CHAPTER 16

CLISP

The syntax of LISP is very simple, in the sense that it can be described concisely, but not in the sense that LISP programs are easy to read or write! This simplicity of syntax is achieved by, and at the expense of, extensive use of explicit structuring, namely grouping through parenthesization. Unlike many languages, there are no reserved words in LISP such as `IF`, `THEN`, `FOR`, `DO`, etc., nor reserved characters like `+`, `-`, `=`, `_`, etc. The only special characters are left and right parentheses and period, which are used for indicating structure, and space and end-of-line, which are used for delimiting identifiers. This eliminates entirely the need for parsers and precedence rules in the LISP interpreter and compiler, and thereby makes program manipulation of LISP programs straightforward. In other words, a program that ‘‘looks at’’ other LISP programs does not need to incorporate a lot of syntactic information. For example, a LISP interpreter can be written in one or two pages of LISP code. It is for this reason that LISP is by far the most suitable, and frequently used, programming language for writing programs that deal with other programs as data, e.g., programs that analyze, modify, or construct other programs.

However, it is precisely this same simplicity of syntax that makes LISP programs difficult to read and write (especially for beginners). ’Pushing down’ is something programs do very well, and people do poorly. As an example, consider the following two ‘‘equivalent’’ sentences:

‘‘The rat that the cat that the dog that I owned chased caught ate the cheese.’’

versus

‘‘I own the dog that chased the cat that caught the rat that ate the cheese.’’

Natural language contains many linguistic devices such as that illustrated in the second sentence above for minimizing embedding, because embedded sentences are more difficult to grasp and understand than equivalent non-embedded ones (even if the latter sentences are somewhat longer). Similarly, most high level programming languages offer syntactic devices for reducing apparent depth and complexity of a program: the reserved words and infix operators used in ALGOL-like languages simultaneously delimit operands and operations, and also convey meaning to the programmer. They are far more intuitive than parentheses. In fact, since LISP uses parentheses (i.e., lists) for almost all syntactic forms, there is very little information contained in the parentheses for the person reading a LISP program, and so the parentheses tend mostly to be ignored: the meaning of a particular LISP expression for people is found almost entirely in the *words*, not in the structure. For example, the expression

```
(COND (EQ N 0) 1) (T TIMES N FACTORIAL ((SUB1 N)))
```

is recognizable as factorial even though there are five misplaced or missing parentheses. Grouping words together in parentheses is done more for LISP’s benefit, than for the programmer’s.

CLISP is designed to make Interlisp programs easier to read and write by permitting the user to employ various infix operators, `IF` statements (page 4.4), and iterative statements (page 4.5), which are automatically converted to equivalent Interlisp expressions when they are first interpreted. For example, factorial could be written in CLISP:

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

Note that this expression would become an equivalent COND after it had been interpreted once, so that programs that might have to analyze or otherwise process this expression could take advantage of the simple syntax.

There have been similar e orts in other LISP systems. CLISP di ers from these in that it does not attempt to *replace* the LISP syntax so much as to *augment* it. In fact, one of the principal criteria in the design of CLISP was that users be able to freely intermix LISP and CLISP without having to identify which is which. Users can write programs, or type in expressions for evaluation, in LISP, CLISP, or a mixture of both. In this way, users do not have to learn a whole new language and syntax in order to be able to use selected facilities of CLISP when and where they nd them useful.

CLISP is implemented via the error correction machinery in Interlisp (see page 15.1). Thus, any expression that is well-formed from Interlisp's standpoint will never be seen by CLISP (i.e., if the user de ned a function IF, he would e ectively turn o that part of 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. It operates as before, and when an erroneous form is encountered, the interpreter calls an error routine which in turn invokes the Do-What-I-Mean (DWIM) analyzer which contains CLISP. If the expression in question turns out to be a CLISP construct, the equivalent Interlisp form is returned to the interpreter. In addition, the original CLISP expression, is modi ed so that it *becomes* the correctly translated Interlisp form. In this way, the analysis and translation are done only once.

Integrating CLISP into the Interlisp system (instead of, for example, implementing it as a separate preprocessor) makes possible Do-What-I-Mean features for CLISP constructs as well as for pure LISP expressions. For example, if the user has de ned a function named GET-PARENT, CLISP would know not to attempt to interpret the form (GET-PARENT) as an arithmetic in x operation. (Actually, CLISP would never get to see this form, since it does not contain any errors.) If the user mistakenly writes (GET-PRAENT), CLISP would know he meant (GET-PARENT), and not (DIFFERENCE GET PRAENT), by using the information that PRAENT 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 rst expression is correct both from the standpoint of CLISP syntax and semantics and the change would be made without the user being noti ed. In the other cases, the user would be informed or consulted about what was taking place. For example, to take an extreme case, suppose the expression (LIST X*FCCT N) were encountered, where there was both a function named FACT and a variable named FCT. The user would rst be asked if FCCT were a misspelling of FCT. If he said YES, the expression would be interpreted as (LIST (X*FCT) N). If he said NO, the user would be asked if FCCT were a misspelling of FACT, i.e., if he intended X*FCCT N to mean X*(FACT N). If he said YES to this question, the indicated transformation would be performed. If he said NO, the system would then ask if X*FCCT should be treated as CLISP, since FCCT is not the name of a (bound) variable.[1] If he said YES, the expression would be transformed, if NO, it would 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 would be considered after the user said NO to X*FCCT N -> X*(FACT N).

Note: Through the discussion above, we speak of CLISP or DWIM asking the user. Actually, if the expression in question was typed in by the user for immediate execution, the user is simply informed of the transformation, on the grounds that the user would prefer an occasional misinterpretation rather than being continuously bothered, especially since he can always retype what he 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 user programs, the user can inform CLISP whether he wishes to operate in CAUTIOUS or TRUSTING mode. In the former case (most typical) the user 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 the user.

CLISP can also handle parentheses errors caused by typing 8 or 9 for ‘‘(’’ or ‘‘)’’. (On most terminals, 8 and 9 are the lower case characters for ‘‘(’’ and ‘‘)’’, i.e., ‘‘(’’ and 8 appear on the same key, as do ‘‘)’’ and 9.) For example, if the user writes 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 5 spelling corrections and fixing the parenthesis error.[2]

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

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

However, the user 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, he can also misspell PRINT, PRIMEP, M or N! In this example, the only thing the user 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.

[1]This question 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 the user if A/B is intended to be CLISP, in case the user really does have a free variable named A/B.

[2]CLISP also contains a facility for converting from Interlisp back to CLISP, so that after running the above incorrect definition of FACTORIAL, the user could ‘‘clispify’’ the now correct version to obtain (IF N=0 THEN 1 ELSE N*(FACTORIAL N-1)).

CLISP is well integrated into the Interlisp system. For example, the above iterative statement translates into an following 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 the user PRETTYPRINTs his program, PRETTYPRINT ‘‘knows’’ to print the original CLISP at the corresponding point in his function. Similarly, when the user edits his program, the editor keeps the translation invisible to the user. If the user modi es 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 (page 8.1). Whereas the programmer’s assistant is an invisible intermediary agent between the user’s console requests and the Interlisp executive, CLISP sits between the user’s programs and the Interlisp interpreter.

Only a small e ort has been devoted to de ning the core syntax of CLISP. Instead, most of the e ort 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 e ect in the recipient. CLISP operates under the assumption that what the user said was *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 the user with many di erent ways of saying the same thing, but to enable him to worry less about the *syntactic* aspects of his communication with the system. In other words, it gives the user a new degree of freedom by permitting him 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 the user de nes 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 the user, a warning message is printed, e.g., (WHILE DEFINED, THEREFORE DISABLED IN CLISP).

16.1 CLISP INTERACTION WITH USER

Syntactically and semantically well formed CLISP transformations are always performed without informing the user. Other CLISP transformations described in the previous section, e.g., misspellings of operands, in x operators, parentheses errors, unary minus - binary minus errors, all follow the same protocol as other DWIM transformations (page 15.1). That is, if DWIM has been enabled in TRUSTING mode, or the transformation is in an expression typed in by the user for immediate execution, user approval is not requested, but the user is informed.[3] However, if the transformation involves a user program, and DWIM was enabled in CAUTIOUS mode, the user will be asked to approve. If he says 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 the user is in

[3]However, in certain situations, DWIM will ask for approval even if DWIM is enabled in TRUSTING mode. For example, the user will always be asked to approve a spelling correction that might also be interpreted as a CLISP transformation, as in LAST-ELL -> LAST-EL.

CAUTIOUS mode and the error is in a program, the corresponding transformation will be performed only if the user approves (or defaults by not responding). If the user says 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) the user would be asked A*B [IN FOO] TREAT AS CLISP ?[4]

If the user 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 the user said NO, A*B would be left alone.[5]

16.2 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 litatom A+B to the form (IPLUS A B). Note that CLISP is envoked, 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 litatom A+B. Therefore the user may choose not to use these facilities with no penalty, similar to other CLISP facilities.

The user has a lot of flexability in using CLISP character operators. A list, can always be substituted for a litatom, 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 litatom "LIST" and the litatom "A+B". Since (LIST (FOO X)+(FIE Y)) is indistinguishable from (LIST (FOO X)t+t(FIE Y)) because spaces before or after parentheses have no effect on the Interlisp READ program,[6] to be consistent, extra spaces have no effect on atomic operands either. In other words, CLISP will treat (LIST A+tB), (LIST At+B), and (LIST At+tB) the same as (LIST A+B).

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

CLISP recognizes +, -, *, /, and ^ as the normal arithmetic infix operators. - is also recognized as the prefix operator, unary minus. These are converted to IPLUS, IDIFFERENCE (or in the case of unary minus, IMINUS), ITIMES, IQUOTIENT, and EXPT.

---

[4]The waiting time on such interactions is three times as long as for simple corrections, i.e., 3*DWIMWAIT.

[5]If the value of CLISPHELPFLG = NIL (initally T), the user 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 the user had been asked and said NO.

[6]CLISP does not use its own special READ program because this would require the user to explicitly identify CLISP expressions, instead of being able to intermix Interlisp and CLISP.

CLISP Character Operators

The I in IPLUS denotes integer arithmetic, i.e., IPLUS converts its arguments to integers, and returns an integer value. Interlisp also contains oating point arithmetic functions as well as mixed arithmetic functions. Floating point arithmetic functions are used in the translation if one or both of the operands are themselves oating point numbers, e.g., X+1.5 translates as (FPLUS X 1.5). In addition, 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 (see page 16.9).

The usual precedence rules apply (although these can be easily changed by the user), 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 rst 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 the user is 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 in x 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 in x operators for ‘‘Equal’’, ‘‘Greater Than’’, ‘‘Less Than’’, ‘‘Greater Than or Equal’’, and ‘‘Less Than or Equal’’.

GT, LT, GE, and LE are all a ected by the same declarations as + and *, with the initial default to use IGREATERP and ILESSP.

Note that only single character operators, e.g., +, _, =, etc., can appear in the *interior* of an atom. All other operators must be set o from identi ers 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 in x operators.[7] AND is higher than OR, and both AND and OR are lower than the other in x operators, so

[7]Currently the complete list is MEMBER, MEMB, FMEMB, ILESSP, IGREATERP, LESSP, GREATERP, FGTP, EQ, NEQ, EQP, EQUAL, OR, and AND. New in x operators can be easily added, as described in page 16.21. Spelling correction on misspelled in x operators is peformed using CLISPINFIXSPLST as a spelling list.

(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 in x 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, the user can easily change this.

: [CLISP Operator]

X:N extracts the Nth element of the list X. FOO:3 speci es 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 rst element of FOO, or (CADAR FOO).

The : operator can also be used for extracting substructures of records (see page 3.1). Record operations are implemented by replacing expressions of the form X:FOO by (fetch FOO of X). Both lower and upper case are acceptable.

: is also used to indicate operations in the pattern match facility (page 23.1).

:: [CLISP Operator]

X:N, returns the Nth *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 modi cation. 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).

The user can indicate he wants /RPLACA and /RPLACD used (undoable version of RPLACA and RPLACD, see page 8.22), or FRPLACA and FRPLACD (fast versions of RPLACA and RPLACD, see page 2.15), by means of CLISP declarations (page 16.9). 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 (page 23.1).

_ has di erent 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 typein, $_FORM (<esc>_FORM) is equiv alent to set the ‘‘last thing mentioned''.[8] For example, immediately after examining the value of LONGVARIABLENAME, the user 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 ’tC> 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.

[8] i.e., is equivalent to (SET LASTWORD FORM). See page 15.15.

On typein, '$ (i.e., '<esc>) is equivalent to (QUOTE VALUE-OF-LASTWORD) (see page 15.15). For example, after calling PRETTYPRINT on LONGFUNCTION, the user could move its definition to FOO by typing (MOVD '$ 'FOO).[9]

~ [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 Prededence 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

16.3 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 (IPLUS A B), (FPLUS A B), or (PLUS 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.

[9] *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.

The user 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 `INTEGER` and `STANDARD`.

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

Declaration	Interlisp Functions to be used
`INTEGER` or `FIXED`	`IPLUS`, `IMINUS`, `IDIFFERENCE`, `ITIMES`, `IQUOTIENT`, `ILESSP`, `IGREATERP`
`FLOATING`	`FPLUS`, `FMINUS`, `FDIFFERENCE`, `FTIMES`, `FQUOTIENT`, `LESSP`, `FGREATERP`
`MIXED`	`PLUS`, `MINUS`, `DIFFERENCE`, `TIMES`, `QUOTIENT`, `LESSP`, `GREATERP`
`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

16.3.1 Local Declarations

The user 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.[10] The user 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. Note that 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).

16.4 CLISP OPERATION

CLISP is a part of the basic Interlisp system. Without any special preparations, the user 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.[11]

However, if a CLISP construct contains an error, an appropriate diagnostic is generated, and the form is left unchanged. For example, if the user writes (LIST X+Y*), the error diagnostic MISSING OPERAND AT X+Y* IN (LIST X+Y*) would be generated. Similarly, if the user writes (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, the user might have meant to

[10] ‘‘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.

[11] This entire discussion also applies to CLISP transformation initiated by calls to DWIM from DWIMIFY.

say(LAST+EL*X).) Note that if LAST+EL were the name of a de ned 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).[12] For example, suppose the user types (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.

If a CLISP in x construct is well formed from a syntactic standpoint, but one or both of its operands are atomic and not bound,[13] it is possible that either the operand is misspelled, e.g., the user 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 example, if the user has a variable named LAST-EL, and writes (LIST LAST-ELL). Therefore, CLISP computes, but does not actually perform, the indicated in x 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 the user whether CLISP was intended, in this case by printing LAST-ELL TREAT AS CLISP ?[14]

The same sort of procedure is followed with 8 and 9 errors. For example, suppose the user writes FOO8*X where FOO8 is not bound. The CLISP transformation is noted, and DWIM proceeds. It next asks the user to approve FOO8*X -> FOO (*X. (For example, this would make sense if the user has (or plans to de ne) a function named *X.) If he refuses, the user is asked whether FOO8*X is to be treated as CLISP. Similarly, if FOO8 were the name of a variable, and the user writes FOOO8*X, he will rst be asked to approve FOOO8*X -> FOOO (XX,[15] and if he refuses, then be o ered the FOOO8 -> FOO8 correction.

CLISP also contains provision for correcting misspellings of in x 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 the user writes 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 nd any other interpretation for ELSSE, and using the

---

[12]Except that CLISP error messages are not printed on type-in. For example, typing X+*Y will just produce a U.B.A. X+*Y message.

[13]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.

[14]If more than one in x 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), the user will simply be informed of the correction, e.g., X+Y+Z TREATED AS CLISP. Otherwise, even if DWIM was enabled in TRUSTING mode, the user will be asked to approve the correction.

[15]The 8-9 transformation is tried before spelling correction since it is empirically more likely that an unbound atom or unde ned function containing an 8 or a 9 is a parenthesis error, rather than a spelling error.

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 the user. 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.

16.5 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.[16] 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 the user 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 the user to configure his Interlisp system so that all expressions are automatically `CLISPIFY`ed immediately before they are presented to him. For example, he can define an edit macro to use in place of `P` which calls `CLISPIFY` on the current expression before printing it. Similarly, he 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[17] elsewhere, usually in the hash array `CLISPARRAY`.[18] The interpreter automatically checks this array when

---

[16]If `CLISPIFTRANFLG` is `T`, the original CLISP for `IF` statements (modified to correct errors) is retained. See page 16.20.

[17]by the function `CLISPTRAN` (page 16.19).

[18]The user can also indicate that he wants 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 in the text. Furthermore, such expressions will be treated as CLISP even if infix and prefix transformations have been disabled by setting `CLISPFLG` to `NIL` (page 16.19). In other words, the user can instruct the system to interpret as CLISP infix or prefix constructs only those expressions that are specifically flagged as such. The user 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. Note this feature provides a way of including CLISP declarations in macro definitions.

given a form CAR of which is not a function.[19] 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 the user *changes* a CLISP expresson by editing it, the editor automatically deletes its translation (if one exists), so that the next time it is evaluated or dwimi ed, the expression will be retranslated.[20] The function PPT and the edit commands PPT and CLISP: are available for examining translations (page 16.20). If PRETTYTRANFLG is T, PRETTYPRINT will print the translations instead of the corresponding CLISP expression (see page 16.20). This can be used for exporting programs to systems that do not provide CLISP, and to examine translations for debugging purposes.

16.6 DWIMIFY

DWIMIFY is e ectively 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 ‘‘ x’’ it. DWIMIFY performs *all* DWIM transformations, not just CLISP transformations, so it does spelling correction, xes 8-9 errors, handles F/L, etc. Thus the user will see the same messages, and be asked for approval in the same situations, as he 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 that the argument of NLAMBDAs are not evaluated.[21] It also knows how variables are bound.[22] 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.[23] Similarly, DWIMIFY will attempt to correct forms for which CAR is unde ned, even when the form is not in error from the user’s standpoint, but the corresponding function has simply not yet been de ned.

---

[19]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 de nition cells. In all cases, if CAR of the object is not LAMBDA or NLAMBDA, the interpreter and compiler will check CLISPARRAY.

[20]If the value of CLISPRETRANFLG is T, DWIMIFY will also (re)translate any expressions which have translations stored remotely. See page 16.16.

[21]The user can inform DWIMIFY that an NLAMBDA function *does* evaluate its arguments (presumably by direct calls to EVAL), by including on its property list the property INFO with value EVAL or a list which contains the atom EVAL.

[22]The user can inform DWIMIFY that a particular function or construct binds variables by including the atom BINDS on the INFO property for CAR of the form. In this case, DWIMIFY assumes that CADR of the form is the variable list, i.e. a list of atoms, or lists of the form (VAL VALUE). LAMBDA, NLAMBDA, PROG, and RESETVARS are handled in this fashion.

[23]DWIMIFY rebinds FIXSPELLDEFAULT to N, so that if the user is not at the terminal when dwimifying (or compiling), spelling corrections will not be performed.

DWIMIFY will also inform the user when it encounters an expression with too *many* arguments,[24] 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 the user 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,[25] and if so, will alert the user 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 the user intended will have no e ect (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, the user 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 the user could achieve the same e ect by simply setting the corresponding variables, and giving the functions dummy de nitions.

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. The user 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 x 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 unde ned at the time FOO is dwimi ed, DWIMIFY will not bother with FIE after the rst 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 de ned 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 de ned or mentioned, if the function is DWIMIFYed again after FOO has been de ned, the correction will be made.

The user can undo selected transformations performed by DWIMIFY, as described on page 8.11.

(DWIMIFY X QUIETFL G 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 QUIETFL G = T.

---

[24]unless DWIMCHECK#ARGSFLG = NIL (initially T).

[25]unless DWIMCHECKPROGLABELSFLG = NIL (initially T), or the label is a member of NOFIXVARSLST.

If X is an atom and L is NIL, X is treated as the name of a function, and its entire definition is dwimified. If X is a list or L is not NIL, X is the expression to be dwimified. 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]

Dwimifies 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 dwimified. 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 dwimified.

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 the user for approval of any CLISP transformations. Instead, in those situations where approval would be required, the effect is the same as though the user 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.

16.7 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 rst 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.[26] CLISPIFY will not convert expressions appearing as arguments to NLAMBDA functions.[27]

Note: Disabling a CLISP operator with CLDISABLE (page 16.19) will also disable the corresponding CLISPIFY transformation. Thus, if _ is ‘‘turned o ’’, A_B will not transform to (SETQ A B), nor vice versa.

(CLISPIFY X L) [Function]

Clispi es X. If X is an atom and L is NIL, X is treated as the name of a function, and its de nition (or EXPR property) is clispi ed. After CLISPIFY has nished, X is rede ned (using /PUTD) with its new CLISP de nition. 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 L is not NIL, X itself is the expression to be clispi ed. If L 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 clispi ed version of X.

(CLISPIFYFNS FN_1 FN_N) [NLambda NoSpread Function]

Like DWIMIFYFNS (page 16.16) except calls CLISPIFY instead of DWIMIFY.

---

[26]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.

[27]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.

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 --). Note that regardless of the setting of this flag, 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 At+tBt*tC.

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 the user has 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. The user can prevent this from happening by adding A*B to the list FUNNYATOMLST. Then, (ITIMES A B) would CLISPIFY to At*tB.

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, the user should write (SETQ X* FORM) in CLISP as X*t_FORM , not X*_FORM . In general, it is best to avoid use of identifiers containing CLISP character operators as much as possible.

16.8 MISCELLANEOUS FUNCTIONS AND VARIABLES

CLISPFLG [Variable]

If set to 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 (+ - * / ^ ~ ' = _ : < > +- ~= @ !).

CLISPCHARRAY [Variable]

A bit table of the characters on CLISPCHARS used for calls to STRPOSL (page 2.31). 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.

(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 (see page 16.9). 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]

A ects handling of translations of IF|THEN|ELSE statements (see page 4.4). If T, the translations are stored elsewhere, and the (modi ed) 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 de nitions before printing them according to the following interpretations of CLISPIFYPRETTYFLG:

ALL	Clispify all functions.
T or EXPRS	Clispify all functions currently de ned 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 le 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-clispi ed and re-prettyprinted when the le was written out.

PRETTYTRANFLG [Variable]

If T, causes PRETTYPRINT to print translations instead of CLISP expressions. This is useful for exporting to a LISP system that does not have CLISP. PRETTYTRANFLG is (temporarily) reset to T when MAKEFILE is called with the option NOCLISP. PRETTYTRANFLG is initially NIL.

(PPT x) [NLambda NoSpread Function]

Both a function and an edit macro for prettyprinting translations. It performs a PP after rst 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. `DWIMIFY`s 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 the user 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`.

16.9 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 litatom ‘‘`+`’’. 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., the user could not himself add such ‘‘special’’ operators, although he 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 the user makes ELT be an infix operator, e.g. #, by putting appropriate CLISPTYPE and LISPFN properties on the property list of # then he 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 (page 16.19) to store the translation.

As an example, to make & be an infix character operator meaning OR, the user 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))
```