15. BREAKING, TRACING, AND ADVISING

Medley provides several different facilities for modifing the behavior of a function without actually editing its definition. By "breaking" a function, you can cause breaks to occur at various times in the running of an incomplete program, so that the program state can be inspected. "Tracing" a function causes information to be printed every time the function is entered or exited.

"Advising" is a facility for specifying longer-term function modifications. Even system functions can be changed through advising.

Breaking Functions and Debugging

Debugging a collection of Lisp functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. In the Medley, there are three facilities which allow you to (temporarily) modify selected function definitions so that you can follow the flow of control in your programs, and obtain this debugging information. All three redefine functions in terms of a system function, BREAK1 (see Chapter 14).

BREAK modifies the definition of a function *FN*, so that whenever *FN* is called and a break condition (user-defined) is satisfied, a function break occurs. You can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE modifies a definition of a function *FN* so that whenever *FN* is called, its arguments (or some other user-specified values) are printed. When the value of *FN* is computed it is printed also. TRACE is a special case of BREAK.

BREAKIN allows you to insert a breakpoint inside an expression defining a function. When the breakpoint is reached and if a break condition (defined by you) is satisfied, a temporary halt occurs and you can again investigate the state of the computation.

The following two examples illustrate these facilities. In the first example, the function FACTORIAL is traced. TRACE redefines FACTORIAL so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fifth recursion, a full interactive break occurs. The situation is then the same as though (BREAK FACTORIAL) had been performed instead of (TRACE FACTORIAL), now you can evaluate various Interlisp forms and direct the course of the computation. In this case, the variable N is examined, and BREAK1 is instructed to return 1 as the value of this cell to FACTORIAL. The rest of the tracing proceeds without incident. Presumably, FACTORIAL would be edited to change L to 1.

```
←PP FACTORIAL
(FACTORIAL
       [LAMBDA (N)
       (COND
          ((ZEROP N)
             L)
          (T (ITIMES N (FACTORIAL (SUB1 N])
FACTORIAL
←(TRACE FACTORIAL)
(FACTORIAL)
←(FACTORIAL 4)
FACTORIAL:
N = 4
```

```
    FACTORIAL:
    N = 3
        FACTORIAL:
        N = 2
            FACTORIAL:
            N = 1
                FACTORIAL:
                N = 0
UNBOUND ATOM
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
                FACTORIAL = 1
            FACTORIAL = 1
        FACTORIAL = 2
    FACTORIAL = 6
FACTORIAL = 24
24
←
```

In the second example, a non-recursive definition of FACTORIAL has been constructed. BREAKIN is used to insert a call to BREAK1 just after the PROG label LOOP. This break is to occur only on the last two iterations, when N is less than 2. When the break occurs, in trying to look at the value of N, NN is mistakenly typed. The break is maintained, however, and no damage is done. After examining N and M the computation is allowed to continue by typing OK. A second break occurs after the next iteration, this time with N = 0. When this break is released, the function FACTORIAL returns its value of 120.

```
←PP FACTORIAL
(FACTORIAL
  [LAMBDA (N)
     (PROG ((M 1))
        LOOP (COND
                    ((ZEROP N)
                     (RETURN M)))
               (SETQ M (ITIMES M N))
               (SETQ N (SUB1 N))
               (GO LOOP])
FACTORIAL

←(BREAKIN FACTORIAL (AFTER LOOP) (ILESSP N 2]
SEARCHING...
FACTORIAL

←((FACTORIAL 5)
((FACTORIAL) BROKEN)
:NN
U.B.A.
NN
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)
```

```
((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

Note: BREAK and TRACE can also be used on CLISP words which appear as CAR of form, e.g. FETCH, REPLACE, IF, FOR, DO, etc., even though these are not implemented as functions. For conditional breaking, you can refer to the entire expression via the variable EXP, e.g. (BREAK (FOR (MEMB 'UNTIL EXP))).

(**BREAK0** *FN WHEN COMS — —*) [Function]

Sets up a break on the function *FN*; returns *FN*. If *FN* is not defined, returns (*FN* NOT DEFINED).

The value of *WHEN*, if non-NIL, should be an expression that is evaluated whenever *FN* is entered. If the value of the expression is non-NIL, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of conditionally breaking a function.

The value of *COMS*, if non-NIL, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCOMS* argument to BREAK1, Chapter 14.)

BREAK0 sets up a break by doing the following:

Redefines *FN* as a call to BREAK1 (Chapter 14), passing an equivalent definition of *FN*, *WHEN*, *FN*, and *COMS* as the BRKEXP, *BRKWHEN*, *BRKFN*, and *BRKCOMS* arguments to BREAK1

Defines a GENSYM (Chapter 2) with the original definition of *FN*, and puts it on the property list of *FN* under the property BROKEN

Puts the form (BREAK0 *WHEN COMS*) on the property list of *FN* under the property BRKINFO (for use in conjunction with REBREAK)

Adds *FN* to the front of the list BROKENFNS.

If *FN* is non-atomic and of the form (*FN_1* IN *FN_2*), BREAK0 breaks every call to *FN_1* from within *FN_2*. This is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to BREAKIN described below, but can be performed even when *FN_2* is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If *FN_1* is not found in *FN_2*, BREAK0 returns the value (*FN_1* NOT FOUND IN *FN_2*).

BREAK0 breaks one function *inside* another by first calling a function which changes the name of *FN_1* wherever it appears inside of *FN_2* to that of a new function, *FN1*-IN-*FN2*, which is initially given the same function definition as *FN_1*. Then BREAK0 proceeds to

break on *FN1*-IN-*FN2* exactly as described above. In addition to breaking *FN1*-IN-*FN2* and adding *FN1*-IN-*FN2* to the list BROKENFNS, BREAK0 adds *FN1* to the property value for the property NAMESCHANGED on the property list of FN_2 and puts (FN_2 . FN_1) on the property list of *FN1*-IN-*FN2* under the property ALIAS. This will enable UNBREAK to recognize what changes have been made and restore the function FN_2 to its original state.

If *FN* is nonatomic and not of the above form, BREAK0 is called for each member of *FN* using the same values for *WHEN*, *COMS*, and *FILE*. This distributivity permits you to specify complicated break conditions on several functions. For example,

```
(BREAK0 '(FOO1 ((PRINT PRIN1) IN (FOO2 FOO3)))
   '(NEQ X T)
   '(EVAL ?= (Y Z) OK) )
```

will break on FOO1, PRINT-IN-FOO2, PRINT-IN-FOO3, PRIN1-IN-FOO2 and PRIN1-IN-FOO3.

If *FN* is non-atomic, the value of BREAK0 is a list of the functions broken.

(**BREAK** *X*) [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T). For each list, it performs (APPLY 'BREAK0 *LIST*). For example, (BREAK FOO1 (FOO2 (GREATERP N 5) (EVAL))) is equivalent to (BREAK0 'FOO1 T) and (BREAK0 'FOO2 '(GREATERP N 5) '(EVAL)).

(**TRACE** *X*) [NLambda NoSpread Function]

For each atomic argument, it performs (BREAK0 *ATOM* T '(TRACE ?= NIL GO)). The flag TRACE is checked for in BREAK1 and causes the message *"FUNCTION :"* to be printed instead of (*FUNCTION* BROKEN).

For each list argument, CAR is the function to be traced, and CDR the forms to be viewed, i.e., TRACE performs:

```
(BREAK0 (CAR LIST) T (LIST 'TRACE '?= (CDR LIST) 'GO))
```

For example, (TRACE FOO1 (FOO2 Y)) causes both FOO1 and FOO2 to be traced. All the arguments of FOO1 are printed; only the value of Y is printed for FOO2. In the special case when you want to see *only* the value, you can perform (TRACE (*FUNCTION*)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: You can always call BREAK0 to obtain combination of options of BREAK1 not directly available with BREAK and TRACE. These two functions merely provide convenient ways of calling BREAK0, and will serve for most uses.

Note: BREAK0, BREAK, and TRACE print a warning if you try to modify a function on the list UNSAFE.TO.MODIFY.FNS (Chapter 10).

(**BREAKIN** *FN WHERE WHEN COMS*) [NLambda Function]

BREAKIN enables you to insert a break, i.e., a call to BREAK1 (Chapter 14), at a specified location in the interpreted function *FN*. BREAKIN can be used to insert breaks before or after PROG labels, particular SETQ expressions, or even the evaluation of a variable. This

is because BREAKIN operates by calling the editor and actually inserting a call to BREAK1 at a specified point *inside* of the function. If *FN* is a compiled function, BREAKIN returns (*FN* UNBREAKABLE) as its value.

WHEN should be an expression that is evaluated whenever the break is entered. If the value of the expression is non-NIL, a break is entered, otherwise the function simply called and returns without causing a break. This provides the means of creating a conditional break. For BREAKIN, unlike BREAK0, if *WHEN* is NIL, it defaults to T.

COMS, if non-NIL, should be a list of break commands, that are interpreted and executed if a break occurs. (See the *BRKCONMS* argument to BREAK1, Chapter 14.)

WHERE specifies where in the definition of *FN* the call to BREAK1 is to be inserted. *WHERE* should be a list of the form (BEFORE ...), (AFTER ...), or (AROUND ...). You specify where the break is to be inserted by a sequence of editor commands, preceded by one of the symbols BEFORE, AFTER, or AROUND, which BREAKIN uses to determine what to do once the editor has found the specified point, i.e., put the call to BREAK1 BEFORE that point, AFTER that point, or AROUND that point. For example, (BEFORE COND) will insert a break before the first occurrence of COND, (AFTER COND 2 1) will insert a break after the predicate in the first COND clause, (AFTER BF (SETQ X &)) after the *last* place X is set. Note that (BEFORE TTY:) or (AFTER TTY:) permit you to type in commands to the editor, locate the correct point, and verify it, and exit from the editor with OK. BREAKIN then inserts the break BEFORE, AFTER, or AROUND that point.

Note: A STOP command typed to TTY: produces the same effect as an unsuccessful edit command in the original specification, e.g., (BEFORE CONDD). In both cases, the editor aborts, and BREAKIN types (NOT FOUND).

If *WHERE* is (BEFORE ...) or (AFTER ...), the break expression is NIL, since the value of the break is irrelevant. For (AROUND ...), the break expression will be the indicated form. In this case, you can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if you inserted a break after a COND predicate, e.g., (AFTER (EQUAL X Y)), you would be powerless to alter the flow of computation if the predicate were not true, since the break would not be reached. However, by breaking (AROUND (EQUAL X Y)), you can evaluate the break expression, i.e., (EQUAL X Y), look at its value, and return something else if desired.

If *FN* is interpreted, BREAKIN types SEARCHING... while it calls the editor. If the location specified by *WHERE* is not found, BREAKIN types (NOT FOUND) and exits. If it is found, BREAKIN puts T under the property BROKEN-IN and (*WHERE WHEN COMS*) under the the property BRKINFO on the property list of *FN*, and adds *FN* to the front of the list BROKENFNS.

Multiple break points, can be inserted with a single call to BREAKIN by using a list of the form ((BEFORE ...) ... (AROUND ...)) for *WHERE*. It is also possible to call BREAK or TRACE on a function which has been modified by BREAKIN, and conversely to BREAKIN a function which has been redefined by a call to BREAK or TRACE.

The message typed for a BREAKIN break is ((*FN*) BROKEN), where *FN* is the name of the function inside of which the break was inserted. Any error, or typing control-E, will cause the full identifying message to be printed, e.g., (FOO BROKEN AFTER COND 2 1).

A special check is made to avoid inserting a break inside of an expression headed by any member of the list NOBREAKS, initialized to (GO QUOTE *), since this break would never be activated. For example, if (GO L) appears before the label L, BREAKIN (AFTER L) will not insert the break inside of the GO expression, but skip this occurrence of L and go on to the next L, in this case the label L. Similarly, for BEFORE or AFTER breaks, BREAKIN checks to make sure that the break is being inserted at a "safe" place. For example, if you request a break (AFTER X) in (PROG ... (SETQ X &) ...), the break will actually be inserted after (SETQ X &), and a message printed to this effect, e.g., BREAK INSERTED AFTER (SETQ X &).

(**UNBREAK** *X*) [NLambda NoSpread Function]

UNBREAK takes an indefinite number of functions modified by BREAK, TRACE, or BREAKIN and restores them to their original state by calling UNBREAK0. Returns list of values of UNBREAK0.

(UNBREAK) will unbreak all functions on BROKENFNS, in reverse order. It first sets BRKINFOLST to NIL.

(UNBREAK T) unbreaks just the first function on BROKENFNS, i.e., the most recently broken function.

(**UNBREAK0** *FN* —) [Function]

Restores *FN* to its original state. If *FN* was not broken, value is (NOT BROKEN) and no changes are made. If *FN* was modified by BREAKIN, UNBREAKIN is called to edit it back to its original state. If *FN* was created from (FN_1 IN FN_2), (i.e., if it has a property ALIAS), the function in which *FN* appears is restored to its original state. All dummy functions that were created by the break are eliminated. Adds property value of BRKINFO to the front of BRKINFOLST.

Note: (UNBREAK0 '(FN_1 IN FN_2)) is allowed: UNBREAK0 will operate on (FN_1-IN-FN_2) instead.

(**UNBREAKIN** *FN*) [Function]

Performs the appropriate editing operations to eliminate all changes made by BREAKIN. *FN* may be either the name or definition of a function. Value is *FN*.

UNBREAKIN is automatically called by UNBREAK if *FN* has property BROKEN-IN with value T on its property list.

(**REBREAK** *X*) [NLambda NoSpread Function]

Nlambda nospread function for rebreaking functions that were previously broken without having to respecify the break information. For each function on *X*, REBREAK searches BRKINFOLST for break(s) and performs the corresponding operation. Value is a

list of values corresponding to calls to BREAK0 or BREAKIN. If no information is found for a particular function, returns (*FN* - NO BREAK INFORMATION SAVED).

(REBREAK) rebreaks everything on BRKINFOLST, so (REBREAK) is the inverse of (UNBREAK).

(REBREAK T) rebreaks just the first break on BRKINFOLST, i.e., the function most recently unbroken.

(**CHANGENAME** *FN FROM TO*) [Function]

Replaces all occurrences of *FROM* by *TO* in the definition of *FN*. If *FN* is defined by an expr definition, CHANGENAME performs (ESUBST *TO FROM* (GETD *FN*)) (see Chapter 16). If *FN* is compiled, CHANGENAME searches the literals of *FN* (and all of its compiler generated subfunctions), replacing each occurrence of *FROM* with *TO*.

Note that *FROM* and *TO* do not have to be functions, e.g., they can be names of variables, or any other literals.

CHANGENAME returns *FN* if at least one instance of *FROM* was found, otherwise NIL.

(**VIRGINFN** *FN FLG*) [Function]

The function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by PRETTYPRINT, DEFINE, and the compiler.

If *FLG* = NIL, as for PRETTYPRINT, it does not modify the definition of *FN* in the process of producing a "clean" version of the definition; it works on a copy.

If *FLG* = T, as for the compiler and DEFINE, it physically restores the function to its original state, and prints the changes it is making, e.g., FOO UNBROKEN, FOO UNADVISED, FOO NAMES RESTORED, etc.

Returns the virgin function definition.

Advising

The operation of advising gives you a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the *interface* between functions as opposed to modifying the function definition itself, as in editing. BREAK, TRACE, and BREAKDOWN, are examples of the use of this technique: they each modify user functions by placing relevant computations *between* the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows you to treat anyone's functions as "black boxes," and to modify them without concern for their contents or details of operations. For example, you could modify SYSOUT to set SYSDATE to the time and date of creation by (ADVISE 'SYSOUT '(SETQ SYSDATE (DATE))).

As with BREAK, advising works equally well on compiled and interpreted functions. Similarly, it is possible to make a change which only operates when a function is called from some other specified function. For example, you can modify the interface between two particular functions, instead of the

interface between one function and the rest of the world. This latter feature is especially useful for changing the *internal* workings of a system function.

For example, suppose you wanted TIME (Chapter 22) to print the results of your measurements to the file FOO instead of the terminal. You can accomplish this by (ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO)).

Advising PRIN1, PRINT, or SPACES directly would have affected all calls to these frequently used functions, whereas advising ((PRIN1 PRINT SPACES) IN TIME) affects just those calls to PRIN1, PRINT, and SPACES from TIME.

Advice can also be specified to operate after a function has been evaluated. The value of the body of the original function can be obtained from the variable !VALUE, as with BREAK1.

Implementation of Advising

After a function has been modified several times by ADVISE, it will look like:

```
(LAMBDA arguments
   (PROG (!VALUE)
      (SETQ !VALUE
         (PROG NIL
               advice1
                  .
                  .      advice before
                  .
               advicen
               (RETURN BODY)))
      advice1
         .
         .      advice after
         .
      advicem
      (RETURN !VALUE)))
```

where *BODY* is equivalent to the original definition. If *FN* was originally an expr definition, *BODY* is the body of the definition, otherwise a form using a GENSYM which is defined with the original definition.

The structure of a function modified by ADVISE allows a piece of advice to bypass the original definition by using the function RETURN. For example, if (COND ((ATOM X) (RETURN Y))) were one of the pieces of advice *before* a function, and this function was entered with X atomic, Y would be returned as the value of the inner PROG, !VALUE would be set to Y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared *after* the function, Y would be returned as the value of the entire advised function.

The advice (COND ((ATOM X) (SETQ !VALUE Y))) *after* the function would have a similar effect, but the rest of the advice *after* the function would still be executed.

Note: Actually, ADVISE uses its own versions of PROG, SETQ, and RETURN, (called ADV-PROG, ADV-SETQ, and ADV-RETURN) to enable advising these functions.

Advise Functions

ADVISE is a function of four arguments: *FN*, *WHEN*, *WHERE*, and *WHAT*. *FN* is the function to be modified by advising, *WHAT* is the modification, or piece of advice. *WHEN* is either BEFORE, AFTER, or AROUND, and indicates whether the advice is to operate BEFORE, AFTER, or AROUND the body of the function definition. *WHERE* specifies exactly where in the list of advice the new advice is to be placed, e.g., FIRST, or (BEFORE PRINT) meaning before the advice containing PRINT, or (AFTER 3) meaning after the third piece of advice, or even (: TTY:). If *WHERE* is specified, ADVISE first checks to see if it is one of LAST, BOTTOM, END, FIRST, or TOP, and operates accordingly. Otherwise, it constructs an appropriate edit command and calls the editor to insert the advice at the corresponding location.

Both *WHEN* and *WHERE* are optional arguments, in the sense that they can be omitted in the call to ADVISE. In other words, ADVISE can be thought of as a function of two arguments (ADVISE *FN WHAT*), or a function of three arguments: (ADVISE *FN WHEN WHAT*), or a function of four arguments: (ADVISE *FN WHEN WHERE WHAT*). Note that the advice is always the *last* argument. If *WHEN* = NIL, BEFORE is used. If *WHERE* = NIL, LAST is used.

(**ADVISE** *FN WHEN WHERE WHAT*) [Function]

FN is the function to be advised, *WHEN* = BEFORE, AFTER, or AROUND, *WHERE* specifies where in the advice list the advice is to be inserted, and *WHAT* is the piece of advice.

If *FN* is of the form (*FN*$_1$ IN *FN*$_2$), *FN*$_1$ is changed to *FN1*-IN-*FN2* throughout *FN*$_2$, as with break, and then *FN1*-IN-*FN2* is used in place of *FN*. If *FN*$_1$ and/or *FN*$_2$ are lists, they are distributed as with BREAK0.

If *FN* is broken, it is unbroken before advising.

If *FN* is not defined, an error is generated, NOT A FUNCTION.

If *FN* is being advised for the first time, i.e., if (GETP *FN* 'ADVISED) = NIL, a GENSYM is generated and stored on the property list of *FN* under the property ADVISED, and the GENSYM is defined with the original definition of *FN*. An appropriate expr definition is then created for *FN*, using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised. Finally, *FN* is added to the (front of) ADVISEDFNS, so that (UNADVISE T) always unadvises the last function advised.

If *FN* has been advised before, it is moved to the front of ADVISEDFNS.

If *WHEN* = BEFORE or AFTER, the advice is inserted in *FN*'s definition either BEFORE or AFTER the original body of the function. Within that context, its position is determined by *WHERE*. If *WHERE* = LAST, BOTTOM, END, or NIL, the advice is added following all other advice, if any. If *WHERE* = FIRST or TOP, the advice is inserted as the first piece of advice. Otherwise, *WHERE* is treated as a command for the editor, similar to BREAKIN, e.g., (BEFORE 3), (AFTER PRINT).

If *WHEN* = AROUND, the body is substituted for * in the advice, and the result becomes the new body, e.g., (ADVISE 'FOO 'AROUND '(RESETFORM (OUTPUT T) *)). Note that if several pieces of AROUND advice are specified, earlier ones will be embedded inside later ones. The value of *WHERE* is ignored.

Finally (LIST *WHEN WHERE WHAT*) is added (by ADDPROP) to the value of property ADVICE on the property list of *FN*, so that a record of all the changes is available for subsequent use in readvising. Note that this property value is a list of the advice in order of calls to ADVISE, not necessarily in order of appearance of the advice in the definition of *FN*.

The value of ADVISE is *FN*.

If *FN* is non-atomic, every function in *FN* is advised with the same values (but copies) for *WHEN*, *WHERE*, and *WHAT*. In this case, ADVISE returns a list of individual functions.

Note: Advised functions can be broken. However if a function is broken at the time it is advised, it is first unbroken. Similarly, advised functions can be edited, including their advice. UNADVISE will still restore the function to its unadvised state, but any changes to the body of the definition will survive. Since the advice stored on the property list is the same structure as the advice inserted in the function, editing of advice can be performed on either the function's definition or its property list.

(**UNADVISE** *X*) [NLambda NoSpread Function]

An nlambda nospread like UNBREAK. It takes an indefinite number of functions and restores them to their original unadvised state, including removing the properties added by ADVISE. UNADVISE saves on the list ADVINFOLST enough information to allow restoring a function to its advised state using READVISE. ADVINFOLST and READVISE thus correspond to BRKINFOLST and REBREAK. If a function contains the property READVICE, UNADVISE moves the current value of the property ADVICE to READVICE.

(UNADVISE) unadvises all functions on ADVISEDFNS in reverse order, so that the most recently advised function is unadvised last. It first sets ADVINFOLST to NIL.

(UNADVISE T) unadvises the first function of ADVISEDFNS, i.e., the most recently advised function.

(**READVISE** *X*) [NLambda NoSpread Function]

An nlambda nospread like REBREAK for restoring a function to its advised state without having to specify all the advise information. For each function on *X*, READVISE retrieves the advise information either from the property READVICE for that function, or from ADVINFOLST, and performs the corresponding advise operation(s). It also stores this information on the property READVICE if not already there. If no information is found for a particular function, value is (*FN* - NO ADVICE SAVED).

(READVISE) readvises everything on ADVINFOLST.

(READVISE T) readvises the first function on ADVINFOLST, i.e., the function most recently unadvised.

A difference between ADVISE, UNADVISE, and READVISE versus BREAK, UNBREAK, and REBREAK, is that if a function is not rebroken between successive (UNBREAK)s, its break information is forgotten. However, once READVISE is called on a function, that function's advice is permanently saved on its property list (under READVICE); subsequent calls to

UNADVISE will not remove it. In fact, calls to UNADVISE update the property READVICE with the current value of the property ADVICE, so that the sequence READVISE, ADVISE, UNADVISE causes the augmented advice to become permanent. The sequence READVISE, ADVISE, READVISE removes the "intermediate advice" by restoring the function to its earlier state.

(**ADVISEDUMP** *X FLG*) [Function]

Used by PRETTYDEF when given a command of the form (ADVISE ...) or (ADVICE ...). If *FLG* = T, ADVISEDUMP writes both a DEFLIST and a READVISE; this corresponds to (ADVISE ...). If *FLG* = NIL, only the DEFLIST is written; this corresponds to (ADVICE ...). In either case, ADVISEDUMP copies the advise information to the property READVICE, thereby making it "permanent" as described above.