pattern given by its first argument and the body given by the rest of them. The above method can be put in the current context by

```
(ADD (IF-ADDED (HAS ?WHO ?SQUARE)
         "AUX" (WHO SQUARE)
         (REMOVE ´(FREE ,SQUARE))   ))
```

and removed by REMOVing an object EQ to the one added.

This EQ-restriction means that an attempt by a user to re-read and ADD a file full of such anonymous methods (say, after editing a bug out of one) will put equivalent copies of all of them in the data base twice, all to be called twice when needed. To avoid this problem, an if-added (or if-removed) can be associated with an atomic name; thus

```
(ADD (IF-ADDED HAS-FREE (HAS ?WHO ?SQUARE)
         "AUX" (WHO SQUARE)
        (REMOVE '(FREE ,SQUARE))   ))
```

causes the atom HAS-FREE to be associated with the method (under the indicator 'METHOD), and to be passed around by the indexing routines. Executing the above expression a second time will now cause the method to be re-constructed (in case it had bugs in its previous incarnation), and associated with HAS-FREE, but not to be re-indexed, because the atom is equivalent to the method in the eyes of the system, and therefore already present. In fact, if (IF-ADDED HAS-FREE...) has been executed,

```
                (ADD 'HAS-FREE)
```

is equivalent to the ADD above.

The third method of data base-control structure interaction is by use of <u>if-needed</u> <u>methods</u>, which cooperate as intimately with FETCH as if-addeds and if-removeds with ADD and REMOVE. Often there is a class of data items which are to be regarded as "present" in a context, but on the basis of some procedural criterion rather than by virtue of actually being there and FETCHable. An if-needed can be used to associate such a procedure with the pattern of a typical item of the class. Any if-neededs present in a context will be found by FETCH, if their patterns match its pattern argument, and stuck at the end of its possibilities list. They are invoked by TRY-NEXT when it comes to them; their auxiliary variables (signalled as usual by "AUX") are bound, and their pattern variables assigned by a match. If it succeeds, execution begins in the method, which behaves like a generator function with respect to the possibilities list TRY-NEXT is working on.

Within an if-needed method, the function INSTANCE of no arguments returns an instantiation of the method's pattern, with all variables given their current values. Then (NOTE (INSTANCE)) (or simply (NOTE)) causes such an <u>instance</u> <u>proposal</u> to be added to PROPOSALS and ultimately (as an item possibility) to POSSIBILITIES, thus simulating very nicely the presence of that instance as an item in the current context. ADIEU and AU-REVOIR work in the same way as before.

For example, to express the idea that all dwarves are vicious, in such a way as to insure that FETCH finds all dwarves when it looks for vicious persons, one might execute

```
(ADD (IF-NEEDED VD (VICIOUS ?X)
          "AUX" (X (P (FETCH '(DWARF ?X))))
   :LOOP (TRY-NEXT P '(ADIEU))
         (AU-REVOIR (INSTANCE))
         (GO 'LOOP)    ))
```

This method notes one vicious dwarf each time TRY-NEXT is called.

The discussion has brought us round to proposals and possibilities again, and it is worth stopping here to explain the format and contents of their lists.

While in a generator (including an if-needed method), PROPOSALS is a simple list, started at NIL the first time it is bound and every time the generator is re-entered via an AU-REVOIR tag. The generator pushes proposals onto this list with NOTE, (NOTE x) being equivalent in effect to (CSETQ PROPOSALS (CONS x PROPOSALS)).

The system supports only one special type of proposal, that produced by INSTANCE, which is an object of the form (*ITEM (instantiated-method-pattern) result-of-match), where result-of-match is an association list (as described in "On Pattern-Directed Invocation," below), which specifies the values of the variables in the calling pattern that will make it EQUAL to the present instance. Instance proposals are added to PROPOSALS as the non-special kinds (simply called <u>values</u>) are (but note will not admit them to the list if result-of-match is NOMATCH). For example, if the method VD finds a dwarf named MILHOUS, the instance proposal

(*ITEM ((VICIOUS MILHOUS)) ((X MILHOUS)))

will be created.

A generator usually quits using ADIEU or AU-REVOIR, either of which reverses the proposals list and adds the atom *POSSIBILITIES to the front of the result before returning it.

A possibilities list is just like a proposals list with the flag *POSSIBILITIES at its front. However, since TRY-NEXT may get possibilities from several sources, there are more types of standard possibilities than proposals. The various types have the following formats:

(1) (*ITEM item-or-simulated-item result-of-match) These are

produced by FETCH (from the current context) and by if-needed methods (from thin air, using INSTANCE). The way to tell if the item is simulated or not is to use the predicate REAL. (See "Hairy Data Structure," below. It should be noted that in both cases, the item is the item datum associated with the list structure mentioned at the beginning of "Basic Conniver," not that list structure itself; ((VICIOUS MILHOUS)), not (VICIOUS MILHOUS). In terms of the data structure, a simulated item is just an absent one; the next chapter must be read to understand this fully.) When TRY-NEXT sees an item possibility, it returns the item-or-simulated-item, besides assigning the variables as directed by result-of-match.

(2) (*METHOD pattern method) These are produced by FETCH (pattern is the FETCH-pattern), but there is no reason a generator could not CONS one up as a value proposal.

(3) (*GENERATOR form) TRY-NEXT evaluates form when it comes across one of these; it expects the value to be a possibilities list, which it splices into the one it has in place of this possibility, just with *METHOD's and *AU-REVOIR's. Form is usually (generator...).

(4) (*AU-REVOIR body fr) (Don't try to print one of these.) Such a possibility differs from a tag to its body and frame only in having *AU-REVOIR as its flag instead of *TAG, but don't try GOing to one, either; only TRY-NEXT is allowed to do that. It should be clear how these get into proposals lists.

(5) Anything else is a value, which TRY-NEXT returns when it pops it off a possibilities list, as in FORCEWIN.

Having described the "internal syntax" of the generator-possibilities interaction, I now return to consideration of control structure, in particular, consideration of the fundamental operations on frames. I start with the observation that methods may have closures just like functions and generators do, and these, too, can be added to the data-base. If such a method is invoked by a data-base change, control will be in a procedure with an access link that differs from that of its caller (like functional arguments in Lisp). This raises the possibility of a process in an old environment being awakened by the addition of an item to its context, or the removal of one from it. In fact, the function HANG can bring exactly this state of affairs about. (HANG is not a Conniver primitive.) (HANG release expression) evaluates expression (typically a transfer or return), but only after ADDing a method closure that implements a test for the release condition. This condition is of the form (IF-ADDED pattern) or (IF-REMOVED pattern). If an item matching pattern is ever added (or removed, as the case may be), HANG returns as its value the frame of the process which was interrupted while adding (or removing) the item, with the side effect of assigning the variables of the pattern.

For example,

```
    (HANG '(IF-ADDED (WIN ?PLAYER)) '(GO 'FOO))
```

goes to :FOO, but execution will resume with a return from HANG if anyone adds (WIN someone) to the data base, and PLAYER will have gotten value someone.

HANG can be defined as

```
(CDEFUN HANG (RELEASE EXPRESSION)
             "AUX" (VALRET (C (CONTROL)))
   (ADD (CLOSURE
          (CEVAL (CONS (CAR RELEASE)
                       (CONS (CADR RELEASE)
                             ("AUX" ((F (FRAME)))
                              (CSETQ VALRET F)
                              (GO 'HANGRET)    ))))))
   (CEVAL EXPRESSION C)
:HANGRET
   (RETURN VALRET)    )
```

By adding the CLOSURE of the method, the HANG is assured of the continued existence of its activation. When the pattern is seen, the method sets VALRET (in the environment it was closed in, naturally) to point to its (the method's) own frame, which has a control pointer to the frame (of an activation of a subroutine of ADD or REMOVE) that invoked it. Then it GOes to HANGRET. The atom HANGRET is searched for in the access frame of the closure (i.e., the frame it was closed in), the correct label is found, and control is suddenly back in HANG, which returns the given frame. Notice that, having added to the current context the closure that does these marvelous things, HANG evaluates (CEVALuates) EXPRESSION in its (HANG's) control frame, the frame of its caller, which is what the user presumably intended.

HANG thus exploits the fact that every frame has two superior frames it points to, an access frame used for free variable evaluation and atom tag searching, and a control super-frame that control is expected to return to. Usually (as in Lisp), they are identical, or the access pointer points a few frames above the beginning of the control chain, to the last frame where variables were bound. But there is no reason for things to be so prosaic.

Several functions have been provided for use in manipulating these objects. The function (FRAME) returns the current frame, one level up from the frame of the invocation of FRAME. (On the top level of a CEXPR, this will always be the frame in which its local variables are bound). (CONTROL frame) and (ACCESS frame) return the control and access pointers of frame. The CONTROL and ACCESS of a tag or closure are legal also. (SETACCESS frame1 frame2) and (SETCONTROL frame1 frame2) reset the appropriate super-frame of frame1 to be frame2.

Closure and relative evaluation are ways of treating frames as access environments. By EXITING a frame (with (EXIT value frame)), the user utilizes the control functions of frames. The following fragment of code is illustrative:

```
(PROG "AUX" (X WAYOUT)
   (CSETQ WAYOUT
          (HANG '(IF-ADDED (?X BERG)) '(GO 'USEFULWORK)))
   (PRINT '(SOME OF MY BEST FRIENDS ARE BERGS))
   (EXIT T WAYOUT)   )
```

which GOes to :USEFULWORK when executed, but prints its message if (anything BERG) is ever ADDed to its context. WAYOUT, for control purposes a subframe of ADD, is then returned from to allow the ADD to proceed normally.

When no frame value is given, EXIT exits from the most immediately enclosing COND, PROG, or CEXPR. RETURN bypasses COND, so is often more convenient.

In these terms it can be explained how TRY-NEXT interacts with various generators; a generator is an (otherwise ordinary) function with PROPOSALS bound in it by the system, and a way to tell, from the deepest AU-REVOIR tag into it, where its top frame is. ADIEU and AU-REVOIR help it to manipulate and return the proposals list, but (RETURN (CONS '*POSSIBILITIES 'FOO)) would work just as well as (ADIEU 'FOO). The trick is in TRY-NEXT; when it finds an AU-REVOIR tag in a possibilities list, it replaces the control link in the top frame of the generator structure to point to the new TRY-NEXT, and just goes to the tag. Finding the top frame is very simple; within any generator activation, *GENERATOR is always bound to it.

Please note that there is only one type of frame, suitable for both access and control functions. Any frame can be used for relative evaluation, or can be exited; The user can do relative evaluation with respect to the same tag he later GOes to. Other examples of the dual function of frames are in GET-POSSIBILITIES and SET-POSSIBILITIES, which operate by using the control link of a generator as an access environment for the variable POSSIBILITIES. This works because of the fact that after its first call a generator's control link points to a sub-frame of TRY-NEXT, where POSSIBILITIES is bound.

This control structure is intended to be manipulable by the user. HANG, for example, is written in Conniver, not Lisp. The control structure primitives of Planner can be written fairly simply in Conniver as follows:

```
(CSETQ FAILURE-STACK NIL)

(CDEFUN FAIL () "AUX" (T1)
   (COND ((NULL FAILURE-STACK) (PRINT 'FAILED) (GO EAR-1)))
   (CSETQ T1 (CAR FAILURE-STACK))
   (CSETQ FAILURE-STACK (CDR FAILURE-STACK))
   (GO T1)   )
```

(EAR-1 is explained under "Using Conniver," below; (GO EAR-1) gets a program to the top level.) This version of Planner maintains a list FAILURE-STACK of environments to fail back to. The list is taken apart by FAIL, which pops off the next element and GOes to it. The list is built by FAILSET:

```
(CDEFUN FAILSET (T)
   (CSETQ FAILURE-STACK (CONS (TAG T) FAILURE-STACK))   )
```

Note that since FAILURE-STACK is an ordinary Conniver variable, there may be local bindings of it, hence a complex structure of failure stacks bound at different levels.

```
(CDEFUN GOAL (PATTERN) "AUX" ((DATA (FETCH PATTERN)))
:GOALF
   (FAILSET 'GOALF)
   (TRY-NEXT DATA '(PROG (CSETQ FAILURE-STACK
                                (CDR FAILURE-STACK))
                         (FAIL)))   )
```

This version of GOAL obeys Conniver conventions for data base search, pattern format, etc., but behaves like the Planner version in that it responds to a failure by TRYing-the-NEXT matching datum unless there aren't any, in which case it continues the failure. Since this GOAL works with if-needed methods instead of consequent theorems, a Planner version of NOTE must be invented. It looks like this:

```
(CDEFUN THNOTE ()
   (FAILSET 'THNOTEF)
   (RETURN (NOTE))
:THNOTEF
   (CSETQ PROPOSALS (CDR PROPOSALS))
   (FAIL))
```

THNOTE behaves exactly like NOTE of no arguments, except that it excises its proposal from the proposals list and continues failing if a failure hits it. A program executing (THNOTE) (ADIEU) can use the abbreviation (SUCCEED):

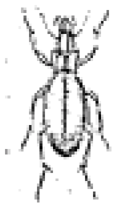
```
(CDEFUN SUCCEED ()
   (FAILSET 'SUCCEEDF)
   (ADIEU (INSTANCE))
:SUCCEEDF
   (CSETQ PROPOSALS (CDR PROPOSALS))
   (FAIL)   )
```

The two remaining functions simulate ASSERT and ERASE in that the r effects are undone on failure:

```
(CDEFUN ASSERT (SKELETON)
   (FAILSET 'ASSERTF)
   (RETURN (ADD SKELETON))
:ASSERTF
   (KILL SKELETON)
   (FAIL)   )

(CDEFUN ERASE (SKELETON)
   (FAILSET 'ERASEF)
   (RETURN (REMOVE SKELETON))
:ERASEF
   (INSERT SKELETON)
   (FAIL)   )
```

KILL and INSERT are versions of REMOVE and ADD which do not search for and invoke if-removed or if-added methods; here they are used to undo the effect of ASSERT and ERASE before failure is allowed to propagate.



Beetle² (def. 1) Pantomor-us godmani (Length ½ in.)

Hairy Data Structure

Hopefully, the user has understood the references to data base manipulation so far. In fact, the range of operations open to him is much larger than might be supposed.

In this chapter, I am going to build up again the notion of context, starting this time with meaningless "fragments" of world models called context-frames. Contexts are implemented as ordered branches of a tree of such frames. A context-frame (c-frame) is used to define changes to a world model as one takes as his context longer and longer hunks of a branch of the tree, starting from the root; this semantics is exactly reflected in the definition of PUSH-CONTEXT, which gives a program one more frame in which to indicate such changes.

To be precise, a c-frame's sole function is that every datum in the data base may be thought of as realized, unrealized, or unspecified by every c-frame; for now, these frames have no relevant internal structure; they might as well be buckets full of mentions of data as "realized" or "unrealized."

The user, of course, is building contexts out of them, lists flagged with *CONTEXT as their first elements, followed by c-frames. In a particular context, a datum is always either present or absent, its status depending on search rules through the c-frames of a context, which are ordered from "most local" or "most recently pushed," to "most global."

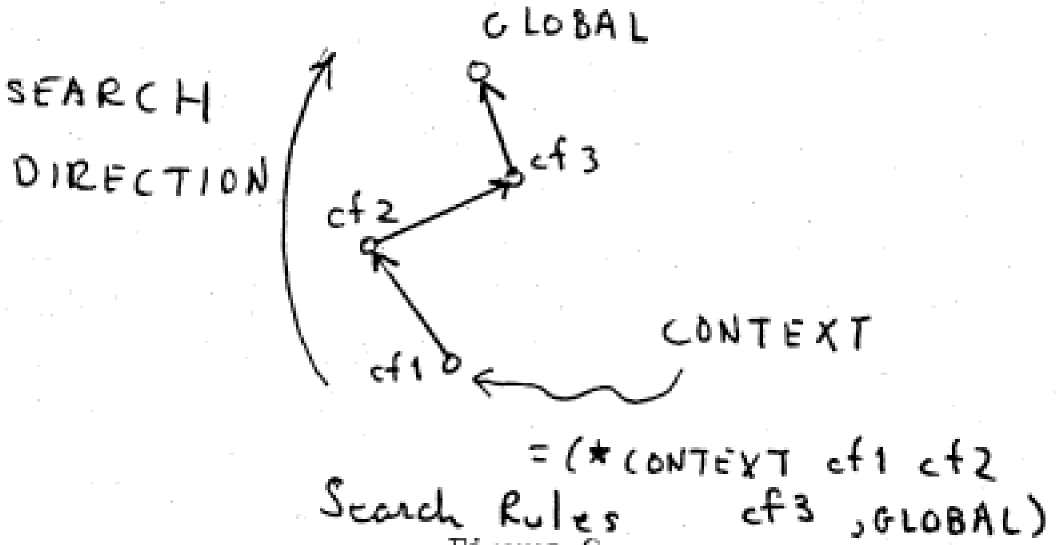


Figure 2.

To determine the status of a datum in a context, find the first (most local) c-frame of the context where the datum has specified status, and use that status, either realized or unrealized, to specify its status in the context as present or absent, respectively. If none of the c-frames define a status, it is absent. For example, in Fig. 3, if the marks "+" and "-" indicate reality or unreality of a certain datum in the c-frames next to them, the examples of contexts built from these frames show under what circumstances a datum is present:

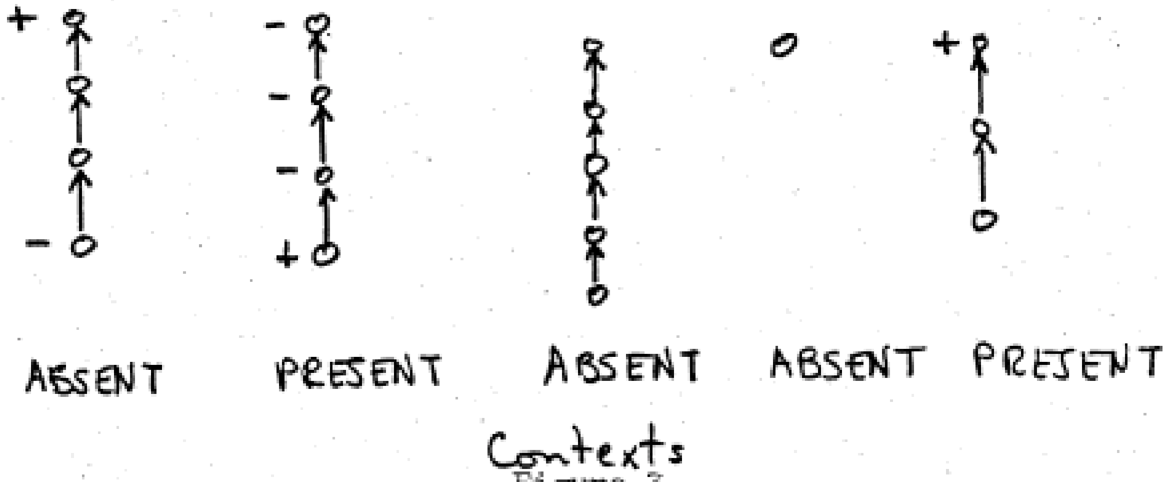


Figure 3.

Notice that a datum may have unspecified status in most c-frames.

I illustrate these search rules with examples of perhaps the most primitive data in this scheme, those of type object, created with the function OBJECT. Objects are viewed by the data-base managers as arbitrary list structures whose only system-maintained properties are presence and absence. The programmer can use these properties to model any semantic features he likes.

For example, a vision program, as it reconstructs a visual scene from a vidissector image, must consider more than one set of possible real-world objects, and decide what is really there on the basis of which is most consistent with the evidence. This world might be modeled as a list of Conniver objects, only some of which are present in any context. Thus, an object proposer might summarize its conclusions by adding a new data object to the list POSSIBLE-OBJECTS:

```
(CSETQ POSSIBLE-OBJECTS
       (CONS (CSETQ NEW-OBJECT (OBJECT '(R4 R5 R9)))
             POSSIBLE-OBJECTS)).
```

This form creates a possible object, NEW-OBJECT, considered to consist of regions 4, 5, and 9. (A realistic data structure would undoubtedly have to contain more information.) This object looks like (*OBJECT (R4 R5 R9)), and has structure (R4 R5 R9), which the system ignores. New objects are, of course, absent in all contexts.

To make this datum present in the current context by virtue of realization in its top c-frame, one executes (REALIZE NEW-OBJECT); to make it absent (by virtue of unrealization there), he xecutes (UNREALIZE NEW-OBJECT). The predicate REAL returns its argument if it is present, or NIL if it is absent; UNREAL, the opposite. (These predicates test for presence and absence in the current context, and thus consider all c-frames according to the search rules given; they do not merely look at the top c-frame.)

To illustrate the use of these primitives, imagine a data structure for tic-tac-toe as follows. Let XS be a Lisp array of 9 data objects like that above, such that (XS n) is the X in the square n; let OS be a similar array of O objects. With this data structure, the predicate (FREE square) can be defined thus:

```
(CDEFUN FREE (SQUARE)
   (NOT (OR (REAL (XS SQUARE)) (REAL (OS SQUARE))))   )
```

To put an X in square 5 (the center), for example, execute

```
                (REALIZE (XS 5))
```

If this is done in a particular context, the board will "have an X in the center" in that context and all contexts sprouted from it. By resetting or rebinding CONTEXT to a higher point in the branch, the "X" modeled as (XS 5) can be made to "vanish," as (XS 5) reverts to absence.

To summarize, a context is a branch of a structure of context-frames. The structure can be grown at its tips (using PUSH-CONTEXT as described in "Basic Conniver"), and manipulated in other ways to be mentioned. The search rules through a context are such that the presence of a datum is not disturbed by pushing new c-frames onto a context unless it is specifically

UNREALIZEd.

So far this section has concerned itself with data objects whose only properties are presence and absence. The reason for this focus is to isolate the search rules that define "present in a context," as predicated of any datum, and the functions REALIZE and UNREALIZE, and the predicates REAL and UNREAL, that make the concept useful.

In fact data can have many more useful attributes than presence. First, as pointed out, an object may be created with an arbitrary structure. For example, still another representation of a tic-tac-toe situation would be a Lisp array SQUARES of 9 objects, each having as structure its number in the "magic square" representation of tic-tac-toe (in which the numbers in every row, column, and diagonal add to 15; this would be of use in THIRD-IN-ROW). The upper left-hand square would be created by (STORE (SQUARES 0) (OBJECT 2)), for instance; the object (SQUARES 0)'s structure is then "2"; this object looks like (*OBJECT 2), and, at creation time, is present in no context.

Second, any datum may have <u>properties</u> in any c-frame. This feature is dealt with below.

Third, some types of data are <u>indexable</u> and can be <u>searched</u> for (internal operations of ADD, REMOVE, and FETCH), by virtue of association with a pattern. These are, of course, items, methods, and closures of methods, which are like non-indexable objects in every other respect. The difference between an item and an object is that an item may be specified by a skeleton or pattern (although it doesn't have to be), but an object must be mentioned directly. The user should perceive the similarity between ADDing (HAS X ,SQUARE) in FORCEMIN and REALIZing (XS SQUARE) in the simpler OBJECT array representation of a state of a game of tic-tac-toe. ADD simply REALIZEs the item its skeleton represents. Both routines, given indexable data arguments, make sure the data referred to are indexed and all relevant if-addeds are called. A similar relation holds between REMOVE and UNREALIZE. The choice between the item or object representation should be based, among other things, on how the user wishes his data to interact with his programs.

Notice that, since ADD and REMOVE are merely ways of referring to items by skeletons, using them to handle methods, referred to directly or by name, is <u>synonymous</u> with use of REALIZE and UNREALIZE. In fact, even an item, once in the data base, looks something like

```
        ((HAS O 4) (52 -) (O +))
```

(see description of c-markers near the end of this section), where the item ADDed (a new list structure derived from, say, (HAS ,X ,Y)) is only the first element of the actual <u>item datum</u>. Therefore, item data, as data returned by ADD, REMOVE, PRESENT, ABSENT, (TRY-NEXT (FETCH...)), etc., indexed or unindexed in the data base, can be REALIZED and UNREALIZED just like the others.

Although items and objects can be arbitrary list structures, it is very often desirable to separate a datum's "essence" (e.g., (FRED FLOGS FOOLS)) from its "accidents," or <u>its properties by virtue of being present in a context</u>, such as

(REASON (FRED IS-A SADIST)).

Association of <u>indicators</u> like REASON and <u>properties</u> like (FRED IS-A SADIST) must be relative to a context-frame. Every datum is <u>mentioned</u> by a set of context-frames, as realized, unrealized, or as having properties, and associated with each such mention is a property list which contains pairs as shown.

To associate indicator with property in context, on datum, use

(DPUT datum property indicator context),

where context is optional with default value CONTEXT. This causes the first c-frame of context to mention datum if it does not already (it leaves datum's present or absent status unaltered), and associates indicator with property with respect to that mention.

(DGET datum indicator context)

returns the first indicator-property pair found by searching through <u>all</u> mentions of datum by c-frames in the current context; or NIL if there is no pair with indicator in any mention of datum by context's c-frames. Finally,

(DREM datum indicator context)

does a DGET, but removes the pair if it finds it.

Usually one does not wish to refer to the mention of a datum by all the c-frames of the current context, but only to the mentions that specify realization, those created by ADD or REALIZE. These functions mention the datum as being realized in a particular c-frame; such a frame will be the first status-defining frame in the context in the case of any present datum. To add to the datum's properties in this c-frame, use the function DPUT+. (If the datum is in fact absent, an error will occur.) To retrieve and remove properties only from the set of all "+"-marked frames up to the first "-"-marked one, use the functions DGET+ and DREM+. (If they are given absent arguments, they always returns NIL.)

For example, if CONTEXT is used by a program to mean "the world as it stands now," and YESTERYEAR points to a higher branch (a super-context) of CONTEXT, a program may find out that something was true that no longer is, and indicate its current status, in the following way:

```
(COND ((CSETQ ITEM (PRESENT '(EXIST 5-CENT CIGARS)
                            YESTERYEAR))
       (COND ((UNREAL ITEM)
              (DPUT+ ITEM
                     'BY-GONE
                     'CURRENT-STATUS
                     YESTERYEAR))   ))   )
```

which construction saves it from having to discover which c-frame of YESTERYEAR it was realized in. Notice how REAL and UNREAL work with items and methods as well as objects.

As another example, I return to the representation of the tic-tac-toe board as an array SQUARES of 9 objects. Let each such object specify the occupant of the corresponding square in a particular context as its property under the indicator OCCUPANT; if it is empty, let the object be absent in that context. Then FREE can be written

```
(CNFUN FREE (SQUARE) (UNREAL (SQUARES SQUARE))   )
```

and the occupant of a square in the current context might be found by

```
(AND (REAL (SQUARES SQUARE)) (CADR (DGET+ (SQUARES SQUARE)
'OCCUPANT)))
```

which returns X, O, or NIL. Then FORCEMIN could be written

```
(CNFUN FORCEMIN (PLAYER SQUARE)
                 "AUX" ((CONTEXT (PUSH-CONTEXT)))
   (DPUT+ (REALIZE (SQUARES SQUARE)) PLAYER 'OCCUPANT)
   (MAKEMOVE (OTHER PLAYER))
   (TRY-NEXT (WINMOVES PLAYER) NIL)   )
```

If the semantics of property-list manipulators does not quite fit your needs, there are more primitive functions, described in the Appendix, which enable you to tailor-make your own versions of them.

If the user is to understand this data structure completely, he should know the formats and properties of context-frames; each one is simply a list of the form (*CFRAME cnum *data*), where cnum is a number unique to the frame, and data are the data it mentions; GLOBAL, however, for internal reasons, always looks like (*CFRAME 0). A context is a list (*CONTEXT *c-frames*), where c-frames are context-frames <u>in order of decreasing cnums</u>, the last of which is always GLOBAL.

It is a useful feature of context frames that they vanish when no one points to them, i.e., they are garbage-collectable. When one so vanishes, it takes every mention of a datum in it with it, along with all properties and, of course, any status definition. However, although the mention of the datum in that c-frame is gone, all other c-frames' references are intact. Thus one may PUSH-CONTEXT, assign some properties to a datum in the new context (using DPUT, DGET, and DREM), while doing a little computing. If he then flushes the frame, all the properties will vanish, while the datum's status remains the same.

The rest of this section is concerned with the details of system interaction with the index, garbage-collection of indexable data, the semantics of absence vs. presence, formats of the markers on data that define their context-sensitive properties, and esoteric property manipulators. Rather than attempt it on a first pass, you may wish merely to skip to the last page and heed the warnings printed there.

For those with confidence in themselves, I begin with the problems raised by esoteric property-handlers. Notice that functions like UNREAL and REMOVE return data whose first status-defining mention in the current context specifies unrealization. To manipulate properties in the c-frames mentioning a datum this way, use DPUT-, DGET-, and DREM-, which are almost completely analogous to their "+" counterparts. (Hence, DPUT- will cause an error if the datum it is given is <u>present</u>.) However, the analogy is somewhat flawed.

The description of presence vs. absence of a datum that I have given has not differentiated the properties they share and do not share. In a sense, they are equivalent; a datum, once realized in a context-frame, cannot be made absent there by operations at a higher level, and unrealization is equally tenacious. But "absent" can also mean "having unspecified status in all c-frames of a context." What are primitives like DPUT- to do in such cases? Conniver's solution is to treat c-frame GLOBAL as special, in that having no mention but "unrealized in GLOBAL," is equivalent to having no mention at all.

This requires a detailed explanation. (But, since in most cases the intuitively desirable thing happens, it is not very important that this explanation be understood.) Every datum keeps track of its mentions with <u>context</u> <u>markers</u> (c-markers), each of the form

(cnum status *property-pairs*)

where cnum identifies a c-frame and status is +, -, or NIL. Formally, a <u>mention</u> of a datum by a c-frame is its assignment of non-NIL status or properties or both, but with (0 -) excluded (0 being the cnum of context frame GLOBAL). This means that c-markers of the form (52 NIL (FOO BAR)), (0 +), or (3 -) are mentions, but (52 NIL), (0 NIL), and (0 -) are not; when a c-marker like one of the latter arises by the action of a system function, it is deleted from its datum. The same happens (eventually) when nobody points to the c-frame with a marker's cnum (as it is flushed by the garbage collector). Indexable data are indexed only when there is at least one mention of them by a living c-frame, to allow for garbage-collection of totally worthless data which would otherwise be protected by the index.

This state of affairs clears the way for GLOBAL to be the default c-frame used by DPUT-, DGET-, and DREM- if the data they work on are absent by virtue of being unspecified in all c-frames of the current context.

If an example will help, consider this. Starting with a fresh data base, you type

```
                (CSETQ D1 (REMOVE '(LYNDON PULLS PIG-EARS))),
```

which returns, as it should, the item datum referred to, with c-markers as its tail:

```
                     ((LYNDON PULLS PIG-EARS)).
```

Its CAR is the instantiated skeleton (EQUAL to it), its CDR (and list of c-markers), NIL. It is not mentioned or indexed; no one can tell it is there. You REMOVEd it, right? The (0 -) you may have intended is invisible. Now you try

```
                  (DPUT- D1 'INSTEAD 'DOG-EARS)
```

and out comes

```
                      (DOG-EARS INSTEAD),
```

the new property pair DPUT- created. But look at the value of D1:

```
        ((LYNDON PULLS PIG-EARS) (O NIL (DOG-EARS INSTEAD))).
```

Not only that, but (ABSENT '(LYNDON PULLS PIG-EARS)) will find D1; it's indexed. Now, typing

```
                    (DREM- D1 'DOG-EARS)
```

prints out

```
                    (DOG-EARS INSTEAD),
```

the deleted pair. Deleting it has made the c-marker (O NIL) a non-mention, so it is deleted from the item, leaving no mentions of D1, which is therefore unindexed. D1 looks like:

```
                    ((LYNDON PULLS PIG-EARS))
```

again.

In this way GLOBAL is treated uniformly as the c-frame where things aren't when they aren't anywhere else. (Ontologists take notice.)

This treatment means that REMOVal at the top level allows garbage-collection of items, which, because of the tenacity of absence in general, cannot take place at lower levels whose c-frames are still alive. Since all contexts need such a c-frame at the bottom, GLOBAL must be the last c-frame in every context.

If the user has noticed the delicacy of this structure, he will be more than glad to heed the following warnings:

(1) Don't build contexts by any method but with the system functions provided for this purpose.

(2) Don't play with a datum's c-markers on your own; you may create an illegal mention, scramble their order, or be caught by

a context-frame garbage collection. Once you have a property pair, however, you may safely do anything you wish to it.

With these caveats in mind, the user may turn to the descriptions of the following context-building functions:

(1) (NEW-CONTEXT c-frame-list) makes sure the c-frames in its argument are in the proper order, then adds a *CONTEXT at the front, returning the result. If necessary, it splices in a pointer to GLOBAL at the end, to make sure it's there.

(2) (CFRAME) creates a c-frame with a unique cnum, higher than any in use, suitable mostly for use in expressions like (NEW-CONTEXT (LIST (CFRAME) GLOBAL)).

(3) (PUSH-CONTEXT context) behaves just like

```
(LAMBDA (CONTEXT)
   (CONS '*CONTEXT (CONS (CFRAME) (CDR CONTEXT)))   ),
```

but its argument is optional with default value CONTEXT.

(4) (POP-CONTEXT context) behaves like

```
(LAMBDA (CONTEXT)
   (CONS '*CONTEXT (CDDR CONTEXT))   ),
```

but it, too, will take CONTEXT as its default argument if it is applied to NIL.

(5) (SPLICE context) adds a brand-new c-frame just _after_ the first of context, and returns its argument, _with its structure changed_.

<u>On Pattern-Directed Invocation</u>

Methods can be invoked in association with adding items to, fetching items from, and removing items from the data base. The invocation depends on a match between the method's pattern and the item.

The matcher used in Conniver is very simple, and is biased in favor of taking a constant list on one side. (MATCH varpat datapat dataenv), where dataenv is optional, assumes varpat is a pattern of a FETCH or method. A pattern is a non-circular list structure with certain sub-structures that are expansions of expressions starting with the macro characters "?", "!", ",", and "@". "?var" (which expands into (GIVEN var)) indicates a variable that is to receive a value during the match; "!var" ((ASSIGN var)), a variable that must match a variable-free expression for the match to succeed; ",var" ((CVALUE var)), a variable whose value is to be substituted in the pattern <u>before the match begins.</u> (Varpat variables are looked up in (FRAME), as are datapat variables if no dataenv is given.) "@expr" expands into (/@ . expr) and means "the Lisp value of expr," which is substituted into the pattern, again, prior to the match. (See "Lisp and Conniver," below, for further information about "@".) MATCH does not actually assign the pattern variables; it returns an association list pairing each pattern variable that matched a constant expression with that expression. If the match failed, however, the atom NOMATCH is returned instead. The matcher is multi-level (that is, variables can occur below the top level of list structure), and dots are allowed in patterns, as (DINO DESI . ?X). Hence, the pattern ((FREDS ?X) . ?REST) matches

((FREDS FATHER) WHISTLES)

((FREDS FATHER) WHISTLES DIXIE)

((FREDS GONE) HE SAID),

generating association lists

((X FATHER) (REST (WHISTLES)))

((X FATHER) (REST (WHISTLES DIXIE)))

((X GONE) (REST (HE SAID))),

respectively.

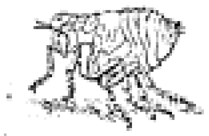
TRY-NEXT takes lists generated this way (as it finds them, associated with item possibilities), and assigns the variables as they direct.

If-addeds and if-removeds work nicely with this matcher. To invoke one (or the closure of one), Conniver binds its variables and, in the method's frame, matches its pattern against an item. Since there are no variables in items, all variables in the pattern get values. The invoker then assigns them and starts the method.

If-neededs' patterns must often be matched against patterns (of FETCH's) that themselves have variables. Invocation proceeds as with other methods, but a method variable matched against an expression containing question-marked atoms in the FETCH-pattern simply does not receive a value. If the method variable is preceded by "!" instead of "?", the match fails immediately. In fact, sub-patterns of the form "!var" find their greatest use in patterns of if-neededs that, armed only with "?", would have to have as a first line, (COND ((UNASSIGNED 'var) (ADIEU))).

Notice also that TRY-NEXT in no case assigns a variable in the FETCH-pattern while invoking the if-needed method, when the FETCH-pattern will be datapat. Thus matching an if-needed's (FOO A ?Z) against calling pattern (FOO ?X (FREDS ?Y)) does not assign X, Y, or Z in either environment. (And, if the if-needed's pattern were (FOO A !Z), the method would not be executed at all.) It is only as the method finds and NOTEs instances, INSTANCE matches the FETCH-pattern against them, and TRY-NEXT uses the results that such assignments may take place. Since instances, like items, have no variables, all FETCH-pattern variables are guaranteed to be assigned by TRY-NEXT. (Planner users please note.) In the example given, if the method assigns Z to (FREDS GALORE) and notes this instance of its pattern, TRY-NEXT will assign X to A and Y to GALORE when it comes across it. If, on the other hand, the method assigns Z to something like LINCOLN or (SALLYS FURS), the match by INSTANCE on the instance (FOO A LINCOLN) or (FOO A (SALLYS FURS)) will fail, and NOTE will reject it as a possibility.

A word should be said here about skeletons, the list structures ADD and REMOVE use to specify items; "skeleton" is defined just like "pattern," except that only "," and "@" are allowed; that is, every skeleton must expend into an expression with no variables.


Dog flea, *Ctenocephalides canis* (Length ⅛ in.)

Using Conniver

Conniver is a remarkably friendly language to use, because its control structure is "open to the public." The command CNVR ^K typed at DDT causes Conniver to print out its version number, set up an initially empty global context assigned to GLOBAL, and print

```
EAR-1
←
```

The "←" is printed out whenever Conniver wants input. The ear it is listening with initially is EAR-1. This is not a joke, but a tag into a READ-CEVAL-PRINT loop at the top level. Interacting with such a loop ought to be very easy for an experienced Lisp user; Conniver will attempt to CEVAL everything typed at it, and will print the result.

If input is switched to a new file (using UREAD), masses of CEXPR's can be defined using

(CDEFUN name (*variable-declarations*) *body*).

CFEXPR's, CLEXPR's, or something similar, are not needed because of the flexibility of variable declarations. Declarations can be just a list of atoms, but the construction

"OPTIONAL" *declarations*

enables function to supply default values for missing trailing arguments. For example, the declaration (X "OPTIONAL" (Y CONTEXT) Z) specifies one required and two optional arguments; if Y is missing, it receives the value of CONTEXT; a missing third argument leaves Z rebound but unassigned.

If the last two elements of the declaration are

"REST" var

var is bound to a list of the remaining arguments, each evaluated.

In place of a declared variable, the form (QUOTE var) may appear in any of the variable declaration slots, including "REST" ′var. This has the effect of blocking evaluation of the corresponding argument, or list of arguments in the case of "REST". A FEXPR of one argument L in Lisp, therefore, has as counterpart a CEXPR with declaration ("REST" ′L).

(It should be pointed out that this entire variable declaration syntax was taken from MUDDLE.)

In similar fashion, CDEFGEN can be used to create generators using the same variable declaration syntax, and ADD can be used to create an initial context of items and methods.

When an error occurs (either a Lisp error or a call to CERROR), the system creates a new frame with the frame of the error as its control pointer, prints a message, defines a new ear, and enters its READ-CEVAL-PRINT loop, printing

```
EAR-n
←
```

The function BACKTRACE can be used to get a lucid summary of the control pointer chain from (FRAME) upwards. Variable values can be inspected, functions can be called, etc. Quitting completely is done by typing (GO EAR-1), EAR-1 being the always-defined top level. To continue execution, EXIT from EAR-n. Since the value may be irrelevant, the function (DISMISS frame) has been provided to exit from it with no particular value.

DISMISS comes in handy in conjunction with the ~A-interrupt feature. To stop Conniver between elementary steps, hit ~A. This will cause an "interrupt," and create a new ear. To restart the process use (DISMISS); DISMISS takes (FRAME) as its default argument.

If a Conniver program is in the middle of executing a piece of Lisp that it called, it will correctly intercept any ~X's not caught by ERRSET's in the Lisp itself. However, since Conniver is written in Lisp, it is possible to mangle it by hitting ~X in the middle of doing some internal Conniverish thing.

Simply DISMISSing from an error will cause Conniver to try again what it choked on before. If it hit a ~A, that means it will continue. If it choked on a ~X which turned off an infinite printout, DISMISSing will start the infinite printout again. If it barfed because of an unassigned variable reference, it will barf again unless you assign it before DISMISSing.

You can get out of Conniver at any time by calling STOP. This leaves all Conniver structures intact, but puts you in a lisp READ-EVAL-PRINT loop, where Lisp errors don't generate annoying new ears. To restart in <u>exactly</u> <u>the</u> <u>state</u> before (STOP), call (RUN); you're back in Conniver. (RUN and STOP have more sophisticated uses; see the appendix.)

Lisp and Conniver

Lisp functions do not usually call Conniver CEXPR's, GENERATOR's, and CINT's (the analogue of FSUBR's in Lisp), because Lisp stacks are far more perishable than Conniver's frame-trees. (But see the description of CLVAL, below.) Conniver can call any Lisp function, though, and Lisp EXPR's, FEXPR's, LSUBR's, and SUBR's can take Conniver arguments in forms evaluated by Conniver. For example,

```
(PRINT (TRY-NEXT P NIL))
```

is perfectly legal. Lisp functions called by Conniver can reference Conniver variables free by use of the function (CVALUE var), abbreviated ",var". For example, Lisp functions should refer to CONTEXT as ,CONTEXT.

Since Lisp can't call CEXPR's, functions that do Conniving things must be written in Conniver down to a low level. The resulting slowdown may make one cringe, but there is a remedy. Any piece of pure Lisp may be made more efficient by prefixing it with the "@" macro-character, and making all Conniver variable references explicit by use of ",". For example,

```
@(THIRD-IN-ROW ,SQUARE1 ,SQUARE2)
```

where THIRD-IN-ROW is an EXPR, is much more efficient than

```
(THIRD-IN-ROW SQUARE1 SQUARE2)
```

because it expands into

```
(/@ THIRD-IN-ROW (CVALUE SQUARE1) (CVALUE SQUARE2)),
```

/@ being a FEXPR, namely

(LAMBDA (EXP) (EVAL EXP)).

Conniver always gives FEXPR's complete control over their argument evaluation, so just hands the expression (/@ ...) to EVAL, saving generating a frame and interpreting the expression. The @ macro is thus a way of hand-compiling arbitrary sections of code involving no CEXPR's, GENERATOR's, or CINT's. The @ may be used inside skeletons and patterns used by ADD and REMOVE, and FETCH; just as "," in such a context means "substitute the value of a variable," so "@" means "substitute the (Lisp) value of an expression." Another use of the @ macro is getting the Lisp value of a variable within Conniver; @CONTEXT, for instance, gets the Lisp value of CONTEXT, just as "," gets its Conniver value. (It should be pointed out here that if Conniver can find no binding of an atom while looking for its CVALUE, it takes the global Lisp value, if any, so that sometimes @ and , do the same thing; this has the consequence that Conniver concludes a variable is unbound only if it is globally unassigned by Lisp.)

A Lisp program, if it really wants to, can use CEVAL to Conniver-evaluate a form. If it is a well-behaved form, this is just like using EVAL, but there are pitfalls. Some of the problems stem from the fact that the frame and its daughters generated by execution of the form may hang around (with a HANG, for example), after an EXIT back to the Lisp. While control is in this structure the first time, Lisp variables bound in its caller may be accessed (with @), and in general everything is cool. After it returns, however, the Lisp return point vanishes, along with its frame, bindings, etc., and even the frame of the EXP: CEVAL.

If control re-enters the Conniver structure, the new Lisp stack-state above it will have nothing to do with the original, God will know _what_ Lisp variables are referenceable, and a return from the structure's top level will have no obvious meaning. This is not to say that a process created in this manner has no use, but merely to emphasize the dangers in creating one. Attempting to @ a Lisp variable will probably find it unbound (creating a Lisp-error in Conniver), and an attempt to return from the control structure again generates a Conniver error (whose EAR has as control pointer a frame created _above_ the structure for just this purpose the first time control returned from it).

There is still another problem which is even worse. If, during a CEVALuation, control leaves the new Conniver control structure it created (e.g., by GOing to an old tag), and never returns, the entire old Conniver process will be running with a Lisp stack slightly different from what it started with. In particular, all the Lisp frames that were around when CEVAL was called are still there, but there is no way to detect or flush them. In such a situation, STOP (see Appendix) no longer does the right thing, and the stack has been enlarged in perpetuity. Enough such pathological CEVALs can cause a pdl overflow. The user is strongly encouraged to use RUN and STOP for Lisp-Conniver interaction, even if they are trickier.

One pleasant thought is that many Conniver functions are actually EXPR's, or have EXPR versions which do almost the same thing. (In the compiled Conniver system, of course, these are SUBR's or FSUBR's, but I will continue to use the term EXPR in the loose sense "Lisp function.") For example, the CEVAL you get if you call it from Lisp is clearly different from the CINT version the Conniver interpreter would find. All functions with EXPR versions can, of course, be called from Lisp. Happily, they include all the data base-manipulation functions, but the EXPR versions of ADD, REMOVE, REALIZE, and UNREALIZE differ slightly from the CEXPR versions because the invocation of any if-added or if-removed methods must be CEVAL'ed. Since if-addeds and if-removeds are probably not too closely linked with the process that triggers them, these are probably safe CEVALs.

One worry the user <u>doesn't</u> have to have is whether his Lisp functions will clobber or rebind internal Lisp variables used by the Conniver interpreter. All Conniver atoms Conniver doesn't want you to see have been "half-killed" in such a way that they will print out but cannot be recognized during user input.

Appendix

The Conniver Reference Source

Whereas the previous section of this manual is a discursive overview of Conniver for the purpose of illustration of and introduction to the ideas embodied in Conniver, this section is an attempt to provide a reference source for the active user. Thus, it contains a detailed description of each primitive of the language, enumerating the possible error conditions that are associated with that primitive and its limitations which might not be immediately apparent. Besides primitive operators, every language has a set of reserved words (syntactic indicators and significant variables). These will be duly noted. A comprehensive index to the primitives, reserved words, and error comments is in the rear. This reference material is divided into a section on the evaluator and a section on the data base functions. Each section is preceeded by a summary of general information followed by a listing of primitives organized into categories. Not surprisingly some functions appear more than once. The formal conventions of this section are:

Actual code is in upper case

Syntactic variables are lower case

Optional arguments are delimited by brackets ([,]) surrounding the syntactic variable and its default value.

Segment syntactic variables are delimited by stars (*). Every function defined has its type (or types) specified next to a sample call. CINTs and CEXPRs are invisible from Lisp and thus are only defined in Conniver code, avoiding interference with LISP functions of the same name.

The Evaluator

The Conniver interpreter evaluates expressions in a manner similar to that of LISP. The basic syntax is as follows:

conniver expression = number | atom |
'S-expression | @s-expression | (function
arguments)
arguments = empty | conniver expression1 ...
conniver expression N

The evaluation rules are:

1. As in LISP quoted expressions and numbers evaluate to themselves.

2. The value of an atom is its value as a variable. If it is bound in Conniver that value is used; if not the LISP value is used. Thus, if a variable is unbound in both LISP and Conniver its evaluation results in the LISP unbound variable error. An error also results from the evaluation of a variable which is unassigned (though bound):

UNASSIGNED VARIABLE offending-variable

3. An expression following an @ is passed directly off to LISP for evaluation. We recommend that code be written so that as much as possible happens in LISP because of the considerable speedup attainable.

4. Functional applications are processed as follows:

If the function is atomic, it is checked for CINT, CEXPR, GENERATOR, FEXPR, FSUBR definitions. If an atom has two such definitions, the first on its property list is taken; this means that if the user wants a function to be a FEXPR in Lisp code and a CEXPR in Conniver code, the CEXPR must be defined last so as to be first on the property list. If it is none of the above, it is assumed to be a LISP EXPR, SUBR, or LSUBR, thus undefined function errors come from LISP.

If the function is a FEXPR or FSUBR the form is passed to LISP for immediate evaluation.

If it is a CINT (such as COND) the form is evaluated by the appropriate internal Conniver routine.

If the function is a CEXPR or GENERATOR, the arguments are paired with the formal parameters of the function (and perhaps evaluated) as specified by the declaration in the function (see CDEFUN, CDEFGEN for details). After binding, the body of the function is executed.

If the function is an EXPR, SUBR, or LSUBR the arguments are evaluated by Conniver and then the LISP function is applied to the resulting argument list with LISP APPLY.

If the function is non-atomic then either it is an anonymous CLAMBDA expression (CEXPR) or it is an anonymous LAMBDA expression (EXPR) and treated accordingly.

Note that there are no other cases. The function position is never evaluated as in LISP. Functional arguments are handled explicitly, preventing ambiguity, using the function CALL.

Execution of the body of a CEXPR, GENERATOR, PROG or METHOD proceeds as follows:

If it begins with the reserved word "AUX" then the second element of the body is taken as a declaration of auxiliary variables (PROG variables in LISP). Such a declaration is a mixed list of atoms and initializations. Each atom is bound but left unassigned. An initialization is a list of an atom and an expression. The atom is bound and assigned to the value of the expression.

The rest of the body is then executed sequentially (unless the sequence is changed by a GO). The value of the body (and hence of the function) is the value of the last expression in the body, unless a return is forced by RETURN, EXIT, or DISMISS.

I. Communication with LISP

A. (RUN [stuff NIL]) FSUBR (but it evaluates its argument)

B. (STOP [stuff NIL]) LSUBR

These functions allow LISP and Conniver programs to treat each other as co-routines. Control is passed from Conniver to LISP via STOP and from LISP to Conniver via RUN. The argument to STOP is returned as the LISP value of RUN and the argument of RUN is returned as the Conniver value of STOP. STOP may only be called if Conniver is running, otherwise:

Conniver-NOT-RUNNING—STOP

RUN may only be called if Conniver is not running, otherwise:

Conniver ALREADY RUNNING.

Example: To have Conniver evaluate expressions passed to it from LISP, we put Conniver into the loop:

```
(PROG "AUX" ((MESSAGE 'HI-LISP))
 :LOOP (CSETQ MESSAGE (CEVAL (STOP MESSAGE)))
       (GO 'LOOP))
```

Conniver returns to LISP with the value HI-LISP. Thereafter LISP may get an expression evaluated by Conniver by calling

(RUN expression)

The value of RUN will be the Conniver value of the expression.

Within a (Lisp) CEVAL, STOP causes its argument to be returned as the CEVAL's value; this will be true even if Conniver control has left the structure that CEVAL set up. RUN will <u>not</u> get the program back to the execution point of that STOP, because after leaving the CEVAL, Conniver is already running. So, if you want STOP to do the right thing, don't use CEVAL.

If, for some reason, the Conniver interpreter (not the data-base — see DATA-INIT) needs to be re-initialized, it can be done so by executing (from LISP only):

C. (START) SUBR

START resets all of the Conniver internal variables (including the ear#) and goes into the top-level listen loop. The data base is not disturbed, but all contexts previously bound only to Conniver variables will be lost to garbage collection. START binds CONTEXT to a global context containing GLOBAL as its only c-frame.

A Conniver program may safely be stopped for examination by hitting ^a (control-a). This causes a new ear to be generated. The program can be resumed at the place it left off by exiting from the resulting listen loop via:

D. (DISMISS) CINT

Also see: BACKTRACE, LISTEN When in Conniver IBASE=BASE=10. and all character macros are in effect; these return to their LISP defaults when returning via STOP.

II. Flow of control modification

A. (COND clause1 ... clauseN) CINT

COND in Conniver is almost identical to COND in LISP except for the fact that the CDR of a clause is a general PROG body. Thus it may contain an "AUX" declaration (See Definition of procedures, PROG) and statement labels (tags). Thus entering a COND clause produces a new activation block so remember this when using EXIT etc. This is a legal use of COND:

```
        (COND ((= N 1) "AUX" ((M 2) (P (ACTBLOCK)))
                 :LOOP (COND ((= (CSETQ M (1- M)) 0)
                                (EXIT 3 p)))
                       (GO 'LOOP))
               (T 2))
```

Next we consider the unconditional transfer:

B. (GO <u>atom or tag</u>) CINT

GO always evaluates its argument, avoiding the ambiguity of LISP. If its argument is an atom, GO searches its environment for the nearest body containing (CTAG atom). This is abbreviated by the use of a LISP character macro as :atom.

<u>Caution</u>: do not read Conniver code into LISP as the macros are not in effect.

Execution then proceeds from the statement label found. If its argument is a tag (see TAG and ACTBLOCK) control is transferred to the tag, perhaps non-locally. If the argument is of the wrong type or an atomic tag cannot be found we get:

BAD TAG

C. (EXIT value [frame (ACTBLOCK)]) CINT

D. (RETURN value) CINT

E. (DISMISS [frame <u>first non-COND frame</u>]) CINT

EXIT returns from the frame indicated with the value indicated. If no frame is given it returns from the nearest activation block. <u>Caution</u>: COND causes an activation block. RETURN, returns from the nearest non-COND activation block. DISMISS is EXIT from the frame specified with the value NIL. If no frame is given it does a (RETURN NIL). If there is no activation block to RETURN from or EXIT from we get:

```
        RETURN FROM WHAT?
 or EXIT FROM WHAT?
```

If DISMISS or EXIT is given a non-frame they complain:

```
        BAD FRAME
```

F. (ADIEU proposal1 ... proposalN) CEXPR

G. (AU-REVOIR proposal1 ... proposalN) CEXPR

These functions return a possibilities list from a generator, NOTEing proposals 1 ... N in that order. (None may be supplied.) (see NOTE). ADIEU leaves for good but AU-REVOIR finishes by noting a tag inside AU-REVOIR so that TRY-NEXT can resume the generator where it left off. The value of AU-REVOIR, on resumption, is the message passed in TRY-NEXT. (see TRY-NEXT).

III. Frame Manipulators:

A. Constructors:

1. (TAG atom) SUBR

2. (ACTBLOCK) SUBR

TAG searches the environment for the first activation block containing a statement label :atom. It returns a tag structure whose frame is that activation block and whose body-pointer is to that statement label.

ACTBLOCK searches for the first activation block (frame with a body) in its environment and returns a tag to the beginning of the body. If either a TAG or ACTBLOCK is unsuccessful in its search it returns NIL.

3. (FRAME) SUBR

4. (ACCESS [frame (FRAME)]) LSUBR

5. (CONTROL [frame (FRAME)]) LSUBR

FRAME returns the frame with respect to which it was evaluated.

ACCESS returns the access frame of its argument.

CONTROL returns the control frame of its argument.

The argument to ACCESS or CONTROL must be a legitimate frame (*tag, *frame, *closure). If it is not we get the error message:

    BAD FRAME SUPPLIED

6. (CLOSURE procedure) SUBR

CLOSURE produces the lambda-closure of the procedure (function, generator, method) indicated. Later invocation of the closure (see CALL) causes the environment of the procedure (its access

preter, where it searches for bindings of free variables, tags, etc.) to be the environment in which the closure was constructed.
e.g. If X = 4 then:

```
  (CALL ((CLAMBDA (X) (CLOSURE '(CLAMBDA (Y) (+ X Y)))) 3) 5)
```

has the value 8 but

```
  (CALL ((CLAMBDA (Y) (+ X Y))) 3) 5)
```

has the value 9.
This is the classical "FUNARG" device.

B. Modifiers

1. (SETACCESS frame1 frame2) SUBR
2. (SETCONTROL frame1 frame2) SUBR

These very dangerous functions (for experts only) are used for modifing frames. They set the access or control pointer of frame1 to frame2.

C. Interrogation

(EXPRESSION frame) SUBR

This function returns the expression whose evaluation created the frame supplied. It is useful for hunting arround in the frame structure.

IV. Relative Evaluation:

A. (CEVAL expression [frame (FRAME)])   CINT,LSUBR

This is the standard relative evaluation function. The expression is evaluated with respect to the frame specified (default, the current environment) as its access frame. If the frame supplied is not legitimate, we get:

    BAD FRAME

The LSUBR definition of CEVAL can be used to do Conniver evaluations from Lisp. Unfortunately, if you use it to do something really clever, you probably are doing the wrong thing. See "Lisp and Conniver" for an account of the dangers involved.

B. (CALL functional argument arg1 ... argN)   CINT

CALL applies the functional argument to the arguments supplied. It avoids the LISP ambiguity in the case that a functional argument is the value of a variable and we have no way of guaranteeing that it has no function property. The functional argument may be a function, generator, or closure of a function or generator.

V. Variable manipulators:

A. Interrogators:

```
1. (VLOC atom [frame (FRAME)]) LSUBR
2. (RVALUE atom [frame (FRAME)]) LSUBR
3. (CVALUE atom)             FSUBR
4. (LVALUE atom)             FSUBR
5. (ASSIGNED atom)           FSUBR
```

VLOC returns a locative to the value of the atom supplied if it is found in the frame specified, if not, it returns NIL.

RVALUE returns the real value of the atom given in the frame specified (it does not check for *UNASSIGNED). If the variable is unbound in Conniver, the LISP value is taken. If it is unbound in LISP, the appropriate LISP error occurs. If either VLOC or RVALUE are given an illegal frame, we get:

```
BAD FRAME SUPPLIED
```

(CVALUE atom) (abbreviated ,atom via macro-characters) gets the current Conniver value of the atom. This is how LISP code called by Conniver code gets the value of Conniver variables.

LVALUE gets the LISP value of its argument. (LVALUE atom) is equivalent to (but not identical to) @atom.

ASSIGNED returns as its value, T if its argument has a value (other than *UNASSIGNED) and NIL if it is unassigned.

B. Modifiers:

```
1. (CSET atom value [frame (FRAME)]) LSUBR
```

2. (CSETQ atom value)   CINT,SUBR

3. (UNASSIGN atom)   SUBR

CSET is the most powerful assignment operator; it sets the atom to the value relative to the frame specified.

CSETQ is a minor convenience; it does not evaluate its first argument.

UNASSIGN sets its argument to *UNASSIGNED.

C. there is one more function for working with variables:

    (BIND atom value)   SUBR

This is for experts only; it binds the atom to the value in the current frame.

VI. Definition of Procedures:

1. (CDEFUN atom declaration *body*)   FSUBR
2. (CDEFGEN atom declaration *body*)   FSUBR

These functions are used to define the atom to be a function (or generator) with the formal parameters specified by the declaration and with the body given. Functions are placed under the indicator CEXPR and generators under the indicator GENERATOR. The body is simply a sequence of statements to be evaluated sequentially. It may (or may not) begin with a declaration of auxiliary variables (described later). The formal parameter declaration syntax is as follows:

declaration = (obligatory variables optional variables excess)

obligatory variables = empty | par1 ... parN

parI = atom | ʹatom

optional variables = empty | "OPTIONAL" op1 ... opN

opI = atom | ʹatom | (atom default) | (ʹatom default)

excess = empty | "REST" atom | "REST" ʹatom

The semantics is as follows:

1) Formal parameters are matched against arguments from left to right.

2) There must be at least one argument for each obligatory variable.

3) Unless there is an excess collector declared, there may not be more arguments than declared variables.

4) Arguments are evaluated unless the corresponding formal parameter is quoted (′).

5) If the arguments run out while binding optionals, they are filled with either *UNASSIGNED, or if an expression for the default value is given, the value of the default expression (in the frame of the function with all previously processed variables bound) is used.

6) An excess collector gets the list of arguments or values of arguments (depending upon the existence of a ′) left over.

This elegant syntax is due to Chris Reeve of MUDDLE. Note how beautifully this does away with FEXPR′s and LEXPR′s and how much more flexible than LISP it is.

If the evaluator is not satisfied that the number of arguments is right for a function it complains:

    WRONG # OF ARGUMENTS

If the syntax of a declaration is not as specified above the error comment:

    BAD DECLARATION

will be generated.

To create a method we use the constructors:

C. (IF-ADDED atom pattern *body*) FSUBR

D. (IF-REMOVED atom pattern *body*) FSUBR

E. (IF-NEEDED atom pattern *body*) FSUBR

The given atom is defined to be a method of the type indicated, invoked by the given pattern, with the given body. The method required is the value of the constructor. If the atom is not specified, the method is not named, but of course, it may be saved as the value of a variable. To be accessible, a method must be put into the data base via INSERT or ADD.

The body of a procedure is as follows: The value of a function is the value of the last expression in the body (or of a RETURN, EXIT, or DISMISS). The body is just a sequence of expressions to be evaluated. If it begins with "AUX" (a reserved word) then the next element of the body is taken as a declaration of auxiliary variables (PROG variables in LISP). Such a declaration is a list of atoms and initializations. Each atom is bound but left unassigned. An initialization is a list of an atom and an expression. The atom is bound and assigned to the value of the expression.

VII Possibilities lists

A possibilities list (created by FETCH or a generator function) has the following format.

```
        possibilities = (*POSSIBILITIES pos1 ... posN)
        posI =  (*METHOD pattern method)|
                (*GENERATOR form)|
                (*AU-REVOIR body fr)|
                (*ITEM item alist)|
                anything else
```

Thus anything may be a possibility but the specifically mentioned types have special interpretation in:

A. (TRY-NEXT possibilities [nomore NIL] [message NIL]) CSUBR

TRY-NEXT is used to try the first possibility on the possibilities list. In doing so, it clobbers the list, removing the first one. If there are none, it evaluates nomore and returns the value. The action taken by TRY-NEXT on each type of possibility is as follows:

1. (*METHOD pattern <u>if-needed method</u>)

The method is invoked. It generates and returns a possibilities list (probably by either ADIEU or AU-REVOIR, though it may CONS one up). This new possibilities list is then spliced into the given one, replacing the method possibility which created it. TRY-NEXT then loops back to try the first possibility in the newly constructed possibilities list. The pattern is used by INSTANCE inside the method as the calling pattern. If an if-needed or other kind of generator doesn't return a possibilities list to TRY-NEXT, what it does return is ignored.

2. (*GENERATOR form)

Exactly the same as a method except that the form is evaluated rather than the method invoked.

3. (*AU-REVOIR body fr)

This is the way AU-REVOIR can be resumed. The TRY-NEXT goes off to the appropriate place in the AU-REVOIR which passed this back. The AU-REVOIR returns to its caller (the generator or method) with the optional TRY-NEXT message as its value.

4. (*ITEM item alist)

The alist is a list of variable-value pairs probably constructed by the matcher. The variables are set to the indicated values and the item is returned as the value of TRY-NEXT.

5. Anything else is returned as the value of the TRY-NEXT.

Thus we see that TRY-NEXT does not terminate until either the possibilities list is empty i.e. (*POSSIBILITIES) or an item possibility or an "anything else" is first on the list. If TRY-NEXT is given a bad possibility list we get.

BAD POSSIBILITIES LIST

A method makes item possibilities as instances of its invocation pattern with:

B. (INSTANCE) FSUBR

which returns the current instance. It will get upset if there are unassigned variables in the pattern and will cry:

IMPURE INSTANCE

A generator or method may note a new possibility via

C. (NOTE [possibility (INSTANCE)]) LSUBR

The list of possibilities that have been noted since the generator was started (or last resumed at an AU-REVOIR) is stored in a variable, PROPOSALS, in the reverse order of notation (chronologically). This is the list which is reversed and returned (with *POSSIBILITIES CONSed on) by AU-REVOIR and ADIEU. If a generator (or a method) wants to get (to see or clobber) the possibilities list of the TRY-NEXT it feeds, it can:

D. (GET-POSSIBILITIES) FSUBR

It can replace the possibilities list of that TRY-NEXT by:

E. (SET-POSSIBILITIES possibilities list) EXPR

VIII. Debugging Aids

A. (BACKTRACE [number 696969]) LEXPR

BACKTRACE types out, in a very readable form, the expressions corresponding to each frame of the current process, starting with the current frame, and proceeding by control links to the top level. The optional numerical argument may be supplied to limit the typeout to that many frames.

B. (EXPRESSION <u>frame object</u>) EXPR

Has as its value the expression of the frame, tag, or closure passed to it. This is useful to find out what in the hell is coming off when you are playing with control structures.

C. (CTAG atom) FEXPR

This is the internal representation of :atom, a statement label. Please note that the LISP trace package can be used to trace CTAG, thus showing your flow of control.

The Data Base

The Conniver data base is a hierarchical structure of <u>contexts</u>, or a "tree" of <u>context-frames</u>, containing four types of <u>data</u>: <u>objects</u>, <u>items</u>, <u>methods</u>, and <u>method</u> <u>closures</u>.

Objects are of the form:

(*OBJECT arbitrary-structure *c-markers*).

Items are of the form:

(name *c-markers*)

where name is any non-circular list structure.

Methods look like

(type name pattern body *c-markers*),

where type is IF-NEEDED, IF-ADDED, or IF-REMOVED; name is an atom which is the method's name unless it is NIL; pattern is a non-circular list structure with all variables (if any) marked as (GIVEN var) ("?var") or (ASSIGN var) ("!var"); and body is a generator-function body if type is IF-NEEDED, and a CEXPR body otherwise. Any atom with such a structure on its property list under the indicator METHOD is equivalent to that method in every situation as far as the system is concerned, but the atom must be the name of the method.

Method closures look like

(*CLOSURE method fr),

where method is a method (possibly specified by name), and fr is an internal frame pointer.

All such data have (possibly NIL) lists of c-markers associated with them. A c-marker is of the form

(cnum status *property-pairs*)

where cnum is a c-frame number; status is +, -, or NIL, indicating realization, unrealization, or unspecification in that c-frame, respectively; and property-pairs are non-atomic s-expressions. The c-markers on each datum are in order of decreasing cnums.

A c-marker indicates a mention of its datum by its context-frame. A context-frame (c-frame) is of the form

(*CFRAME cnum *data*)

where cnum is its unique context-frame number, and data are the data it mentions. If cnum = 0, this is the global context-frame GLOBAL, for which data always = NIL.

A context is a list like

(*CONTEXT *c-frames*),

where c-frames must be in order of decreasing cnums, with c-frame GLOBAL = (*CFRAME 0) as the last one. It is worth mentioning here that none of the functions that depend on an explicit or implicit context argument check for the presence of the *CONTEXT flag at the beginning of the context. Hence, any list with a list of c-frames as its cdr is a legal context; in particular, (CDR context) = (super-context context) for all practical purposes.

Each context rigorously defines the status of every datum as present or absent, as follows: if the first status-defining mention of the datum by a c-frame in the context specifies realization (status = +), it is present, else absent; in particular, if it is unspecified by all c-frames in the context, it is absent.

Every c-marker must specify either non-NIL status, or non-NIL property-pairs, or both, and cannot be (0 -), or it does not constitute a mention. System functions delete all c-markers of the form (n NIL) or (0 -).

When a c-frame is not pointed to by anything, it is subject to garbage collection. All c-markers embodying a mention by it will be deleted from their data.

Items, methods, and method closures are indexable data; they can be referred to by pattern in FETCH and other functions. This indexing is done automatically by the system whenever an unmentioned datum becomes mentioned (by ADD, REMOVE, DPUT, and other functions); unindexing occurs when its last mention is removed (by DREM, the garbage collector, etc.). Unindexed items and anonymous methods are subject to garbage collection if unprotected.

<u>Data-Base Functions</u>

These functions are tightly interwoven. They all call a common body of invisible functions which analyze their arguments; it is these that print most error messages. Many functions generate the following two messages:

```
        TOO FEW ARGUMENTS
        TOO MANY ARGUMENTS.
```

These will be accompanied by a print-out of the form that caused the trouble.

Many functions use system routines to break a datum into usable chunks. They can generate the message

```
        MEANINGLESS DATUM — function,
```

where function is one of ANALYZE, CHARKERS, or PATTERN.

Functions that take skeletons instead of patterns as arguments resent finding "?" or "!" in them. They generate the message

```
        VARIABLES IN A SKELETON — INSTANTIATE
```

(INSTANTIATE being the routine that generates an item from a skeleton).

I. Data-Base Initialization

(DATA-INIT n m) SUBR

This function wipes out all currently existing contexts, and unindexes all indexable data. It creates a brand-new data base governed by the paramters n and m. n is the total number of c-frames allowed; if the data-base functions ever attempt to maintain more than this number at once, the message

TOO MANY CONTEXT-FRAmES — CFRAME

will occur. (See CFRAME for a more complete account.)

The second parameter, m, is the increment between the numbers of context-frames consecutively generated by CFRAME. Given the ordering constraint on c-frames, and the fact that SPLICE (q.v.) must be able to generate c-frames with cnums between those of any two c-frames, even if they were generated consecutively, they cannot be numbered 0, 1, 2,..., but 0, m, 2m, 3m,....

Conniver does a (DATA-INIT 100. 10.) when it is loaded, creating a data base with at most 100 c-frames at a time, numbered 0 (GLOBAL), 10., 20., 30.,....

II. Datum Creation

A. (OBJECT [structure NIL]) LSUBR

creates a brand-new object of the form (*OBJECT structure), where structure is arbitrary. This object is initially absent in all contexts, and, of course, not EQ to any other.

If OBJECT has too many arguments, it errs with the message

TOO MANY ARGUMENTS

b. (DATUM skeleton) SUBR

Item data are normally created implicitly whenever the user names one with a skeleton that does not refer to any currently indexed item datum. If, however, the user creates an item datum himself, by using LIST on a name, or using the "simulated item" of an instance proposal, etc., it is obviously guaranteed not to be EQ to an indexed item datum with the same name (if any). Thus, if he executes (REALIZE (LIST '(LINE GOO1)) and ((LINE GOO) (9 -)) is already indexed, the new one will be indexed as well. (The indexer could check for this, but it would slow things down.) Then FETCH will find both, and PRESENT will find an unpredictable one of them. To get around this problem, use DATUM instead of LIST. After instantiating skeleton (like ADD), DATUM returns LIST of the result only if it can't find it in the index; if it can, it returns the unique item datum with that instantiated skeleton as its name.

Mosquito.
Culex pipiens
(Length ¼ in.)

III. Enlarging, Depleting, and Searching the Data Base

```
A. (REALIZE datum [context CONTEXT])            CEXPR,LSUBR
   (UNREALIZE datum [context CONTEXT])          CEXPR,LSUBR
   (ADD skeleton [context CONTEXT])             CEXPR,LSUBR
   (REMOVE skeleton [context CONTEXT])          CEXPR,LSUBR
   (INSERT skeleton [context CONTEXT])          LSUBR
   (KILL skeleton [context CONTEXT])            LSUBR
```

These functions make datum present (REALIZE, ADD, INSERT) or absent (UNREALIZE, REMOVE, KILL), by virtue of realization or unrealization in context's first context-frame. Here, "datum" means datum (REALIZE, UNREALIZE) or "item datum referred to by ske eton" (ADD, REMOVE, INSERT, KILL). ADD and REMOVE can be used to alter the status of data not referred to by skeleton; see III.B.

The effects of these functions are invisible in all super-contexts of context; these effects will be collected as garbage if the top c-frame is ever caught unprotected by the garbage collector.

If ADD or REALIZE is given a skeleton or indexable datum argument, respectively, all if-added methods matching datum's name that are present in context will be invoked. Similarly, UNREALIZE and REMOVE invoke if-removed methods matching datum's name.

Warning! The LSUBR versions of these four functions execute hidden CEVALs to accomplish the method invocations. If the methods do anything really clever and subtle, invoking them will probably screw your program.

```
B. (ADD method [context CONTEXT])                      CEXPR,LSUBR
   (REMOVE method [context CONTEXT])                   CEXPR,LSUBR
   (INSERT method [context CONTEXT])                   LSUBR
   (KILL method [context CONTEXT])                     LSUBR
```

If ADD and REMOVE are given method, method name, or method closure arguments, they are synonymous with REALIZE and UNREALIZE; the same is true of INSERT and KILL. These functions can not be given any other datum types as arguments. (Since there is no way to tell whether ((FOO BAR)), for example, is meant to be an item or the name of one.)

```
C. (FETCH pattern [context CONTEXT])                   LSUBR
   (FETCHI pattern [context CONTEXT])                  LSUBR
   (FETCHM pattern [type IF-NEEDED] [context CONTEXT])
                                                       LSUBR
```

FETCH returns a possibilities list consisting of item possibilities for all items present in context that match pattern; followed by method possibilities for all if-needed methods in context whose patterns match pattern. For the format of these lists, see "Hairy Control Structure."

FETCHI returns a possibilities list containing only the item possibilities. FETCHM returns a list of only the method possibilities of type type that are present in context and match pattern.

FETCHM may spawn the error message

TOO MANY ARGUMENTS.

Cicada,
Magicicada
septendecim
(Length about
1 in.)

IV. Properties of Data

```
A. (REAL datum [context CONTEXT])                    LSUER
   (UNREAL datum [context CONTEXT])                  LSUER
   (PRESENT pattern [context CONTEXT])               LSUER
   (ABSENT skeleton [context CONTEXT])               LSUER
```

These functions return datum if and only if it is present (REAL, PRESENT) or absent (UNREAL, ABSENT) in context, and NIL otherwise. REAL and UNREAL are handed their data arguments directly; PRESENT tries to return a randomly chosen present item that matches pattern; ABSENT takes DATUM (qv.) of its ske eton and then calls UNREAL.

PRESENT behaves a lot like (TRY-NEXT (FETCH pattern)); in particular, it assigns any "?"- or "!"-marked variables in pattern to the pieces of the item that they matched.

```
B. (DPUT datum property indicator [context CONTEXT])
                                                   LSUER
   (DGET datum indicator [context CONTEXT])        LSUER
   (DREM datum indicator [context CONTEXT])        LSUER

   (DPUT+ datum property indicator [context CONTEXt])
                                                   LSUER
   (DGET+ datum indicator [context CONTEXT])       LSUER
   (DREM+ datum indicator [context CONTEXT])       LSUER

   (DPUT- datum property indicator [context CONTEXt])
                                                   LSUER
   (DGET- datum indicator [context CONTEXT])       LSUER
   (DREM- datum indicator [context CONTEXT])       LSUER
```

DPUT associates the pair (indicator property) with datum in the first ("most local") c-frame of context; like REALIZE, UNREALIZE, and their ilk, these effects are invisible above context and garbage-collectable if its top c-frame is reclaimed. DGET finds the first pair associated with indicator in any c-frame of context, starting with its first c-frame; if there is no such pair, its value is NIL. DREM has the same value, but removes the pair as a side effect.

DPUT+, DGET+, and DREM+ are exactly the same, but they attend only to the c-frames in which datum has realized ("+") status, <u>up to the first c-frame in which datum is unrealized.</u> If DPUT+ is given a datum which is absent in context, the error message

ABSENT DATUM — DPUT+

occurs.

DPUT-, DGET-, and DREM- are exactly analogous, but they ignore all frames with status $\neq$ "-", and all frames after the first in which datum is realized (marked with "+"). In addition, if the datum is absent by virtue of unspecified status in all c-frames of context, DPUT- uses GLOBAL as the repository of its properties; DGET- and DREM- treat GLOBAL as specifying unrealization if its c-marker on datum is marked NIL as well as if it is marked -.

If DPUT-'s argument is in fact present, the error message

PRESENT DATUM — DPUT-

is generated.

```
C. (DPUTCF datum property indicator c-frame)    SUBR
   (DGETCF datum indicator c-frame)             SUBR
   (DremCF datum indicator c-frame)             SUBR
```

These functions manipulate properties in an explicitly given context-frame. DPUTCF associates property with indicator in the c-marker for c-frame on datum; if there is no such c-marker on datum, DPUTCF puts it there, without changing its status. As usual, these effects are invisible in super-contexts not containing c-frame, and will be garbage-collected if c-frame is. DGETCF and DREMCF search that c-marker for a pair with first element = indicator, and return it, or NIL if there isn't one; DREMCF removes what it finds.

V. Manipulating Contexts

```
A. (CFRAME)                                          LSUBR
```

CFRAME returns a new c-frame with a number higher than that of any other. (It uses the second argument to DATA-INIT (qv.), adding it to the previous one it generated.) If there are as many contexts already as provided for by DATA-INIT, CFRAME calls the c-frame garbage collector to free space for more. If all the places are taken, the message

TOO MANY CONTEXT-FRAMES — CFRAME

is generated.

```
B. (PUSH-CONTEXT [context CONTEXT])                  LSUBR
   (POP-CONTEXT [context CONTEXT])                   LSUBR
   (NEW-CONTEXT c-frame-list)                        SUBR
   (SPLICE context)                                  SUBR
```

These functions create new contexts, and return them. PUSH- and POP-CONTEXT return contexts with one new c-frame added to, or the front c-frame removed from, context, respectively. If POP-CONTEXT tries to pop the last c-frame (i.e., GLOBAL) off, it errs with message

EMPTY CONTEXT — POP-CONTEXT.

NEW-CONTEXT creates a context by CONSing the flag *CONTEXT onto c-frame-list. The c-frames in the list must be in order of decreasing cnums, or the message

UNORDERED CONTEXT — NEW-CONTEXT

appears. In addition, the system refuses to create a new context without GLOBAL as its last frame, and <u>will</u> <u>RPLACD</u> <u>GLOBAL</u> <u>in</u> <u>if</u> <u>it</u> <u>isn't</u> <u>there.</u>

SPLICE adds a brand-new c-frame to context, <u>just</u> <u>after</u> its first frame. This c-frame will have a currently unused number between those of its successor and predecessor. If all such numbers are in use, the error message is

```
        NO NEW CNUM BETWEEN low AND high -- NEWCNUM
```

SPLICE is called for its side effect. Its value is EQ to its argument, but changed, of course.

Since SPLICE and PUSH-CONTEXT call CFRAME, they can cause its error.

C. (IN-CONTEXT context form)   CEXPR

CEVALuates form with CONTEXT rebound to context. Thus, for example,

```
    (IN-CONTEXT C1 '(ADD '(FALL SKY)))
```

is equivalent to

```
    (ADD '(FALL SKY) C1).
```

In general, IN-CONTEXT allows you to pretend any function takes an optional context argument. (There is no SUBR version of this program because it must rebind a Conniver variable.)

D. (MENTIONERS datum [sign NIL] [context CONTEXT])
                                                LSUBR
   (C-MARKER datum c-frame)   SUBR

MENTIONERS returns a list, in decreasing c-frame-number order, of all the c-frames in context that mention datum. If sign is non-NIL (i.e., + or -), it ignores all mentions with status ≠ sign, except that if sign = –, a mention by GLOBAL specifying no status is counted as unrealization in GLOBAL. If sign does = NIL, MENTIONERS returns all mentioning c-frames.

C-MARKER returns the c-marker of datum with cnum = number of c-frame, or NIL if c-frame doesn't mention datum. If c-frame is subsequently garbage-collected, or the c-marker ceases being a mention, the c-marker will no longer be attached to datum. Never say Conniver didn't give you enough rope.

E. (PATH context) SUBR

is a debugging convenience. Its value is a list whose first element is *CONTEXT, followed by the cnums of context's c-frames. Such an object serves no useful purpose, but it is much more more lucidly printable than context itself, in general.

Moth.
Callosamia
promethea
(Wingspread
4½ in.)