20. CLISP

The syntax of Lisp is very simple. It can be described concisely, but it makes Lisp difficult to read and write without tools. Unlike many languages, there are no reserved words in Lisp such as IF, THEN, FOR, DO, etc., nor reserved characters like +, −, =, _, etc. The only components of the language are atoms and delimiters. This eliminates the need for parsers and precedence rules, and makes Lisp programs easy to mainpuilate. For example, a Lisp interpreter can be written in one or two pages of Lisp code. This makes Lisp the most suitable programming language for writing programs that deal with other programs as data.

Human language is based on more complicated structures and relies more on special words to carry the meaning. The definiton of the factorial function looks like this in Lisp:

```
(COND ((ZEROP N) 1) (T (TIMES N (FACTORIAL ((SUB1 N))))))
```

This definition is easy to read for a machine but difficult to read for a human. CLISP is designed to make Interlisp programs easier to read and write. CLISP does this by translating various operators, conditionals, and iterative statements to Interlisp. For example, factorial can be written in CLISP:

```
(IF N=0 THEN 1 ELSE N*(FACTORIAL N-1))
```

CLISP will translate this expression to the form in the example above. The translation will take place when the form is read so there are no performance penalties.

You should view CLISP as a shothand for produceing Lisp programs. CLISP makes a program easy to read and sometimes more compact.

CLISP is implemented via the error correction machinery in Interlisp (see Chapter 20). Any expression that Interlisp thinks is well-formed will never be seen by CLISP This means that interpreted programs that do not use CLISP constructs do not pay for its availability by slower execution time. In fact, the Interlisp interpreter does not know about CLISP at all. When the interpreter finds an error it calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer. The DWIM analyzer knows how to deal with CLISP expressions. If the expression in question turns out to be a CLISP construct, the translated form is returned to the interpreter. In addition, the original CLISP expression is modified so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into Medley makes possible Do-What-I-Mean features for CLISP constructs as well as for pure Lisp expressions. For example, if you have defined a function named GET−PARENT, CLISP would know not to attempt to interpret the form (GET−PARENT) as an arithmetic infix operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If you mistakenly write (GET−PRAENT), CLISP would know you meant (GET−PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PARENT is not the name of a variable, and that GET−PARENT is the name of a user function whose spelling is "very close" to that of GET−PRAENT. Similarly, by using information about the program's environment not readily available to a

preprocessor, CLISP can successfully resolve the following sorts of ambiguities:

1. `(LIST X*FACT N)`, where `FACT` is the name of a variable, means `(LIST (X*FACT) N)`.

2. `(LIST X*FACT N)`, where `FACT` is *not* the name of a variable but instead is the name of a function, means `(LIST X*(FACT N))`, i.e., `N` is `FACT`'s argument.

3. `(LIST X*FACT(N))`, `FACT` the name of a function (and not the name of a variable), means `(LIST X*(FACT N))`.

4. Cases 1, 2 and 3 with `FACT` misspelled!

The first expression is correct both from the standpoint of CLISP syntax and semantics so the change would be made notification. In the other cases, you would be informed or consulted about what was taking place. For example, suppose you write the expression `(LIST X*FCCT N)`. Assume also that there was both a function named `FACT` and a variable named `FCT`.

1. You will first be asked if `FCCT` is a misspelling of `FCT`. If you say `YES`, the expression will be interpreted as `(LIST (X*FCT) N)`. If you say `NO`, you will be asked if `FCCT` was a misspelling of `FACT`, i.e., if you intended `X*FCCT N` to mean `X*(FACT N)`.

2. If you say `YES` to this question, the indicated transformation will be performed. If you say `NO`, the system will ask if `X*FCCT` should be treated as CLISP, since `FCCT` is not the name of a (bound) variable.

3. If you say `YES`, the expression will be transformed, if `NO`, it will be left alone, i.e., as `(LIST X*FCCT N)`. Note that we have not even considered the case where `X*FCCT` is itself a misspelling of a variable name, e.g., a variable named `XFCT` (as with `GET-PRAENT)`. This sort of transformation will be considered after you said `NO` to `X*FCCT N -> X*(FACT N)`.

The question of whether `X*FCCT` should be treated as CLISP is important because Interlisp users may have programs that employ identifiers containing CLISP operators. Thus, if CLISP encounters the expression `A/B` in a context where either `A` or `B` are not the names of variables, it will ask you if `A/B` is intended to be CLISP, in case you really do have a free variable named `A/B`.

Note: Through the discussion above, we speak of CLISP or DWIM asking you. Actually, if you typed in the expression in question for immediate execution, you are simply informed of the transformation, on the grounds that you would prefer an occasional misinterpretation rather than being continuously bothered, especially since you can always retype what you intended if a mistake occurs, and ask the programmer's assistant to `UNDO` the effects of the mistaken operations if necessary. For transformations on expressions in your programs, you can tell CLISP whether you wish to operate in `CAUTIOUS` or `TRUSTING` mode. In the former case (most typical) you will be asked to approve transformations, in the latter, CLISP will operate as it does on type-in, i.e., perform the transformation after informing you.

CLISP can also handle parentheses errors caused by typing `8` or `9` for `(` or `)`. (On most terminals, `8`

and 9 are the lowercase characters for (and), i.e., (and 8 appear on the same key, as do) and 9.) For example, if you write N*8FACTORIAL N-1, the parentheses error can be detected and fixed before the infix operator * is converted to the Interlisp function TIMES. CLISP is able to distinguish this situation from cases like N*8*X meaning (TIMES N 8 X), or N*8X, where 8X is the name of a variable, again by using information about the programming environment. In fact, by integrating CLISP with DWIM, CLISP has been made sufficiently tolerant of errors that almost everything can be misspelled! For example, CLISP can successfully translate the definition of FACTORIAL:

```
(IFF N = 0 THENN1 ESLE N*8FACTTORIALNN-1)
```

to the corresponding COND, while making five spelling corrections and fixing the parenthesis error. CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, you could "clispify" the now correct version to obtain (IF N = 0 THEN 1 ELSE N*(FACTORIAL N-1)).

This sort of robustness prevails throughout CLISP. For example, the iterative statement permits you to say things like:

```
(FOR OLD X FROM M TO N DO (PRINT X) WHILE (PRIMEP X))
```

However, you can also write OLD (X_M), (OLD X_M), (OLD (X_M)), permute the order of the operators, e.g., (DO PRINT X TO N FOR OLD X_M WHILE PRIMEP X), omit either or both sets of parentheses, misspell any or all of the operators FOR, OLD, FROM, TO, DO, or WHILE, or leave out the word DO entirely! And, of course, you can also misspell PRINT, PRIMEP, M or N! In this example, the only thing you could not misspell is the first X, since it specifies the *name* of the variable of iteration. The other two instances of X could be misspelled.

CLISP is well integrated into Medley. For example, the above iterative statement translates into an equivalent Interlisp form using PROG, COND, GO, etc. When the interpreter subsequently encounters this CLISP expression, it automatically obtains and evaluates the translation. Similarly, the compiler "knows" to compile the translated form. However, if you PRETTYPRINT your program, PRETTYPRINT "knows" to print the original CLISP at the corresponding point in your function. Similarly, when you edit your program, the editor keeps the translation invisible to you. If you modify the CLISP, the translation is automatically discarded and recomputed the next time the expression is evaluated.

In short, CLISP is not a language at all, but rather a system. It plays a role analagous to that of the programmer's assistant (Chapter 13). Whereas the programmer's assistant is an invisible intermediary agent between your console requests and the Interlisp executive, CLISP sits between your programs and the Interlisp interpreter.

Only a small effort has been devoted to defining the core syntax of CLISP. Instead, most of the effort has been concentrated on providing a facility which "makes sense" out of the input expressions using context information as well as built-in and acquired information about user and system programs. It has been said that communication is based on the intention of the speaker to produce an effect in the recipient. CLISP operates under the assumption that what you say is *intended* to represent a meaningful operation, and therefore tries very hard to make sense out of it. The motivation behind CLISP is not to provide you with many different ways of saying the same thing, but to enable you to

worry less about the *syntactic* aspects of your communication with the system. In other words, it gives you a new degree of freedom by permitting you to concentrate more on the problem at hand, rather than on translation into a formal and unambiguous language.

DWIM and CLISP are invoked on iterative statements because CAR of the iterative statement is not the name of a function, and hence generates an error. If you define a function by the same name as an i.s. operator, e.g., WHILE, TO, etc., the operator will no longer have the CLISP interpretation when it appears as CAR of a form, although it will continue to be treated as an i.s. operator if it appears in the interior of an i.s. To alert you, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

CLISP Interaction with User

Syntactically and semantically well formed CLISP transformations are always performed without informing you. Other CLISP transformations described in the previous section, e.g., misspellings of operands, infix operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (Chapter 19). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression you typed in for immediate execution, your approval is not requested, but you are informed. However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, you will be asked to approve. If you say NO, the transformation is not performed. Thus, in the previous section, phrases such as "one of these (transformations) succeeds" and "the transformation LAST-ELL -> LAST-EL would be found" etc., all mean if you are in CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if you approve (or defaults by not responding). If you say NO, the procedure followed is the same as though the transformation had not been found. For example, if A*B appears in the function FOO, and B is not bound (and no other transformations are found) you would be asked A*B [IN FOO] TREAT AS CLISP ? (The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*DWIMWAIT).

In certain situations, DWIM asks for approval even if DWIM is enabled in TRUSTING mode. For example, you are always asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in LAST-ELL -> LAST-EL.

If you approved, A*B would be transformed to (ITIMES A B), which would then cause a U.B.A.B. error in the event that the program was being run (remember the entire discussion also applies to DWIMifying). If you said NO, A*B would be left alone.

If the value of CLISPHELPFLG = NIL (initally T), you will not be asked to approve any CLISP transformation. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said NO.

CLISP Character Operators

 CLISP recognizes a number of special characters operators, both prefix and infix, which are translated

into common expressions. For example, the character + is recognized to represent addition, so CLISP translates the symbol A+B to the form (IPLUS A B). Note that CLISP is invoked, and this translation is made, only if an error occurs, such as an unbound atom error or an undefined function error for the perfectly legitamate symbol A+B. Therefore you may choose not to use these facilities with no penalty, similar to other CLISP facilities.

You have a lot of flexability in using CLISP character operators. A list can always be substituted for a symbol, and vice versa, without changing the interpretation of a phrase. For example, if the value of (FOO X) is A, and the value of (FIE Y) is B, then (LIST (FOO X)+(FIE Y)) has the same value as (LIST A+B). Note that the first expression is a list of *four* elements: the atom "LIST", the list "(FOO X)", the atom "+", and the list "(FIE X)", whereas the second expression, (LIST A+B), is a list of only *two* elements: the symbol "LIST" and the symbol "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X) + (FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program, to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+ B), (LIST A +B), and (LIST A + B) the same as (LIST A+B).

Note: CLISP does not use its own special READ program because this would require you to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

+ [CLISP Operator]
- [CLISP Operator]
***** [CLISP Operator]
/ [CLISP Operator]
^ [CLISP Operator]

CLISP recognizes +, −, *, /, and ^ as the normal arithmetic infix operators. The − is also recognized as the prefix operator, unary minus. These are converted to PLUS, DIFFERENCE (or in the case of unary minus, MINUS), TIMES, QUOTIENT, and EXPT.

Normally, CLISP uses the "generic" arithmetic functions PLUS, TIMES, etc. CLISP contains a facility for declaring which type of arithmetic is to be used, either by making a global declaration, or by separate declarations about individual functions or variables.

The usual precedence rules apply (although you can easily change them), i.e., * has higher precedence than + so that A+B*C is the same as A+(B*C), and both * and / are lower than ^ so that 2*X^2 is the same as 2*(X^2). Operators of the same precedence group from left to right, e.g., A/B/C is equivalent to (A/B)/C. Minus is binary whenever possible, i.e., except when it is the first operator in a list, as in (−A) or (−A), or when it immediately follows another operator, as in A*−B. Note that grouping with parentheses can always be used to override the normal precedence grouping, or when you are not sure how a particular expression will parse. The complete order of precedence for CLISP operators is given below.

Note that + in front of a number will disappear when the number is read, e.g., (FOO X +2) is indistinguishable from (FOO X 2). This means that (FOO X +2) will not be interpreted as CLISP, or be converted to (FOO (IPLUS X 2)). Similarly, (FOO X −2) will not be interpreted the same as (FOO X−2). To circumvent this, always type a space between the + or − and a number if an infix operator is intended, e.g., write (FOO X +

2).

= [CLISP Operator]
GT [CLISP Operator]
LT [CLISP Operator]
GE [CLISP Operator]
LE [CLISP Operator]

These are infix operators for "Equal", "Greater Than", "Less Than", "Greater Than or Equal", and "Less Than or Equal".

GT, LT, GE, and LE are all affected by the same declarations as + and *, with the initial default to use GREATERP and LESSP.

Note that only single character operators, e.g., +, _, =, etc., can appear in the *interior* of an atom. All other operators must be set off from identifiers with spaces. For example, XLTY will not be recognized as CLISP. In some cases, DWIM will be able to diagnose this situation as a run-on spelling error, in which case after the atom is split apart, CLISP will be able to perform the indicated transformation.

A number of Lisp functions, such as EQUAL, MEMBER, AND, OR, etc., can also be treated as CLISP infix operators. New infix operators can be easily added (see the CLISP Internal Convetions section below). Spelling correction on misspelled infix operators is peformed using CLISPINFIXSPLST as a spelling list.

AND is higher than OR, and both AND and OR are lower than the other infix operators, so (X OR Y AND Z) is the same as (X OR (Y AND Z)), and (X AND Y EQUAL Z) is the same as (X AND (Y EQUAL Z)). All of the infix predicates have lower precedence than Interlisp forms, since it is far more common to apply a predicate to two forms, than to use a Boolean as an argument to a function. Therefore, (FOO X GT FIE Y) is translated as ((FOO X) GT (FIE Y)), rather than as (FOO (X GT (FIE Y))). However, you can easily change this.

: [CLISP Operator]

X:*N* extracts the *N*th element of the list *X*. FOO:3 specifies the third element of FOO, or (CADDR FOO). If *N* is less than zero, this indicates elements counting from the end of the list; i.e. FOO:-1 is the last element of FOO. : operators can be nested, so FOO:1:2 means the second element of the first element of FOO, or (CADAR FOO).

The : operator can also be used for extracting substructures of records (see Chapter 8). Record operations are implemented by replacing expressions of the form X:FOO by (fetch FOO of X). Both lower- and uppercase are acceptable.

: is also used to indicate operations in the pattern match facility (see Chapter 12). X:(& 'A -- 'B) translates to (match X with (& 'A -- 'B))

. [CLISP Operator]

In combination with :, a period can be used to specify the "data path" for record operations. For example, if FOO is a field of the BAR record, X:BAR.FOO is translated into (fetch (BAR FOO) of X). Subrecord fields can be specified with multiple periods:

X:BAR.FOO.BAZ translates into (fetch (BAR FOO BAZ) of X).

Note: If a record contains fields with periods in them, CLISPIFY will not translate a record operation into a form using periods to specify the data path. For example, CLISPIFY will NOT translate (fetch A.B of X) into X:A.B.

:: [CLISP Operator]

X:*N*, returns the *N*th *tail* of the list *X*. For example, FOO::3 is (CDDDR FOO), and FOO::-1 is (LAST FOO).

_ [CLISP Operator]

_ is used to indicate assignment. For example, X_Y translates to (SETQ X Y). If X does not have a value, and is not the name of one of the bound variables of the function in which it appears, spelling correction is attempted. However, since this may simply be a case of assigning an initial value to a new free variable, DWIM will always ask for approval before making the correction.

In conjunction with : and ::, _ can also be used to perform a more general type of assignment, involving structure modification. For example, X:2_Y means "make the second element of X *be* Y", in Interlisp terms (RPLACA (CDR X) Y). Note that the *value* of this operation is the value of RPLACA, which is (CDR X), rather than Y. Negative numbers can also be used, e.g., X:-2←Y, which translates to (RPLACA (NLEFT X 2) Y).

You can indicate you want /RPLACA and /RPLACD used (undoable version of RPLACA and RPLACD, see Chapter 13), or FRPLACA and FRPLACD (fast versions of RPLACA and RPLACD, see Chapter 3), by means of CLISP declarations. The initial default is to use RPLACA and RPLACD.

_ is also used to indicate assignment in record operations (X:FOO_Y translates to (replace FOO of X with Y).), and pattern match operations (Chapter 12).

_ has different precedence on the left from on the right. On the left,_ is a "tight" operator, i.e., high precedence, so that A+B_C is the same as A+(B_C). On the right, _ has broader scope so that A_B+C is the same as A_(B+C).

On type-in, $_*FORM* (where $ is the escape key) is equivalent to set the "last thing mentioned", i.e., is equivalent to (SET LASTWORD *FORM*) (see Chapter 20). For example, immediately after examining the value of LONGVARIABLENAME, you could set it by typing $_ followed by a form.

Note that an atom of the form X_Y, appearing at the top level of a PROG, will not be recognized as an assignment statement because it will be interpreted as a PROG label by the Interlisp interpreter, and therefore will not cause an error, so DWIM and CLISP will never get to see it. Instead, one must write (X_Y).

< [CLISP Operator]

> [CLISP Operator]

Angle brackets are used in CLISP to indicate list construction. The appearance of a "<" corresponds to a "(" and indicates that a list is to be constructed containing all the elements up to the corresponding ">". For example, `<A B <C>>` translates to `(LIST A B (LIST C))`. `!` can be used to indicate that the next expression is to be inserted in the list as a *segment*, e.g., `<A B ! C>` translates to `(CONS A (CONS B C))` and `<! A ! B C>` to `(APPEND A B (LIST C))`. `!!` is used to indicate that the next expression is to be inserted as a segment, and furthermore, all list structure to its right in the angle brackets is to be physically attached to it, e.g., `<!! A B>` translates to `(NCONC1 A B)`, and `<!!A !B !C>` to `(NCONC A (APPEND B C))`. Not `(NCONC (APPEND A B) C)`, which would have the same value, but would attach `C` to `B`, and not attach either to `A`. Note that `<`, `!`, `!!`, and `>` need not be separate atoms, for example, `<A B ! C>` may be written equally well as `< A B !C >`. Also, arbitrary Interlisp or CLISP forms may be used within angle brackets. For example, one can write `<FOO_(FIE X) ! Y>` which translates to `(CONS (SETQ FOO (FIE X)) Y)`. `CLISPIFY` converts expressions in `CONS`, `LIST`, `APPEND`, `NCONC`, `NCONC1`, `/NCONC`, and `/NCONC1` into equivalent CLISP expressions using `<`, `>`, `!`, and `!!`.

Note: brackets differ from other CLISP operators. For example, `<A B 'C>` translates to `(LIST A B (QUOTE C))` even though following `'`, all *operators* are ignored for the rest of the identifier. (This is true only if a previous unmatched **<** has been seen, e.g., `(PRINT 'A>B)` will print the atom `A>B`.) Note however that `<A B ' C> D>` is equivalent to `(LIST A B (QUOTE C>) D)`.

' [CLISP Operator]

CLISP recognizes `'` as a prefix operator. `'` means `QUOTE` when it is the first character in an identifier, and is ignored when it is used in the interior of an identifier. Thus, `X = 'Y` means `(EQ X (QUOTE Y))`, but `X = CAN'T` means `(EQ X CAN'T)`, *not* `(EQ X CAN)` followed by `(QUOTE T)`. This enables users to have variable and function names with `'` in them (so long as the `'` is not the first character).

Following `'`, all operators are ignored for the rest of the identifier, e.g., `'*A` means `(QUOTE *A)`, and `'X=Y` means `(QUOTE X=Y)`, not `(EQ (QUOTE X) Y)`. To write `(EQ (QUOTE X) Y)`, one writes `Y='X`, or `'X =Y`. This is one place where an extra space does make a difference.

On type-in, `'$` (escape) is equivalent to `(QUOTE` *VALUE-OF-LASTWORD*`)` (see Chapter 19). For example, after calling `PRETTYPRINT` on `LONGFUNCTION`, you could move its definition to `FOO` by typing `(MOVD '$ 'FOO)`.

Note that this is not `(MOVD $ 'FOO)`, which would be equivalent to `(MOVD LONGFUNCTION 'FOO)`, and would (probably) cause a `U.B.A. LONGFUNCTION` error, nor `MOVD($ FOO)`, which would actually move the definition of `$` to `FOO`, since DWIM and the spelling corrector would never be invoked.

~ [CLISP Operator]

CLISP recognizes `~` as a prefix operator meaning `NOT`. `~` can negate a form, as in `~(ASSOC X Y)`, or `~X`, or negate an infix operator, e.g., `(A ~GT B)` is the same as `(A`

LEQ B). Note that ~A = B means (EQ (NOT A) B).

When ~ negates an operator, e.g., ~=, ~LT, the two operators are treated as a single operator whose precedence is that of the second operator. When ~ negates a function, e.g., (~FOO X Y), it negates the whole form, i.e., (~(FOO X Y)).

Order of Precedence of CLISP Operators:

'
:
_ (left precedence)
– (unary), ~
^
*, /
+, – (binary)
_ (right precedence)
=

Interlisp forms

LT, GT, EQUAL, MEMBER, etc.
AND
OR
IF, THEN, ELSEIF, ELSE
iterative statement operators

Declarations

CLISP declarations are used to affect the choice of Interlisp function used as the translation of a particular operator. For example, A+B can be translated as either (PLUS A B), (FPLUS A B), or (IPLUS A B), depending on the declaration in effect. Similarly X:1_Y can mean (RPLACA X Y), (FRPLACA X Y), or (/RPLACA X Y), and <!! A B> either (NCONC1 A B) or (/NCONC1 A B). Note that the choice of function on all CLISP transformations are affected by the CLISP declaration in effect, i.e., iterative statements, pattern matches, record operations, as well as infix and prefix operators.

(**CLISPDEC** *DECLST*) [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

You can makes (changes) a global declaration by calling CLISPDEC with *DECLST* a list of declarations, e.g., (CLISPDEC '(FLOATING UNDOABLE)). Changing a global declaration does not affect the speed of subsequent CLISP transformations, since all CLISP transformation are table driven (i.e., property list), and global declarations are accomplished by making the appropriate internal changes to CLISP at the time of the declaration. If a function employs *local* declarations (described below), there will be a slight loss in efficiency owing to the fact that for each CLISP transformation, the

declaration list must be searched for possibly relevant declarations.

Declarations are implemented in the order that they are given, so that later declarations override earlier ones. For example, the declaration FAST specifies that FRPLACA, FRPLACD, FMEMB, and FLAST be used in place of RPLACA, RPLACD, MEMB, and LAST; the declaration RPLACA specifies that RPLACA be used. Therefore, the declarations (FAST RPLACA RPLACD) will cause FMEMB, FLAST, RPLACA, and RPLACD to be used.

The initial global declaration is MIXED and STANDARD.

The table below gives the declarations available in CLISP, and the Interlisp functions they indicate:

Declaration:	Interlisp Functions to be used:
MIXED	PLUS, MINUS, DIFFERENCE, TIMES, QUOTIENT, LESSP, GREATERP
INTEGER or FIXED	IPLUS, IMINUS, IDIFFERENCE, ITIMES, IQUOTIENT, ILESSP, IGREATERP
FLOATING	FPLUS, FMINUS, FDIFFERENCE, FTIMES, FQUOTIENT, LESSP, FGREATERP
FAST	FRPLACA, FRPLACD, FMEMB, FLAST, FASSOC
UNDOABLE	/RPLACA, /RPLACD, /NCONC, /NCONC1, /MAPCONC, /MAPCON
STANDARD	RPLACA, RPLACD, MEMB, LAST, ASSOC, NCONC, NCONC1, MAPCONC, MAPCON
RPLACA, RPLACD, /RPLACA, etc.	corresponding function

You can also make local declarations affecting a selected function or functions by inserting an expression of the form (CLISP: . *DECLARATIONS*) immediately following the argument list, i.e., as CADDR of the definition. Such local declarations take precedence over global declarations. Declarations affecting selected variables can be indicated by lists, where the first element is the name of a variable, and the rest of the list the declarations for that variable. For example, (CLISP: FLOATING (X INTEGER)) specifies that in this function integer arithmetic be used for computations involving X, and floating arithmetic for all other computations, where "involving" means where the variable itself is an operand. For example, with the declaration (FLOATING (X INTEGER)) in effect, (FOO X)+(FIE X) would translate to FPLUS, i.e., use floating arithmetic, even though X appears somewhere inside of the operands, whereas X+(FIE X) would translate to IPLUS. If there are declarations involving *both* operands, e.g., X+Y, with (X FLOATING) (Y INTEGER), whichever appears first in the declaration list will be used.

You can also make local record declarations by inserting a record declaration, e.g., (RECORD --), (ARRAYRECORD --), etc., in the local declaration list. In addition, a local declaration of the form (RECORDS A B C) is equivalent to having copies of the global

declarations A, B, and C in the local declaration. Local record declarations override global record declarations for the function in which they appear. Local declarations can also be used to override the global setting of certain DWIM/CLISP parameters effective only for transformations within that function, by including in the local declaration an expression of the form (*VARIABLE* = *VALUE*), e.g., (PATVARDEFAULT = QUOTE).

The CLISP: expression is converted to a comment of a special form recognized by CLISP. Whenever a CLISP transformation that is affected by declarations is about to be performed in a function, this comment will be searched for a relevant declaration, and if one is found, the corresponding function will be used. Otherwise, if none are found, the global declaration(s) currently in effect will be used.

Local declarations are effective in the order that they are given, so that later declarations can be used to override earlier ones, e.g., (CLISP: FAST RPLACA RPLACD) specifies that FMEMB, FLAST, RPLACA, and RPLACD be used. An exception to this is that declarations for specific variables take precedence of general, function-wide declarations, regardless of the order of appearance, as in (CLISP: (X INTEGER) FLOATING).

CLISPIFY also checks the declarations in effect before selecting an infix operator to ensure that the corresponding CLISP construct would in fact translate back to this form. For example, if a FLOATING declaration is in effect, CLISPIFY will convert (FPLUS X Y) to X+Y, but leave (IPLUS X Y) as is. If (FPLUS X Y) is CLISPIFYed while a FLOATING declaration is under effect, and then the declaration is changed to INTEGER, when X+Y is translated back to Interlisp, it will become (IPLUS X Y).

CLISP Operation

CLISP is a part of the basic Medley system. Without any special preparations, you can include CLISP constructs in programs, or type them in directly for evaluation (in EVAL or APPLY format), then, when the "error" occurrs, and DWIM is called, it will destructively transform the CLISP to the equivalent Interlisp expression and evaluate the Interlisp expression. CLISP transformations, like all DWIM corrections, are undoable. User approval is not requested, and no message is printed. This entire discussion also applies to CLISP transformation initiated by calls to DWIM from DWIMIFY.

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if you write (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if you write (LAST+EL X), CLISP knows that ((IPLUS LAST EL) X) is not a valid Interlisp expression, so the error diagnostic MISSING OPERATOR IN (LAST+EL X) is generated. (For example, you might have meant to say (LAST+EL*X).) If LAST+EL were the name of a defined function, CLISP would never see this form.

Since the bad CLISP transformation might not be CLISP at all, for example, it might be a misspelling of a user function or variable, DWIM holds all CLISP error messages until after trying other corrections. If one of these succeeds, the CLISP message is discarded. Otherwise, if all fail, the message is printed (but no change is made). For example, suppose you type (R/PLACA X Y). CLISP generates a diagnostic, since ((IQUOTIENT R PLACA) X Y) is obviously not right. However, since R/PLACA spelling corrects to /RPLACA, this diagnostic is never printed.

Note: CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

If a CLISP infix construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound, it is possible that either the operand is misspelled, e.g., you wrote X+YY for X+Y, or that a CLISP transformation operation was not intended at all, but that the entire expression is a misspelling. For the purpose of DWIMIFYing, "not bound" means no top level value, not on list of bound variables built up by DWIMIFY during its analysis of the expression, and not on NOFIXVARSLST, i.e., not previously seen.

For example, if you have a variable named LAST-EL, and write (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated infix transformation. DWIM then continues, and if it is able to make another correction, does so, and ignores the CLISP interpretation. For example, with LAST-ELL, the transformation LAST-ELL -> LAST-EL would be found.

If no other transformation is found, and DWIM is about to interpret a construct as CLISP for which one of the operands is not bound, DWIM will ask you whether CLISP was intended, in this case by printing LAST-ELL TREAT AS CLISP ?.

Note: If more than one infix operator was involved in the CLISP construct, e.g., X+Y+Z, or the operation was an assignment to a variable already noticed, or TREATASCLISPFLG is T (initially NIL), you will simply be informed of the correction, e.g., X+Y+Z TREATED AS CLISP. Otherwise, even if DWIM was enabled in TRUSTING mode, you will be asked to approve the correction.

The same sort of procedure is followed with 8 and 9 errors. For example, suppose you write FOO8*X where FOO8 is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks you to approve FOO8*X -> FOO (*X. For example, this would make sense if you have (or plan to define) a function named *X. If you refuses, you are asked whether FOO8*X is to be treated as CLISP. Similarly, if FOO8 were the name of a variable, and you write FOOO8*X, you will first be asked to approve FOOO8*X -> FOOO (XX, and if you refuse, then be offered the FOOO8 -> FOO8 correction. The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or undefined function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

CLISP also contains provision for correcting misspellings of infix operators (other than single characters), IF words, and i.s. operators. This is implemented in such a way that the user who does not misspell them is not penalized. For example, if you write IF N = 0 THEN 1 ELSSE N*(FACT N-1) CLISP does *not* operate by checking each word to see if it is a misspelling of IF, THEN, ELSE, or ELSEIF, since this would seriously degrade CLISP's performance on *all* IF statements. Instead, CLISP assumes that all of the IF words are spelled correctly, and transforms the expression to (COND ((ZEROP N) 1 ELSSE N*(FACT N-1))). Later, after DWIM cannot find any other interpretation for ELSSE, and using the fact that this atom originally appeared in an IF statement, DWIM attempts spelling correction, using (IF THEN ELSE ELSEIF) for a spelling list. When this is successful, DWIM "fails" all the way back to the original IF statement, changes ELSSE to ELSE, and starts over. Misspellings of AND, OR, LT, GT, etc. are handled similarly.

CLISP also contains many Do-What-I-Mean features besides spelling corrections. For example, the

form (LIST +X Y) would generate a MISSING OPERATOR error. However, (LIST -X Y) makes sense, if the minus is unary, so DWIM offers this interpretation to you. Another common error, especially for new users, is to write (LIST X*FOO(Y)) or (LIST X*FOO Y), where FOO is the name of a function, instead of (LIST X*(FOO Y)). Therefore, whenever an operand that is not bound is also the name of a function (or corrects to one), the above interpretations are offered.

CLISP Translations

The translation of CLISP character operators and the CLISP word IF are handled by *replacing* the CLISP expression with the corresponding Interlisp expression, and discarding the original CLISP. This is done because (1) the CLISP expression is easily recomputable (by CLISPIFY) and (2) the Interlisp expressions are simple and straightforward. Another reason for discarding the original CLISP is that it may contain errors that were corrected in the course of translation (e.g., FOO_FOOO:1, N*8FOO X), etc.). If the original CLISP were retained, either you would have to go back and fix these errors by hand, thereby negating the advantage of having DWIM perform these corrections, or else DWIM would have to keep correcting these errors over and over.

Note that CLISPIFY is sufficiently fast that it is practical for you to configure your Interlisp system so that all expressions are automatically CLISPIFYed immediately before they are presented to you. For example, you can define an edit macro to use in place of P which calls CLISPIFY on the current expression before printing it. Similarly, you can inform PRETTYPRINT to call CLISPIFY on each expression before printing it, etc.

Where (1) or (2) are not the case, e.g., with iterative statements, pattern matches, record expressions, etc. the original CLISP *is* retained (or a slightly modified version thereof), and the translation is stored elsewhere (by the function CLISPTRAN, in the Miscellaneous Functions and Variables), usually in the hash array CLISPARRAY. The interpreter automatically checks this array when given a form CAR of which is not a function. Similarly, the compiler performs a GETHASH when given a form it does not recognize to see if it has a translation, which is then compiled instead of the form. Whenever you *change* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or DWIMIFIed, the expression will be retranslated (if the value of CLISPRETRANFLG is T, DWIMIFY will also (re)translate any expressions which have translations stored remotely, see the CLISPIFY section). The function PPT and the edit commands PPT and CLISP: are available for examining translations (see the Miscellaneous Functions and Variables section).

You can also indicate that you want the original CLISP retained by embedding it in an expression of the form (CLISP . *CLISP-EXPRESSION*), e.g., (CLISP X:5:3) or (CLISP <A B C ! D>). In such cases, the translation will be stored remotely as described above. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting CLISPFLG to NIL (see the Miscellaneous Functions and Variables section). In other words, you can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. You can also include CLISP declarations by writing (CLISP *DECLARATIONS* . *FORM*), e.g., (CLISP (CLISP: FLOATING) ...). These declarations will be used in place of any CLISP declarations in the function definition. This feature provides a way of

including CLISP declarations in macro definitions.

Note: CLISP translations can also be used to supply an interpretation for function objects, as well as forms, either for function objects that are used openly, i.e., appearing as CAR of form, function objects that are explicitly APPLYed, as with arguments to mapping functions, or function objects contained in function definition cells. In all cases, if CAR of the object is not LAMBDA or NLAMBDA, the interpreter and compiler will check CLISPARRAY.

DWIMIFY

DWIMIFY is effectively a preprocessor for CLISP. DWIMIFY operates by scanning an expression as though it were being interpreted, and for each form that would generate an error, calling DWIM to "fix" it. DWIMIFY performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, fixes 8-9 errors, handles F/L, etc. Thus you will see the same messages, and be asked for approval in the same situations, as you would if the expression were actually run. If DWIM is unable to make a correction, no message is printed, the form is left as it was, and the analysis proceeds.

DWIMIFY knows exactly how the interpreter works. It knows the syntax of PROGs, SELECTQs, LAMBDA expressions, SETQs, et al. It knows how variables are bound, and that the argument of NLAMBDAs are not evaluated (you can inform DWIMIFY of a function or macro's nonstandard binding or evaluation by giving it a suitable INFO property, see below). In the course of its analysis of a particular expression, DWIMIFY builds a list of the bound variables from the LAMBDA expressions and PROGs that it encounters. It uses this list for spelling corrections. DWIMIFY also knows not to try to "correct" variables that are on this list since they would be bound if the expression were actually being run. However, note that DWIMIFY cannot, a priori, know about variables that are used freely but would be bound in a higher function if the expression were evaluated in its normal context. Therefore, DWIMIFY will try to "correct" these variables. Similarly, DWIMIFY will attempt to correct forms for which CAR is undefined, even when the form is not in error from your standpoint, but the corresponding function has simply not yet been defined.

Note: DWIMIFY rebinds FIXSPELLDEFAULT to N, so that if you are not at the terminal when DWIMIFYing (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform you when it encounters an expression with too *many* arguments (unless DWIMCHECK#ARGSFLG = NIL), because such an occurrence, although does not cause an error in the Interlisp interpreter, nevertheless is frequently symptomatic of a parenthesis error. For example, if you wrote (CONS (QUOTE FOO X)) instead of (CONS (QUOTE FOO) X), DWIMIFY will print:

```
POSSIBLE PARENTHESIS ERROR IN
(QUOTE FOO X)
TOO MANY ARGUMENTS (MORE THAN 1)
```

DWIMIFY will also check to see if a PROG label contains a clisp character (unless DWIMCHECKPROGLABELSFLG = NIL, or the label is a member of NOFIXVARSLST), and if so, will alert you by printing the message SUSPICIOUS PROG LABEL, followed by the label. The PROG label will

not be treated as CLISP.

Note that in most cases, an attempt to transform a form that is already as you intended will have no effect (because there will be nothing to which that form could reasonably be transformed). However, in order to avoid needless calls to DWIM or to avoid possible confusion, you can inform DWIMIFY *not* to attempt corrections or transformations on certain functions or variables by adding them to the list NOFIXFNSLST or NOFIXVARSLST respectively. Note that you could achieve the same effect by simply setting the corresponding variables, and giving the functions dummy definitions.

DWIMIFY will never attempt corrections on global variables, i.e., variables that are a member of the list GLOBALVARS, or have the property GLOBALVAR with value T, on their property list. Similarly, DWIMIFY will not attempt to correct variables declared to be SPECVARS in block declarations or via DECLARE expressions in the function body. You can also declare variables that are simply used freely in a function by using the USEDFREE declaration.

DWIMIFY and DWIMIFYFNS (used to DWIMIFY several functions) maintain two internal lists of those functions and variables for which corrections were unsuccessfully attempted. These lists are initialized to the values of NOFIXFNSLST and NOFIXVARSLST. Once an attempt is made to fix a particular function or variable, and the attempt fails, the function or variable is added to the corresponding list, so that on subsequent occurrences (within this call to DWIMIFY or DWIMIFYFNS), no attempt at correction is made. For example, if FOO calls FIE several times, and FIE is undefined at the time FOO is DWIMIFYed, DWIMIFY will not bother with FIE after the first occurrence. In other words, once DWIMIFY "notices" a function or variable, it no longer attempts to correct it. DWIMIFY and DWIMIFYFNS also "notice" free variables that are set in the expression being processed. Moreover, once DWIMIFY "notices" such functions or variables, it subsequently treats them the same as though they were actually defined or set.

Note that these internal lists are local to each call to DWIMIFY and DWIMIFYFNS, so that if a function containing FOOO, a misspelled call to FOO, is DWIMIFYed before FOO is defined or mentioned, if the function is DWIMIFYed again after FOO has been defined, the correction will be made.

You can undo selected transformations performed by DWIMIFY, as described in Chapter 13.

(**DWIMIFY** *X QUIETFLG L*) [Function]

Performs all DWIM and CLISP corrections and transformations on *X* that would be performed if *X* were run, and prints the result unless *QUIETFLG* = T.

If *X* is an atom and *L* is NIL, *X* is treated as the name of a function, and its entire definition is DWIMIFYed. If *X* is a list or *L* is not NIL, *X* is the expression to be DWIMIFYed. If *L* is not NIL, it is the edit push-down list leading to *X*, and is used for determining context, i.e., what bound variables would be in effect when *X* was evaluated, whether *X* is a form or sequence of forms, e.g., a COND clause, etc.

If *X* is an iterative statement and *L* is NIL, DWIMIFY will also print the translation, i.e., what is stored in the hash array.

(**DWIMIFYFNS** FN_1 ... FN_N) [NLambda NoSpread Function]

DWIMIFYs each of the functions given. If only one argument is given, it is evalued. If its value is a list, the functions on this list are DWIMIFYed. If only one argument is given, it is atomic, its value is not a list, and it is the name of a known file, DWIMIFYFNS will operate on (FILEFNSLST FN_1), e.g. (DWIMIFYFNS FOO.LSP) will DWIMIFY every function in the file FOO.LSP.

Every 30 seconds, DWIMIFYFNS prints the name of the function it is processing, a la PRETTYPRINT.

Value is a list of the functions DWIMIFYed.

DWIMINMACROSFLG [Variable]

Controls how DWIMIFY treats the arguments in a "call" to a macro, i.e., where the CAR of the form is undefined, but has a macro definition. If DWIMINMACROSFLG is T, then macros are treated as LAMBDA functions, i.e., the arguments are assumed to be evaluated, which means that DWIMIFY will descend into the argument list. If DWIMINMACROSFLG is NIL, macros are treated as NLAMBDA functions. DWIMINMACROSFLG is initially T.

INFO [Property Name]

Used to inform DWIMIFY of nonstandard behavior of particular forms with respect to evaluation, binding of arguments, etc. The INFO property of a symbol is a single atom or list of atoms chosen from among the following:

EVAL Informs DWIMIFY (and CLISP and Masterscope) that an nlambda function *does* evaluate its arguments. Can also be placed on a macro name to override the behavior of DWIMINMACROSFLG = NIL.

NOEVAL Informs DWIMIFY that a macro does *not* evaluate all of its arguments, even when DWIMINMACROSFLG = T.

BINDS Placed on the INFO property of a function or the CAR of a special form to inform DWIMIFY that the function or form binds variables. In this case, DWIMIFY assumes that CADR of the form is the variable list, i.e., a list of symbols, or lists of the form (*VAL VALUE*). LAMBDA, NLAMBDA, PROG, and RESETVARS are handled in this fashion.

LABELS Informs CLISPIFY that the form interprets top-level symbols as labels, so that CLISPIFY will never introduce an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

NOFIXFNSLST [Variable]

List of functions that DWIMIFY will not try to correct.

NOFIXVARSLST [Variable]

List of variables that DWIMIFY will not try to correct.

NOSPELLFLG [Variable]

If T, DWIMIFY will not perform any spelling corrections. Initially NIL. NOSPELLFLG is reset to T when compiling functions whose definitions are obtained from a file, as opposed to being in core.

CLISPHELPFLG [Variable]

If NIL, DWIMIFY will not ask you for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though you had been asked and said NO. Initially T.

DWIMIFYCOMPFLG [Variable]

If T, DWIMIFY is called before compiling an expression. Initially NIL.

DWIMCHECK#ARGSFLG [Variable]

If T, causes DWIMIFY to check for too many arguments in a form. Initially T.

DWIMCHECKPROGLABELSFLG [Variable]

If T, causes DWIMIFY to check whether a PROG label contains a CLISP character. Initially T.

DWIMESSGAG [Variable]

If T, suppresses all DWIMIFY error messages. Initially NIL.

CLISPRETRANFLG [Variable]

If T, informs DWIMIFY to (re)translate all expressions which have remote translations in the CLISP hash array. Initially NIL.

CLISPIFY

CLISPIFY converts Interlisp expressions to CLISP. Note that the expression given to CLISPIFY need *not* have originally been input as CLISP, i.e., CLISPIFY can be used on functions that were written before CLISP was even implemented. CLISPIFY is cognizant of declaration rules as well as all of the precedence rules. For example, CLISPIFY will convert (IPLUS A (ITIMES B C)) into A+B*C, but (ITIMES A (IPLUS B C)) into A*(B+C). CLISPIFY handles such cases by first DWIMIFYing the expression. CLISPIFY also knows how to handle expressions consisting of a mixture of Interlisp and CLISP, e.g., (IPLUS A B*C) is converted to A+B*C, but (ITIMES A B+C) to (A*(B+C)). CLISPIFY converts calls to the six basic mapping functions, MAP, MAPC, MAPCAR, MAPLIST, MAPCONC, and MAPCON, into equivalent iterative statements. It also converts certain easily recognizable internal PROG loops to the corresponding iterative statements. CLISPIFY can convert all iterative statements input in CLISP back to CLISP, regardless of how complicated the translation was, because the original CLISP is saved.

CLISPIFY is not destructive to the original Interlisp expression, i.e., CLISPIFY produces a new

expression without changing the original. The new expression may however contain some "pieces" of the original, since CLISPIFY attempts to minimize the number of CONSes by not copying structure whenever possible.

CLISPIFY will not convert expressions appearing as arguments to NLAMBDA functions, except for those functions whose INFO property is or contains the atom EVAL. CLISPIFY also contains built in information enabling it to process special forms such as PROG, SELECTQ, etc. If the INFO property is or contains the atom LABELS, CLISPIFY will never create an atom (by packing) at the top level of the expression. PROG is handled in this fashion.

Note: Disabling a CLISP operator with CLDISABLE (see the Miscellaneous Functions and Variables section) will also disable the corresponding CLISPIFY transformation. Thus, if _ is "turned off", A_B will not transform to (SETQ A B), nor vice versa.

(**CLISPIFY** *X EDITCHAIN*) [Function]

Clispifies *X*. If *X* is an atom and *EDITCHAIN* is NIL, *X* is treated as the name of a function, and its definition (or EXPR property) is clispified. After CLISPIFY has finished, *X* is redefined (using /PUTD) with its new CLISP definition. The value of CLISPIFY is *X*. If *X* is atomic and not the name of a function, spelling correction is attempted. If this fails, an error is generated.

If *X* is a list, or *EDITCHAIN* is not NIL, *X* itself is the expression to be clispified. If *EDITCHAIN* is not NIL, it is the edit push-down list leading to *X* and is used to determine context as with DWIMIFY, as well as to obtain the local declarations, if any. The value of CLISPIFY is the clispified version of *X*.

(**CLISPIFYFNS** FN_1 ... FN_N) [NLambda NoSpread Function]

Like DWIMIFYFNS except calls CLISPIFY instead of DWIMIFY.

CL:FLG [Variable]

Affects CLISPIFY's handling of forms beginning with CAR, CDR, ... CDDDDR, as well as pattern match and record expressions. If CL:FLG is NIL, these are not transformed into the equivalent : expressions. This will prevent CLISPIFY from constructing any expression employing a : infix operator, e.g., (CADR X) will not be transformed to X:2. If CL:FLG is T, CLISPIFY will convert to : notation only when the argument is atomic or a simple list (a function name and one atomic argument). If CL:FLG is ALL, CLISPIFY will convert to : expressions whenever possible.

CL:FLG is initially T.

CLREMPARSFLG [Variable]

If T, CLISPIFY will remove parentheses in certain cases from simple forms, where "simple" means a function name and one or two atomic arguments. For example, (COND ((ATOM X) --)) will CLISPIFY to (IF ATOM X THEN --). However, if CLREMPARSFLG is set to NIL, CLISPIFY will produce (IF (ATOM X) THEN --). Regardless of the flag setting, the expression can be input in either form.

CLREMPARSFLG is initially NIL.

CLISPIFYPACKFLG [Variable]

CLISPIFYPACKFLG affects the treatment of infix operators with atomic operands. If CLISPIFYPACKFLG is T, CLISPIFY will pack these into single atoms, e.g., (IPLUS A (ITIMES B C)) becomes A+B*C. If CLISPIFYPACKFLG is NIL, no packing is done, e.g., the above becomes A + B * C.

CLISPIFYPACKFLG is initially T.

CLISPIFYUSERFN [Variable]

If T, causes the function CLISPIFYUSERFN, which should be a function of one argument, to be called on each form (list) not otherwise recognized by CLISPIFY. If a non-NIL value is returned, it is treated as the clispified form. Initially NIL

Note that CLISPIFYUSERFN must be both set and defined to use this feature.

FUNNYATOMLST [Variable]

Suppose you have variables named A, B, and A*B. If CLISPIFY were to convert (ITIMES A B) to A*B, A*B would not translate back correctly to (ITIMES A B), since it would be the name of a variable, and therefore would not cause an error. You can prevent this from happening by adding A*B to the list FUNNYATOMLST. Then, (ITIMES A B) would CLISPIFY to A * B.

Note that A*B's appearance on FUNNYATOMLST would *not* enable DWIM and CLISP to decode A*B+C as (IPLUS A*B C); FUNNYATOMLST is used only by CLISPIFY. Thus, if an identifier contains a CLISP character, it should always be separated (with spaces) from other operators. For example, if X* is a variable, you should write (SETQ X* *FORM*) in CLISP as X* *_FORM*, not X**_FORM*. In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

Miscellaneous Functions and Variables

CLISPFLG [Variable]

If CLISPFLG = NIL, disables all CLISP infix or prefix transformations (but does not affect IF/THEN/ELSE statements, or iterative statements).

If CLISPFLG = TYPE-IN, CLISP transformations are performed only on expressions that are typed in for evaluation, i.e., not on user programs.

If CLISPFLG = T, CLISP transformations are performed on all expressions.

The initial value for CLISPFLG is T. CLISPIFYing anything will cause CLISPFLG to be set to T.

CLISPCHARS [Variable]

A list of the operators that can appear in the interior of an atom. Currently (+ - 20-19^

```
~ ' = _ : < > +- ~= @ !).
```

CLISPCHARRAY [Variable]

A bit table of the characters on CLISPCHARS used for calls to STRPOSL (Chapter 4). CLISPCHARRAY is initialized by performing (SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS)).

CLISPINFIXSPLST [Variable]

A list of infix operators used for spelling correction.

CLISPARRAY [Variable]

Hash array used for storing CLISP translations. CLISPARRAY is checked by FAULTEVAL and FAULTAPPLY on erroneous forms before calling DWIM, and by the compiler.

(**CLEARCLISPARRAY** *NAME* --) [Function]

Macro and CLISP expansions are cached in CLISPARRAY, the systems CLISP hash array. When anything changes that would invalidate an expansion, it needs to be removed from the cache. CLEARCLISPARRAY does this for you. The system does this automatically whenever you define redefine a CLISP or macro form. If you have changed something that a CLISP word or a macro depends on the system will not be able to detect this, so you will have to invalidate the cahce by calling CLEARCLISPARRAY. You can clear the whole cache by calling (CLRHASH CLISPARRAY).

(**CLISPTRAN** *X TRAN*) [Function]

Gives *X* the translation *TRAN* by storing (key *X*, value *TRAN*) in the hash array CLISPARRAY. CLISPTRAN is called for all CLISP translations, via a non-linked, external function call, so it can be advised.

(**CLISPDEC** *DECLST*) [Function]

Puts into effect the declarations in *DECLST*. CLISPDEC performs spelling corrections on words not recognized as declarations. CLISPDEC is undoable.

(**CLDISABLE** *OP*) [Function]

Disables the CLISP operator *OP*. For example, (CLDISABLE '-) makes - be just another character. CLDISABLE can be used on all CLISP operators, e.g., infix operators, prefix operators, iterative statement operators, etc. CLDISABLE is undoable.

Note: Simply removing a character operator from CLISPCHARS will prevent it from being treated as a CLISP operator when it appears as part of an atom, but it will continue to be an operator when it appears as a separate atom, e.g. (FOO + X) vs FOO+X.

CLISPIFTRANFLG [Variable]

Affects handling of translations of IF-THEN-ELSE statements (see Chapter 9). If T, the translations are stored elsewhere, and the (modified) CLISP retained. If NIL, the

corresponding COND expression replaces the CLISP. Initially T.

CLISPIFYPRETTYFLG [Variable]

If non-NIL, causes PRETTYPRINT (and therefore PP and MAKEFILE) to CLISPIFY selected function definitions before printing them according to the following interpretations of CLISPIFYPRETTYFLG:

ALL	Clispify all functions.
T or **EXPRS**	Clispify all functions currently defined as EXPRs.
CHANGES	Clispify all functions marked as having been changed.
a list	Clispify all functions in that list.

CLISPIFYPRETTYFLG is (temporarily) reset to T when MAKEFILE is called with the option CLISPIFY, and reset to CHANGES when the file being dumped has the property FILETYPE value CLISP. CLISPIFYPRETTYFLG is initially NIL.

Note: If CLISPIFYPRETTYFLG is non-NIL, and the only transformation performed by DWIM are well formed CLISP transformations, i.e., no spelling corrections, the function will *not* be marked as changed, since it would only have to be re-clispified and re-prettyprinted when the file was written out.

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

Both a function and an edit macro for prettyprinting translations. It performs a PP after first resetting PRETTYTRANFLG to T, thereby causing any translations to be printed instead of the corresponding CLISP.

CLISP: [Editor Command]

Edit macro that obtains the translation of the correct expression, if any, from CLISPARRAY, and calls EDITE on it.

CL [Editor Command]

Edit macro. Replaces current expression with CLISPIFYed current expression. Current expression can be an element or tail.

DW [Editor Command]

Edit macro. DWIMIFYs current expression, which can be an element (atom or list) or tail.

Both CL and DW can be called when the current expression is either an element or a tail and will work properly. Both consult the declarations in the function being edited, if any, and both are undoable.

(**LOWERCASE** *FLG*) [Function]

If *FLG* = T, LOWERCASE makes the necessary internal modifications so that CLISPIFY will

use lower case versions of AND, OR, IF, THEN, ELSE, ELSEIF, and all i.s. operators. This produces more readable output. Note that you can always type in *either* upper or lower case (or a combination), regardless of the action of LOWERCASE. If *FLG* = NIL, CLISPIFY will use uppercase versions of AND, OR, et al. The value of LOWERCASE is its previous "setting". LOWERCASE is undoable. The initial setting for LOWERCASE is T.

CLISP Internal Conventions

CLISP is almost entirely table driven by the property lists of the corresponding infix or prefix operators. For example, much of the information used for translating the + infix operator is stored on the property list of the symbol "+". Thus it is relatively easy to add new infix or prefix operators or change old ones, simply by adding or changing selected property values. (There *is* some built in information for handling minus, **:**, **'**, and ~, i.e., you could not yourself add such "special" operators, although you can disable or redefine them.)

Global declarations operate by changing the LISPFN and CLISPINFIX properties of the appropriate operators.

CLISPTYPE [Property Name]

The property value of the property CLISPTYPE is the precedence number of the operator: higher values have higher precedence, i.e., are tighter. Note that the actual value is unimportant, only the value relative to other operators. For example, CLISPTYPE for **:**, ^, and * are 14, 6, and 4 respectively. Operators with the same precedence group left to right, e.g., / also has precedence 4, so A/B*C is (A/B)*C.

An operator can have a different left and right precedence by making the value of CLISPTYPE be a dotted pair of two numbers, e.g., CLISPTYPE of _ is (8 . -12). In this case, CAR is the left precedence, and CDR the right, i.e., CAR is used when comparing with operators on the *left*, and CDR with operators on the *right*. For example, A*B_C+D is parsed as A*(B_(C+D)) because the left precedence of _ is 8, which is higher than that of *, which is 4. The right precedence of _ is -12, which is lower than that of +, which is 2.

If the CLISPTYPE property for any operator is removed, the corresponding CLISP transformation is disabled, as well as the inverse CLISPIFY transformation.

UNARYOP [Property Name]

The value of property UNARYOP must be T for unary operators or brackets. The operand is always on the right, i.e., unary operators or brackets are always prefix operators.

BROADSCOPE [Property Name]

The value of property BROADSCOPE is T if the operator has lower precedence than Interlisp forms, e.g., LT, EQUAL, AND, etc. For example, (FOO X AND Y) parses as ((FOO X) AND Y). If the BROADSCOPE property were removed from the property list of AND, (FOO X AND Y) would parse as (FOO (X AND Y)).

LISPFN [Property Name]

The value of the property LISPFN is the name of the function to which the infix operator translates. For example, the value of LISPFN for ^ is EXPT, for ' QUOTE, etc. If the value of the property LISPFN is NIL, the infix operator itself is also the function, e.g., AND, OR, EQUAL.

SETFN [Property Name]

If FOO has a SETFN property FIE, then (FOO --)_X translates to (FIE -- X). For example, if you make ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then you can also make # followed by _ translate to SETA, e.g., X#N_Y to (SETA X N Y), by putting SETA on the property list of ELT under the property SETFN. Putting the list (ELT) on the property list of SETA under property SETFN will enable SETA forms to CLISPIFY back to ELT's.

CLISPINFIX [Property Name]

The value of this property is the CLISP infix to be used in CLISPIFYing. This property is stored on the property list of the corresponding Interlisp function, e.g., the value of property CLISPINFIX for EXPT is ^, for QUOTE is ' etc.

CLISPWORD [Property Name]

Appears on the property list of clisp operators which can appear as CAR of a form, such as FETCH, REPLACE, IF, iterative statement operators, etc. Value of property is of the form (*KEYWORD* . *NAME*), where *NAME* is the lowercase version of the operator, and *KEYWORD* is its type, e.g. FORWORD, IFWORD, RECORDWORD, etc.

KEYWORD can also be the name of a function. When the atom appears as CAR of a form, the function is applied to the form and the result taken as the correct form. In this case, the function should either physically change the form, or call CLISPTRAN to store the translation.

As an example, to make & be an infix character operator meaning OR, you could do the following:

```
_(PUTPROP '& 'CLISPTYPE (GETPROP 'OR 'CLISPTYPE))
_(PUTPROP '& 'LISPFN 'OR)
_(PUTPROP '& 'BROADSCOPE T)
_(PUTPROP 'OR 'CLISPINFIX '&)
_(SETQ CLISPCHARS (CONS '& CLISPCHARS))
_(SETQ CLISPCHARRAY (MAKEBITTABLE CLISPCHARS))
```