MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARTIFICIAL INTELLIGENCE LABORATORY

Artificial Intelligence
Memo No. 259a

May 1972
Updated January, 1974

THE CONNIVER REFERENCE MANUAL

Drew V. McDermott and Gerald Jay Sussman

ABSTRACT

This manual is an introduction and reference to the latest version of the Conniver programming language, an A.I. language with general control and data-base structures.

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.

Apology

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.

Conniver embodies few original ideas, but is hopefully 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. Our philosophy has been greatly influenced by Joel Moses.

Several people read the first versions of this manual and influenced this one, especially David McDonald, Terry Winograd, Sidney Markowitz, Michael Speciner, 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 toward avenging the confusion of an entire generation of programmers at his Lisp 1.5 manual.

Some of the notational conventions of the manual are:

Actual code is in upper case
Syntactic variables are lower case
Optional arguments or list components are delimited by brackets ([,]) surrounding the syntactic variable and its default value, if any.
Segment syntactic variables are delimited by stars (☆).
Quoted arguments are flagged with a quote ('). Unevaluated segment arguments start with '☆.
"Control-letter" is indicated by "~letter". "Up-arrow" is denoted by ^; "Left arrow," by .
"Altmode" is denoted by $.
Arguments are often given mnemonic names, as in (ADD atom). If the argument is not quoted, this notation means it is to evaluate to something described by the name; hence, (ADD (CAR X)) is legal if X starts with an atom.

DVM

Contents

I BASIC CONNIVER

I.1 Introduction

Conniver is a programming language designed to make easy the definition of processes cooperating to solve problems in the realm of Artificial Intelligence. These problems are often characterized by unpredictability of data-structure formats and flow of control, since programmers must try to use them to model flexibly some ill-defined part of the world. Each programmer wishes to make additions or changes to the data of his model in a way that is as close as possible to what he is thinking, without having to translate into some internal representation. It is just as true that he cannot be bothered with representations of processes and procedures; he would like to be able to refer to environments he perceives as "here" or "where control was a moment ago," without regard to where these environments exist on some internal stack, or whether they can even be referred to without advance preparation. This second aspect of A.I. problems is especially prominent when procedures are regarded as data ("beliefs"), to be monitored and understood as well as executed.

Conniver is a Lisp-like language which clears up some of these deficiencies in Lisp with two additions:

(1) a system-maintained data base

(2) the ability to manipulate arbitrary control environments.

Neither of these features is new. Readers who are acquainted with predicate calculus or PLANNER will soon recognize the more obvious characteristics of the data base. Those with knowledge of PAL and to some extent the lambda-calculus and Lisp 1.5 will already understand Conniver's control structure.

I.2 Pattern-Accessible Data

The data base is the place to begin. In some computer applications, "data" means huge arrays of numbers, which it is desired to store efficiently and crunch quickly; or a group of strings stored in well-defined variables, to be manipulated in a well-defined sequence. In A.I., the process that creates a datum does not usually know who wants it or what he wants it for. It cannot, therefore, store it in a standard variable. Furthermore, data are often not numbers but _facts_, such as "Mary is the mother of God." These facts are usually not so much computed as discovered, and they are not usually passed on to the next stage of the computation, but merely "made available" to whomever needs them. For example, some process may later ask, "Who is the mother of God?" or "Who are the children of Mary?" and the fact given must be accessible.

This is achieved by letting facts be modelled as arbitrary non-circular list structures, which are accessible via any combination of their components. These data structures are called _items_, and they are just "there" in the data base, rather than being thought of as properties pointed to by their individual atoms. Thus, a process may execute

```
        (ADD '(MARY MOTHER-OF GOD))
```

and that item will be _present_ to other processes that need it. (Notice that Conniver syntax is that of M.I.T. Lisp. The given _form_ calls for the application of the function ADD to one quoted (unevaluated) argument, namely (MARY MOTHER-OF GOD).) The effect of ADD can be undone with REMOVE, as in

```
        (REMOVE '(MARY MOTHER-OF GOD))
```

which leaves the data base in the state it was in before the ADD.

After an ADD, if another process wishes to access a fact, it may do so in several ways. All of them rest on the notion of specifying part of a desired fact and letting the system find all present items which agree in the specified part. This is a form of associative memory in which the user is free to let any piece of an item be its key, and all the rest to be what is retrieved. The specified and unspecified parts are intermixed in a _pattern_ which resembles the item, but has _variables_ instead of _constants_ in the parts to be filled in by the memory system. For example, the pattern (!>WHO MOTHER-OF GOD) specifies "mother" and "God," but has a slot labeled WHO to be filled in (i.e., be assigned as a variable) if there is an item (someone MOTHER-OF GOD) present in the data base. The characters "!>" specify that WHO is a variable, i.e., the name of the contents of the slot, not the actual contents.

The most fundamental way to use these patterns is with the function FETCH, which takes a pattern as an argument, and returns a <u>possibilities list</u> containing all items which <u>match</u> it in the sense described. In my example, (FETCH '(!>WHO MOTHER-OF GOD)) returns

```
((☆POSSIBILITIES (!>WHO MOTHER-OF GOD))
 ☆IGNORE
 (☆ITEM ((MARY MOTHER-OF GOD) (0 (0 . +)))
        ((WHO MARY)))).
```

This list contains all the items (here there is only one) which match the pattern, and a good deal more besides. (FETCH returns a lot of information which it must compute anyway; usually most of it is discarded.) The detailed format of this list will be described later.

I.3 Conniver Programs

Usually, a list of items such as this will be processed by looping through its elements, looking for one with some desired property or doing something to each of them. This is done by using the function TRY-NEXT, which removes the next possibility from the list and returns it. In the case of item possibilities, it returns the item and sets the pattern variables as the association list in its item possibility directs. If the variable P is bound to the list given, (TRY-NEXT P) has the value ((MARY MOTHER-OF GOD) (0 (0 . +))), and the side effect of setting variables as the association list ((WHO MARY)) dictates; i.e., it gives the variable WHO the value MARY. (The markers, of the form (0 (0 . +)), on the item returned may be ignored for now.)

Now imagine the items (MARY MOTHER-OF GOD), (RHEA MOTHER-OF GOD), and (ISIS MOTHER-OF GOD) are in the data base. (The idea that a creature can have only one mother is, of course, not built into the system.) The following program will print out all these mothers:

```
(PROG "AUX" (WHO (P (FETCH '(!>WHO MOTHER-OF GOD))))
:LOOP (TRY-NEXT P '(RETURN NIL))
      (PRINT WHO)
      (GO 'LOOP)    )
```

This program is reminiscent of a Lisp PROG, with some differences:

(1) Local variables are declared as auxiliaries explicitly by putting the atom "AUX" before the bound variable list for the PROG. (The list and marker could be omitted entirely.) These variables are not automatically bound to NIL. Atoms appearing in the list (like WHO) are bound but unassigned. List of the form (atom expression), such as (P (FETCH...)), specify that atom is to be bound to the value of expression.

(2) The second argument to TRY-NEXT gives its value when there are no more possibilities. (It must be quoted to avoid being evaluated when passed to TRY-NEXT.)

(3) Tags are preceded by ":" to distinguish them from ordinary atoms appearing in PROG bodies. This is necessary because Conniver PROG's return the value of the last expression in their bodies. For example,

```
(PROG "AUX" (WHO (P (FETCH '(!>WHO MOTHER-OF GOD))))
   (TRY-NEXT P '(RETURN NIL))
   WHO)
```

returns the first mother of God it finds, or NIL if there aren't any.

(4) GO always evaluates its argument.

I.4 Contexts

So far, I have spoken in terms of one data base or world model, in which items can be added and fetched. The major novel feature of the Conniver data base is that any number of world models, called <u>contexts</u>, are allowed. Each may be manipulated entirely independently. These different states may be used to model hypothetical worlds, alternative cases of a proof, different board positions in a game, or different times, or all of these.

The user starts with a single, global context, bound to the variable CONTEXT. He may change it by executing ADD to store a new item, or REMOVE to delete an old one. These changes will cause items actually to appear or disappear from the "index" which implements the associative data base. (See Chapter III.) A rather different kind of change is achieved if the form

```
        (CSETQ CON1 (PUSH-CONTEXT CONTEXT))
```

is evaluated, which stores a new context as the value of variable CON1. (CSETQ is the Conniver analogue of Lisp's SETQ.) The result is a new data base whose contents are initially exactly the same as those of CONTEXT. CON1 is said to be a <u>sub-context</u> of CONTEXT; this is meant purely in the sense that a stack frame of a language processor is a sub-frame of the frame beneath it on the stack. (See Chapter II. Contexts are implemented as stacks of "layers," which are analogous to stacks of frames.) Although initially CON1 is identical to CONTEXT, arbitrary additions and removals may happen to it which have no effect at all on CONTEXT.

For example, assume that the "theology problem solver" I have been describing has in its data base (CONTEXT) the item (MARY MOTHER-OF GOD) (and that the other mothers have been removed). A routine might perform the following actions, which involve the specification of a new religion (or a heresy):

```
          (CSETQ CON1 (PUSH-CONTEXT CONTEXT))
          (REMOVE '(MARY MOTHER-OF GOD) CON1)
          (ADD '(KONNIVA MOTHER-OF GOD) CON1)
```

Now CON1 differs from CONTEXT only in whom it represents as the mother of God. Notice that ADD, REMOVE, and FETCH take optional second arguments which specify which context they apply to. The default value of this argument is the value of CONTEXT. Now (FETCH '(!>WHO MOTHER-OF GOD)) has the same effect as before, but (FETCH '(!>WHO MOTHER-OF GOD) CON1) returns

```
((☆POSSIBILITIES (!>WHO MOTHER-OF GOD))
 ☆IGNORE
 (☆ITEM ((KONNIVA MOTHER-OF GOD) (10 (0 . +)))
         ((WHO KONNIVA)))).
```

I shall work into a less sacrilegious example, which will serve as a justification for Conniver control structure, by making the following observation. A common technique in Conniving is to <u>rebind</u> CONTEXT (e.g., in an "AUX" list) to a sub-context. This has the effect of making "hypothetical" all data-base changes performed inside the scope of the new binding. It also means that returning from this scope causes all these changes to disappear, unless special precautions are taken.

I.5 Defining Conniver Functions

The usual place besides PROG's in which one statement after another is evaluated is inside a function body.

I mentioned before that contexts may be used to model board positions in a game. The example I will now pursue is a part of a tic-tac-toe program which has never been completed. For this program, I organize the data base as a collection of items about the squares. A square is a number from 1 to 9; a player is one of the symbols O or X. In the initial context, there is an item of the form (FREE s) for all squares. When a player p makes a move in square s, the item (HAS p s) replaces (FREE s).

One of the subroutines required in a tic-tac-toe program is (FORCEWIN player square), which is non-NIL if and only if a player can force a win on the next move of a game by playing in square. It is defined using CDEFUN, which is analogous to Lisp's DEFUN:

```
(CDEFUN FORCEWIN (PLAYER SQUARE)
       "AUX" ((CONTEXT (PUSH-CONTEXT CONTEXT))
              (WM !"((☆POSSIBILITIES)
                    ☆IGNORE
                    (☆GENERATOR (WINMOVES PLAYER)))))
   (ADD !"(HAS ,PLAYER ,SQUARE))
   (REMOVE !"(FREE ,SQUARE))
   (COND ((MAKEMOVE (OTHER PLAYER))
          (TRY-NEXT WM NIL))   )).
```

In English, square is a forced win for player if player still has a winning move after the other makes his best reply to square. Notice that this statement has a hypothetical ("if...") clause in it, which corresponds to the pushing-down of CONTEXT before the move to square is made. When FORCEWIN is exited, none of the effects of ADD, REMOVE, or MAKEMOVE will be visible.

FORCEWIN illustrates several new Conniver features. The macro-characters !" are used to signify <u>skeleton</u> <u>expansion</u>. !"(☆elements☆) is like '(☆elements☆), except that some of the elements are evaluated and their values substituted into the result, so that it is not EQ to the original "skeleton." Within a skeleton, "," indicates that the value of a Conniver variable is to be substituted. Thus, if PLAYER=X and SQUARE=5, !"(HAS ,PLAYER ,SQUARE) has value (HAS X 5). Other characters have other uses; see Sect. VII.1.

(MAKEMOVE player) is the main tic-tac-toe player being called recursively; FORCEWIN is itself a subroutine of MAKEMOVE. It adds to the data base the best move player can make. (OTHER player) is defined as (OTHER X)=O and (OTHER O)=X.

I.6 Generators

The most important feature of FORCEWIN is the generalization it makes of possibilities lists. The possibilities list WM contains a new kind of possibility, of type ☆GENERATOR, whose second element is a form, (WINMOVES PLAYER). When TRY-NEXT sees such a thing, it evaluates the form, in this case calling the function WINMOVES.

Such a function might do anything. If it just returns, TRY-NEXT just goes on to the next possibility (if any). In the usual case, however, the function behaves like a <u>generator</u>, adding new possibilities to the list in its place. Thus, such a possibility "stands for" the "real" possibilities which the function is capable of generating. In a rudimentary sense which will be expanded, the list of possibilities is a communication channel between the generator and the function that called it.

What WINMOVES wishes to communicate is all the winning moves its PLAYER has in the board position in which it is called. If there are any, they are simply put into the possibilities list as numbers. When TRY-NEXT sees an element of this sort (with no flag like ☆ITEM or ☆GENERATOR), it assumes it is a <u>value possibility</u> and merely pops it off and returns it. So, in this case, the last line of FORCEWIN means, "return a winning move for PLAYER, if any, else NIL."

The description of generators that is to come will introduce and justify Conniver control structure. Before I give it, let me summarize what has been created so far. So far, Conniver is an ordinary Lisp-like language with a system-maintained data base. This data base has the interesting property that it consists of any number of contexts, related in a tree structure. [It might look as if restricting them to be bound only to bindings of the variable CONTEXT would collapse this tree into a stack, of which only the topmost context would be important at any one time. This assumption will be shown to be false.]

A simple version of the generator WINMOVES would look like:

```
(CDEFUN 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)  ).
```

There are some new functions and notations to be explained here. The new functions are COND, PRESENT, THIRD-IN-ROW, NOTE, and ADIEU. (LESSP is a Lisp function, which is callable from Conniver.) COND is the Conniver version of Lisp's COND, which differs from it only in that a COND clause, after the test form, is just like a PROG; "AUX" variables and statement labels are allowed; but these features are not used here. (PRESENT pattern) is a system function almost like (TRY-NEXT (FETCH pattern)); it returns an item and sets the pattern variables if there is one present that matches the pattern. THIRD-IN-ROW returns the third square in a line with its arguments, or NIL if they are not collinear.

Before I describe NOTE and ADIEU, which allow WINMOVES to communicate with its caller through its calling possibilities list, let me explain the prefix "!," used in the three FETCH patterns (including that of PRESENT). "!,PLAYER" in this program matches only the current Conniver value of PLAYER. (It actually has a slightly more general meaning; see Chapter IV for a complete description of this and several other pattern prefixes.) The FETCHes in this program, like that of my earlier program for printing mothers of God, create possibilities lists (P1 and P2) of items, which are used in TRY-NEXT-driven loops. Notice that P1 and P2 are substantially the same list, except that each sets a different variable, and that P2 is re-created more often than P1. Each is a list of all items corresponding to squares owned by PLAYER. They are used to generate all pairs of squares owned by PLAYER. (The LESSP clause of the first COND is used to discard redundant or degenerate pairs. This is an inefficient, but clear, way of doing things.)

Whenever WINMOVES has found two collinear occupied squares with a free third collinear square, it must insert this third (winning) square in the possibilities list. This it does with NOTE. When all winning moves have been discovered, P1 will be exhausted, and the TRY-NEXT at statement :OUTERLOOP will execute (ADIEU). In this case, ADIEU merely returns to TRY-NEXT, which returns the first NOTEd winning square, if there are any.

I.7 Generalized Control Structure

The type of generator I have described so far is merely an odd function which is capable of returning zero or many values instead of just one. As such, it is not very interesting. In some cases, also, it is inefficient. It may be, for example, much more expensive to generate possibilities than to use them; or the expense of generating them may grow as fewer and fewer remain; or the number may be infinite. Another type of difficulty is that the generation of successive possibilities may depend on what the calling function did with previous ones; the caller may want to advise the generator as to how to proceed. Or it may simply be that the generator has no idea how many possibilities its caller wants. (Notice that FORCEWIN is interested in only one winning move.)

What is needed is a way of returning some of the possibilities while maintaining the generator in existence for further duty if required. I will describe in a moment the AU-REVOIR feature that allows this to happen, but first the question must be answered, what is being maintained in existence? In this case, what is being saved is a description of the "process" embodied by the generator. This description must include the values and names of the variables bound there, the body of the generator and where control was before it returned, and who it returned to. All of this information is saved in the <u>frame</u> of the generator. This frame is created when the generator is called (as it is for every function). Normally, frames are lost when control returns from them; they are garbage-collected along with their bound variables, including any bindings of CONTEXT they may have. This is why I said the context tree might look like a stack. It is also why, in most programming languages, frames are stored on a stack ("frame" originally meant "section of stack"), and always go away when they are popped off.

In Conniver, however, frames are accessible data structures which can be protected from garbage collection merely by being pointed to. Protecting a frame in this way means that its bindings must remain in potential existence, since they are always capable of being resurrected. The ways in which such frames can be used will be described. For now, notice that one implication of this design is that Conniver control and context structures must be at least as complex as trees (Cf. sect. II.1).

One thing that can be done with frames is to make tags, which can be GOne to like atoms; however, GOing to a tag restores its bindings, no matter when they were created. So, a generator can keep itself in existence by generating a kind of tag as a possibility. This tag stands for the further possibilities it can generate the way its original generator stood for all its possibilities. Such a tag is called an *AU-REVOIR possibility, and is generated by calling (AU-REVOIR). This form behaves like (ADIEU), but, by saving a tag to its own frame, can potentially return "again," inside the generator, causing it to note new values, and repeat the AU-REVOIR or do an ADIEU.

As an example, consider the following version of WINMOVES, which returns one winning move at a time:

```
(CDEFUN 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 generates just two possibilities: 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 _in WINMOVES_ 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 _user_ 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 (Sect. VII.1.8).

II HAIRY CONTROL STRUCTURE

Chapter I has demonstrated that the control structures necessary to support things like generators are of a sort which are illegal in most languages. Just how illegal will now be made clear.

II.1 What a Frame Is

II.1.1 How to Be a Programming Language

If you were to simulate a PROG, there are two things you would have to keep track of: which line of the program you were working on; and what the current values of its "AUX" variables were. If this PROG evaluated another one, you would have to stop what you were doing, and do something similar to the new one.

In evaluating the new one, however, there are two new things to remember: which line of the old program to work on when you get back to it, and what the values of the "AUX" variables of the old program are:

```
(PROG "AUX" ((X 5) (Y 10))
   (PROG "AUX" ((X 50))
      (PLUS X Y))
   (PLUS X Y)).
```

The inner PROG requires the values of the outer X and Y for two reasons: it must be able to refer to them free (as it does here with Y), and must be able to restore their values when it returns (as it does here for X). Thus, the inner sum of this example has value 60; the outer, and the whole expression, 15.

A language interpreter has need of the same information. It stores it in an object called a "fr," with four slots:

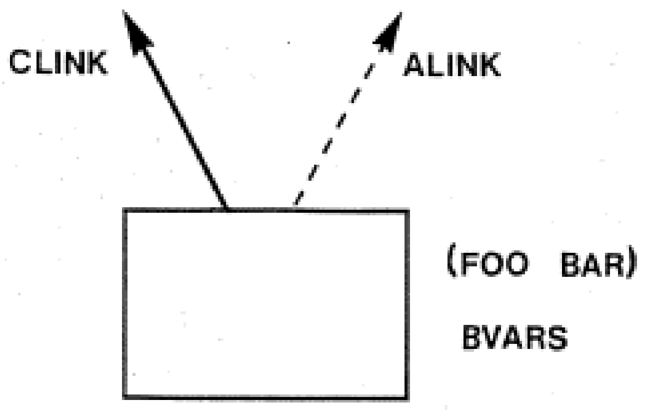
(1) BVARS (Bound VARiableS): a pairing of a location with each bound variable name.

(2) IVARS (Internal VARiableS): a specification of what the interpreter is now doing. (For example, which line of a PROG it is working on, or which argument of a PLUS it is evaluating.)

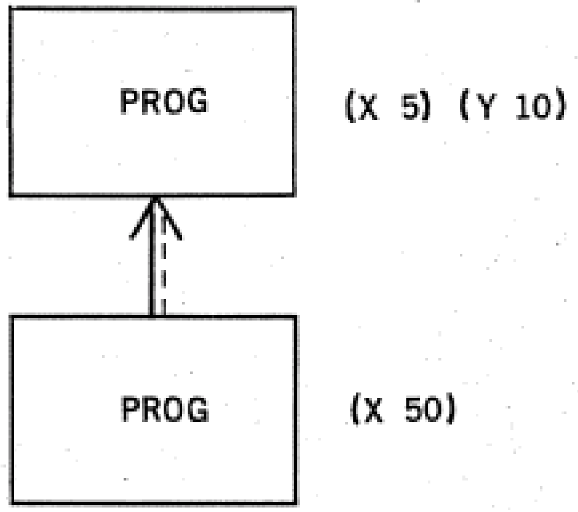
(3) ALINK (Access LINK): the fr whose BVARS and ALINK are to be searched for any free variables that are not bound in this fr's BVARS.

(4) CLINK (Control LINK): the fr to which control is to return when it leaves this one. The IVARS of CLINK specify how the running of that fr is to proceed.

We will represent a fr by a box, with bound variables indicated beside it, whose CLINK is an arrow pointing to another fr, and whose ALINK is a dotted arrow pointing to yet another one. (Fig. 1(a).) If ALINK(fr) = CLINK(fr), a double arrow will be used. (Our terminology and notation are a simplified version of that of Bobrow and Wegbreit (1972).)



A. A fr.



B. TYPICAL CASE

FIGURE 1

The control structure during execution of the PROG's given before is shown in Fig. 1(b).

It might be asked whether anything more complicated than Fig. 1(b) is necessary or possible. In some languages, it is not. PL/I, for example, allows this structure and no other. FORTRAN imposes the additional restriction that no two fr's in a chain of fr's be created by the same function; hence, Fig. 1(b) is not even possible in FORTRAN.

However, other structures are imaginable.

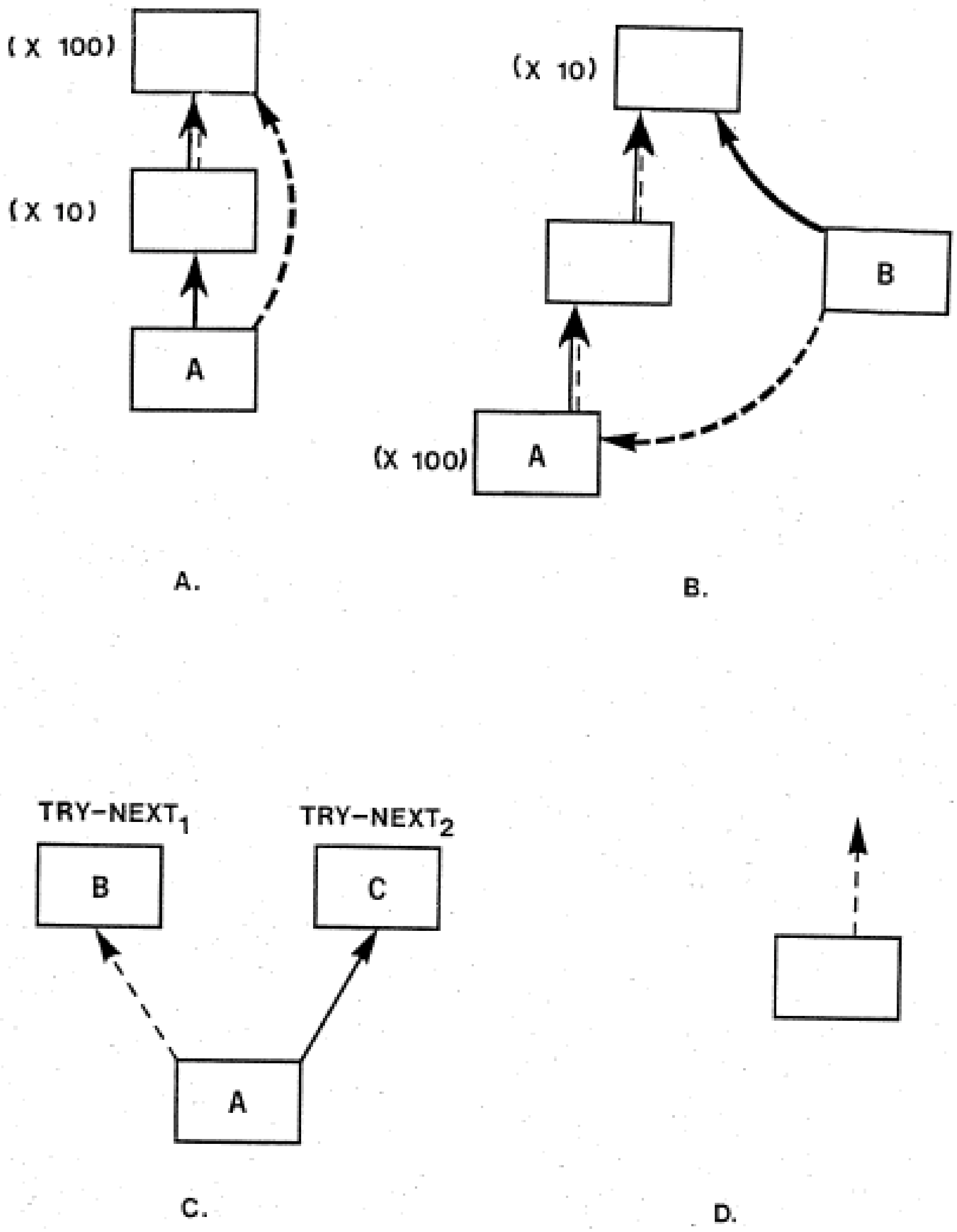


FIGURE 2

Figure 2.

In Fig. 2(a), X has value 100 rather than 10 in fr A. The same is true for fr B of Fig. 2(b). Fig. 2(a) is a legal structure in Algol, MAC Lisp, and Lisp 1.5. (In the last, however, access fr's and control fr's are different kinds of entities.) Fig. 2(b) is legal in Lisp 1.5 only. (These structures arise from the application of "FUNARG's"; see below, sect. 1.2.3.) The other cases are unusual. Fig. 2(c) shows the typical situation of a generator revived after an AU-REVOIR. No one has yet thought of a use for Fig. 2(d).

These abstract objects may be implemented in various ways. In FORTRAN, a fr is not clearly distinguished from a function; in addition, each function has as ALINK only the COMMON area. In most languages, fr's are implemented as stack frames, which can be piled up as Fig. 1(b). Once such a frame returns control to its caller, that frame is no longer referenceable. This condition rules out the structures of Fig. 2(b) and (c). (In Lisp 1.5, the BVARS and ALINK of a fr are preserved after it is popped off, so Fig. 2(b) is possible.)

Notice that Fig. 3

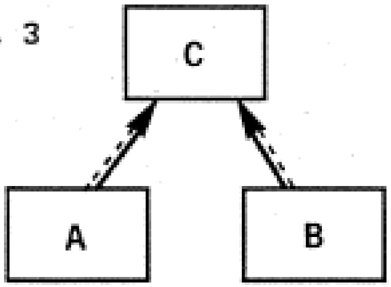


FIGURE 3

is ambiguous, in that the two subfr's of C may be meant to be chronologically exclusive or not. In the former case, A and B may share stack space; otherwise, as in Fig. 2(a), they may not. It is clear what the chronological interpretation of Fig. 3 is: C called A, A returned, then C called B. What is the other interpretation? Simply that A stands ready to begin execution again or supply its bound variables' values (as in Fig. 2(b)).

[We leave out a notation for IVARS in fr's of Fig. 3 and elsewhere, to make things simpler. It is clear that A and B must see different IVARS for C, so that different things may happen when they return. (There is nothing to prevent A from returning several times.) Leaving out IVARS allows us to be vague about exactly which point in the execution of a fr we intend; remember only that each CLINK must specify its superior's IVARS.]

II.1.2 Conniver Control Structure

In Conniver, a fr is an internal list structure which specifies BVARS, IVARS, ALINK, CLINK, and EXP, the last being the form whose evaluation led to the creation of the fr. Whenever a non-atomic expression is CEVALuated, a new fr is created. (The only exception is FEXPR applications; Lisp EXPR's, SUBR's, etc. do get Conniver fr's.)

Conniver fr's are used as internal interpreter structures (i.e., parts of other fr's) as described, but they are also accessible to users as parts of frames, tags, closures, and ☆AU-REVOIR possibilities. These concepts will be explained.

II.1.2.1 Frames

A _frame_ is a structure of the form (☆FRAME fr). This is the external representation of an unadorned fr. It may be used in two basic ways, for relative evaluation or continuation. Evaluation of an expression relative to a frame is a way of creating its frame with an abnormal access link (cf. Fig. 2(a) and (b)), namely the fr of the given frame. (Another way to do this is with closures; see below.) Relative evaluation is done with the function (CEVAL expression frame).

Continuation of a frame is returning control to it as directed by the IVARS of its fr. This is done by (CONTINUE frame). To CONTINUE from a frame's control link, use (EXIT value frame), which causes frame to return with value. When no frame value is given, EXIT exits from the most immediately enclosing COND, PROG, CEXPR, or method body. RETURN bypasses COND, so is often more convenient (See sect. VII.1.5).

Frames are created by the function (FRAME), which returns the frame of its caller. (More precise and complete definitions of this and other functions are given in the appendix, sect. VII.1.4.)

For example, after the execution of

```
(PROG "AUX" ((X 50))
   (CSETQ GLOB (FRAME))
   (PRINT 'FOO)),
```

global variable GLOB will be bound to the frame of the PROG. (This is because FRAME returns a frame with the ALINK when it is called as its fr.)

Now (CEVAL 'X GLOB) has value 50. (CONTINUE GLOB) causes FOO to be printed again.

Other functions can be used to manipulate Conniver frames. (ACCESS frame) and (CONTROL frame) return the frames for the ALINK and CLINK of frame's fr, respectively. SETACCESS and SETCONTROL reset the appropriate links of frames. For example, (SETCONTROL A A) causes the state of affairs shown in Fig. 2(d).

II.1.2.2 Tags

Another way to use fr's is to create and use fr's associated with a labelled section of a PROG, function body, or COND clause. Such an object is called a tag, and is of form (*TAG label fr), where fr is a frame whose IVARS instruct any CONTINUEr to begin execution at that labelled piece of body.

Such a tag is created using the function (TAG atom), which searches the current body for a tag of the form ":atom". If it is not found, the CLINK of the current fr is followed, and the process repeated.

Tags may be used in any context that allows frames, including CEVAL and CONTINUE. A synonym for CONTINUE with a tag argument is GO. If GO has an atomic argument atom, it is equivalent to (GO (TAG 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
   (PRIN1 'BAR)
   NIL)

(PRINTFOOBAR)
```

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

II.1.2.3 Closures

Functions and other "procedural" objects (see below) may be associated with fr's to form closures, data of the form (☆CLOSURE function fr). A closure behaves like its function, except that its frame will have ALINK=fr rather than ALINK=CLINK. Consequently, its free variables will be looked up using fr. This may give rise to a structure of form Fig. 2(a) or (b).

Closures are generated by (CLOSURE function), which returns the closure of function in the fr of (FRAME). Since these objects may be returned or assigned to free variables, they may point to exited fr's, as in Fig. 2(b).

For example, the following function of X returns a function which adds X to anything:

```
(CDEFUN PLUSX (X)
   (CLOSURE '(CLAMBDA (Y) (PLUS X Y)))).
```

Then, if F = (PLUSX 5), (CALL F 4) = 9. When F is called, Y is bound to 4.

Figure 4.

Closures can be used in any context as though they were the frames of their fr's. Closures of methods are described below, Sect. II.2.

II.2 Methods

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, 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 _data base-sensitive_ procedures are called _methods_, of type _if-added if-removed_, or _if-needed_.

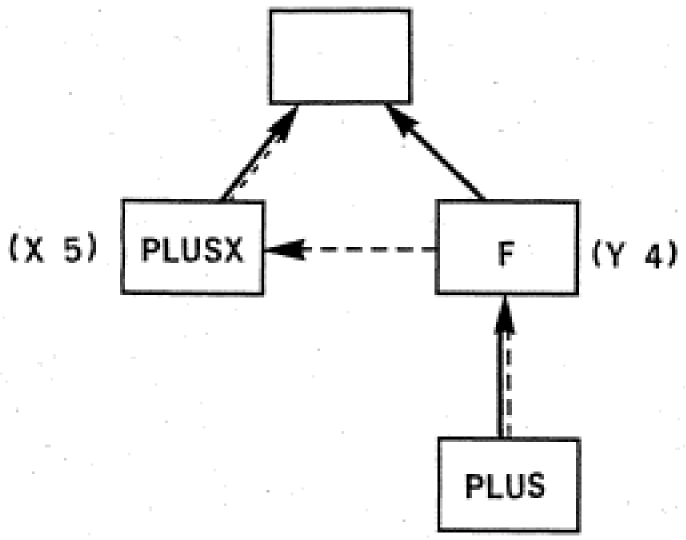


FIGURE 4

II.2.1 If-addeds and If-removeds

When an item is added (removed), any present if-addeds (if-removeds) whose patterns match the item are invoked. When a method is invoked, a new frame is created for it and the result of the match with its pattern (see Chapt. IV) is used to create its initial variable bindings. (If the match fails, the method is, of course, not invoked.) Auxiliary variables may be bound by including an "AUX" declaration at the beginning of the method body. Execution begins at the front of the if-added's (if-removed's) body, right after the "AUX" if there is one. For example, the (anonymous) method

```
(IF-ADDED NIL (HAS !>WHO !>SQUARE)
              ((REMOVE !"(FREE ,SQUARE)))
```

with name NIL, pattern (HAS !>WHO !>SQUARE), and body ((REMOVE !"(FREE ,SQUARE))), 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 (sect. I.5).

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 (see sect. III.1.2.2). 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

```
(REALIZE (IF-ADDED (HAS !>WHO !>SQUARE)
         (REMOVE !"(FREE ,SQUARE))   ))
```

and removed by UNREALIZing 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)
         (REMOVE !"(FREE ,SQUARE))   ))
```

causes the atom HAS-FREE to be associated with the method (under the indicator DATUM), 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 (cf. sect. III.4).

II.2.2 If-neededs

The third method of data base-control structure interaction is by use of if-needed methods, 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 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 in the same way ADD and REMOVE invoke their methods: the result of a successful match (an alist; see Chapt. IV.) will be present in the possibilities list of the method; it will be used to start the bound variables of the method's frame, which are augmented by any "AUX" variables that may be around. Execution begins in the method, which behaves like a generator function (see sect. 3) 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 a note possibility to be appended to POSSIBILITIES. Since TRY-NEXT has the same effect on a note as on an item possibility, NOTE simulates the presence of that instance as an item in the current context. ADIEU and AU-REVOIR work in the same way as before.

The patterns of if-neededs may use match characters which are not usually used elsewhere. This is because if-needed patterns are matched against FETCH-patterns that may include other variables, whereas all other patterns are matched against constant list structures. The most important such special prefix is "!<", variables prefixed which match only with expressions with variables when the method is entered. When the function INSTANCE is called, the variables prefixed with "!<" will have been assigned by execution of the body of the method, and their values will be transmitted back to the variables that they matched in the FETCH-pattern.

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" ((P (FETCH '(DWARF !>X))))
   :LOOP (TRY-NEXT P '(ADIEU))
         (AU-REVOIR (INSTANCE))
         (GO 'LOOP)    ))
```

VD will be invoked when it is found by a call like (FETCH '(VICIOUS !>Y)), i.e., an attempt to generate vicious creatures. It would not be invoked by a call like (FETCH '(VICIOUS LYNDON)), because !<X will not match a constant like LYNDON. This method

notes one vicious dwarf each time TRY-NEXT is called. (The matcher is explained in detail in Sect. IV.)

II.2.3 Closures of Methods

Methods may have closures just as functions 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 <u>not</u> 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,

```
    (CSETQ F1
           (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 in the frame F1.

HANG can be defined as

```
(CDEFUN HANG (RELEASE EXPRESSION)
             "AUX" (HANGFRAME)
   (CSETQ HANGFRAME (FRAME))
   (REALIZE (CLOSURE
               (CEVAL (CONS (CAR RELEASE)
                            (CONS (CADR RELEASE)
                                  '((EXIT (FRAME)
                                          HANGFRAME)   ))))))
   (CEVAL EXPRESSION (CONTROL))   )
```

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 immediately causes the HANG to return for a second time (i.e., have its frame be exited), this time with value (FRAME), which will be the frame of the method closure's activation. The method EXITs from the correct frame because it looks up the value of the variable HANGFRAME by searching up the access chain (see sect. II.1), which points off to the frame of the HANG in which it was closed. Notice that, having added to the current context the closure that does these marvelous things, HANG 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. (See sect. II.1)

The utility of method closures is somewhat reduced by the fact that it is dangerous to ADD closures of methods containing bindings of CONTEXT. This is explained in sect. III.1.2.2.

II.3 Generators

The basic operation of generators was explained in Sect. I as an illustration of Conniver control structure. Since we have explained more of it now, it is worthwhile to describe in detail how generators do what they do.

A generator is any process which communicates with another process through a possibilities list. The list of possibilities is a communication channel which must be set up before the generator is called. As we have seen, the generator is associated with that particular list by being found as a *GENERATOR or *METHOD possibility in it. Once the function or method is called, it behaves like any other, except that the value it might RETURN is ignored; all its important values are to be deposited in the list, as the data the possibility "stood for" to the TRY-NEXT. This means that, unlike most functions, generators may return zero or more values, instead of exactly one.

The mechanics of this are simple. While it is running, a generator can see an invisible variable bound to the rest of the possibilities list, starting from where it was found. NOTE, ADIEU, and AU-REVOIR put new possibilities into this part of the list, in the order in which they are called. AU-REVOIR also leaves an ☆AU-REVOIR possibility, which is like a tag (sect. VII.1.8).

The real trick is in TRY-NEXT. When it finds an AU-REVOIR frame in a possibilities list (while it is chewing on the list after the generator has returned), it replaces the control link in the top frame of the generator structure to point to the new TRY-NEXT, and just EXITs the AU-REVOIR frame. This is how structures of the form of Fig. 2(c) come about. Control stays in the generator on its second round, with all free variables as before, until it is ready to return again, when it will return to the new TRY-NEXT, and ultimately to the caller of that TRY-NEXT.

A generator may return anything as a possibility. However, there may be special types it wants to use, either one of the system-defined types, or something the user has made up. These will have format (type-atom ...). The system-defined types are ☆ITEM, ☆METHOD, ☆GENERATOR, ☆AU-REVOIR, ☆NOTE. These are explained in detail in Sect. VII.1.8.

II.4 Applications of Control Structure

This control structure is intended to be manipulable by the user. HANG, for example, is written in Conniver, not Lisp. In this section I give two quick examples of the kind of manipulations that may be done easily with Conniving control structure.

II.4.1 Backtracking

The backtracking 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.

Clearly, the type of generator we are describing does not work with AU-REVOIR. Instead, we must call SUCCEED explicitly to return:

```
(CDEFUN SUCCEED ()
   (ADIEU (INSTANCE))   ),
```

and generation of more than one instance is done by failing back to failpoints within the body of the if-needed method.

The two remaining functions simulate ASSERT and ERASE in that their 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.

II.4.2 Multiprocessing

It is just as easy to create various types of multiprocessing in Conniver. This comes in handy for building goal trees, for example. The easy way to do this is often just to throw tags and frames around, but you may prefer a more rigid format. The following little number treats all processes as atoms with a tag under the indicator PROCESS which indicates where control is to resume to continue execution of that process. The atom CURPROC always has one such atom as its value:

```
        (CSETQ CURPROC (GENSYM))
```

Processes are created in association with a function of no arguments by the function CREATE:

```
(CDEFUN CREATE (FUNC) "AUX" ((NEWPROC (GENSYM)))
   (PUTPROP NEWPROC (TAG 'APPLY) 'PROCESS)
   (RETURN NEWPROC)
:APPLY
   (CALL FUNC)
   (CERR PROCESS TRIED TO RETURN)   )
```

Processes so created may be resumed at any time with the following function:

```
(CDEFUN RESUME (PROC)
   (PUTPROP CURPROC (TAG 'RESUME) 'PROCESS)
   (CSETQ CURPROC PROC)
   (GO (GET PROC 'PROCESS))
:RESUME).
```

Processes never return (that would generate an error, as in the last line of CREATE), but resume each other back and forth as they wish. To do this, they must know each other's names. The simple scheme shown here doesn't explicitly allow for that kind of communication, but it is not hard to see how RESUME, for example, might be redefined so as to return the name of the process that ultimately resumes the process that called it.

To destroy a process, execute (REMPROP process-atom 'PROCESS). It will then be garbage-collected.

III HAIRY DATA STRUCTURE

The basic features of the Conniver data base are its context structure and indexification. They allow the user to create sets of data which are fetchable by pattern, have property associations, and exist in different configurations.

Until the discussion of implementation in Sect. 3 of this chapter, the right way to think about contexts is as a tree of data bases. Anything that is added (including properties; see sect. 2) to a given context is immediately present in it and all its daughters, and will be automatically present in all new daughters that are sprouted from it using PUSH-CONTEXT. However, the effect is completely invisible in its super-contexts. Exactly the same applies to removal of data (or properties); whatever was removed is gone in sub-contexts, still around in all super-contexts that used to contain it. The mechanism for all this will be explained in sect. 3.

In Chapter I, the data base was thought of as containing items, arbitrary lists that were present or absent in each context. Such items are implemented as types of data, called item data, which can be referred to, as we have seen, by patterns. In this chapter, we will look at how to refer to items by pointers to their data, and thus introduce the concept of manipulating a datum.

A datum is a system-maintained entity, which has characteristics which depend on its type, and which is either present or absent in any given context. The data we have seen so far include item data and three kinds of method. These types are _indexable_ data, which appellation refers to the peculiar pattern-directed way of accessing them. I will return to consideration of the data index later (sect. III.1.2).

III.1 Types of Data

III.1.1 Objects

A more primitive kind of datum has the same behavior with respect to contexts, but has no pattern to be referenced by. It is accessible, like any Lisp structure, by a pointer to it. This is the _object_.

Many people are completely mystified by the notion of an object datum, since they prefer to think of a data base as a set of known assertions. The most simple way they can think of to access it is to ask something like (PRESENT '(BLOCK A)), which checks whether A is a block; or (FETCH '(BLOCK !>X)), which gets a possibilities list of things currently believed to be blocks.

However, this kind of a data base is a recent development in A.I. A more primitive kind just assigns various symbols as properties of other symbols. This type is sometimes simpler and more efficient to use. The first step toward making this kind of data base context-dependent would be to invent a "symbol" which is "present" only some of the time. This is part of the concept of object.

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 modelled 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-THINGS:

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

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 <u>structure</u> (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, one executes (REALIZE NEW-THING); to make it absent, he executes (UNREALIZE NEW-THING). The predicate REAL returns its argument if it is present, or NIL if it is absent; UNREAL, the opposite.

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 <u>higher</u> point in the branch, the "X" modelled as (XS 5) can be made to "vanish," as (XS 5) reverts to absence.

This simple introduction to objects leaves them rather uninteresting, since they have exactly one context-dependent property (presence or absence) and a context-independent structure. In fact, the most interesting uses of objects involve more general properties, which are discussed in Sect. III.2.

III.1.2 Indexable Data

An object behaves like any normal Lisp structure. You use it by having a pointer to it, to which functions and predicates are applied. The only difference is that its properties depend upon the current Conniver context.

Another class of data have the same properties as objects: given a pointer to one, exactly the same operations (REALIZation, UNREALIZation, reality testing, etc.) can be performed. However, the way you obtain a pointer to one is by use of _patterns_. That is, upon specification of all or part of a list structure associated with such a datum, Conniver will generate or regurgitate the data that _match_ what is given. These are called _indexable data_, because the restoration of a datum from a description of a piece of it is not possible without the presence of an _index_ to all data of its type.

The indexable data types implemented in Conniver are _item data_, _methods_ (of several types), and _method closures_.

III.1.2.1 Types of Indexable Data

III.1.2.1.1 Item Data

The obvious operations on item data have been described in Chapter I. From that chapter, it should be clear that item data can always be referred to by their associated items or patterns that match them, using functions like ADD, REMOVE, PRESENT, and FETCH. However, each of these functions returns pointers to the item data involved, the direct accessing of which is more efficient that access through the index.

For example,

```
        (CSETQ D1 (ADD '(HERSHEY BAR)))
```

returns

```
        ((HERSHEY BAR) (0 (0 . +))),
```

or something similar. This structure has a CAR which is the item of interest, and a CDR which describes its properties in various contexts (see sect. III.3). The whole thing is an <u>item</u> <u>datum</u>, and is pointed to by the <u>item</u> <u>index</u>. This means that (in a sub-context this time) if (REMOVE '(HERSHEY BAR)) is executed, the same item datum can be found, certain mumbo-jumbo performed on it, and

```
        ((HERSHEY BAR) (10 (1)) (0 (0 10)))
```

returned. This is the exact same datum (it is EQ to the first), with its structure changed. (As will be described below (sect. 3), its "tail" indicates that it is still present in the original context, but absent in the sub-context.)

It takes effort to go from the structure (HERSHEY BAR), EQUAL to the item wanted, to ((HERSHEY BAR)...), which is EQ to its item datum. However, in this case D1 is a pointer to this datum, so there is no reason why one cannot execute

```
        (UNREALIZE D1)
```

in the same sub-context as the previous REMOVal, with exactly the same result. In particular, REALIZE and UNREALIZE call if-addeed and if-removed methods respectively, just as ADD and REMOVE do.

In fact, ADD can be defined as

```
        (REALIZE (DATUM item))
```

where DATUM is a function that maps items into their associated item data. (If an item datum was not previously indexed, DATUM generates one.) REMOVE has a similar paraphrase.

For a description of the exact meaning of an item's being indexed, see below, III.1.2.2.

III.1.2.1.2 Methods and Method Closures

The use and creation of these objects is dealt with elsewhere, in Sect. II.2 and VII.1.2. Some description of how they behave as <u>data</u> will be given here.

Any datum that starts with the atom *CLOSURE is treated as a closure. If a datum starts with any other atom but *OBJECT, it is supposed to be a method, of the form

```
        (type name pattern body ...),
```

where "..." is its tail (see sect. 3). It can be called by TRY-NEXT or INVOKE (sect. II.2), or (indirectly) by ADD and REMOVE. User-defined method types can be used in any way the user wishes. (Sect. VII.2.2.F)

If a method has a non-NIL atomic name, it is a special case of a <u>named</u> <u>datum</u> (sect. 4), and the name is uniformly used by the system and user to refer to it.

III.1.2.2 The Index

There are two senses in which an indexable datum may be "there." One is the same as for object data: "there" means "present in the current context." This is implemented by the set of markers on the "tail" of every datum. The other sense is "pointed to by the index," that is, findable by asking the index maintainer for potential matches to a pattern. Indexable data will be indexed in this fashion only when they are present or have a property (sect. 2) in at least one context. (The motivation for this is that one version of ((A B C)), a propertyless item datum, is as good as any other version of ((A B C)), so there is no point in making all of them unique.)

From the user's point of view, the index is just a list of all propertied data. When a datum acquires presence or some other property, it is added to the list. When all its properties are flushed or the contexts it has properties in are garbage-collected (see sect. 3), it is deleted from it.

When the data base is initialized, each index (one for items and each method type) is indeed just a list of its contents. However, when this list exceeds a certain size, it is broken down into subindexes, according to the contents of the CAR's and CDR's of its elements. Each subindex specifies an index for each different atom that appears in a slot. These indexes are themselves indexified if they become too large.

Now when a pattern is given to the data base, it can isolate a small subset of candidates for matching before running the relatively costly matcher. It does this by looking for a bucket, in the index, corresponding to each constant position of the pattern, and taking the smallest bucket it finds. For methods, in each position, it must take into account the bucket for methods with a variable in that position.

Thus, for example, the function FETCHI, which fetches items, works as follows:

a. Search the item index for candidates.

b. Throw away all those absent from the current context.

c. Match the others and make up a possibilities list.

The indexer's efficiency depends on two numbers, ☆ATOMINDEXTHRESHOLD and ☆STRUCTINDEXTHRESHOLD, which determine at what size buckets of the two types (which we really didn't describe) are broken down. No one knows what the "best" values are for the numbers, or what they depend on. If you want to worry about this, talk to DVM.

The indexes point at item data and therefore keep them from being garbage-collected. This is, of course, essential, since as long as the contexts that contain them are around, such data might have to be accessed by pattern. However, this feature can lead to difficulty. If an item datum has a property that includes a pointer to a context in which that property is accessible, the resulting circular structure (context -> datum -> property -> context; see sect. 3) will be uncollectable. So some interesting properties (like "frame in which this item was added," if it points even indirectly to a binding of CONTEXT) are unfeasible.

The biggest bummer is that method closures almost always point to frames with a binding of context that includes the closure. Thus the presence of many closures will choke up free storage irrevocably. You might have to write your own special garbage collector to delete these data by hand.

III.2 Properties of Data

It is always possible to given any datum a name (sect. 4) and assert items about it, in order to record its context-dependent properties. However, for reasons of efficiency (and because items are not always the best means of representing everything), it is also possible to associate <u>indicators</u> with <u>properties</u> on a datum, and retrieve them in a context-dependent fashion.

To associate indicator with property in context, use

```
(DPUT datum property indicator context).
```

This causes datum to have the pair (indicator property ...) on its tail, where "..." is garbage described in sect. 3. This pair is returned. This property pair will be associated with the datum in all subcontexts of context unless it is removed or overridden in one of them.

```
(DGET datum indicator context)
```

returns the first indicator-property pair found on the datum by searching the context, its super-context, etc., or NIL if there isn't one. Finally,

```
        (DREM datum indicator context)
```

does a DGET, but makes all pairs found with that indicator invisible in context and all its sub-contexts, so that future DGET's in them will return NIL (See sect. 3).

As an example of what can be done with property functions, consider 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

```
(CDEFUN 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 FORCEWIN could be written

```
(CDEFUN FORCEWIN (PLAYER SQUARE)
                 "AUX" ((CONTEXT (PUSH-CONTEXT)))
   (DPUT (REALIZE (SQUARES SQUARE)) PLAYER 'OCCUPANT)
   (MAKEMOVE (OTHER PLAYER))
   (TRY-NEXT (GENERATE (WINMOVES PLAYER)) NIL)   )
```

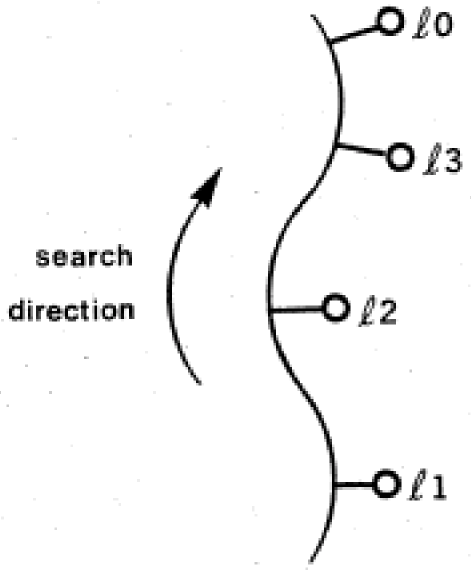
If the semantics of property-list manipulators does not quite fit your needs, there are more primitive functions, described in Sect. VII.2.4, which enable you to tailor-make your own versions of them.

III.3 Implementation of Contexts

A context is profitably considered an abstract object whose only interesting characteristics are how it got started and what has been done to it and its superiors since. However, for some purposes it may be useful to know that a context is implemented as a list of _context layers_, each of which describes the differences between a context containing that layer and one not containing it. Actually, it is of the form (*CONTEXT *layers*) for debugging purposes.

III.3.1 Presence and Absence

To be precise, a layer's functions are to indicate which data are to be thought of as _realized_ in contexts containing that layer, and which such mentions by other layers are to be _cancelled_ in contexts containing it. The first function is easy to grasp: every datum is present in a context if and only if a search up the context from most recently pushed to most global finds a layer that mentions that datum. (Fig. 1.)



CONTEXT = (★CONTEXT ℓ1 ℓ2 ℓ3 ℓ0)

FIGURE 1. Seach Rules.

However, if that mention has been cancelled when the datum was "unrealized" in some subcontext, it does not count.

To make this clear. imagine the following context structure:

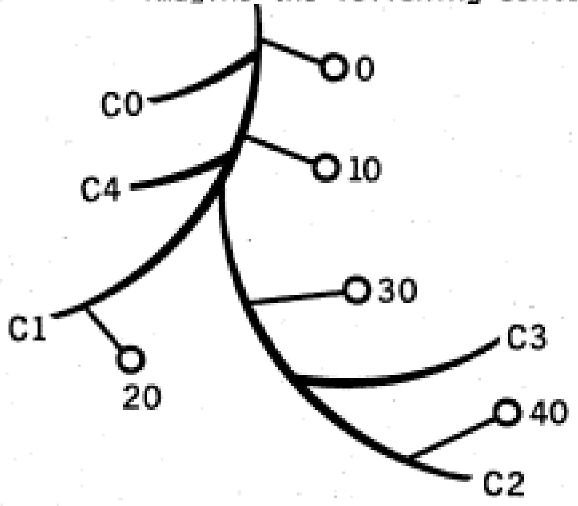
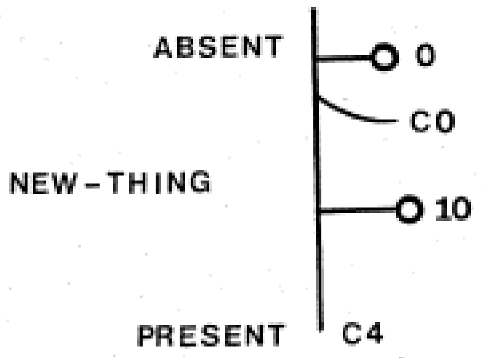


FIGURE 2

The numbers are context-layer numbers. Four contexts, c1 (20, 10, 0), c2 (40, 30, 10, 0), c3 (30, 10, 0), and c4 (10, 0), are identified (besides the global context c0 (= (0))). C4 is super-context of all but c0.

Now imagine that the structure of Fig. 2 is begun by a process that sprouts a sub-context from the global context cØ, and hypothesizes that the object datum NEW-THING (cf. sect. 1.1) is really "there," in a visual scene. (Fig. 3)



```
(PROG "AUX"
  ((CONTEXT
    (CSETQ C4 (PUSH-CONTEXT))))
       :
  (REALIZE NEW-THING)
       :
       )
```

FIGURE 3

This causes NEW-THING to sprout a tail of context-markers (c-markers), so that it looks like (#OBJECT (R4 R5 R9) (1Ø (Ø . +))), and is present in all sub-contexts of c4, actual and potential (The format of c-markers is described in detail in Sect. VII.2).

Now the process creates two sub-processes, each with its own context, in this case, c1 and c3:

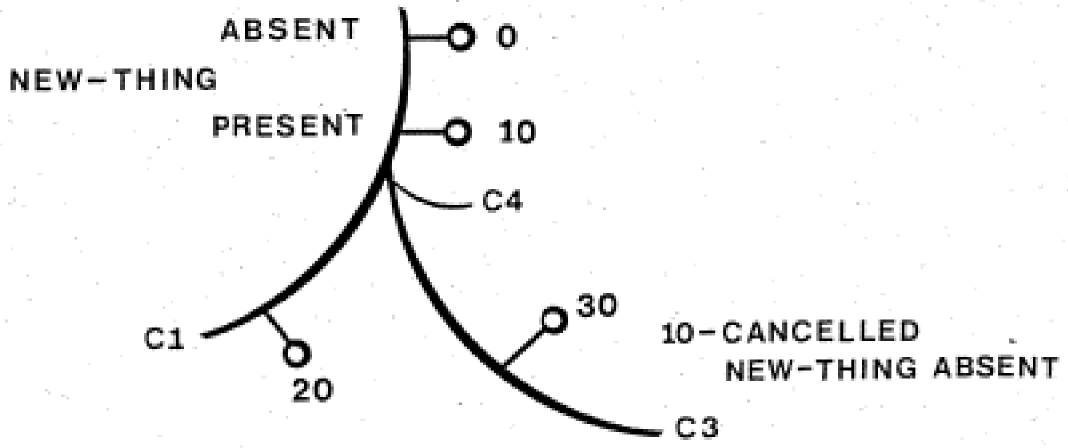


FIGURE 4

Each process investigates the interaction of NEW-THING with previous parts of the scene. The investigation in c3 leads the machine to doubt that NEW-THING is there, so it executes (UNREALIZE NEW-THING C3), making it absent there by "hiding" the fact that c4 mentions the object. The object remains absent in any sub-contexts of c3 generated by further pushing. (Fig. 5)

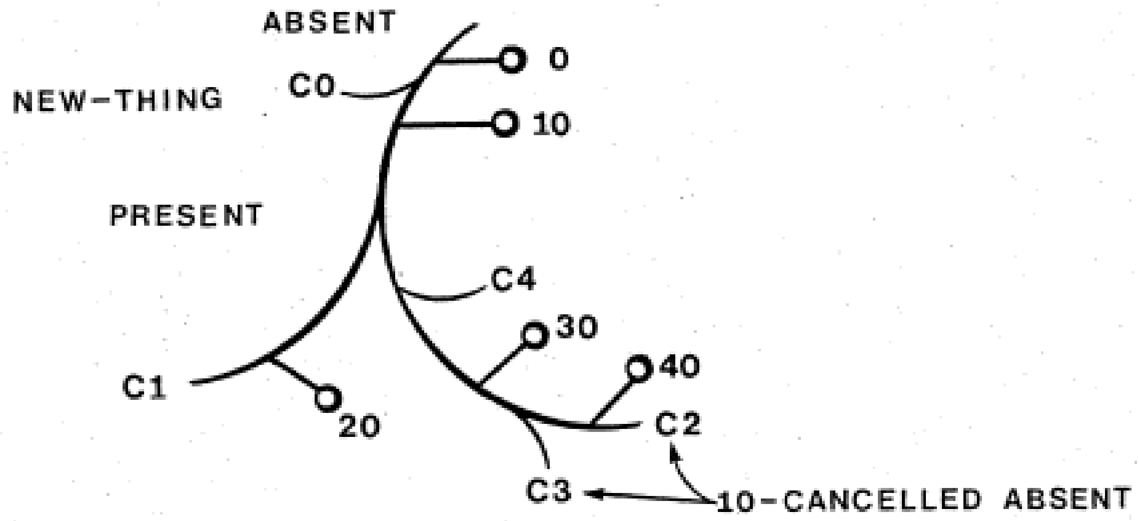


FIGURE 5

Now NEW-THING = (*OBJECT (R4 R5 R9) (30 (1)) (10 (0 30))), indicating "present in all contexts with layer 10 except those with layer 30 as well."

Meanwhile, the process running in c1 still believes NEW-THING is there. Imagine that it discovers it to be the only possible supporter of another object which is known to exist, and hence that NEW-THING is certain to exist also. It executes (REALIZE NEW-THING C0), which makes it present in all the contexts:

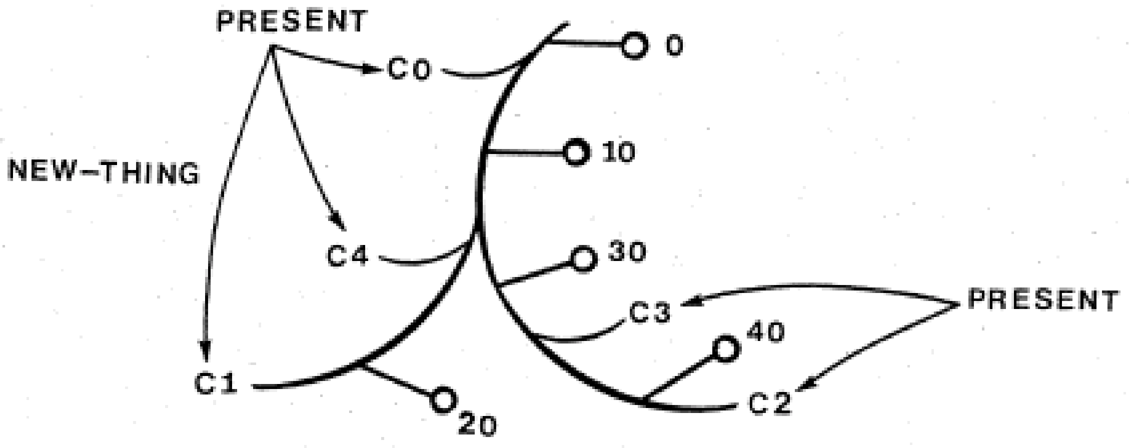


FIGURE 6

Now NEW-THING looks like (☆OBJECT (R4 R5 R9) (30 (1)) (10 (0 30)) (0 (0 . +))); that is, a cancellation applies only to outstanding, not future, realizations of a datum. Later realizations will override it. (If (REALIZE NEW-THING C4) had been executed, the same effect would have occurred, except for NEW-THING's remaining absent in c0. Its tail would have consisted of (10 (0 . +)) again.)

Not all UNREALIZations cause cancellation. If NEW-THING were UNREALIZED in c4 at the second step, its tail would be emptied rather than be made to contain, say, the c-marker (10 (1 10)). That is, c-markers never cancel themselves.

In any given context, the predicates REAL and UNREAL can be used to determine if a datum is present or absent. REAL returns its argument if there are any outstanding (uncancelled) mentions of it in the current context branch, or NIL otherwise; UNREAL, the opposite.

All of these primitives operate by altering or examining the tails of their arguments, which consist of c-markers of the form

```
        (lnum (refco . status) *property-pairs*)
```

where lnum identifies a context layer, status is +, NIL, or a list of canceling lnums, and property-pairs are explained below (cf. sect. VII.2).

III.3.2 Implementation of Properties

The association of properties with data is in some sense a generalization of the notions of presence and absence. "Presence" can be considered an indicator whose associated property is ignored. It is either there or not there. Properties, on the other hand, have distinguishable values, so they may be overridden as well as cancelled.

Properties are implemented as cancellable "pairs" which are associated with c-markers. If a c-marker on a datum looks like:

```
        (n (...)...(ind prop . status)...),
```

it means that datum has the ind-prop association in contexts with layer n, except those which cancel it, as indicated by status, which is "+" if there are no cancellations. (Details are given in sect. VII.2.)

III.3.3 Context Layers

A context layer is implemented as a list (*LAYER lnum *data*), where lnum is its unique layer number, and data are all data with at least one occurrence of lnum in their tails. The reason the layer must point to each such datum is that when it is garbage-collected, the magic Conniver garbage collector must be able to remove all these occurrences. (Unfortunately, this means contexts point at the data they contain, so the loops mentioned in Sect. 1.2.2 are possible.)

Anyway, it is possible to loop through the layers of a context and the data in the layers, and thus apply some function to, say, all the data in a context.

III.3.4 Nonstandard Contexts

Most contexts are generated by PUSHing and POPping -CONTEXT. These processes cause the generation and discarding of context layers. Since individual layers are sometimes important, it is desirable to be able to string them together into new contexts. Then a layer which has a c-marker of non-NIL status on a datum will make that datum present in any context in which it appears. A layer whose number appears as a canceller of c-markers or properties can be thrown in to make certain data or their properties go away.

The only restriction on generating new contexts is that the layers must appear in order of decreasing lnums. This is because operations like testing the visibility of c-markers and property pairs involve taking the intersection of two set of lnums (the status and the context layers), which is done faster with ordered sets.

The functions which create new contexts are described in Sect. VII.2.5.

III.4 Named Data

Occasionally it is convenient to refer to an arbitrary datum by an atom rather than a pointer to the datum itself. Sometimes this is a matter of convenience. Sometimes, as with functions, the datum must mention itself without being circular. In the case of methods, as mentioned before (sect. II.2), one needs to be able to refer to them by an EQ name when redefining them in order to avoid having two around with the same pattern; an atom fills this need. Item data are also occasionally to be thought of as EQ things: an item that refers to another item wants to mean that particular thing, not a data structure EQUAL to it.

Of course, the user may use any *ad hoc* scheme he wishes to associate an atom with a datum. If he knows atoms might have data under the property FOO, the item (POSSIBLE FRAB) might mean "the item under indicator FOO on FRAB is possible."

Another possibility is to create a more intimate association between an atom and a datum, in which the system considers the atom to stand for that datum in every situation, much as an atom with an EXPR property always stands for the associated function, in forms, as an argument to APPLY, etc. in a Lisp program. This association is needed for named methods, and has been extended for use with all types of data. Any datum may be given a name with the function NAME-DATUM. If it already has a name, the name will be changed. This function alters every old system-maintained copy of the datum to be the new name, even down to its name slot if it is a named method. Thus the index, context layers, etc., will point to the atom instead of the datum directly.

The other ways to give a datum a name are with the method-defining functions, and the function DATUM with two arguments. All of these are documented in the appendix, sect. VII.2.2.

The association of an atom with an item datum is a bit imperfect, because of a problem with the identity of a propertyless datum. As mentioned in sect. III.1.2, an item datum with no properties is not pointed to by the index. DATUM of something like (A B C) is a brand-new ((A B C)) each time if the item has no properties or presence. This will be true even if there is an atomic name for a particular such propertyless datum.

If the thing did have properties, this atom would be what the system would return as the DATUM of its item, but if it is unindexed, there is no way to find the atom, and the system is fooled into returning a new non-atomic item datum each time DATUM is called. Thus, if item (A B C) corresponds to ((A B C) (10 (0) (PLAUS 12 . +))), and you execute (DATUM '(A B C) 'FOO), then FOO will become associated with that datum, and (DATUM '(A B C)) will thenceforth return FOO.

However, if the item datum is just ((A B C)), (DATUM '(A B C)) will be ((A B C)) no matter how many times you name it. So be careful on the timing of naming data.

IV ON PATTERN-DIRECTED INVOCATION

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, a match being an assignment of values to the pattern's variables that will make it EQUAL to the item. Since when if-needed methods are called, it is necessary to match two patterns against each other, the matcher always returns a list of two alists that specify assignments of as many variables on both sides as possible. If there is no match, NIL is returned.

The matcher may be called by (MATCH varpat datapat); MATCH is asymmetric in that it is biased toward assigning variables in varpat to constants from the other side. A pattern is a non-circular list structure with "variable parts" marked by the prefixes "!>" and "!,". "!>var" must match a variable-free section of datapat. (This restriction will always be met when patterns are matched against items.) Matching "!>var" against something causes var to be bound on the alist for variables in varpat, with a value corresponding to what is matched. For example, (MATCH '(FOO !>X) '(FOO BAR)) returns (((X BAR)) NIL). (Here, "NIL" is the alist for variables in (FOO BAR).)

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. (The second lists are always NIL in these cases.)

When FETCH calls the matcher, it uses the varpat alist to make up an item possibility. Thus, if items (SPIRO LIKES ROCKS) and (SPIRO LIKES DICK) are present in the current context, (FETCH '(SPIRO LIKES !>WHAT)) might return

```
((☆POSSIBILITIES (SPIRO LIKES !>WHAT)) ☆IGNORE
   (☆ITEM ((SPIRO LIKES ROCKS) (10 (0 . +))) ((WHAT ROCKS)))
   (☆ITEM ((SPIRO LIKES DICK) (0 (0 . +))) ((WHAT DICK)))).
```

TRY-NEXT takes the association lists from item possibilities and assigns the variables as they direct.

The other principal prefix is "!,", which refers to the current binding of its variable in the match so far, or the current Conniver binding, and matches its value. For example, the pattern (GRANDFATHER !>X !,X) matches all items corresponding to people who are their own grandfathers.

Another frill is the ability to specify a <u>restricted match</u>. If "!>" prefixes a non-atomic expression, its CDR is a list of forms that must all evaluate (in Lisp) to non-NIL after assignment of its CAR. For example, !>(X (ATOM !,X)) matches only atoms, and !>(CREATURE (FEATHERLESS !,CREATURE) (EQ (NUMBER-OF-LEGS !,CREATURE) 2)) matches featherless bipeds. If it is desired to bind and initialize a variable in its pattern's alist, one can write "!,(var initial-value)." For example, (FUNCT-OF !>FORM !,(F (CAR !,FORM))) matches (FUNCT-OF (FACTORIAL 5) FACTORIAL) but not (FUNCT-OF (PLUS 2 2) MINUS). Finally, if it is desired to specify an item shape without naming or saving its parts, the prefix characters "!>" can stand alone. Thus, (FETCH '(FOO !>)) returns a possibilities list of items of the form (FOO x), but applying TRY-NEXT to it sets no variables.

If-addeds and if-removeds work nicely with MATCH. To invoke one, Conniver applies MATCH to its pattern and the item that triggered it. If the result is non-NIL, the varpat alist is used to start the variable bindings in the method's frame (which may be augmented by "AUX" bindings).

An if-needed method is really an entirely different kind of entity. First, its match occurs at FETCH-time, its alists being saved in a ☆METHOD possibility until TRY-NEXT calls it. Second, such a method is a kind of callable subroutine, which should be capable of more than verifying or achieving conditions represented by constant patterns. In particular, one would like to be able to specify that a slot represents an <u>output</u> <u>variable</u>, to be set by the method but <u>not</u> by the match. This is accomplished by use of the prefix "!<"; "!<var" matches only

expressions with variables (var being assigned to that expression, but only so the user can see what is going to happen on output).

Thus, when (NOTE (INSTANCE)) is called to create a value to be added to the possibilities list, MATCH is called again in a special (secret) way, this time with the FETCH-pattern as varpat, which treats the method pattern as an essentially <u>constant</u> structure whose variable slots are to be filled as indicated by the values the variables have picked up in the method.

As an example, consider the method

```
(IF-NEEDED FIND-SUPPORTERS
        (ON !>X !<Y)
       "AUX" ((P (FETCH '(SUPPORTS !>Y !,X))))
:LOOP
       (TRY-NEXT P '(ADIEU))
       (AU-REVOIR (INSTANCE))
       (GO 'LOOP)    ).
```

This method will appear in a FETCH-possibilities-list generated by, e.g., (FETCH '(ON BOX1 !>B)), but not one generated by, e.g., (FETCH '(ON !>A TABLE)), which is looking for "supportees" of an object called TABLE. When FIND-SUPPORTERS is entered, X will have the value BOX1, and Y the value !>B. At statement :LOOP, Y is set to the next supporter of BOX1. One such supporter is returned each time the method is entered or re-entered.

Notice that this method has a completely different purpose from one with the pattern (ON !<X !>Y), which would find the "supportees" of a given object Y; or one with pattern (ON !>X

!>Y), which verifies a support relation.

Occasionally, however, one wishes the decision of what to do in cases differing in this way to be made after the method is entered. In this case, one can use the prefix "!?" which is ambiguous; it matches anything, but its variable is assigned if and only if it matches a variable-free expression. The complementary ambiguity on the FETCH-side is handled by the prefix "!;" which means "!>" if its variable is unbound in the current match alist and unassigned by Conniver; or "!," if its variable is assigned or bound in the match alist but unassigned. These characters are typically handy in situations where a pattern is to be expanded according to its definition, regardless of exactly what is variable in it. For example, if men are always to be treated as featherless bipeds, the following method does the conversion if one is looking for men or attempting to verify that something is a man:

```
(IF-NEEDED IS-MAN
       (IS !?X MAN)
   "AUX" ((P (FETCH '(IS !;X BIPED))))
:LOOP
      (TRY-NEXT P '(ADIEU))
      (COND ((PRESENT '(FEATHERLESS !,X))
             (AU-REVOIR (INSTANCE)))   )
      (GO 'LOOP)   ).
```

It takes the place of the two methods that would be required without "!?" and "!;", which would have "!<" and "!>", or "!>" and "!," instead. (Micro-Planner users please note that the micro-Planner prefix "$?" includes both ambiguities, and would take the place of all prefixes used in IS-MAN.)

Finally, some if-needed methods claim to be able to expand on the <u>syntactic forms</u> of calling patterns rather than to be able to generate items similar to those represented by those patterns. The corresponding method-pattern variables are signaled by the prefix "!'", which is analogous to the "'" of CEXPR bound-variable declarations. "!'var" binds var to an expression without examining its internal structure in any way; its variables are neither substituted or bound. For example, a method for causing (FETCH '(AND *conjuncts*)) to set the variables in the conjuncts correctly might look like:

```
(IF-NEEDED AND-EXPANDER
        (AND . !'CONJUNCTS)
       (COND (CONJUNCTS
                  "AUX" ((P1 (FETCH (CAR CONJUNCTS)))
                         (P2 (FETCH !"(AND . @(CDR CONJUNCTS))))
                         COP2)
               :LOOP1
                  (TRY-NEXT P1 '(ADIEU))
                  (CSETQ COP2 (COPY P2))
               :LOOP2
                  (TRY-NEXT COP2 '(GO 'LOOP1))
                  (AU-REVOIR (INSTANCE))
                  (GO 'LOOP2))
              ((AU-REVOIR (INSTANCE)))   )).
```

For example, if the items (GREEN BOX3), (GREEN BOX1), (GREEN BOX4), (ON BOX1 BOX2), and (ON BOX4 BOX5) are in the data base, repeatedly TRY-NEXTing (FETCH '(AND (GREEN !>X) (ON !,X !>Y))) will set X to BOX1 and Y to BOX2, then X to BOX4 and Y to BOX5, then quit. The method's value is always (*NOTE NIL), because it never concerns itself with binding calling pattern variables. (A more efficient implementation, by the way, is obviously possible.)

All the syntactic frills such as restrictions and bound variable initialization are legal in method patterns. However, it is generally meaningless to restrict a "!<"-marked variable. If !<(X (ATOM !,X)) appears in a pattern, it is not clear what is being restricted. It is certainly _not_ possible by such a restriction to prevent _future_ assignments of X to anything non-atomic. All restrictions apply only at match time. In the case of "!?", restrictions will be run only if the prefixed variable is assigned in the match.

V 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-CPRINT 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, formatting the output so that variables and special data types print in a lucid fashion.

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 declarations 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.)

When a Lisp or Conniver error occurs, the system initially causes a Lisp READ-EVAL-CPRINT loop to be created as close to the error as possible. (Such a loop is created by the function CERR.) Within this loop, only Lisp evaluations can take place. If it is desired to continue from the CERR, type (RETURN value-desired). Altmode-P is equivalent to (RETURN NIL). Users' instances of CERR (and CBREAK, which prints a less alarming message) may do anything they wish with a returned value. The system usually ends an error message with "// <something> <- ?" This means that (RETURN value) will cause that something to be value. For example, if an attempt is made to evaluate X when it is unassigned, the message

```
UNASSIGNED VARIABLE X // VAL <- ?
```

is printed. If the user types (RETURN 5), X will get value 5, and the evaluation will proceed from the CERR. If there is no obvious thing to be returned, the system will type "// GO ON?" Any value returned will be ignored, but if the user wishes to do his own patching and proceed, he may. Finally, if there is nothing to be done, an attempt to proceed will land him in the nearest EAR loop.

Within such a context (or any piece of Lisp), if it is desired to return to the closest Conniver frame and create a listen loop, evaluate (EAR). This creates a Conniver EAR-n loop, whose creation is signalled by the printing of

```
EAR-n

_
```

which behaves just like the top-level one. A variant function is NEAR, which returns to the nearest already-existing Conniver listen loop. Finally, the function TOP flushes the entire existing control structure, resets the EAR-counter to 1, and starts up a new EAR-1. (GO EAR-1) is similar, but can only be executed from Conniver.

The function BACKTRACE can be used to get a lucid summary in a CERR- or EAR-loop of the <u>control</u> pointer chain from the current Conniver frame upwards. Variable values can be inspected, functions can be called, etc. The functions UP, DOWN, and J can be used to move an EAR-loop around in the control structure.

This is handy for "editing" the stack, checking out variable bindings in the top-most activation of a recursive structure, etc. (See sect. VII.3.1.)

Since a tag is a sort of frame for most purposes, a Conniver listen loop can be flushed by EXITing EAR-n. The function (DISMISS frame) has been provided to exit from it with no particular value. DISMISS takes (FRAME) as its default argument. Within a LISTEN loop, $P is equivalent to (DISMISS).

To stop a Conniver program externally, use control-H (~H). This will generate a READ-EVAL-PRINT loop (as it would in Lisp), which can be exited with $P. From this loop, (EAR) will get you to Conniver, which remembers the Lisp expression it was working on. DISMISSing this EAR-n will cause the Lisp evaluation to be re-attempted (not resumed).

Another way to interrupt a Conniver is with ~A, whose action depends on the next character read. In general, this character must have a ^A ("uparrow A") property on its property list; this property should be a function of one argument (the character), which is called when "~A that-character" is typed at Conniver. If there is no such function property, a question mark is printed as ~A's sole function.

There are several built-in system functions that are handled by this mechanism. ~AE causes (EAR) to be executed at the next interruptible place in Conniver; ~AN, (NEAR). ~AT causes (TOP) to be executed immediately. A Lisp READ-EVAL-PRINT loop may be caused with ^AL; this is equivalent to a CERR loop. Two others do not cause listen loops: ^AF flushes the current input buffer if control is already in a READ, and is thus equivalent to a million rubouts; ^AX prints the current expression being CEVALuated inside Conniver, and continues. ^AX is a way of checking up on what the evaluator is doing without stopping it.

The Conniver error system operates with the more general Conniver interrupt system. The Lisp variable CINTERRUPT contains a list of expressions to be CEVALuated at the next opportunity. The function (CINTERRUPT exp) adds exp to the end of the list. It will cause it to be evaluated the next time control returns to Conniver. (Besides the error system, if-added and if-removed methods also cause interrupts to happen.) The function (ALLOW T-or-NIL) enables or disables all interrupts. If interrupts are disabled, CINTERRUPT queues them until (ALLOW T) is executed.

In many places in Conniver ^G or ^X will cause Lisp to go to a very-top-level READ-EVAL-PRINT loop; in such situations they are equivalent to STOP (see below). However, just as Lisp must protect itself from such things during garbage collection, Conniver disables all such Lisp interrupts when its data base is in a momentarily inconsistent state--i.e., when it is making changes to it. At such times, there is no way to stop the thing, so don't give it, e.g., a circular list as a context.

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. To restart in _exactly the state_ before (STOP), call (RUN); you're back in Conniver. (RUN and STOP have more sophisticated uses; see the appendix.)

VI LISP AND CONNIVER

Lisp functions do not usually call Conniver CEXPR'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 CEVAL, below.) Conniver can call any Lisp function, though, and Lisp EXPR's, LEXPR'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 (/, var), abbreviated ",var". For example, Lisp functions should refer to CONTEXT as ,CONTEXT.

EAR, NEAR, STOP, and other Conniver system functions use labeled CATCHes and THROWs to do non-local control jumps. The way Lisp currently works, there is no way to prevent unwanted interaction with users' unlabeled CATCHes should they be in the way. For example, (CATCH (NEAR)) will return something meaningless rather than go to the nearest ear.

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 (/, SQUARE1) (/, 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 or CINT's. 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.

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

EXPR CEVAL.

If control re-enters the Conniver structure, the new Lisp stack-state above it will have nothing to do with the original, none of the old, now unbound, Lisp variables will be 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 causes the entire Conniver to return to Lisp (thinking it is returning to the Lisp frame that started the CEVAL), in such a way that RUNning or STARTing is impossible.

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 doesn't 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.

VII APPENDIX: THE CONNIVER REFERENCE SOURCE

Whereas the previous sections of this manual are 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.

This appendix has three sections, one describing the evaluator, one for the data base functions, and one for various debugging aids. 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.

VII.1 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
                       ^ !"S-expression ^ (function ☆arguments☆)
```

where the arguments are themselves Conniver expressions (and zero is a possible number of them).

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. This value will be determined by its value from some local binding or a hidden list of global values.

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. An expression following !" is called a <u>skeleton</u>, which is <u>expanded</u> as follows: atoms expand to themselves; ",atom" expands to the Conniver value of atom; "@expression" expands to the Lisp value of expression; "(!@expression . rest)" expands to (APPEND

Lisp-value-of-expression expansion-of-rest); "(exp1 . rest)" expands to (CONS expansion-of-exp1 expansion-of-rest). For example, if X= (A B C), !"(,X D @(CDR ,X) (!@(CDDR ,X) . ,X)) = ((A B C) D (B C) (C A B C)).

5. Functional applications are processed as follows:

If the function is atomic, it is checked for CINT, CEXPR, 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. In this case, Conniver does not define a new frame for its evaluation.

In all other cases, a new frame is created, with appropriate access and control links.

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

If the function is a CEXPR, the arguments are paired with the formal parameters of the function (and perhaps evaluated) as specified by the declaration in the function (see CDEFUN 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, PROG or METHOD proceeds as follows:

If it begins with the reserved word "AUX", then the auxiliary variables that follow it are bound. (See below, Sect. VII.1.2.)

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.

VII.1.1 Communication with Lisp

```
A. (RUN [stuff NIL])                                LSUBR
   (STOP [stuff NIL])                               LSUBR
```

When a Conniver program is running, its control structure is "serviced" by a set of Lisp programs which use the Lisp stack. Control repeatedly returns to one called RUN1, which is the "inner loop" of the system. Beneath the frame of RUN1 on the stack is its caller, which is expecting a value to be returned. The function STOP can be used to return such a value.

When it does, the state of the Conniver computation is not disturbed, because it must all be saved in various frames anyway. STOP leaves everything in such a state that (RUN x) will cause the Conniver to start again, as though STOP had returned x. RUN does this by calling RUN1. Hence, a later (STOP y) will make RUN1 return to RUN with value y. RUN returns this value.

Hence, 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--RUN1.
```

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 _not_ get the program back to the execution point of that STOP, because after leaving the CEVAL, Conniver is already running. So, be careful with using STOP to return a value for a CEVAL.

If, for some reason, the Conniver interpreter (not the data-base -- see DATA-INIT) needs to be re-initialized, it can be done 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. Global Conniver variable bindings and their values are not changed. The data base is not disturbed, but all contexts previously saved only as values of Conniver variables will be lost to garbage collection.

When in Conniver IBASE=BASE=10. and all character macros are in effect; these return to their Lisp defaults when returning via STOP.

VII.1.2 Procedure definition

All of the functions below define procedures which include a slot called the "body." The body of a procedure is evaluated 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. This expression must not evaluate to a tag or frame for the current activation of the procedure in which the "AUX" appears. To initialize a variable to a tag, you must allow it to be bound to ☆UNASSIGNED, then CSETQ it to the tag value desired.

A. (CDEFUN 'atom 'declaration '☆body☆)   FSUBR

This function is used to define the atom to be a function with the formal parameters specified by the declaration and with the body given. The definition will be placed on the atom's property list under the indicator CEXPR. 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 = (<u>obligatory variables</u> <u>optional variables</u> <u>excess</u>)

<u>obligatory variables</u> = <u>empty</u> ^ <u>par1</u> ... <u>parN</u>

<u>par1</u> = atom ^ 'atom

<u>optional variables</u> = <u>empty</u> ^ "OPTIONAL" <u>op1</u> ... <u>opN</u>

<u>op1</u> = atom ^ 'atom ^ (atom default) ^ ('atom default)

<u>excess</u> = <u>empty</u> ^ "REST" atom ^ "REST" 'atom

The semantics is as follows:

1) Formal paramaters 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 prints either

```
        TOO FEW ARGUMENTS--VARS = remaining vars // ARGS <- ?
```

and wants arguments for the leftover variables, or

```
        TOO MANY ARGUMENTS--ARGS = remaining args // VARS <- ?
```

and wants variables to bind the leftover arguments to. If the syntax of a declaration is not as specified above the error comment:

```
     BAD DECLARATION--VARS = rotten variables // VARS <- ?
```

will be generated, and anything RETURNed will replace the rotten variables.

To create a method we use one of the constructors:

```
B. (IF-ADDED ['atom] 'pattern '☆body☆)           FSUBR
   (IF-REMOVED ['atom] 'pattern '☆body☆)         FSUBR
   (IF-NEEDED ['atom] 'pattern '☆body☆)          FSUBR
```

The given atom is defined or redefined 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 by pattern, a method must be put into the data base via INSERT or ADD. Once a named method has been added to some contexts, redefinition of the same name will cause it to remain present in the same contexts.

The pattern is the analogue of the variable declarations of a CEXPR; in particular, the appearance of any type of match variable (except "!,var") signals that variable is to be bound when the method is invoked. The pattern is used as described in Chapter IV.

VII.1.3 Sequence Evaluators

```
A. (PROG ["AUX" 'aux-variable-declarations] '☆body☆)
                                                  CINT
   (PROGBIND aux-variable-declaration '☆body☆)    CINT
```

The value of a Conniver PROG is the value of the last expression in its body. The expressions in the body are evaluated in order after any "AUX" variables are bound. These variables are subject to the same format and restrictions as those for CEXPR's and methods. The sequence of evaluation may be altered by use of GO (Sect. VII.1.5).

PROGBIND is like PROG except that it evaluates its first argument to give a list of auxiliary variables. For example, if X=A,

```
    (PROGBIND (LIST (LIST X 5)) (PRINT A))
```

binds A to 5 and prints "5".

```
B. (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)
                     (CSETQ P (ACTBLOCK))
           :LOOP (COND ((= (CSETQ M (1- M)) 0)
                         (EXIT 3 P)))
                 (GO 'LOOP))
         (T 2))
```

C. (AND '☆body☆) CINT
(OR '☆body☆) CINT

These are exactly equivalent to their Lisp counterparts. AND returns the value of the last element of its body or NIL if one of the elements evaluates to NIL. (AND) = T. OR returns the value of the first non-NIL element of its body, or NIL if all of its elements evaluate to NIL. (OR) = NIL.

VII.1.4 Frame Creators and Manipulators

Conniver keeps track of what it is doing by maintaining a structure called a <u>fr</u> for each invocation of all kinds of functions except Lisp FEXPR's or FSUBR's. A fr is basically a structure with five slots: IVARS, BVARS, FORM, ALINK, and CLINK. IVARS are the internals of the interpreter; BVARS are the variable locatives for the variables bound in the frame; FORM is the expression whose evaluation gave rise to this frame; and ALINK and CLINK are the access and control fr's where free variables will be looked up and where control will return, respectively.

These objects are not explicitly touchable by the user, but are parts of frames, tags, and closures, the data types returned or manipulated by the functions of this section.

```
A. (FRAME)                                          SUBR
   (ACCESS [frame (FRAME)])                         LSUBR
   (CONTROL [frame (FRAME)])                        LSUBR
   (EXPRESSION frame)                               SUBR
```

FRAME returns the frame with respect to which it was evaluated. This means the nearest enclosing frame in which variables are bound. This means that in most reasonable places in a PROG or CEXPR, (FRAME) will be the frame of that PROG or CEXPR's activation. This means that constructions like (ACCESS (ACCESS (FRAME))) will have the correct meaning, no matter how deeply nested they are.

ACCESS returns the access frame of its argument.

CONTROL returns the control frame of its argument.

EXPRESSION returns the expression whose evaluation created the frame supplied. It is useful for hunting around in the frame structure.

The argument to ACCESS, CONTROL, or EXPRESSION must be a legitimate frame. If it is not we get the error message:

```
 BAD FRAME SUPPLIED // FRAME <- ?
```

By "legitimate frame" is meant anything that contains a pointer to an internal fr. This includes all the data types of this section, frames, tags, and closures. In what follows, "frame" is used ambiguously to refer to any of these or "*FRAMEs" in particular. The ambiguity is harmless because the system never cares which you mean.

```
B. (TAG [atom])                                LSUBR
   (ACTBLOCK)                                  SUBR
```

TAG searches the control link chain from (FRAME) 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.

TAG of no arguments is equivalent to ACTBLOCK, which 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.

C. (VFRAME atom [frame (FRAME)])   LSUBR

VFRAME searches up the access link chain from frame until it finds a frame in which atom is bound as a Conniver variable. It returns that frame.

D. (CLOSURE procedure [frame (FRAME)])   LSUBR

CLOSURE produces the lambda-closure of the procedure (function, method) indicated. This is an object of the form (☆CLOSURE procedure fr). Later invocation of the closure (see CALL) causes the environment of the procedure (its access pointer, where it searches for bindings of free variables, tags, etc.) to be frame E.g. If X = 4 then:

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

has the value 8 but

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

has the value 9.

This is the classical "FUNARG" device.

E. (SETACCESS frame1 frame2)   SUBR
   (SETCONTROL frame1 frame2)   SUBR

These functions are used to alter the ALINK and CLINK, respectively, of frame1 to be frame2. These will alter the locatives of free variables referred to in frame1, and the frame to which it returns. Each function returns its second argument.

F. (SAMEFRAME frame1 frame2) SUBR

Functions like FRAME cons a new list structure each time they are called, so EQ will not work as an identity test on frames. (EQUAL will not work because frames may be circular.) SAMEFRAME should be used. It returns non-NIL if and only if frame1 and frame2 actually refer to the same fr.

VII.1.5 Alteration of Flow of Control

```
A. (CONTINUE frame)                                CINT
   (GO atom-or-tag)                                CINT
```

These two functions cause a given fr to become the current process description; that is, they cause control to resume in a previously constructed frame. GO is a special case of CONTINUE which takes only a tag argument; (GO tag) is equivalent to (CONTINUE tag), but GO has other uses. GO always evaluates its argument, avoiding the ambiguity of Lisp. If its argument is an atom, (GO arg) is equivalent to (GO (TAG arg)); that is, it searches up the control link chain from (FRAME) for a statement label ":arg." Execution then proceeds from the statement label found. If the argument is of the wrong type or an atomic tag cannot be found we get:

```
        FOLLOWING NOT SEEN AS TAG--argument--GO // ARG <- ?
```

CONTINUE can cause the error

```
        BAD FRAME SUPPLIED // FRAME <- ?
```

```
B. (EXIT value [frame (ACTBLOCK)])                CINT
   (RETURN value)                                  CINT
   (DISMISS [frame first-non-COND-frame])          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:

```
        NO FRAME WITH BODY--EXIT // FRAME <- ?
```

or

```
        NO NON-COND FRAME WITH BODY--RETURN // FRAME <- ?
```

and the frame you RETURN will be EXITed with no further checking for bodies.

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

```
        BAD FRAME SUPPLIED // FRAME <- ?
```

```
C. (ADIEU pos1...posN)                              CEXPR
   (AU-REVOIR pos1...posN)                          CEXPR
```

These functions return to TRY-NEXT from a generator, NOTEing possibilities 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).

VII.1.6 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 SUPPLIED // 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 Chapt. VI 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.

C. (INVOKE method pattern)   CINT

INVOKE attempts to match the pattern of the given method against pattern. If the match fails, the value is NIL. Otherwise, the method-pattern alist generated is used to start the method's variable bindings, and its body is executed as a PROG, its last expression yielding its value (unless the flow of control is altered).

VII.1.7 Variable manipulators:

```
A. (VLOC atom [frame (FRAME)])                    LSUBR
   (RVALUE atom [frame (FRAME)])                  LSUBR
   (/, 'atom)                                     FSUBR
   (LVALUE 'atom)                                 FSUBR
   (ASSIGNED? atom)                               SUBR
```

VLOC returns a locative to the value of the atom supplied if it is found in some frame in the access link chain starting with 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 either VLOC or RVALUE are given an illegal frame, we get:

BAD FRAME SUPPLIED // FRAME <- ?

(/, 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. (CSET atom value [frame (FRAME)])              LSUBR
   (CSETQ 'atom1 value1 ... 'atomN valueN)        CINT,FSUBR
   (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 odd-numbered arguments (the atoms).

UNASSIGN sets its argument to ☆UNASSIGNED.

VII.1.8 Possibilities lists

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

```
possibilities = ((☆POSSIBILITIES thing)
                 last-possibility
                 pos1 ... posN)
thing = expression or pattern that created this list
posI =  (☆METHOD method methalist callalist pattern)^
        (☆GENERATOR form)^
        (☆AU-REVOIR fr)^
        (☆ITEM item-datum alist)^
        (☆NOTE alist) ^
        (user-defined-pos-type ...)^
        anything else
  last-possibility = ☆IGNORE ^ (☆BLOCK ☆processes☆)  ^
previous-pos1
```

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

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

CINT

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 method methalist callalist callpattern)

The method is invoked, with initial bindings given by methalist. (The two alists are usually from the MATCH that FETCH used to filter out useless methods.) It may generate new possibilities using ADIEU or AU-REVOIR. The new possibilities are then spliced into the current one, replacing the method possibility which generated them. TRY-NEXT then loops back to try the first possibility in the newly augmented possibilities list. The callpattern is used by INSTANCE inside the method for generating output alists.

Method possibilities are assumed to behave as a kind of generator, as just described. If they return a value (e.g., by running off the end of their bodies), the value is ignored.

2. (☆GENERATOR form)

Exactly the same as a method except that the form is evaluated rather than the method invoked. Within a generator, NOTE (see below) works, but INSTANCE does not.

3. (☆AU-REVOIR 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-datum 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-datum is returned as the value of TRY-NEXT.

5. (☆NOTE alist)

This type of possibility has the same side effect as a ☆ITEM possibility with the same alist, but returns the entire possibility instead of an item. These are produced by the function INSTANCE mentioned below, and are a method's way of simulating items.

6. (user-defined-pos-type...)

If a possibility is non-atomic, and begins with an atom with a ☆POSSIBILITY property, that property is assumed to be a function of one argument. TRY-NEXT calls that function with the possibility as argument, and returns whatever value the function produces. For example, to create possibilities of the form (☆ASSUMPTION item con), which return item and have the side effect of setting CONTEXT to con, define

```
(DEFUN ASPOS (POS)
   (CSETQ CONTEXT (CADDR POS))
   (CADR POS))

(DEFPROP ☆ASSUMPTION ASPOS ☆POSSIBILITY)
```

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

At any given time, the last-possibility slot of the possibilities list contains the last possibility that was returned. Initially, this is ☆IGNORE; when control is in the process of entering a method, it is (☆BLOCK ☆ready-processes☆), which structure is used to avoid certain timing errors. This elaborate machinery is present so that two not-necessarily-synchronized processes may suck possibilities out of the same list and be sure of getting exactly the same possibilities in exactly the same order. I (DVM) am not to blame for it.

Thus we see that TRY-NEXT does not stop churning back for more possibilities created by called generators until either the possibilities list is empty (i.e., ((☆POSSIBILITIES...) ☆IGNORE) or an item possibility or an "anything else" is first on the list. If TRY-NEXT is given a bad possibilities list we get.

BAD POSSIBILITIES LIST

B. (GENERATE 'form) CEXPR

This function takes one unevaluated argument, which it assumes is a generator form. It returns a possibilities list which starts with the first non-method or generator possibility returned by the form. Thus, (TRY-NEXT (GENERATE form)) is equivalent to (TRY-NEXT !"((☆POSSIBILITIES form) ☆IGNORE (☆GENERATOR form))), but (GENERATE form) is <u>not</u> equivalent to !"((☆POSSIBILITIES form) ☆IGNORE (☆GENERATOR form))), because the generator is actually called by GENERATE.

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

C. (INSTANCE) FSUBR

which returns the current instance, in the form (*NOTE alist), where alist is a pairing of the variables in the callalist of the current method with values obtained in a new match of the method alist. It will get upset if there are unassigned variables in the pattern and will ask you to assign them.

A generator or method may note a new possibility via

D. (NOTE [possibility (INSTANCE)] pos2...posN) LSUBR

This function adds each of its arguments to the current possibilities list; hence, it can be called only in a generator. It cannot be called with zero arguments; (NOTE) means (NOTE (INSTANCE)).

E. (ADIEU pos1...posN) CEXPR
(AU-REVOIR pos...posN) CEXPR

If a generator (including a method) wants to get the possibilities list of the TRY-NEXT it feeds, it can:

See Sect. VII.1.5.

F. (GET-POSSIBILITIES) FSUBR

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

G. (SET-POSSIBILITIES possibilities-list)   SUBR

VII.1.9 The Interrupt System

In this section we outline the Conniver interrupt system in its crudest form. The system uses it for errors and calling if-added methods. These uses are described in the remaining sections of the appendix.

```
A. (CINTERRUPT expression)                        SUBR
   (NOW expression)                               SUBR
```

These two functions both cause expression to be evaluated as soon as control is next in the Conniver evaluator (if interrupts are allowed; see B). They may be called from Lisp, in which case the interruptions will be deferred until the current Lisp evaluation is over. The difference between them is that CINTERRUPT stacks its interrupt so that all previously ordered interrupts will be run first, whereas NOW causes expression to be evaluated before the old ones.

```
B. (ALLOW 'T-or-NIL)                               FSUBR
```

(ALLOW NIL) causes interrupts to be disabled; i.e., CINTERRUPT and NOW will stack expressions that are not evaluated. ALLOW of anything else enables interrupts; at the next possible place, all pending interrupts will be run.

VII.2 The Data Base

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

Objects are of the form:

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

Item data are of the form:

(item ☆c-markers☆)

where item 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, or a user-defined method type; 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 !>var, !<var, !,var, etc.; and body is a function body.

Method closures look like

(☆CLOSURE method fr ☆c-markers☆),

where method is a method, and fr is an internal frame pointer.

Any datum may be given an atomic name by PUTPROPing it on the atom under the indicator DATUM. This is done automatically by the method-defining functions, but must be done manually for objects, item data, and closures. The function NAME-DATUM may be used for this.

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

    (lnum (refco . status) ☆property-pairs☆)

where lnum is a layer number; refco is a reference count of the number of references besides this one to the layer number lnum by this datum; status is +, NIL, or a list of lnums of layers where the c-marker is cancelled; and property-pairs are of the form (ind prop . status), where ind and prop are arbitrary, and status is as for the whole datum. The c-markers on each datum are in order of decreasing lnums, as are the lnums in a status.

A c-marker or status with a given lnum indicates a mention of its datum by the context-layer associated with that lnum. A context-layer is of the form

    (☆LAYER lnum ☆data☆)

where lnum is its unique layer number, and data are the data it mentions.

A context is a list like

    (☆CONTEXT ☆layers☆),

where layers must be in order of decreasing lnums. 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 layers as its cdr is a legal context; in particular, (CDR context) = (POP-CONTEXT context) for all practical purposes.

Every datum has various properties, including presence or absence, which vary from context to context. Typically, data-base changing functions (like ADD, REMOVE, DPUT, and DREM) apply only to the current context and its subcontexts, while data-base searchers (like FETCH, PRESENT, and DGET) search the present context from the most local layer upwards, ignoring all canceled c-markers or pairs. These notions will now be made precise. (The next two paragraphs may be ignored.)

Each context rigorously defines the status of every datum as present or absent, as follows: if the datum has a c-marker whose lnum corresponds to some layer of this context and whose status is + or a list with no lnums corresponding to layers of the context, it is present, else absent. In other words, it is present if it has at least one uncancelled c-marker. In particular, if it is unmentioned by all layers in the context, it is absent.

Each context also determines the properties that a datum has, as follows: its property for indicator ind is the prop of the first property pair in a c-marker with lnum corresponding to some layer of the context, such that that pair is uncancelled in the context; i.e., the first pair whose status shares no lnums with layers of the context. If there is no such pair, the datum has no such property.

Every c-marker must specify either non-NIL status, or non-NIL property-pairs, or non-zero refco, or any combination, and cannot be cancelled by its own lnum, or it does not constitute a mention. System functions delete all c-markers of the forms (n (0)); a status of the form (...n...) which appears anywhere in a c-marker (n...) is converted to NIL. For example, if (5 (0 . 5)) ever arises, it is converted to (5 (0)) and deleted.

When a layer 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.

Item data, 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, DPUT, and other functions); _unindexing_ occurs when its last mention is removed (by REMOVE, DREM, the garbage collector, etc.). Anonymous unindexed item data and methods are subject to garbage collection if unprotected.

Data-Base Errors

The data-base 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--function // RESULT <- ?
TOO MANY ARGUMENTS--function // GO ON?
```

The first will cause whatever you RETURN to be the value of the function. The second will ignore the extra arguments if you proceed.

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

```
MEANINGLESS DATUM -- function // DATUM <- ?
```

If you RETURN a better datum, the system will use it in place of the bad one.

VII.2.1 Data-Base Initialization

(DATA-INIT [n 100] [m 10]) 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 context layers allowed; if the data-base functions ever attempt to maintain more than this number at once, the message

TOO MANY CONTEXT LAYERS -- LAYER

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

The second parameter, m, is the increment between the numbers of context layers consecutively generated by LAYER. Given the ordering constraint on layers, and the fact that SPLICE (qv.) must be able to generate layers with lnums between those of any two layers, 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 layers at a time, numbered 0, 10., 20., 30.,....

VII.2.2 Datum Creation and Manipulation

A. (OBJECT [structure NIL]) LSUBR

creates and returns 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.

B. (NAME-DATUM datum atom) SUBR

This function causes datum to be called by the name atom in all future dealings with the system. It returns the atom. If datum is as yet unmentioned by any contexts, this is equivalent to (PUTPROP atom datum), but NAME-DATUM avoids certain timing errors associated with the other method.

Once a datum is named, the name should be used thenceforth in referring to it. If an already-named datum is renamed, the system will use the new name from then on.

One reason for naming data is so they can be used in items. A pointer to an actual non-atomic datum as a component of an item (as, (POSSIBLE ((INNOCENT NIXON) (0 (0 . +))))) is a no-no.

C. (DATUM item [name]) LSUBR

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 an item, it is obviously guaranteed not to be EQ to an indexed item datum for the same item (if any). Thus, if he executes (REALIZE (LIST '(LINE G001))) and ((LINE G001) (9 (0) (ABSENT T . +))) 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. 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 item.

If DATUM is given a second argument, it becomes the name of the datum, and is the value returned.

D. (ITEM item-datum) SUBR

This function is equivalent to CAR for the usual type of item datum, but also works on atomic-named data. It is the inverse of DATUM. Thus, if you execute (NAME-DATUM (ADD '(GZORN ZEP)) 'FOO) then (ITEM 'FOO) = (GZORN ZEP), and

```
        (DATUM (ITEM 'FOO)) = FOO
        (ITEM (DATUM '(GZORN ZEP))) = (GZORN ZEP)
```

```
E. (IF-ADDED ['atom] 'pattern '☆body☆)          FSUBR
   (IF-REMOVED ['atom] 'pattern '☆body☆)        FSUBR
   (IF-NEEDED ['atom] 'pattern '☆body☆)         FSUBR
```

See sect. VII.1.2.B.

```
F. (METHOD-TYPE atom)                                    SUBR
   (DELETE-METHOD-TYPE atom)                             SUBR
```

These functions are used to define new method types, in addition to IF-ADDED, IF-REMOVED, and IF-NEEDED. (METHOD-TYPE atom) causes atom to become defined as a method-defining function just like IF-ADDED, with its own index. If DATA-INIT is performed subsequently, all such methods will be deleted. For example, following (METHOD-TYPE 'PRE-ADD),

```
(ADD
  (PRE-ADD P1 (ON !>X !>Y)
     (AND (PRESENT (ON !,X !>Z))
          (REMOVE !"(ON ,X ,Z)))))
```

would define and add a new method of this type. Presumably, such a new method is intended to be used with a user's own ADD function; he is responsible for setting up routines (using FETCHM, INVOKE, and TRY-NEXT) to use the method properly.

If the user wishes to define methods of the new type with some function of a different format from that of the standard kinds, he should define the defining function first; METHOD-TYPE will avoid redefining it.

VII.2.3 Enlarging, Depleting, and Searching the Data Base

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

These functions make datum present (REALIZE, ACTUALIZE, ADD, INSERT) or absent (UNREALIZE, UNACTUALIZE, REMOVE, KILL), by giving it "+" status in the c-marker for context's first layer, or by canceling all outstanding c-markers, respectively. Here, "datum" means datum (REALIZE, UNREALIZE, ACTUALIZE, UNACTUALIZE) or "item datum referred to by item" (ADD, REMOVE, INSERT, KILL). All of them return datum. (ADD and REMOVE can be used to alter the status of data not referred to by skeleton; see VII.2.3.B.)

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

If ADD or REALIZE is given a item 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. In all cases, the data base change occurs before any methods are called. Methods are called only if datum's status changes; i.e., realizing a present, or unrealizing an absent, datum is a no-op. Methods are called by stacking invocations of them as Conniver interrupts. Hence, (ALLOW NIL) will cause all if-addeds to remain uninvoked until interrupts are re-enabled.

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 atom [context CONTEXT])                    CEXPR,LSUBR
   (REMOVE atom [context CONTEXT])                 CEXPR,LSUBR
   (INSERT atom [context CONTEXT])                 LSUBR
   (KILL atom [context CONTEXT])                   LSUBR
```

If ADD and REMOVE are given atomic arguments, they are synonymous with REALIZE and UNREALIZE; INSERT and KILL with such arguments are synonymous with ACTUALIZE and UNACTUALIZE. The most common use of this extra meaning is in using ADD to add methods, which usually have atomic names.

```
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 Sect. VII.1.8.

FETCHI returns a possibilities list containing only the item possibilities. FETCHM returns a list of only the method possibilities of the type type, that are present in context and match pattern. (Type may be a user-defined type (Sect VII.2.2).)

VII.2.4 Properties of Data

```
A.  (REAL datum [context CONTEXT])                  LSUBR
    (UNREAL datum [context CONTEXT])                LSUBR
    (PRESENT pattern [context CONTEXT])             LSUBR
    (ABSENT item [context CONTEXT])                 LSUBR
```

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 argument and then calls UNREAL.

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

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

    (DPUT+ datum property indicator [context CONTEXT])
                                                    LSUBR
    (DGET+ datum indicator [context CONTEXT])       LSUBR
    (DREM+ datum indicator [context CONTEXT])       LSUBR
```

DPUT associates the pair (indicator property . +) with datum in the first ("most local") layer of context; like REALIZE, UNREALIZE, and their ilk, these effects are invisible above context and garbage-collectable if its top layer is reclaimed. DGET finds the first uncancelled pair associated with indicator in any layer of context, starting with its first layer; if there is no such pair, its value is NIL. DREM has the same value, but, as a side effect, cancels all uncancelled pairs starting with indicator; it does it by adding the lnum for the top layer of context to the status for all these pairs. Thus, after a DREM, DGET for the same datum, indicator, and context will return NIL.

DPUT+, DGET+, and DREM+ are exactly the same, but they ignore all context layers before the first in which datum has uncancelled status. Thus, DPUT+ gives a datum properties in the context in which it is realized. This is useful if a property happens to be calculated for the first time in a hypothetical context, but is itself non-hypothetical; DPUT+ makes sure it is visible from all subcontexts of the one in which the datum first appeared, and saves having to calculate it repeatedly, once per hypothesis. If DPUT+ is given a datum which is absent in context, the error message

```
ABSENT DATUM -- DPUT+ // GO ON?
```

occurs.

```
C. (DPUTL datum property indicator layer)        SUBR
   (DGETL datum indicator layer)                 SUBR
   (DREML datum indicator layer)                 SUBR
```

These functions manipulate properties in an explicitly given context layer. DPUTL associates property with indicator in the c-marker for layer on datum. As usual, these effects are invisible in super-contexts not containing layer, and will be garbage-collected if layer is. DGETL and DREML search the c-marker of layer on datum for a pair with first element = indicator, and return it, or NIL if there isn't one; DREML removes what it finds.

VII.2.5 Manipulating Contexts

```
A. (LAYER)                                      LSUBR
   (FLUSH layer)                                SUBR
```

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

```
        TOO MANY CONTEXT LAYERS -- LAYER
```

is generated.

FLUSH removes all copies of the lnum of its argument from all data mentioned by it. If some datum loses all of its c-markers because of it, it will be unindexed. The error messages that can come out are due to Conniver errors, which should be ignored, since we do not wish to hear of unpleasant things. They are:

```
        NO REFERENCE COUNT FOR LAYER lnum
            ON DATUM datum --FLUSH1
        TOO FEW REFERENCES TO LAYER lnum
            ON DATUM datum --FLUSH1
```

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

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

```
        EMPTY CONTEXT -- POP-CONTEXT // SUPER-CONTEXT <- ?
```

and returns what you give it.

FINALIZE has the same value as POP-CONTEXT, with the side effect of making its argument an equivalent context to its superior. That is, all data will have the same properties in the super-context that they had in the original one be equivalent.

NEW-CONTEXT creates a context by CONSing the flag ☆CONTEXT onto layer-list. The layers in the list must be in order of decreasing lnums, or the message

```
        UNORDERED CONTEXT -- NEW-CONTEXT // LAYERS <- ?
```

appears, and NEW-CONTEXT tries again with the list you give it.

SPLICE adds a brand-new layer to context, <u>just</u> <u>after</u> its first frame. This layer 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 LAYER, they can cause its error.

C. (IN-CONTEXT context form)   CEXPR,SUBR

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. The SUBR version of IN-CONTEXT calls the Lisp CEVAL.

D. (MENTIONERS datum [sign NIL] [context CONTEXT])
                                                LSUBR
   (CONTEXT+ datum [context CONTEXT])   SUBR

MENTIONERS returns a list, in decreasing lnum order, of all the layers in context that mention datum. If sign is non-NIL, it ignores all cancelled c-markers. If sign does = NIL, MENTIONERS returns all mentioning layers.

CONTEXT+ returns the closest super-context of context in which datum was realized; i.e., whose first layer has a c-marker on datum with uncancelled status. Hence, (DPUT+ datum prop ind context) means the same as (DPUT datum prop ind (CONTEXT+ context)).

E. (C-MARKER datum layer)   SUBR

This function returns the c-marker for layer on datum, or NIL if layer doesn't mention datum. If layer is subsequently garbage-collected, or the c-marker degenerates to the form "(n (0))," the c-marker will no longer be attached to datum. Never say Conniver didn't give you enough rope.

VII.2.6 The Matcher

The matcher is documented in detail in Sect. IV. Here we merely summarize the meaning of each of the variable prefixes that it knows about.

A. !>var -- The basic matcher variable, which matches any expression which does not contain any variables (after substitution for "!,var's"), and binds var to it on the alist for its side of the match. The only exception is that a !>var appearing in a FETCH-pattern need not match a variable-less expression in a method pattern; it will be bound to ☆UNASSIGNED, and, when the method returns, will be assigned to the piece of method pattern it matches, with method variable values substituted. The form !>(var ☆restrictions☆) matches any variable-less expression such that all the restrictions are non-NIL when evaluated.

B. !,var -- This form does not bind a variable, but refers to the value associated with a previous binding; either one produced by !> or the Conniver binding in existence when the match is started.

C. !,(var value) -- This binds var to value, and matches anything that value would match.

D. !<var -- In general, !<var makes sense only in a method pattern. It matches only an expression with variables in it after substitution of values for "!,'s", and binds var on the proper matcher alist to that expression.

E. !?var -- This is also for method patterns only. It matches any expression that either !>var or !<var would match; in the former case, it binds var to the variable-less expression matched; in the latter, to ☆UNASSIGNED. Hence, inside a method, such a variable will be assigned only if it matched a definite expression.

F. !;var -- This is another ambiguous expression, which only works in FETCH-patterns. If var is unassigned, it behaves like !>var; otherwise, like !,var.

G. !'var -- This expression appears to have no use except in method patterns. It matches any expression, without looking at it, and binds var to it, even if it contains variables whose values could be substituted. It is useful for doing obscure syntactical decompositions on patterns.

All these features are explained in greater detail in Sect. IV.

VII.3 Debugging in Conniver

There are four classes of debugging functions in Conniver: listen-loop (breakpoint) functions, information printers, a trace package, and variable monitors. The first three classes of function are discussed in the three sections below.

Variable monitors are not a particular bunch of functions, but a mechanism for using Lisp functions for performing debugging actions when variables are referenced or changed. It is implemented as follows: Conniver variable locatives, at the top and lower levels, are implemented as lists of the form (atom value [monitor]). The optional monitor is a Lisp LSUBR or LEXPR, of two or three arguments. It will be called whenever the variable is accessed or set, in the former case with two arguments, in the latter with three. The two arguments for the case of accessing are the name of the accessing function (usually "/,") and the locative involved. For setting, the three arguments are the setting function (e.g., CSET), the locative before the set, and the new value.

There are no special functions for placing a monitor. It can be done with RPLACD in the following fashion:

(RPLACD (VLOC atom) (LIST monitor))

Another debugging feature is the ~A-interrupts, described in Sect. V. Each function of ~A is reprinted in this section in its logical category; a complete listing is found in Sect. V.

Remember that others may be added by the user.

VII.3.1 Listen-Loop Builders and Manipulators

A.	(LISTEN message)	CEXPR
	(EAR)	SUBR
	(NEAR)	SUBR
	(TOP)	SUBR

These functions create and return to listen loops whose bodies contain loops referred to by tags of the form EAR-n. These are called "ears." LISTEN creates a new such loop, which is a READ-CEVAL-CPRINT loop just like the top level, printing its message argument, followed by EAR-n. Whenever it is ready for the next input, it types "_" (left-arrow or underscore). Within such a loop, the following internal Lisp variables may prove useful: ** is bound to the last expression read; *, to the value of the last expression; and _ to the expression before last. These must be accessed using "@." If you wish to flush the current input line, type ~AF, which responds with a carriage return and a reprint of "_."

Within such a loop, the tag EAR points to a place in the body which prints EAR-n and restarts the loop; the variable EAR-n is bound to that tag. Thus (GO 'EAR) and (GO EAR-n) have the same effect. Like all other tags, ears may be used for relative evaluations and EXITing as well as GOing; therefore, to cause a LISTEN to return a value, use (EXIT value EAR-n). $P or (DISMISS) causes NIL to be returned.

The remaining functions manipulate such listen loops. (EAR) interrupts Conniver in the next possible place, sprouting a new ear; ~AE calls this function. (NEAR) interrupts Conniver with (GO 'EAR); i.e., it causes it to return to the nearest already-existing ear; ~AN calls it. (TOP) flushes the current Conniver stack, resets the ear counter to 1, unbinds all previous ears, and sprouts a new EAR-1; typing ~AT has the same effect. Both ~AE and ~AN work only at places where Conniver is interruptible, i.e., between steps in evaluation. They cannot be used in the middle of infinite printouts, Lisp evaluations, or READ's.

```
B. (UP [n 1] [action 'BT] [whichlink 'CONTROL]) CEXPR
   (DOWN [n 1] [action 'BT])                    CEXPR
   (J [frame original-LISTEN-access-frame] [action 'BT])
                                                CEXPR
```

When a LISTEN loop is created, its access and control links are the same. Evaluations are with respect to them. The functions of this section enable you to move this entire loop around the control tree to examine and alter variables and control structures. UP moves the EAR-loop n frames up either the control or access links, depending on whichlink. When you arrive, the value of action will be printed, unless it is BT (the default), which causes the same data to be printed that BACKTRACE (Sect. VII.3.2.B) would print. DOWN moves n frames back down the links followed by UP. J jumps to a new frame, from which it is meaningless to go DOWN. (J) returns you to the original place in the tree.

All this movement occurs by clobbering the access link on the LISTEN frame. The current one is stored as CURFRAME. The control link is not disturbed, so (DISMISS) or $P work even if you have moved the frame away from where it started.

If you attempt to go UP off the top-level EAR or DOWN further than you've come up, the message

```
        excess FRAMES TOO FAR
```

will appear, and no action will be taken.

```
C. (CERR '☆messages☆)                              FSUBR
   (CBREAK '☆messages☆)                            FSUBR
   (CERRMESS)                                      SUBR
```

The messages are printed on the same line, one after another, those of the form "@exp" being Lisp-evaluated, after which a Lisp READ-EVAL-CPRINT loop is created. CERR first prints "☆☆ERROR☆☆" and the form Conniver was working on. Expressions of the form (RETURN value) cause the CERR or CBREAK to return the given value; $P is equivalent to (RETURN NIL). A listen loop like this is created at the next Lisp-interruptible place by ~AL.

Most catchable data base errors cause CERR-loops to be created. Within such a loop, an ERRSET will catch all Lisp errors, including ~X. ~AF will flush the current input buffer. (CERRMESS) causes the messages to be reprinted. If the priority interrupt system has been disabled while the data base is in an inconsistent state, CERR or CBREAK will turn it on for the duration of the loop; if it is left with RETURN, it will go off again. If ^G is executed, however, the data base may be screwed; it might be right to DATA-INIT following such a hasty retreat.

VII.3.2 Data Printers

A. (CPRINT exp) SUBR
   (CPRIN1 exp) SUBR

These functions behave exactly like their Lisp counterparts PRINT and PRIN1, except that they are "programmable," in the sense that special data types are printed in different ways according to their CAR's. CPRIN1 prints atomic argument on the current line; if its argument's CAR is an atom with a CPRINT property, it does something special; otherwise, it CPRIN1's its CAR, then its CDR. (We are not being very precise.) If the argument does have a marked CAR, the CPRINT property is assumed to be a FEXPR or FSUBR; CPRIN1 merely applies it to the thing to be CPRINTed. In all cases, CPRIN1 returns the thing printed. CPRINT prints a carriage return, CPRIN1's its argument, and prints a space, then returns its argument.

The built-in data types treated specially by CPRIN1 are as follows: all quoted expressions, statements labels, matcher variables, and other system macro'd data are printed in the same format as they are input; tags, frames, and closures are printed in such a way that internal fr pointers are replaced by the expressions that gave rise to those fr's. The user is free to add new data formats to this list by creating FEXPR's attached to atoms used to flag them. For example, to make all contexts print out as lists of numbers, execute

```
(DEFUN CP-CONTEXT FEXPR (CON)
   (PRIN1 (PATH CON))    )

(DEFPROP ☆CONTEXT CP-CONTEXT CPRINT)
```

B. (EXPRESSION frame)   SUBR
   (BACKTRACE [number 696969])   LEXPR

EXPRESSION returns the form whose evaluation gave rise to frame; it is documented in Sect. VII.1.4.

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. The backtrace is programmable in the following sense: if an expression's CAR has a BACKTRACE property, the property is assumed to be an EXPR or SUBR of two arguments, a frame and a list of arguments; BACKTRACE applies the function to the frame and CDR of the expression and does nothing else. This is used internally to print EAR-frames, PROG's, COND's, and other things in special formats. Another way to use expressions is with ~AX, which causes the current (EXPRESSION (FRAME)) to be printed. Execution then continues.

C. (PATH [context CONTEXT])   LSUBR

The value of PATH is a list whose first element is ☆CONTEXT, followed by the inums of context's layers. Such an object serves no useful purpose, but it is much more more lucidly printable than context itself, in general.

VII.3.3 Tracing in Conniver

For those who like to trace, there is a crude trace package which exists as CNVR;CTRACE >. It is not normally part of the Conniver system. Suggestions and volunteers for improving it are welcome.

The Lisp tracer may also be used while Conniving.

A. (CTRACE '☆specs☆) CEXPR

Each spec is of the form (atom EN ☆things-to-do-on-entrance☆ EX ☆things-to-do-on-exit☆). The order of EX and EN is unimportant. If an EN is present, a message will be printed when the function is entered, and the things to do will be EVALuated; EX is similar. Both are optional, although leaving both out is a slow no-op. If a spec is an atom, it is equivalent to (atom EN (CDISPLAY ☆ARGS) EX (CDISPLAY ☆VAL)). (See below.) Within a traced function, ☆ARGS will be bound to a list of evaluated or unevaluated arguments (the status of each of which depends imaginatively on the variable declarations of the function); and ☆VAL will, on output, be bound to the value the function is to return.

Functions, generators, and methods may be traced.

B. (CUNTRACE '☆atoms☆) CEXPR

Each atom must be a CTRACEd function, which is untraced.

C. (CDISPLAY ☆things☆) FEXPR

This is a handy function in tracing. It prints out a table of the Lisp value of each thing, unless it is an atom, when its Conniver value will be printed. Thus, to see the arguments to a function and the CAR of the Conniver variable FOO, use

```
(CDISPLAY ☆ARGS (CAR ,FOO)), and get
☆ARGS = whatever-they-are
(CAR ,FOO) = whatever-it-is.
```

D. (/: 'atom) FEXPR

(/: atom) is the internal representation of :atom. The Lisp trace package can be used to trace the function "/:," thus showing your flow of control.

Bibliography

Bobrow, D.G. and Wegbreit, B. (1972) A Model and Stack Implementation of Multiple Environments, Bolt, Beranek, and Newman Inc. Report 2334.

Hewitt, C. (1971) Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot, M.I.T. A.I. Laboratory Technical Report 258.

McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I. (1962) LISP 1.5 Programmer's Manual, Cambridge, Mass.: The MIT Press.

PDP-6 (1967) PDP-6 LISP, M.I.T. A.I. Laboratory Memo. No. 116A.

Weissman, C. (1967) LISP 1.5 Primer, Belmont, Calif.: Dickenson Publishing Company, Inc.

White, J.L. (1970) An Interim LISP User's Guide, M.I.T. A.I. Laboratory Memo No. 190.