19. DWIM

A surprisingly large percentage of the errors made by Interlisp users are of the type that could be corrected by another Lisp programmer without any information about the purpose of the program or expression in question, e.g., misspellings, certain kinds of parentheses errors, etc. To correct these types of errors we have implemented in Medley a DWIM facility, short for Do-What-I-Mean. DWIM is called automatically whenever an error occurs in the evaluation of an Interlisp expression. (Currently, DWIM only operates on unbound atoms and undefined function errors.) DWIM then proceeds to try to correct the mistake using the current context of computation plus information about what you had previously been doing (and what mistakes you had been making) as guides to the remedy of the error. If DWIM is able to make the correction, the computation continues as though no error had occurred. Otherwise, the procedure is the same as though DWIM had not intervened: a break occurs, or an unwind to the last ERRORSET (see Chapter 14). The following protocol illustrates the operation of DWIM.

For example, suppose you define the factorial function (FACT N) as follows:

```
_DEFINEQ((FACT (LAMBDA (N) (COND
((ZEROP N0 1) ((T (ITIMS N (FACCT 9SUB1 N]
(FACT)

_
```

Note that the definition of FACT contains several mistakes: ITIMES and FACT have been misspelled; the 0 in N0 was intended to be a right parenthesis, but the Shift key was not pressed; similarly, the 9 in 9SUB1 was intended to be a left parenthesis; and finally, there is an extra left parenthesis in front of the T that begins the final clause in the conditional.

```
_PRETTYPRNT((FACCT]
       =PRETTYPRINT
       =FACT

       (FACT
         [LAMBDA (N)
           (COND
             ((ZEROP N0 1)
               ((T (ITIMS N (FACCT 9SUB1 N])
       (FACT)


_
```

After defining FACT, you want to look at its definition using PRETTYPRINT, which you unfortunately misspell. Since there is no function PRETTYPRNT in the system, an undefined function error occurs, and DWIM is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by *this* user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, notifies you of this, and calls PRETTYPRINT.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since FACCT has no definition. Note that this is *not* an Interlisp error condition, so that DWIM would not be called as

described above. However, it is obviously not what you *meant*.

This sort of mistake is corrected by having PRETTYPRINT itself explicitly invoke the spelling corrector portion of DWIM whenever given a function with no EXPR definition. Thus, with the aid of DWIM PRETTYPRINT is able to determine that you want to see the definition of the function FACT, and proceeds accordingly.

```
_FACT(3]
       N0 [IN FACT] -> N ) ?  YES
       [IN FACT] (COND -- ((T --))) ->
                     (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES
       FACCT [IN FACT] -> FACT
       9SUB1 [IN FACT] ->  ( SUB1 ?  YES
       6


_PP FACT
       (FACT
         [LAMBDA (N)
           (COND
             ((ZEROP N)
               1)
             (T (ITIMES N (FACT (SUB1 N])
       FACT


_
```

You now call FACT. During its execution, five errors occur, and DWIM is called five times. At each point, the error is corrected, a message is printed describing the action taken, and the computation is allowed to continue as if no error had occurred. Following the last correction, 6 is printed, the value of (FACT 3). Finally, you prettyprint the new, now correct, definition of FACT.

In this particular example, you were operating in TRUSTING mode, which gives DWIM carte blanche for most corrections. You can also operate in CAUTIOUS mode, in which case DWIM will inform you of intended corrections before they are made, and allow you to approve or disapprove of them. If DWIM was operating in CAUTIOUS mode in the example above, it would proceed as follows:

```
_FACT(3)
       N0 [IN FACT] -> N ) ?  YES
       U.D.F. T [IN FACT]   FIX?   YES
       [IN FACT] (COND -- ((T --))) ->
                     (COND -- (T --))
       ITIMS [IN FACT] -> ITIMES ?   ...YES
       FACCT [IN FACT] -> FACT ?   ...YES
       9SUB1 [IN FACT] ->  ( SUB1 ?  NO
       U.B.A.
       (9SUB1 BROKEN)
       :
```

For most corrections, if you do not respond in a specified interval of time, DWIM automatically proceeds with the correction, so that you need intervene only when you do not approve. In the

example, you responded to the first, second, and fifth questions; DWIM responded for you on the third and fourth.

DWIM uses ASKUSER for its interactions with you (see Chapter 26). Whenever an interaction is about to take place and you have typed ahead, ASKUSER types several bells to warn you to stop typing, then clears and saves the input buffers, restoring them after the interaction is complete. Thus if you typed ahead before a DWIM interaction, DWIM will not confuse your type-ahead with the answer to its question, nor will your type-ahead be lost. The bells are printed by the function PRINTBELLS, which can be advised or redefined for specialized applications, e.g. to flash the screen for a display terminal.

A great deal of effort has gone into making DWIM "smart", and experience with a large number of users indicates that DWIM works very well; DWIM seldom fails to correct an error you feel it should have, and almost never mistakenly corrects an error. However, it is important to note that even when DWIM *is* wrong, no harm is done: since an error had occurred, you would have had to intervene anyway if DWIM took no action. Thus, if DWIM mistakenly corrects an error, you simply interrupt or abort the computation, reverse the DWIM change using UNDO (see Chapter 13), and make the correction you would have had to make without DWIM. An exception is if DWIM's correction mistakenly caused a destructive computation to be initiated, and information was lost before you could interrupt. We have not yet had such an incident occur.

(**DWIM** *X*) [Function]

Used to enable/disable DWIM. If *X* is the symbol C, DWIM is enabled in CAUTIOUS mode, so that DWIM will ask you before making corrections. If *X* is T, DWIM is enabled in TRUSTING mode, so DWIM will make most corrections automatically. If *X* is NIL, DWIM is disabled. Medley initially has DWIM enabled in CAUTIOUS mode.

DWIM returns CAUTIOUS, TRUSTING or NIL, depending to what mode it has just been put into.

For corrections to expressions typed in for immediate execution (typed into LISPX, Chapter 13), DWIM always acts as though it were in TRUSTING mode, i.e., no approval necessary. For certain types of corrections, e.g., run-on spelling corrections, 9-0 errors, etc., DWIM always acts like it was in CAUTIOUS mode, and asks for approval. In either case, DWIM always informs you of its action as described below.

Spelling Correction Protocol

One type of error that DWIM can correct is the misspelling of a function or a variable name. When an unbound symbol or undefined function error occurs, DWIM tries to correct the spelling of the bad symbol. If a symbol is found whose spelling is "close" to the offender, DWIM proceeds as follows:

If the correction occurs in the typed-in expression, DWIM prints =*CORRECT-SPELLING* and continues evaluating the expression. For example:

```
_(SETQ FOO (IPLUSS 1 2))
```

```
=IPLUS
3
```

If the correction does not occur in type-in, DWIM prints

BAD-SPELLING [IN *FUNCTION-NAME*] -> *CORRECT-SPELLING*

The appearance of -> is to call attention to the fact that the user's function will be or has been changed.

Then, if DWIM is in TRUSTING mode, it prints a carriage return, makes the correction, and continues the computation. If DWIM is in CAUTIOUS mode, it prints a few spaces and ? and then wait for approval. The user then has six options:

1. Type Y. DWIM types es, and proceeds with the correction.
2. Type N. DWIM types o, and does not make the correction.
3. Type ^. DWIM does not make the correction, and furthermore guarantees that the error will not cause a break.
4. Type Control-E. For error correction, this has the same effect as typing N.
5. Do nothing. In this case DWIM waits for DWIMWAIT seconds, and if you have not responded, DWIM will type ... followed by the default answer.

   The default on spelling corrections is determined by the value of the variable FIXSPELLDEFAULT, whose top level value is initially Y.
6. Type space or carriage-return. In this case DWIM will wait indefinitely. This option is intended for those cases where you want to think about your answer, and want to insure that DWIM does not get "impatient" and answer for you.

The procedure for spelling correction on other than Interlisp errors is analogous. If the correction is being handled as type-in, DWIM prints = followed by the correct spelling, and returns it to the function that called DWIM. Otherwise, DWIM prints the incorrect spelling, followed by the correct spelling. Then, if DWIM is in TRUSTING mode, DWIM prints a carriage-return and returns the correct spelling. Otherwise, DWIM prints a few spaces and a ? and waits for approval. You can then respond with Y, N, Control-E, space, carriage return, or do nothing as described above.

The spelling corrector itself is not ERRORSET protected like the DWIM error correction routines. Therefore, typing N and typing Control-E may have different effects when the spelling corrector is called directly. The former simply instructs the spelling corrector to return NIL, and lets the calling function decide what to do next; the latter causes an error which unwinds to the last ERRORSET, however far back that may be.

Parentheses Errors Protocol

When an unbound symbol or undefined error occurs, and the offending symbol contains 9 or 0,

DWIM tries to correct errors caused by typing 9 for left parenthesis and 0 for right parenthesis. In these cases, the interaction with you is similar to that for spelling correction. If the error occurs in type-in, DWIM types =*CORRECTION*, and continues evaluating the expression. For example:

```
_(SETQ FOO 9IPLUS 1 2]
       = ( IPLUS
       3
```

If the correction does not occur in type-in, DWIM prints

BAD-ATOM [IN *FUNCTION-NAME*] -> *CORRECTION* ?

and then waits for approval. You then have the same six options as for spelling correction, except the waiting time is 3*DWIMWAIT seconds. If you type Y, DWIM operates as if it were in TRUSTING mode, i.e., it makes the correction and prints its message.

Actually, DWIM uses the value of the variables LPARKEY and RPARKEY to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY are initially 9 and 0 respectively, but they can be reset for other keyboard layouts, e.g., on some terminals left parenthesis is over 8, and right parenthesis is over 9.

Undefined Function T Errors

When an undefined function error occurs, and the offending function is T, DWIM tries to correct certain types of parentheses errors involving a T clause in a conditional. DWIM recognizes errors of the following forms:

(COND --) (T --)	The T clause appears outside and immediately following the COND.
(COND -- (-- & (T --)))	The T clause appears inside a previous clause.
(COND -- ((T --)))	The T clause has an extra pair of parentheses around it.

For undefined function errors that are not one of these three types, DWIM takes no corrective action at all, and the error will occur.

If the error occurs in type-in, DWIM simply types T FIXED and makes the correction. Otherwise if DWIM is in TRUSTING mode, DWIM makes the correction and prints the message:

```
[IN FUNCTION-NAME] {BAD-COND} ->
            {CORRECTED-COND}
```

If DWIM is in CAUTIOUS mode, DWIM prints

```
UNDEFINED FUNCTION T
       [IN FUNCTION-NAME]   FIX?
```

and waits for approval. You then have the same options as for spelling corrections and parenthesis errors. If you type Y or default, DWIM makes the correction and prints its message.

Having made the correction, DWIM must then decide how to proceed with the computation. In the first case, (COND --) (T --), DWIM cannot know whether the T clause would have been executed if it had been inside of the COND. Therefore DWIM asks you CONTINUE WITH T CLAUSE (with a default of YES). If you type N, DWIM continues with the form after the COND, i.e., the form that originally followed the T clause.

In the second case, (COND -- (-- & (T --))), DWIM has a different problem. After moving the T clause to its proper place, DWIM must return as the value of & as the value of the COND. Since this value is no longer around, DWIM asks you OK TO REEVALUATE and then prints the expression corresponding to &. If you type Y, or default, DWIM continues by reevaluating &, otherwise DWIM aborts, and a U.D.F. T error will then occur (even though the COND has in fact been fixed). If DWIM can determine for itself that the form can safely be reevaluated, it does not consult you before reevaluating. DWIM can do this if the form is atomic, or CAR of the form is a member of the list OKREEVALST, and each of the arguments can safely be reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

In the third case, (COND -- ((T --))), there is no problem with continuation, so no further interaction is necessary.

DWIM Operation

Whenever the interpreter encounters an atomic form with no binding, or a non-atomic form CAR of which is not a function or function object, it calls the function FAULTEVAL. Similarly, when APPLY is given an undefined function, FAULTAPPLY is called. When DWIM is enabled, FAULTEVAL and FAULTAPPLY are redefined to first call the DWIM package, which tries to correct the error. If DWIM cannot decide how to fix the error, or you disapprove of DWIM's correction (by typing N), or you type Control-E, then FAULTEVAL and FAULTAPPLY cause an error or break. If you type ^ to DWIM, DWIM exits by performing (RETEVAL 'FAULTEVAL '(ERROR!)), so that an error will be generated at the position of the call to FAULTEVAL.

If DWIM can (and is allowed to) correct the error, it exits by performing RETEVAL of the corrected form, as of the position of the call to FAULTEVAL or FAULTAPPLY. Thus in the example at the beginning of the chapter, when DWIM determined that ITIMS was ITIMES misspelled, DWIM called RETEVAL with (ITIMES N (FACCT 9SUB1 N)). Since the interpreter uses the value returned by FAULTEVAL exactly as though it were the value of the erroneous form, the computation will thus proceed exactly as though no error had occurred.

In addition to continuing the computation, DWIM also repairs the cause of the error whenever possible;

in the above example, DWIM also changed (with RPLACA) the expression (ITIMS N (FACCT 9SUB1 N)) that caused the error. Note that if your program had *computed* the form and called EVAL, it would not be possible to repair the cause of the error, although DWIM could correct the misspelling each time it occurred.

Error correction in DWIM is divided into three categories: unbound atoms, undefined CAR of form, and undefined function in APPLY. Assuming that the user approves DWIM's corrections, the action taken by DWIM for the various types of errors in each of these categories is summarized below.

DWIM Correction: Unbound Atoms

If DWIM is called as the result of an unbound atom error, it proceeds as follows:

1. If the first character of the unbound atom is ', DWIM assumes that you (intentionally) typed '*ATOM* for (QUOTE *ATOM*) and makes the appropriate change. No message is typed, and no approval is requested.

   If the unbound atom is just ' itself, DWIM assumes you want the *next* expression quoted, e.g., (CONS X '(A B C)) will be changed to (CONS X (QUOTE (A B C))). Again no message will be printed or approval asked. If no expression follows the ', DWIM gives up.

   > Note: ' is normally defined as a read-macro character which converts 'FOO to (QUOTE FOO) on input, so DWIM will not see the ' in the case of expressions that are typed-in.

2. If CLISP (see Chapter 21) is enabled, and the atom is part of a CLISP construct, the CLISP transformation is performed and the result returned. For example, N-1 is transformed to (SUB1 N), and (... FOO←3 ...) is transformed into (... (SETQ FOO 3) ...).

3. If the atom contains an 9 (actually LPARKEY (see the DWIM Functions and Variables section below), DWIM assumes the **9** was intended to be a left parenthesis, and calls the editor to make appropriate repairs on the expression containing the atom. DWIM assumes that you did not notice the mistake, i.e., that the entire expression was affected by the missing left parenthesis. For example, if you type (SETQ X (LIST (CONS 9CAR Y) (CDR Z)) Y), the expression will be changed to (SETQ X (LIST (CONS (CAR Y) (CDR Z)) Y)). The 9 does not have to be the first character of the atom: DWIM will handle (CONS X9CAR Y) correctly.

4. If the atom contains a 0 (actually RPARKEY, see the DWIM Functions and Variables section below), DWIM assumes the 0 was intended to be a right parenthesis and operates as in the case above.

5. If the atom begins with a 7, the 7 is treated as a '. For example, 7FOO becomes 'FOO, and then (QUOTE FOO).

6. The expressions on DWIMUSERFORMS (see the DWIMUSERFORMS section below) are evaluated in the order that they appear. If any of these expressions returns a non-NIL value, this value is treated as the form to be used to continue the computation, it is

evaluated and its value is returned by DWIM.

7. If the unbound atom occurs in a function, DWIM attempts spelling correction using the LAMBDA and PROG variables of the function as the spelling list.

8. If the unbound atom occurred in a type-in to a break, DWIM attempts spelling correction using the LAMBDA and PROG variables of the broken function as the spelling list.

9. Otherwise, DWIM attempts spelling correction using SPELLINGS3 (see the Spelling Lists section below).

10. If all of the above fail, DWIM gives up.

Undefined CAR of Form

If DWIM is called as the result of an undefined CAR of form error, it proceeds as follows:

1. If CAR of the form is T, DWIM assumes a misplaced T clause and operates as described in the Undefined Function T Errors section above.

2. If CAR of the form is F/L, DWIM changes the "F/L" to "FUNCTION(LAMBDA". For example, (F/L (Y) (PRINT (CAR Y))) is changed to (FUNCTION (LAMBDA (Y) (PRINT (CAR Y)))). No message is printed and no approval requested. If you omit the variable list, DWIM supplies (X), e.g., (F/L (PRINT (CAR X))) is changed to (FUNCTION (LAMBDA (X) (PRINT (CAR X)))). DWIM determines that you have supplied the variable list when more than one expression follows F/L, CAR of the first expression is not the name of a function, and every element in the first expression is atomic. For example, DWIM will supply (X) when correcting (F/L (PRINT (CDR X)) (PRINT (CAR X))).

3. If CAR of the form is a CLISP word (IF, FOR, DO, FETCH, etc.), the indicated CLISP transformation is performed, and the result is returned as the corrected form. See Chapter 21.

4. If CAR of the form has a function definition, DWIM attempts spelling correction on CAR of the definition using as spelling list the value of LAMBDASPLST, initially (LAMBDA NLAMBDA).

5. If CAR of the form has an EXPR or CODE property, DWIM prints *CAR-OF-FORM* UNSAVED, performs an UNSAVEDEF, and continues. No approval is requested.

6. If CAR of the form has a FILEDEF property, the definition is loaded from a file (except when DWIMIFYing). If the value of the property is atomic, the entire file is to be loaded. If the value is a list, CAR is the name of the file and CDR the relevant functions, and LOADFNS will be used. For both cases, *LDFLG* will be SYSLOAD (see Chapter 17). DWIM uses FINDFILE (Chapter 24), so that the file can be on any of the directories on DIRECTORIES, initially (NIL NEWLISP LISP LISPUSERS). If the file is found, DWIM types SHALL I LOAD followed by the file name or list of functions. If you approve, DWIM loads the function(s) or file, and continues the computation.

7. If CLISP is enabled, and CAR of the form is part of a CLISP construct, the indicated

transformation is performed, e.g., (N_N-1) becomes (SETQ N (SUB1 N)).

8. If CAR of the form contains an 9, DWIM assumes a left parenthesis was intended e.g., (CONS9CAR X).
9. If CAR of the form contains a 0, DWIM assumes a right parenthesis was intended.
10. If CAR of the form is a list, DWIM attempts spelling correction on CAAR of the form using LAMBDASPLST as spelling list. If successful, DWIM returns the corrected expression itself.
11. The expressions on DWIMUSERFORMS are evaluated in the order they appear. If any returns a non-NIL value, this value is treated as the corrected form, it is evaluated, and DWIM returns its value.
12. Otherwise, DWIM attempts spelling correction using SPELLINGS2 as the spelling list (see the Spelling Lists section below). When DWIMIFYing, DWIM also attemps spelling correction on function names not defined but previously encountered, using NOFIXFNSLST as a spelling list (see Chapter 21).
13. If all of the above fail, DWIM gives up.

Undefined Function in APPLY

If DWIM is called as the result of an undefined function in APPLY error, it proceeds as follows:

1. If the function has a definition, DWIM attempts spelling correction on CAR of the definition using LAMBDASPLST as spelling list.
2. If the function has an EXPR or CODE property, DWIM prints *FN* UNSAVED, performs an UNSAVEDEF and continues. No approval is requested.
3. If the function has a property FILEDEF, DWIM proceeds as in case 6 of undefined CAR of form.
4. If the error resulted from type-in, and CLISP is enabled, and the function name contains a CLISP operator, DWIM performs the indicated transformation, e.g., type FOO_(APPEND FIE FUM).
5. If the function name contains an 9, DWIM assumes a left parenthesis was intended, e.g., EDIT9FOO].
6. If the "function" is a list, DWIM attempts spelling correction on CAR of the list using LAMBDASPLST as spelling list.
7. The expressions on DWIMUSERFORMS are evaluated in the order they appear, and if any returns a non-NIL value, this value is treated as the function used to continue the computation, i.e., it will be applied to its arguments.
8. DWIM attempts spelling correction using SPELLINGS1 as the spelling list.
9. DWIM attempts spelling correction using SPELLINGS2 as the spelling list.
10. If all fail, DWIM gives up.

DWIMUSERFORMS

The variable DWIMUSERFORMS provides a convenient way of adding to the transformations that DWIM performs. For example, you might want to change atoms of the form $X to (QA4LOOKUP X). Before attempting spelling correction, but after performing other transformations (F/L, 9, 0, CLISP, etc.), DWIM evaluates the expressions on DWIMUSERFORMS in the order they appear. If any expression returns a non-NIL value, this value is treated as the transformed form to be used. If DWIM was called from FAULTEVAL, this form is evaluated and the resulting value is returned as the value of FAULTEVAL. If DWIM is called from FAULTAPPLY, this form is treated as a function to be applied to FAULTARGS, and the resulting value is returned as the value of FAULTAPPLY. If all of the expressions on DWIMUSERFORMS return NIL, DWIM proceeds as though DWIMUSERFORMS = NIL, and attempts spelling correction. Note that DWIM simply takes the value and returns it; the expressions on DWIMUSERFORMS are responsible for making any modifications to the original expression. The expressions on DWIMUSERFORMS should make the transformation permanent, either by associating it with FAULTX via CLISPTRAN, or by destructively changing FAULTX.

In order for an expression on DWIMUSERFORMS to be able to be effective, it needs to know various things about the context of the error. Therefore, several of DWIM's internal variables have been made SPECVARS (see Chapter 18) and are therefore "visible" to DWIMUSERFORMS. Below are a list of those variables that may be useful.

FAULTX [Variable]

For unbound atom and undefined car of form errors, FAULTX is the atom or form. For undefined function in APPLY errors, FAULTX is the name of the function.

FAULTARGS [Variable]

For undefined function in APPLY errors, FAULTARGS is the list of arguments. FAULTARGS may be modified or reset by expressions on DWIMUSERFORMS.

FAULTAPPLYFLG [Variable]

Value is T for undefined function in APPLY errors; NIL otherwise. The value of FAULTAPPLYFLG *after* an expression on DWIMUSERFORMS returns a non-NIL value determines how the latter value is to be treated. Following an undefined function in APPLY error, if an expression on DWIMUSERFORMS sets FAULTAPPLYFLG to NIL, the value returned is treated as a form to be evaluated, rather than a function to be applied.

FAULTAPPLYFLG is necessary to distinguish between unbound atom and undefined function in APPLY errors, since FAULTARGS may be NIL and FAULTX atomic in both cases.

TAIL [Variable]

For unbound atom errors, TAIL is the tail of the expression CAR of which is the unbound atom. DWIMUSERFORMS expression can replace the atom by another expression by performing (/RPLACA TAIL *EXPR*)

PARENT [Variable]

For unbound atom errors, PARENT is the form in which the unbound atom appears. TAIL is a tail of PARENT.

TYPE-IN? [Variable]

True if the error occurred in type-in.

FAULTFN [Variable]

Name of the function in which error occurred. FAULTFN is TYPE-IN when the error occurred in type-in, and EVAL or APPLY when the error occurred under an explicit call to EVAL or APPLY.

DWIMIFYFLG [Variable]

True if the error was encountered while DWIMIFYing (as opposed to happening while running a program).

EXPR [Variable]

Definition of FAULTFN, or argument to EVAL, i.e., the superform in which the error occurs.

The initial value of DWIMUSERFORMS is ((DWIMLOADFNS?)). DWIMLOADFNS? is a function for automatically loading functions from files. If DWIMLOADFNSFLG is T (its initial value), and CAR of the form is the name of a function, and the function is contained on a file that has been noticed by the file package, the function is loaded, and the computation continues.

DWIM Functions and Variables

DWIMWAIT [Variable]

Value is the number of seconds that DWIM will wait before it assumes that you are not going to respond to a question and uses the default response FIXSPELLDEFAULT.

DWIM operates by dismissing for 250 milliseconds, then checking to see if anything has been typed. If not, it dismisses again, etc. until DWIMWAIT seconds have elapsed. Thus, there will be a delay of at most 1/4 second before DWIM responds to your answer.

FIXSPELLDEFAULT [Variable]

If approval is requested for a spelling correction, and you do not respond, defaults to value of FIXSPELLDEFAULT, initially Y. FIXSPELLDEFAULT is rebound to N when DWIMIFYing.

ADDSPELLFLG [Variable]

If NIL, suppresses calls to ADDSPELL. Initially T.

NOSPELLFLG [Variable]

If T, suppresses *all* spelling correction. If some other non-NIL value, suppresses spelling correction in programs but not type-in. NOSPELLFLG is initially NIL. It is rebound to T when compiling from a file.

RUNONFLG [Variable]

If NIL, suppresses run-on spelling corrections. Initially NIL.

DWIMLOADFNSFLG [Variable]

If T, tells DWIM that when it encounters a call to an undefined function contained on a file that has been noticed by the file package, to simply load the function. DWIMLOADFNSFLG is initially T (see above).

LPARKEY [Variable]
RPARKEY [Variable]

DWIM uses the value of the variables LPARKEY and RPARKEY (initially 9 and 0 respectively) to determine the corresponding lower case character for left and right parentheses. LPARKEY and RPARKEY can be reset for other keyboard layouts. For example, on some terminals left parenthesis is over 8, and right parenthesis is over 9.

OKREEVALST [Variable]

The value of OKREEVALST is a list of functions that DWIM can safely reevaluate. If a form is atomic, or CAR of the form is a member of OKREEVALST, and each of the arguments can safely be reevaluated, then the form can be safely reevaluated. For example, (SETQ X (CONS (IPLUS Y Z) W)) is safe to reevaluate because SETQ, CONS, and IPLUS are all on OKREEVALST.

DWIMFLG [Variable]

DWIMFLG = NIL, all DWIM operations are disabled. (DWIM 'C) and (DWIM T) set DWIMFLG to T; (DWIM NIL) sets DWIMFLG to NIL.

APPROVEFLG [Variable]

APPROVEFLG = T if DWIM should ask the user for approval before making a correction that will modify the definition of one of his functions; NIL otherwise.

When DWIM is put into CAUTIOUS mode with (DWIM 'C), APPROVEFLG is set to T; for TRUSTING mode, APPROVEFLG is set to NIL.

LAMBDASPLST [Variable]

DWIM uses the value of LAMBDASPLST as the spelling list when correcting "bad" function definitions. Initially (LAMBDA NLAMBDA). You may wish to add to LAMBDASPLST if you elect to define new "function types" via an appropriate DWIMUSERFORMS entry. For example, the QLAMBDAs of SRI's QLISP are handled in this way.

Spelling Correction

The spelling corrector is given as arguments a misspelled word (word means symbol), a spelling list (a list of words), and a number: *XWORD*, *SPLST*, and *REL* respectively. Its task is to find that word on *SPLST* which is closest to *XWORD*, in the sense described below. This word is called a *respelling* of *XWORD*. *REL* specifies the minimum "closeness" between *XWORD* and a respelling. If the spelling corrector cannot find a word on *SPLST* closer to *XWORD* than *REL*, or if it finds two or more words equally close, its value is NIL, otherwise its value is the respelling. The spelling corrector can also be given an optional functional argument, *FN*, to be used for selecting out a subset of *SPLST*, i.e., only those members of *SPLST* that satisfy *FN* will be considered as possible respellings.

The exact algorithm for computing the spelling metric is described later, but briefly "closeness" is inversely proportional to the number of disagreements between the two words, and directly proportional to the length of the longer word. For example, PRTTYPRNT is "closer" to PRETTYPRINT than CS is to CONS even though both pairs of words have the same number of disagreements. The spelling corrector operates by proceeding down *SPLST*, and computing the closeness between each word and *XWORD*, and keeping a list of those that are closest. Certain differences between words are not counted as disagreements, for example a single transposition, e.g., CONS to CNOS, or a doubled letter, e.g., CONS to CONSS, etc. In the event that the spelling corrector finds a word on *SPLST* with *no* disagreements, it will stop searching and return this word as the respelling. Otherwise, the spelling corrector continues through the entire spelling list. Then if it has found one and only one "closest" word, it returns this word as the respelling. For example, if *XWORD* is VONS, the spelling corrector will probably return CONS as the respelling. However, if *XWORD* is CONZ, the spelling corrector will not be able to return a respelling, since CONZ is equally close to both CONS and COND. If the spelling corrector finds an acceptable respelling, it interacts with you as described earlier.

In the special case that the misspelled word contains one or more $s (escape), the spelling corrector searches for those words on *SPLST* that match *XWORD*, where a $ can match any number of characters (including 0), e.g., FOO$ matches FOO1 and FOO, but not NEWFOO. FOO matches all three. Both completion and correction may be involved, e.g. RPETTY$ will match PRETTYPRINT, with one mistake. The entire spelling list is always searched, and if more than one respelling is found, the spelling corrector prints AMBIGUOUS, and returns NIL. For example, CON$ would be ambiguous if both CONS and COND were on the spelling list. If the spelling corrector finds one and only one respelling, it interacts with you as described earlier.

For both spelling correction and spelling completion, regardless of whether or not you approve of the spelling corrector's choice, the respelling is moved to the front of *SPLST*. Since many respellings are of the type with no disagreements, this procedure has the effect of considerably reducing the time required to correct the spelling of frequently misspelled words.

Synonyms

Spelling lists also provide a way of defining synonyms for a particular context. If a dotted pair appears on a spelling list (instead of just an atom), CAR is interpreted as the correct spelling of the misspelled word, and CDR as the antecedent for that word. If CAR is *identical* with the misspelled word, the antecedent is returned without any interaction or approval being necessary. If the

misspelled word *corrects* to CAR of the dotted pair, the usual interaction and approval will take place, and then the antecedent, i.e., CDR of the dotted pair, is returned. For example,you could make IFLG synonymous with CLISPIFTRANFLG by adding (IFLG . CLISPIFTRANFLG) to SPELLINGS3, the spelling list for unbound atoms. Similarly, you could make OTHERWISE mean the same as ELSEIF by adding (OTHERWISE . ELSEIF) to CLISPIFWORDSPLST, or make L be synonymous with LAMBDA by adding (L . LAMBDA) to LAMBDASPLST. You can also use L as a variable without confusion, since the association of L with LAMBDA occurs only in the appropriate context.

Spelling Lists

Any list of atoms can be used as a spelling list, e.g., BROKENFNS, FILELST, etc. Various system packages have their own spellings lists, e.g., LISPXCOMS, CLISPFORWORDSPLST, EDITCOMSA, etc. These are documented under their corresponding sections, and are also indexed under "spelling lists." In addition to these spelling lists, the system maintains, i.e., automatically adds to, and occasionally prunes, four lists used solely for spelling correction: SPELLINGS1, SPELLINGS2, SPELLINGS3, and USERWORDS. These spelling lists are maintained *only* when ADDSPELLFLG is non-NIL. ADDSPELLFLG is initially T.

SPELLINGS1 [Variable]

SPELLINGS1 is a list of functions used for spelling correction when an input is typed in apply format, and the function is undefined, e.g., EDTIF(FOO). SPELLINGS1 is initialized to contain DEFINEQ, BREAK, MAKEFILE, EDITF, TCOMPL, LOAD, etc. Whenever LISPX is given an input in apply format, i.e., a function and arguments, the name of the function is added to SPELLINGS1 if the function has a definition.

For example, typing CALLS(EDITF) will cause CALLS to be added to SPELLINGS1. Thus if you typed CALLS(EDITF) and later typed CALLLS(EDITV), since SPELLINGS1 would then contain CALLS, DWIM would be successful in correcting CALLLS to CALLS.

SPELLINGS2 [Variable]

SPELLINGS2 is a list of functions used for spelling correction for all other undefined functions. It is initialized to contain functions such as ADD1, APPEND, COND, CONS, GO, LIST, NCONC, PRINT, PROG, RETURN, SETQ, etc. Whenever LISPX is given a non-atomic form, the name of the function is added to SPELLINGS2. For example, typing (RETFROM (STKPOS (QUOTE FOO) 2)) to a break would add RETFROM to SPELLINGS2. Function names are also added to SPELLINGS2 by DEFINE, DEFINEQ, LOAD (when loading compiled code), UNSAVEDEF, EDITF, and PRETTYPRINT.

SPELLINGS3 [Variable]

SPELLINGS3 is a list of words used for spelling correction on all unbound atoms. SPELLINGS3 is initialized to EDITMACROS, BREAKMACROS, BROKENFNS, and ADVISEDFNS. Whenever LISPX is given an atom to evaluate, the name of the atom is added to SPELLINGS3 if the atom has a value. Atoms are also added to SPELLINGS3 whenever they are edited by EDITV, and whenever they are set via RPAQ or RPAQQ. For example, when a file is loaded, all of the variables set in the file are added to SPELLINGS3. Atoms are also added to SPELLINGS3 when they are set by a LISPX input,

e.g., typing `(SETQ FOO (REVERSE (SETQ FIE ...)))` will add both `FOO` and `FIE` to `SPELLINGS3`.

USERWORDS [Variable]

`USERWORDS` is a list containing both functions and variables that you have *referred* to, e.g., by breaking or editing. `USERWORDS` is used for spelling correction by `ARGLIST`, `UNSAVEDEF`, `PRETTYPRINT`, `BREAK`, `EDITF`, `ADVISE`, etc. `USERWORDS` is initially `NIL`. Function names are added to it by `DEFINE`, `DEFINEQ`, `LOAD`, (when loading compiled code, or loading exprs to property lists) `UNSAVEDEF`, `EDITF`, `EDITV`, `EDITP`, `PRETTYPRINT`, etc. Variable names are added to `USERWORDS` at the same time as they are added to `SPELLINGS3`. In addition, the variable `LASTWORD` is always set to the last word added to `USERWORDS`, i.e., the last function or variable referred to by the user, and the respelling of `NIL` is defined to be the value of `LASTWORD`. Thus, if you had just defined a function, you can then prettyprint it by typing `PP()`.

Each of the above four spelling lists are divided into two sections separated by a special marker (the value of the variable `SPELLSTR1`). The first section contains the "permanent" words; the second section contains the temporary words. New words are added to the corresponding spelling list at the front of its temporary section (except that functions added to `SPELLINGS1` or `SPELLINGS2` by `LISPX` are always added to the end of the permanent section. If the word is already in the temporary section, it is moved to the front of that section; if the word is in the permanent section, no action is taken. If the length of the temporary section then exceeds a specified number, the last (oldest) word in the temporary section is forgotten, i.e., deleted. This procedure prevents the spelling lists from becoming cluttered with unimportant words that are no longer being used, and thereby slowing down spelling correction time. Since the spelling corrector usually moves each word selected as a respelling to the front of its spelling list, the word is thereby moved into the permanent section. Thus once a word is misspelled and corrected, it is considered important and will never be forgotten.

The spelling correction algorithm will not alter a spelling list unless it contains the special marker (the value of `SPELLSTR1`). This provides a way to ensure that a spelling list will not be altered.

#SPELLINGS1 [Variable]
#SPELLINGS2 [Variable]
#SPELLINGS3 [Variable]
#USERWORDS [Variable]

The maximum length of the temporary section for `SPELLINGS1`, `SPELLINGS2`, `SPELLINGS3` and `USERWORDS` is given by the value of `#SPELLINGS1`, `#SPELLINGS2`, `#SPELLINGS3`, and `#USERWORDS`, initialized to 30, 30, 30, and 60 respectively.

You can alter these values to modify the performance behavior of spelling correction.

Generators for Spelling Correction

For some applications, it is more convenient to *generate* candidates for a respelling one by one, rather than construct a complete list of all possible candidates, e.g., spelling correction involving a large directory of files, or a natural language data base. For these purposes, *SPLST* can be an array (of any size). The first element of this array is the generator function, which is called with the array itself as

its argument. Thus the function can use the remainder of the array to store "state" information, e.g., the last position on a file, a pointer into a data structure, etc. The value returned by the function is the next candidate for respelling. If NIL is returned, the spelling "list" is considered to be exhausted, and the closest match is returned. If a candidate is found with no disagreements, it is returned immediately without waiting for the "list" to exhaust.

SPLST can also be a generator, i.e. the value of the function GENERATOR (Chapter 11). The generator *SPLST* will be started up whenever the spelling corrector needs the next candidate, and it should return candidates via the function PRODUCE. For example, the following could be used as a "spelling list" which effectively contains all functions in the system:

```
[GENERATOR
       (MAPATOMS (FUNCTION (LAMBDA (X) (if (GETD X) then (PRODUCE
X]
```

Spelling Corrector Algorithm

The basic philosophy of DWIM spelling correction is to count the number of disagreements between two words, and use this number divided by the length of the longer of the two words as a measure of their relative disagreement. One minus this number is then the relative agreement or closeness. For example, CONS and CONX differ only in their last character. Such substitution errors count as one disagreement, so that the two words are in 75% agreement. Most calls to the spelling corrector specify a relative agreement of 70, so that a single substitution error is permitted in words of four characters or longer. However, spelling correction on shorter words is possible since certain types of differences such as single transpositions are not counted as disagreements. For example, AND and NAD have a relative agreement of 100. Calls to the spelling corrector from DWIM use the value of FIXSPELLREL, which is initially 70. Note that by setting FIXSPELLREL to 100, only spelling corrections with "zero" mistakes, will be considered, e.g., transpositions, double characters, etc.

The central function of the spelling corrector is CHOOZ. CHOOZ takes as arguments: a word, a minimum relative agreement, a spelling list, and an optional functional argument, *XWORD*, *REL*, *SPLST*, and *FN* respectively.

CHOOZ proceeds down *SPLST* examining each word. Words not satisfying *FN* (if *FN* is non-NIL), or those obviously too long or too short to be sufficiently close to *XWORD* are immediately rejected. For example, if *REL* = 70, and *XWORD* is 5 characters long, words longer than 7 characters will be rejected.

Special treatment is necessary for words shorter than *XWORD*, since doubled letters are not counted as disagreements. For example, CONNSSS and CONS have a relative agreement of 100. CHOOZ handles this by counting the number of doubled characters in *XWORD* before it begins scanning *SPLST*, and taking this into account when deciding whether to reject shorter words.

If *TWORD*, the current word on *SPLST*, is not rejected, CHOOZ computes the number of disagreements between it and *XWORD* by calling a subfunction, SKOR.

SKOR operates by scanning both words from left to right one character at a time. SKOR operates on the

list of character codes for each word. This list is computed by CHOOZ before calling SKOR. Characters are considered to agree if they are the same characters or appear on the same key (i.e., a shift mistake). The variable SPELLCASEARRAY is a CASEARRAY which is used to determine equivalence classes for this purpose. It is initialized to equivalence lowercase and upper case letters, as well as the standard key transitions: for example, 1 with !, 3 with #, etc.

If the first character in *XWORD* and *TWORD* do *not* agree, SKOR checks to see if either character is the same as one previously encountered, and not accounted-for at that time. (In other words, transpositions are not handled by lookahead, but by *lookback*.) A displacement of two or fewer positions is counted as a tranposition; a displacement by more than two positions is counted as a disagreement.In either case, both characters are now considered as accounted for and are discarded, and SKORing continues.

If the first character in *XWORD* and *TWORD* do not agree, and neither agree with previously unaccounted-for characters, and *TWORD* has more characters remaining than *XWORD*, SKOR removes and saves the first character of *TWORD*, and continues by comparing the rest of *TWORD* with *XWORD* as described above. If *TWORD* has the same or fewer characters remaining than *XWORD*, the procedure is the same except that the character is removed from *XWORD*. In this case, a special check is first made to see if that character is equal to the *previous* character in *XWORD*, or to the *next* character in *XWORD*, i.e., a double character typo, and if so, the character is considered accounted-for, and not counted as a disagreement. In this case, the "length" of *XWORD* is also decremented. Otherwise making *XWORD* sufficiently long by adding double characters would make it be arbitrarily close to *TWORD*, e.g., XXXXXX would correct to PP.

When SKOR has finished processing both *XWORD* and *TWORD* in this fashion, the value of SKOR is the number of unaccounted-for characters, plus the number of disagreements, plus the number of tranpositions, with two qualifications:

1. If both *XWORD* and *TWORD* have a character unaccounted-for in the same position, the two characters are counted only once, i.e., substitution errors count as only one disagreement, not two
2. If there are no unaccounted-for characters and no disagreements, transpositions are not counted.

This permits spelling correction on very short words, such as edit commands, e.g., XRT->XTR. Transpositions are also not counted when FASTYPEFLG = T, for example, IPULX and IPLUS will be in 80% agreement with FASTYPEFLG = T, only 60% with FASTYPEFLG = NIL. The rationale behind this is that transpositions are much more common for fast typists, and should not be counted as disagreements, whereas more deliberate typists are not as likely to combine tranpositions and other mistakes in a single word, and therefore can use more conservative metric. FASTYPEFLG is initially NIL.

Spelling Corrector Functions and Variables

(**ADDSPELL** *X SPLST N*) [Function]

Adds *X* to one of the spelling lists as determined by the value of *SPLST*:

NIL	Adds *X* to USERWORDS and to SPELLINGS2. Used by DEFINEQ.
0	Adds *X* to USERWORDS. Used by LOAD when loading EXPRs to property lists.
1	Adds *X* to SPELLINGS1 (at end of permanent section). Used by LISPX.
2	Adds *X* to SPELLINGS2 (at end of permanent section). Used by LISPX.
3	Adds *X* to USERWORDS and SPELLINGS3.
a spelling list	If *SPLST* is a spelling list, *X* is added to it. In this case, *N* is the (optional) length of the temporary section. If *X* is already on the spelling list, and in its temporary section, ADDSPELL moves *X* to the front of that section.

ADDSPELL sets LASTWORD to *X* when *SPLST* = NIL, 0 or 3.

If *X* is not a symbol, ADDSPELL takes no action.

Note that the various systems calls to ADDSPELL, e.g., from DEFINE, EDITF, LOAD, etc., can all be suppressed by setting or binding ADDSPELLFLG to NIL (see the DWIM Functions and Variables section above).

(**MISSPELLED?** *XWORD REL SPLST FLG TAIL FN*) [Function]

If *XWORD* = NIL or $ (<esc>), MISSPELLED? prints = followed by the value of LASTWORD, and returns this as the respelling, without asking for approval. Otherwise, MISSPELLED? checks to see if *XWORD* is really misspelled, i.e., if *FN* applied to *XWORD* is true, or *XWORD* is already contained on *SPLST*. In this case, MISSPELLED? simply returns *XWORD*. Otherwise MISSPELLED? computes and returns (FIXSPELL *XWORD REL SPLST FLG TAIL FN*) .

(**FIXSPELL** *XWORD REL SPLST FLG TAIL FN TIEFLG DONTMOVETOPFLG*) [Function]

The value of FIXSPELL is either the respelling of or NIL. If for some reason itself is on , then FIXSPELL aborts and calls ERROR!. If there is a possibility that is spelled correctly, MISSPELLED? should be used instead of FIXSPELL. FIXSPELL performs all of the interactions described earlier, including requesting your approval if necessary.

If *XWORD* = NIL or $ (escape), the respelling is the value of LASTWORD, and no approval is requested.

If *XWORD* contains lowercase characters, and the corresponding uppercase word is correct, i.e. on *SPLST* or satisfies *FN*, the uppercase word is returned and no interaction is performed. If FIXSPELL.UPPERCASE.QUIET is NIL (the default), a warning "=XX" is printed when coercing from "xx" to "XX". If FIXSPELL.UPPERCASE.QUIET is non-NIL, no warning is given.

If *REL* = NIL, defaults to the value of FIXSPELLREL (initially 70).

If *FLG* = NIL, the correction is handled in type-in mode, i.e., approval is never requested, and *XWORD* is not typed. If *FLG* = T, *XWORD* is typed (before the =) and approval is requested if APPROVEFLG = T. If *FLG* = NO-MESSAGE, the correction is returned with no further processing. In this case, a run-on correction will be returned as a dotted pair of the two parts of the word, and a synonym correction as a list of the form (*WORD1 WORD2*), where *WORD1* is (the corrected version of) *XWORD*, and *WORD2* is the synonym. The effect of the function CHOOZ can be obtained by calling FIXSPELL with *FLG* = NO-MESSAGE.

If *TAIL* is not NIL, and the correction is successful, CAR of *TAIL* is replaced by the respelling (using /RPLACA).

FIXSPELL will attempt to correct misspellings caused by running two words together, if the global variable RUNONFLG is non-NIL (default is NIL). In this case, approval is always requested. When a run-on error is corrected, CAR of *TAIL* is replaced by the two words, and the value of FIXSPELL is the first one. For example, if FIXSPELL is called to correct the edit command (MOVE TO AFTERCOND 3 2) with *TAIL* = (AFTERCOND 3 2), *TAIL* would be changed to (AFTER COND 2 3), and FIXSPELL would return AFTER (subject to yourapproval where necessary). If *TAIL* = T, FIXSPELL will also perform run-on corrections, returning a dotted pair of the two words in the event the correction is of this type.

If *TIEFLG* = NIL and a tie occurs, i.e., more than one word on *SPLST* is found with the same degree of "closeness", FIXSPELL returns NIL, i.e., no correction. If *TIEFLG* = PICKONE and a tie occurs, the first word is taken as the correct spelling. If *TIEFLG* = LIST, the value of FIXSPELL is a list of the respellings (even if there is only one), and FIXSPELL will not perform any interaction with you, nor modify *TAIL*, the idea being that the calling program will handle those tasks. Similarly, if *TIEFLG* = EVERYTHING, a list of all candidates whose degree of closeness is above *REL* will be returned, regardless of whether some are better than others. No interaction will be performed.

If *DONTMOVETOPFLG* = T and a correction occurs, it will *not* be moved to the front of the spelling list. Also, the spelling list will not be altered unless it contains the special marker used to separate the temporary and perminant parts of the system spelling lists (the value of SPELLSTR1).

(**FNCHECK** *FN NOERRORFLG SPELLFLG PROPFLG TAIL*) [Function]

The task of FNCHECK is to check whether *FN* is the name of a function and if not, to correct its spelling. If *FN* is the name of a function or spelling correction is successful, FNCHECK adds the (corrected) name of the function to USERWORDS using ADDSPELL, and returns it as its value.

Since FNCHECK is called by many low level functions such as ARGLIST, UNSAVEDEF, etc., spelling correction only takes place when DWIMFLG = T, so that these functions can operate in a small Interlisp system which does not contain DWIM.

NOERRORFLG informs FNCHECK whether or not the calling function wants to handle the unsuccessful case: if *NOERRORFLG* is T, FNCHECK simply returns NIL, otherwise it prints fn NOT A FUNCTION and generates a non-breaking error.

If *FN* does not have a definition, but does have an EXPR property, then spelling correction

is not attempted. Instead, if *PROPFLG* = T, *FN* is considered to be the name of a function, and is returned. If *PROPFLG* = NIL, *FN* is *not* considered to be the name of a function, and NIL is returned or an error generated, depending on the value of *NOERRORFLG*.

FNCHECK calls MISSPELLED? to perform spelling correction, so that if *FN* = NIL, the value of LASTWORD will be returned. *SPELLFLG* corresponds to MISSPELLED?'s fourth argument, *FLG*. If *SPELLFLG* = T, approval will be asked if DWIM was enabled in CAUTIOUS mode, i.e., if APPROVEFLG = T. *TAIL* corresponds to the fifth argument to MISSPELLED?.

FNCHECK is currently used by ARGLIST, UNSAVEDEF, PRETTYPRINT, BREAK0, BREAKIN, ADVISE, and CALLS. For example, BREAK0 calls FNCHECK with *NOERRORFLG* = T since if FNCHECK cannot produce a function, BREAK0 wants to define a dummy one. CALLS however calls FNCHECK with *NOERRORFLG* = NIL, since it cannot operate without a function.

Many other system functions call MISSPELLED? or FIXSPELL directly. For example, BREAK1 calls FIXSPELL on unrecognized atomic inputs before attempting to evaluate them, using as a spelling list a list of all break commands. Similarly, LISPX calls FIXSPELL on atomic inputs using a list of all LISPX commands. When UNBREAK is given
the name of a function that is not broken, it calls FIXSPELL with two different spelling lists, first with BROKENFNS, and if that fails, with USERWORDS. MAKEFILE calls MISSPELLED? using FILELST as a spelling list. Finally, LOAD, BCOMPL, BRECOMPILE, TCOMPL, and RECOMPILE all call MISSPELLED? if their input file(s) won't open.