CHAPTER 10

BREAKING, TRACING, AND ADVISING

It is frequently useful to be able to modify the behavior of a function without actually editing its de nition. Interlisp provides several di erent facilities for doing this. By ''breaking'' a function, the user 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. These are very useful debugging tools.

''Advising'' is a facility for specifying longer- term function modi cations. Even system functions can be changed through advising.

10.1 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 Interlisp system, there are three facilities which allow the user to (temporarily) modify selected function de nitions so that he can follow the ow of control in his programs, and obtain this debugging information. All three rede ne functions in terms of a system function, BREAK1 (see page 9.11).

BREAK modi es the de nition of a function FN, so that whenever FN is called and a break condition (de ned by the user) is satis ed, a function break occurs. The user can then interrogate the state of the machine, perform any computation, and continue or return from the call.

TRACE modi es a de nition of a function FN so that whenever FN is called, its arguments (or some other values speci ed by the user) are printed. When the value of FN is computed it is printed also. (TRACE is a special case of BREAK).

BREAKIN allows the user to insert a breakpoint *inside* an expression de ning a function. When the breakpoint is reached and if a break condition (de ned by the user) is satis ed, a temporary halt occurs and the user can again investigate the state of the computation.

The following two examples illustrate these facilities. In the rst example, the user traces the function FACTORIAL. TRACE rede nes FACTORIAL so that it print its arguments and value, and then goes on with the computation. When an error occurs on the fth recursion, a full interactive break occurs. The situation is then the same as though the user had originally performed BREAK(FACTORIAL) instead of TRACE(FACTORIAL), and the user can evaluate various Interlisp forms and direct the course of the computation. In this case, the user examines the variable N, and instructs BREAK1 to return 1 as the value of this cell to FACTORIAL. The rest of the tracing proceeds without incident. The user would then presumably edit FACTORIAL 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

U.B.A.
L
(FACTORIAL BROKEN)
:N
0
:RETURN 1
                FACTORIAL = 1
            FACTORIAL = 1
        FACTORIAL = 2
    FACTORIAL = 6
FACTORIAL = 24
24

_
```

In the second example, the user has constructed a non- recursive de nition of FACTORIAL. He uses
BREAKIN 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, the user tries to look at the value of
N, but mistakenly types NN. The break is maintained, however, and no damage is done. After examining
N and M the user allows the computation 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, the user 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).
>
> BREAK0 redefines FN as a call to BREAK1 (page 9.11), with an equivalent definition of FN as BRKEXP, and WHEN, FN, COMS as BRKWHEN, BRKFN, BRKCOMS. Puts a GENSYM defined with the original definition of FN on the property list of FN under the property BROKEN. Puts (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 (FN1 IN FN2), BREAK0 breaks every call

to FN1 from within FN2. 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 specic function, e.g., (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to BREAKIN described below, but can be performed even when FN2 is compiled or blockcompiled, whereas BREAKIN only works on interpreted functions. If FN1 is not found in FN2, BREAK0 returns the value (FN1 NOT FOUND IN FN2).

BREAK0 breaks one function *inside* another by rst calling a function which changes the name of FN1 wherever it appears inside of FN2 to that of a new function, FN1-IN-FN2, which is initially given the same function denition as FN1. 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 FN2 and puts (FN2 . FN1) 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 FN2 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 the user 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]
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]
Nlambda nospread function. For each atomic argument, it performs (BREAK0 ATOM T '(TRACE ?= NIL GO))[1]

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

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

For example, (TRACE FOO1 (FOO2 Y)) will cause both FOO1 and FOO2 to be traced. All the arguments of FOO1 will be printed; only the value of Y will be printed for FOO2. In the special case that the user wants to see *only* the value,

---

[1]The ag TRACE is checked for in BREAK1 and causes the message "FUNCTION   :" to be printed instead of (FUNCTION   BROKEN).

he can perform (TRACE (FUNCTION)). This sets up a break with commands (TRACE ?= (NIL) GO).

Note: the user can always call BREAK0 himself 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.

(BREAKIN FN WHERE WHEN COMS)   [NLambda Function]
BREAKIN is an nlambda function. WHEN and COMS are similar to WHEN and COMS for BREAK0, except that if WHEN is NIL, T is used. WHERE specifies where in the definition of FN the call to BREAK1 is to be inserted (see below).

If FN is a compiled function, BREAKIN returns (FN UNBREAKABLE) as its value.

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.

BREAKIN enables the user to insert a break, i.e., a call to BREAK1, at a specified location in an interpreted function. For example, if FOO calls FIE, inserting a break in FOO before the call to FIE is similar to breaking FIE. However, 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.

The user specifies where the break is to be inserted by a sequence of editor commands. These commands are preceded by 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 the user to type in commands to the editor, locate the correct point, and verify it for himself using the P command if he desires, and exit from the editor with OK.[2] BREAKIN then inserts the break BEFORE, AFTER, or AROUND that point.

For BREAKIN BEFORE or AFTER, the break expression is NIL, since the value of the break is irrelevant. For breakin AROUND, the break expression will be the indicated form. In this case, the user can use the EVAL command to evaluate that form, and examine its value, before allowing the computation to proceed. For example, if the user inserted a break after a COND predicate, e.g., (AFTER (EQUAL X Y)), he would be powerless to alter the flow of computation if the predicate were not true, since the break would

---

[2]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).

not be reached. However, by breaking (AROUND (EQUAL X Y)), he can evaluate the break expression, i.e., (EQUAL X Y), look at its value, and return something else if he wished.

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 the user requests a break (AFTER X) in (PROG   (SETQ X &)   ), the break will actually be inserted AFTER (SETQ X &), and a message printed to this eect, e.g., BREAK INSERTED AFTER (SETQ X &).

(UNBREAK x)   [NLambda NoSpread Function]
> Nlambda nospread function. It takes an indenite number of functions modied 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 rst sets BRKINFOLST to NIL.
>
> (UNBREAK T) unbreaks just the rst 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 modied by BREAKIN, UNBREAKIN is called to edit it back to its original state. If FN was created from (FN1 IN FN2), (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 (front of) BRKINFOLST.
>
> Note: (UNBREAK0 '(FN1 IN FN2)) is allowed: UNBREAK0 will operate on (FN1 -IN-FN2) instead.

(UNBREAKIN FN)   [Function]
> Performs the appropriate editing operations to eliminate all changes made by BREAKIN. FN may be either the name or denition 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 rst break on BRKINFOLST, i.e., the function most recently unbroken.

(CHANGENAME FN FR OM TO) [Function]

Changes all occurrences of FR OM to TO in FN . FN may be compiled or blockcompiled. Value is FN if FR OM was found, otherwise NIL. Does not perform any modi cations of property lists. Note that FR OM and TO do not have to be functions, e.g., they can be names of variables, or any other literals.

(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 FL G= NIL, as for PRETTYPRINT, it does not modify the de nition of FN in the process of producing a ''clean'' version of the de nition; it works on a copy. If FL G= 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 de nition.

10.2   ADVISING

The operation of advising gives the user 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 de nition 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 the user to treat functions, his or someone else's, as ''black boxes,'' and to modify them without concern for their contents or details of operations. For example, the user 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 e ect a modi cation which only operates when a function is called from some other speci ed function, i.e., to 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 the user wanted TIME (page 14.14) to print the results of his measurements to the le FOO instead of the teletype. He could accomplish this by (ADVISE '((PRIN1 PRINT SPACES) IN TIME) 'BEFORE '(SETQQ U FOO))

Note that advising PRIN1, PRINT, or SPACES directly would have a ected all calls to these very frequently used function, whereas advising ((PRIN1 PRINT SPACES) IN TIME) a ects just those

calls to `PRIN1`, `PRINT`, and `SPACES` from `TIME`.

Advice can also be speci ed 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`. For example, suppose the user wanted to perform some computation following each `SYSIN`, e.g., check whether his les were up to date. He could then: `(ADVISE 'SYSOUT 'AFTER '(COND ((LISTP !VALUE) --)))`.[3]

10.2.1   Implementation of Advising

After a function has been modi ed 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 de nition. [4] Note that the structure of a function modi ed by `ADVISE` allows a piece of advice to bypass the original de nition 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 e ect, 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`) in order to enable advising these functions.

---

[3]After the `SYSIN`, the system will be as it was when the `SYSOUT` was performed, hence the advice must be to `SYSOUT`, not `SYSIN`. See page 14.3 for complete discussion of `SYSOUT`.

[4]If `FN` was originally an `EXPR`, `BODY` is the body of the de nition, otherwise a form using a `GENSYM` which is de ned with the original de nition.

10.2.2   Advise Functions

ADVISE is a function of four arguments: FN, WHEN, WHERE, and WHA T. FN is the function to be modi ed by advising, WHA T is the modi cation, 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 de nition. WHERE speci es 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 speci ed, ADVISE rst 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 WHA T), or a function of three arguments: (ADVISE FN WHEN WHA T), or a function of four arguments: (ADVISE FN WHEN WHERE WHA T). 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 WHA T)   [Function]
> FN is the function to be advised, WHEN = BEFORE, AFTER, or AROUND, WHERE speci es where in the advice list the advice is to be inserted, and WHA T is the piece of advice.
>
> If FN is of the form (FN1 IN FN2), FN1 is changed to FN1-IN-FN2 throughout FN2, as with break, and then FN1-IN-FN2 is used in place of FN. If FN1 and/or FN2 are lists, they are distributed as with BREAK0, page 10.3.
>
> If FN is broken, it is unbroken before advising.
>
> If FN is not de ned, an error is generated, NOT A FUNCTION.
>
> If FN is being advised for the rst 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 de ned with the original de nition of FN. An appropriate S-expression de nition is then created for FN.[5] Finally, FN is added to the (front of) ADVISEDFNS, so that (UNADVISE T) always unadvises the last function advised (see page 10.10).
>
> 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 de nition 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 rst piece of advice. Otherwise, WHERE is treated as a command for the editor, similar to BREAKIN, e.g., (BEFORE 3), (AFTER PRINT).

---

[5]Using private versions of PROG, SETQ, and RETURN, so that these functions can also be advised.

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 speci ed, earlier ones will be embedded inside later ones. The value of WHERE is ignored.

Finally (LIST WHEN WHERE WHA T) 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 de nition 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 WHA T. 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 rst 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 de nition 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 de nition or its property list.

(UNADVISE x)   [NLambda NoSpread Function]
An nlambda nospread like UNBREAK. It takes an inde nite 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 rst sets ADVINFOLST to NIL.

(UNADVISE T) unadvises the rst 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). In addition it 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 rst function on ADVINFOLST, i.e., the function most recently unadvised.

A di erence 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. Note that 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.

Advise Functions