CHAPTER 23

LISPUSERS PACKAGES

This chapter describes packages which are of sufficient utility that they would otherwise be included as part of the Interlisp system, except for virtual address space limitations. These packages normally reside on the directory `<LISPUSERS>`.

23.1 PATTERN MATCH COMPILER

Note: The pattern match compiler is a LispUsers package which can be loaded from the file `MATCH.DCOM`*. The entries have a* `FILEDEF` *property (see page 15.8), so simply using a pattern match construct will cause the file to be loaded automatically.*

The pattern match compiler provides a fairly general pattern match facility within CLISP. This facility allows the user to specify certain tests that would otherwise be clumsy to write, by giving a pattern which the datum is supposed to match. Essentially, the user writes ‘‘Does the (expression) X look like (the pattern) P?’’ For example, `X:(& 'A -- 'B)` asks whether the second element of `X` is an `A`, and the last element a `B`. The implementation of the matching is performed by computing (once) the equivalent Interlisp expression which will perform the indicated operation, and substituting this for the pattern, and *not* by invoking each time a general purpose capability such as that found in FLIP or PLANNER. For example, the translation of `X:(& 'A -- 'B)` is:

```
(AND (EQ (CADR X) 'A)
     (EQ (CAR (LAST X)) 'B))
```

Thus the CLISP pattern match facility is really a Pattern Compiler, and the emphasis in its design and implementation has been more on the efficiency of object code than on generality and sophistication of its matching capabilities. The goal was to provide a facility that could and would be used even where efficiency was paramount, e.g., in inner loops. As a result, the CLISP pattern match facility does not contain (yet) some of the more esoteric features of other pattern match languages, such as repeated patterns, disjunctive and conjunctive patterns, recursion, etc. However, the user can be confident that what facilities it does provide will result in Interlisp expressions comparable to those he would generate by hand.[1]

The syntax for pattern match expressions is `FORM:PATTERN`, where `PATTERN` is a list as described below. As with iterative statements, the translation of patterns, i.e., the corresponding Interlisp expressions, are stored in the hash array `CLISPARRAY` (see page 16.19). The original expression, `FORM:PATTERN`, is replaced by an expression of the form `(MATCH FORM WITH PATTERN)`. CLISP also recognizes expressions input in this form.

[1] Wherever possible, already existing Interlisp functions are used in the translation, e.g., the translation of `($ 'A $)` uses `MEMB`, `($ ('A $) $)` uses `ASSOC`, etc.

If FORM appears more than once in the translation, and it is not either a variable, or an expression that is easy to (re)compute, such as (CAR Y), (CDDR Z), etc., a dummy variable will be generated and bound to the value of FORM so that FORM is not evaluated a multiple number of times. For example, the translation of (FOO X):($ 'A $) is simply (MEMB 'A (FOO X)), while the translation of (FOO X):('A 'B --) is:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CAR (SETQ $$2 (FOO X)))
                'A)
            (EQ (CADR $$2) 'B]
```

In the interests of e ciency, the pattern match compiler assumes that all lists end in NIL, i.e., there are no LISTP checks inserted in the translation to check tails. For example, the translation of X:('A & --) is (AND (EQ (CAR X) (QUOTE A)) (CDR X)), which will match with (A B) as well as (A . B). Similarly, the pattern match compiler does not insert LISTP checks on elements, e.g., X:(('A --) --) translates simply as (EQ (CAAR X) 'A), and X:(($1 $1 --) --) as (CDAR X).[2] Note that the user can explicitly insert LISTP checks himself by using @, as described below, e.g., X:(($1 $1 --)@LISTP --) translates as (CDR (LISTP (CAR X))).

23.1.1 Pattern Elements

A pattern consists of a list of pattern elements. Each pattern element is said to match either an element of a data structure or a segment. (cf. the editor's pattern matcher, ''--'' matches any arbitrary segment of a list, while & or a subpattern match only one element of a list.) Those patterns which may match a segment of a list are called *segment* patterns; those that match a single element are called *element* patterns.

23.1.2 Element Patterns

There are several types of element patterns, best given by their syntax:

$1 or & — Matches an arbitrary element of a list.

'EXPRESSION — Matches only an element which is equal to the given expression e.g., 'A, '(A B).

EQ, MEMB, and ASSOC are automatically used in the translation when the quoted expression is atomic, otherwise EQUAL, MEMBER, and SASSOC.

=FORM — Matches only an element which is EQUAL to the value of FORM , e.g., =X, =(REVERSE Y).

==FORM — Same as =, but uses an EQ check instead of EQUAL.

[2] The insertion of LISTP checks for *elements* is controlled by the variable PATLISTPCHECK. When PATLISTPCHECK is T, LISTP checks are inserted, e.g., X:(('A --) --) translates as: (EQ (CAR (LISTP (CAR (LISTP X)))) 'A). PATLISTPCHECK is initially NIL. Its value can be changed within a particular function by using a local CLISP declaration (see page 16.10).

ATOM — The treatment depends on setting of PATVARDEFAULT. If PATVARDEFAULT is ' or QUOTE, same as 'ATOM . If PATVARDEFAULT is = or EQUAL, same as =ATOM . If PATVARDEFAULT is == or EQ, same as ==ATOM . If PATVARDEFAULT is _ or SETQ, same as ATOM _&. PATVARDEFAULT is initially '.

PATVARDEFAULT can be changed within a particular function by using a local CLISP declaration (see page 16.10).

Note: numbers and strings are always interpreted as though PATVARDEFAULT were =, regardless of its setting. EQ, MEMB, and ASSOC are used for comparisons involving small integers.

(PATTERN $_1$ PATTERN $_N$) N 1

Matches a list which matches the given patterns, e.g., (& &), (-- 'A).

ELEMENT- PATTERN @FN

Matches an element if ELEMENT- PATTERN matches it, and FN (name of a function or a LAMBDA expression) applied to that element returns non-NIL. For example, &@NUMBERP matches a number and ('A --)@FOO matches a list whose rst element is A, and for which FOO applied to that list is non-NIL.

For ‘‘simple’’ tests, the function- object is applied before a match is attempted with the pattern, e.g., ((-- 'A --)@LISTP --) translates as (AND (LISTP (CAR X)) (MEMB 'A (CAR X))), not the other way around. FN may also be a FORM in terms of the variable @, e.g., &@(EQ @ 3) is equivalent to =3.

* — Matches any arbitrary element. If the entire match succeeds, the element which matched the * will be returned as the value of the match.

Note: Normally, the pattern match compiler constructs an expression whose value is guaranteed to be non-NIL if the match succeeds and NIL if it fails. However, if a * appears in the pattern, the expression generated could also return NIL if the match succeeds and * was matched to NIL. For example, X:('A * --) translates as (AND (EQ (CAR X) 'A) (CADR X)), so if X is equal to (A NIL B) then X:('A * --) returns NIL even though the match succeeded.

~ELEMENT- PATTERN

Matches an element if the element is *not* matched by ELEMENT- PATTERN , e.g., ~'A, ~=X, ~(-- 'A --).

(*ANY* ELEMENT- PATTERN ELEMENT- PATTERN)

Matches if any of the contained patterns match.

23.1.3 Segment Patterns

$ or -- — Matches any segment of a list (including one of zero length).

The di erence between $ and -- is in the type of search they generate. For example, X:($ 'A 'B $) translates as (EQ (CADR (MEMB 'A X)) 'B), whereas X:(-- 'A 'B $) translates as:

```
[SOME X
```

```
(FUNCTION (LAMBDA ($$2 $$1)
               (AND (EQ $$2 'A)
                    (EQ (CADR $$1) 'B]
```

Thus, a paraphrase of `($ 'A 'B $)` would be ‘‘Is the element following the *rst* `A` a `B`?’’, whereas a paraphrase of `(-- 'A 'B $)` would be ‘‘Is there *any* `A` immediately followed by a `B`?’’ Note that the pattern employing `$` will result in a more e cient search than that employing `--`. However, `($ 'A 'B $)` will not match with `(X Y Z A M O A B C)`, but `(-- 'A 'B $)` will.

Essentially, once a pattern following a `$` matches, the `$` never resumes searching, whereas `--` produces a translation that will always continue searching until there is no possibility of success. However, if the pattern match compiler can deduce from the pattern that continuing a search after a particular failure cannot possibly succeed, then the translations for both `--` and `$` will be the same. For example, both `X:($ 'A $3 $)` and `(-- 'A $3 --)` translate as `(CDDDR (MEMB (QUOTE A) X))`, because if there are not three elements following the rst `A`, there certainly will not be three elements following subsequent `A`'s, so there is no reason to continue searching, even for `--`. Similarly, `($ 'A $ 'B $)` and `(-- 'A -- 'B --)` are equivalent.

`$2`, `$3`, etc. Matches a segment of the given length. Note that `$1` is not a segment pattern.

`!ELEMENT- PATTERN`

Matches any segment which `ELEMENT- PATTERN` would match as a list. For example, if the value of `FOO` is `(A B C)`, `!=FOO` will match the segment `A B C` etc. Note that `!*` is permissible and means `VALUE-OF-MATCH _$`, e.g., `X:($ 'A !*)` translates to `(CDR (MEMB 'A X))`.

Note: since `!` appearing in front of the last pattern speci es a match with some *tail* of the given expression, it also makes sense in this case for a `!` to appear in front of a pattern that can only match with an atom, e.g., `($2 !'A)` means match if `CDDR` of the expression is the atom `A`. Similarly, `X:($ ! 'A)` translates to `(EQ (CDR (LAST X)) 'A)`.

`!ATOM` treatment depends on setting of `PATVARDEFAULT`. If `PATVARDEFAULT` is `'` or `QUOTE`, same as `!'ATOM` (see above discussion). If `PATVARDEFAULT` is `=` or `EQUAL`, same as `!=ATOM` . If `PATVARDEFAULT` is `==` or `EQ`, same as `!==ATOM` . If `PATVARDEFAULT` is `_` or `SETQ`, same as `ATOM _$`.

`.` The atom ‘‘`.`’’ is treated *exactly* like ‘‘`!`’’. In addition, if a pattern ends in an atom, the ‘‘`.`’’ is rst changed to ‘‘`!`’’, e.g., `($1 . A)` and `($1 ! A)` are equivalent, even though the atom ‘‘`.`’’ does not explicitly appear in the pattern.

One exception where ‘‘`.`’’ is not treated like ‘‘`!`’’: ‘‘`.`’’ preceding an assignment does not have the special interpretation that ‘‘`!`’’ has preceding an assignment (see below). For example, `X:('A . FOO_'B)` translates as:

```
(AND (EQ (CAR X) 'A)
     (EQ (CDR X) 'B)
     (SETQ FOO (CDR X)))
```

but `X:('A ! FOO_'B)` translates as:

```
(AND (EQ (CAR X) 'A)
     (NULL (CDDR X))
```

```
(EQ (CADR X) 'B)
(SETQ FOO (CDR X)))
```

SEGMENT- PATTERN @FUNCTION- OBJECT

Matches a segment if the segment-pattern matches it, and the function object applied to the corresponding segment (as a list) returns non-NIL. For example, ($@CDDR 'D $) matches (A B C D E) but not (A B D E), since CDDR of (A B) is NIL.

Note: an @ pattern applied to a segment will require *computing* the corresponding structure (with LDIFF) each time the predicate is applied (except when the segment in question is a tail of the list being matched).

23.1.4 Assignments

Any pattern element may be preceded by "VARIABLE _", meaning that if the match succeeds (i.e., everything matches), VARIABLE is to be set to the thing that matches that pattern element. For example, if X is (A B C D E), X:($2 Y_$3) will set Y to (C D E). Note that assignments are not performed until the entire match has succeeded, so assignments cannot be used to specify a search for an element found earlier in the match, e.g., X:(Y_$1 =Y --)[3] will *not* match with (A A B C ...), unless, of course, the value of Y was A before the match started. This type of match is achieved by using place-markers, described below.

If the variable is preceded by a !, the assignment is to the *tail* of the list as of that point in the pattern, i.e., that portion of the list matched by the remainder of the pattern. For example, if X is (A B C D E), X:($!Y_'C 'D $) sets Y to (C D E), i.e., CDDR of X. In other words, when ! precedes an assignment, it acts as a modi er to the _, and has no e ect whatsoever on the pattern itself, e.g., X:('A 'B) and X:('A !FOO_'B) match identically, and in the latter case, FOO will be set to CDR of X.

Note: *_PATTERN- ELEMENT and !*_PATTERN- ELEMENT are acceptable, e.g., X:($ 'A *_('B --) --) translates as:

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CAADR (SETQ $$2 (MEMB 'A X))) 'B)
           (CADR $$2]
```

23.1.5 Place-Markers

Variables of the form #N, N a number, are called place-markers, and are interpreted specially by the pattern match compiler. Place-markers are used in a pattern to mark or refer to a particular pattern element. Functionally, they are used like ordinary variables, i.e., they can be assigned values, or used freely in forms appearing in the pattern, e.g., X:(#1_$1 =(ADD1 #1)) will match the list (2 3). However, they are not really variables in the sense that they are not bound, nor can a function called

[3] The translation of this pattern is: (COND ((AND (CDR X) (EQUAL (CADR X) Y)) (SETQ Y (CAR X)) T)). The AND is used because if Y is NIL, the pattern should match with (A NIL), but not with just (A). The T is because (CAR X) might be NIL.

from within the pattern expect to be able to obtain their values. For convenience, regardless of the setting of PATVARDEFAULT, the rst appearance of a defaulted place-marker is interpreted as though PATVARDEFAULT were _. Thus the above pattern could have been written as X:(1 =(ADD1 1)). Subsequent appearances of a place-marker are interpreted as though PATVARDEFAULT were =. For example, X:(#1 #1 --) is equivalent to X:(#1_$1 =#1 --), and translates as (AND (CDR X) (EQUAL (CAR X) (CADR X)). (Note that (EQUAL (CAR X) (CADR X)) would incorrectly match with (NIL).)

23.1.6 Replacements

The construct PATTERN- ELEMENT _FORM speci es that if the match succeeds, the part of the data that matched is to be *replaced* with the value of FORM . For example, if X =(A B C D E), X:($ 'C $1_Y $1) will replace the third element of X with the value of Y. As with assignments, replacements are not performed until after it is determined that the entire match will be successful.

Replacements involving segments splice the corresponding structure into the list being matched, e.g., if X is (A B C D E F) and FOO is (1 2 3), after the pattern ('A $_FOO 'D $) is matched with X, X will be (A 1 2 3 D E F), and FOO will be EQ to CDR of X, i.e., (1 2 3 D E F).

Note that ($ FOO_FIE $) is ambiguous, since it is not clear whether FOO or FIE is the pattern element, i.e., whether _ speci es assignment or replacement. For example, if PATVARDEFAULT is =, this pattern can be interpreted as ($ FOO_=FIE $), meaning search for the value of FIE, and if found set FOO to it, or ($ =FOO_FIE $) meaning search for the value of FOO, and if found, store the value of FIE into the corresponding position. In such cases, the user should disambiguate by not using the PATVARDEFAULT option, i.e., by specifying ' or =.

Note: Replacements are normally done with RPLACA or RPLACD. The user can specify that /RPLACA and /RPLACD should be used, or FRPLACA and FRPLACD, by means of CLISP declarations (see page 16.9).

23.1.7 Reconstruction

The user can specify a value for a pattern match operation other than what is returned by the match by writing FORM $_1$:PATTERN =>FORM $_2$.[4] For example, X:(FOO_$ 'A --) => (REVERSE FOO) translates as:

```
[PROG ($$2)
   (RETURN
      (COND ((SETQ $$2 (MEMB 'A X))
              (SETQ FOO (LDIFF X $2))
              (REVERSE FOO]
```

Place-markers in the pattern can be referred to from within FORM , e.g., the above could also have been written as X:(!#1 'A --)=>(REVERSE #1). If -> is used in place of =>, the expression being

[4]The original CLISP is replaced by an expression of the form (MATCH FORM $_1$ WITH PATTERN => FORM $_2$). CLISP also recognizes expressions input in this form.

matched is also *physically changed* to the value of FORM . For example, X:(#1 'A !#2) -> (CONS #1 #2) would remove the second element from X, if it were equal to A.

In general, $FORM_1$:PATTERN ->$FORM_2$ is translated so as to compute $FORM_2$ if the match is successful, and then smash its value into the rst node of $FORM_1$. However, whenever possible, the translation does not actually require $FORM_2$ to be computed in its entirety, but instead the pattern match compiler uses $FORM_2$ as an indication of what should be done to $FORM_1$. For example, X:(#1 'A !#2) -> (CONS #1 #2) translates as (AND (EQ (CADR X) 'A) (RPLACD X (CDDR X))).

23.1.8 Examples

X:(-- 'A --)

-- matches any arbitrary segment. 'A matches only an A, and the second -- again matches an arbitrary segment; thus this translates to (MEMB 'A X).

X:(-- 'A)

Again, -- matches an arbitrary segment; however, since there is no -- after the 'A, A must be the last element of X. Thus this translates to: (EQ (CAR (LAST X)) 'A).

X:('A 'B -- 'C $3 --)

CAR of X must be A, and CADR must be B, and there must be at least three elements after the rst C, so the translation is:

```
(AND (EQ (CAR X) 'A)
     (EQ (CADR X) 'B)
     (CDDDR (MEMB 'C (CDDR X))))
```

X:(('A 'B) 'C Y_$1 $)

Since ('A 'B) does not end in $ or --, (CDDAR X) must be NIL.

```
(COND
   ((AND (EQ (CAAR X) 'A)
         (EQ (CADAR X) 'B)
         (NULL (CDDAR X))
         (EQ (CADR X) 'C)
         (CDDR X))
    (SETQ Y (CADDR X))
    T))
```

X:(#1 'A $ 'B 'C #1 $)

#1 is implicitly assigned to the rst element in the list. The $ searches for the rst B following A. This B must be followed by a C, and the C by an expression equal to the rst element.

```
[PROG ($$2)
   (RETURN
      (AND (EQ (CADR X) 'A)
           (EQ [CADR (SETQ $$2 (MEMB 'B (CDDR X] 'C)
           (CDDR $$2)
           (EQUAL (CADDR $$2) (CAR X]
```

X:(#1 'A -- 'B 'C #1 $)

Similar to the pattern above, except that `--` speci es a search for *any* `B` followed by a `C` followed by the rst element, so the translation is:

```
[AND (EQ (CADR X) 'A)
     (SOME (CDDR X)
           (FUNCTION (LAMBDA ($$2 $$1)
              (AND (EQ $$2 'B)
                   (EQ (CADR $$1) 'C)
                   (CDDR $$1)
                   (EQUAL (CADDR $$1) (CAR X]
```

23.2 PRINTING REENTRANT AND CIRCULAR LIST STRUCTURES

23.2.1 CIRCLPRINT

Note: CIRCLPRINT is a LispUsers package contained on the le `CIRCLPRINT.DCOM`*.*

`HPRINT` (page 6.24) is designed primarily for dumping circular or reentrant list structures (as well as other data structures for which `READ` is not an inverse of `PRINT`) so that they can be read back in by Interlisp. The CIRCLPRINT package is designed for printing circular or reentrant structures so that the user can look at them and understand them.

A reentrant list structure is one that contains more than one occurrence of the same (`EQ`) structure. For example, `TCONC` (page 2.17) makes uses of reentrant list structure so that it does not have to search for the end of the list each time it is called. Thus, if `X` is a list of 3 elements, `(A B C)`, being constructed by `TCONC`, the reentrant list structure used by `TCONC` for this purpose is:

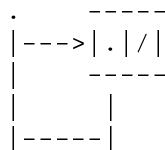
```
-----
|.|.|-----------------|
-----                 |
  |                   |
  V                   V
-----      -----      -----
|A|.|----->|B|.|----->|C|/|
-----      -----      -----
```

This structure would be printed by `PRINT` as `((A B C) C)`. Note that `PRINT` would produce the same output for the non-reentrant structure:

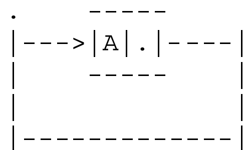
```
-----      -----
|.|.|----->|C|/|
-----      -----
  |
  V
-----      -----      -----
|A|.|----->|B|.|----->|C|/|
-----      -----      -----
```

In other words, PRINT does not indicate the fact that portions of the structure in the rst gure are identical. Similarly, if PRINT is applied to a circular list structure (a special type of reentrant structure) it will never terminate.

For example, if PRINT is called on the structure:



it will print an endless sequence of left parentheses, and if applied to:



will print a left parenthesis followed by an endless sequence of A's.

The function CIRCLPRINT described below produces output that will exactly describe the structure of any circular or reentrant list structure. This output may be in either single or double- line formats. Below are a few examples of the expressions that CIRCLPRINT would produce to describe the structures discussed above.

First Figure, single line:

```
((A B *1* C) {1})
```

First Figure, double- line:

```
((A B  C) {1})
      1
```

Third Figure, single-line:

```
(*1* {1})
```

Third Figure, double- line:

```
({1})
1
```

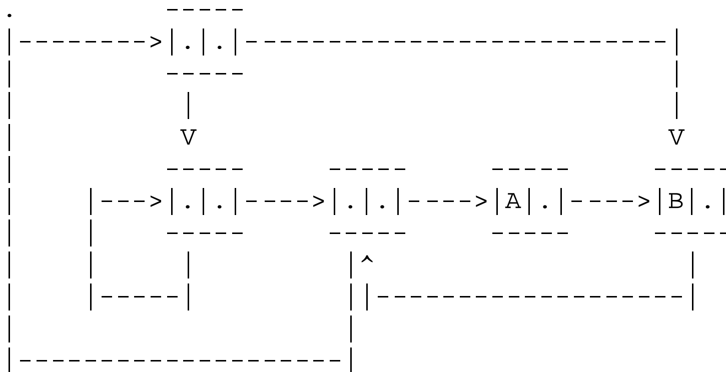
Forth Figure, single-line:

```
(*1* A . {1})
```

Forth Figure, double- line:

```
(A . {1})
1
```

The more complex structure:



is printed as follows:

Single-line:

```
(*2* (*1* {1} *3* {2} A *4* B . {3}) . {4})
```

Double-line:

```
(({1}  {2} A  B . {3}) . {4})
21    3    4
```

In both formats, the reentrant nodes in the list structure are labeled by numbers. (A reentrant node is one that has two or more pointers coming into it.) In the single-line format, the label is printed between asterisks at the beginning of the node (list or tail) that it identifies. In the double-line format, the label is printed below the beginning of the node it identifies. An occurrence of a reentrant node that has already been identified is indicated by printing its label in brackets.

(CIRCLPRINT LIST PRINTFLG RLKNT) [Function]

Prints an expression describing LIST. If PRINTFLG = NIL, double-line format is used, otherwise single-line format. CIRCLPRINT first calls CIRCLMARK, and then calls either RLPRIN1 (if PRINTFLG = T) or RLPRIN2 (if PRINTFLG = NIL). Finally, RLRESTORE is called to restore LIST to its unmarked state. Returns LIST.

(CIRCLMARK LIST RLKNT) [Function]

Marks each reentrant node in LIST with a unique number, starting at RLKNT +1 (or 1, if RLKNT is NIL). Value is (new) RLKNT.

Marking LIST physically alters it. However, the marking is performed undoably. In addition, LIST can always be restored by specifically calling RLRESTORE.

(RLPRIN1 LIST) [Function]

Prints an expression describing LIST in the single-line format. Does not restore LIST to its unCIRCLMARKed state. LIST must previously have been CIRCLMARKed or an error is generated.

(RLPRIN2 LIST) [Function]

Same as RLPRIN1, except that the expression describing LIST is printed in the double-line format.

(RLRESTORE LIST) [Function]

Physically restores list to its original, unmarked state.

Note that the user can mark and print several structures which together share common substructures, e.g., several property lists, by making several calls to CIRCLMARK, followed by calls to RLPRIN1 or RLPRIN2, and nally to RLRESTORE.

(CIRCLMAKER LIST) [Function]

LIST may contain labels and references following the convention used by CIRCLPRINT for printing reentrant structures in single line format, e.g., (*1* . {1}). CIRCLMAKER performs the necessary RPLACA's and RPLACD's to make LIST correspond to the indicated structure. Value is (altered) LIST.

(CIRCLMAKER1 LIST) [Function]

Does the work for CIRCLMAKER. Uses free variables LABELST and REFLST. LABELST is a list of dotted pairs of labels and corresponding nodes. REFLST is a list of nodes containing references to labels not yet seen. CIRCLMAKER operates by initializing LABELST and REFLST to NIL, and then calling CIRCLMAKER1. It generates an error if REFLST is not NIL when CIRCLMAKER1 returns. The user can call CIRCLMAKER1 directly to ‘‘connect up’’ several structures that share common substructures, e.g., several property lists.

23.2.2 PRINTL

Note: PRINTL is a LispUsers package contained on the le PRINTL.COM.

The PRINTL package uses a di erent scheme than CIRCLPRINT to present circular structures in an easily readable format. PRINTL uses indentation a la PRETTYPRINT to make it easier for the user to see the nesting of list structure, and prints index numbers for the beginning and ends of expressions so that the user can nd what is referred back to easily. Note that PRINTL does not provide an output format which can be read back in to reconstruct the original list structure; it is intended primarily as a debugging aid.

The following example illustrates the use of PRINTL:

```
32_(PRINTL (NCONC (SETQQ X (A B C D)) X))
  1: (A B C D . {1})                                  :1
NIL
33_(PRINTL (LIST X (CDR X) (CDDR X) (CDDDR X]
  1: ((A B C D . {2}) {3} {4} {5})                    :1
NIL
34_(PRINTL (LIST X (CONS 'P (CDR X)) (CONS 'Q (CDDR X))
(CONS 'R (CDDDR X]
  1: ((A B C D . {2})                                 :2
  6: (P . {3})                                        :6
  7: (Q . {4})                                        :7
  8: (R . {5}))                                       :1
NIL
35_USE LIST FOR CONS
  1: ((A B C D . {2})                                 :2
  6: (P {3})                                          :6
```

```
  8: (Q {4})                                              :8
  10  (R {5}))                                            :1
NIL
```

PRINTL uses the following algorithm: Each list node that is printed (CAR or CDR) is assigned a number. The second and subsequent appearences of this list node are represented simply by printing the number corresponding to the node in {} brackets. Every line on which the representation of a list begins shows the corresponding number of the *rst* such list, i.e. this number corresponds to the rst open parenthesis on the line. Similarly, to the right of every line on which a list ends is printed the number that corresponds to the *last* close parenthesis on the line. The numbers for those list nodes which do not correspond to the rst open parentheses or the last close parentheses on a line can be obtained by simply counting from the last numbered parenthesis. For example, in the line

```
1: ((A B C D . {2}) {3} {4} {5})                          :1
```

2 is (A B C D), 3 is (B C D), 4 is (C D), and 5 is (D).

(PRINTL ITEM DEPTH LMAR G RMAR G FILE) [Function]

Prints an item which is known to be, or suspected of being a circular list structure, in the form described above. DEPTH controls the depth of recursion in the CAR direction and defaults to the value of the varible PRINTDEPTH (initially 4). Elements of the structure at this depth are printed as ‘‘{--}’’.

LMAR G is the left margin. If NIL, LMAR G defaults to (POSITION FILE). RMAR G is the position at which the righthand column of numbers will be printed. If NIL, RMAR G defaults to (LINELENGTH)-5.

Printing is to FILE, which is opened if necessary.

PRINTDEPTH [Variable]

The default DEPTH argument for PRINTL. Initially 4.

(PRNTL AR GS) [Prog. Asst. Command]

Programmers Assistant command that performs (PRINTL . AR GS) provided (CAR AR GS) is not a number. If it is, or if AR GS = NIL, the item to be printed is taken to be the last event on the history list with a non-null value. Thus PRNTL 6 will print the last non-null value with DEPTH =6.

23.3 INDEXING AND CROSS REFERENCING FILES

23.3.1 SINGLEFILEINDEX

Note: SINGLEFILEINDEX is a LispUsers package that is contained on the le SINGLEFILEINDEX.DCOM.

SINGLEFILEINDEX is a package for giving the user an alphabetical function index on the front of each lisp le listed by Interlisp. This package is similar to the MULTIFILEINDEX package described below, except that SINGLEFILEINDEX provides a table of contents for functions only, and operates on one le at a time. However, SINGLEFILEINDEX is much simpler and faster than MULTIFILEINDEX and

is useful every time a le is made.

The rst page gives the lename, time of creation, and the time of the listing. Following that (on possibly more than one page) are N columns of function names and index numbers, where the index number indicates the function's linear occurrence within the le. The number of columns is determined by the length of the longest function name, as well as by the number of functions in the le as described below. The le is then printed with the lename and page number at the top of every page, and each function is preceded by its index number right-justi ed on the page.

When the SINGLEFILEINDEX package is rst loaded, it rede nes LISTFILES1 (page 11.9) so that all les listed by LISTFILES will be listed by calling (SINGLEFILEINDEX FILE NIL NIL). Note that the le being indexed does not have to be loaded, or even noticed in the le package sense.

(SINGLEFILEINDEX FILE OUTPUTFILE NEWPAGEFLG) [Function]

FILE is the lisp source le. OUTPUTFILE is the destination le. If OUTPUTFILE = NIL, then the value of PRINTER (initially LPT:) is used. NEWPAGEFLG = T means each function will be printed on a new page. The value of FILELINELENGTH determines the position of the index numbers, as well as the placement of the columns. The value of LINESPERPAGE (initially 58) determines the number of lines per page.

23.3.2 MULTIFILEINDEX

Note: MULTIFILEINDEX is a LispUsers package that is contained on the le MULTIFILEINDEX.DCOM.

Many systems built in Interlisp consist of a number of symbolic source les. Finding one's way around in the listings can often be very tedious, even for the implementor of the system, if you don't know the system and the structure of the les intimately. The MULTIFILEINDEX package is an attempt to help users deal with this problem by creating a listing of an entire system or set of les, including an alphabetized table of contents containing entries for each function on any of the les. Information (but not unique index numbers) is included for other entities in the les such as records, blocks, and properties. The function MULTIFILEINDEX implements this mechanism.

(MULTIFILEINDEX SOURCEFILES DESTINATIONFILE NEWPAGEFLG) [Function]

SOURCEFILES is a list of le names (if atomic, (LIST SOURCEFILES) is used). If it is NIL, MULTIFILEINDEX returns immediately. If it is T, the value of FILELST is used (page 11.13). DESTINATIONFILE is the output le. If DESTINATIONFILE is NIL, the value of PRINTER is used (below). If NEWPAGEFLG = T, each function in the listing will be placed on a page by itself.

In the default case, MULTIFILEINDEX does the following:

(1) Outputs an alphabetized table of contents (index) indicating the name of an object (function, record, block, variable, and so on), the le that it belongs to, and its type (property, variable (set or saved), record, block, and so forth). If the object is the name of a function, then the information includes a unique index in the listing for the function, its type (EXPR, FEXPR*, etc.), and its argument list. Note that it handles functions/ les that use DECL (page 23.18). Otherwise, the index represents the index of the function immediately preceeding the de nition of the entity.

(2) Outputs a listing of the les with each function being preceeded by its index number right-justi ed

on the line. Header information is placed at the top of each page, and the pages are numbered.

(3) Undoably removes the names of the les indexed from NOTLISTEDFILES (page 11.9).

MULTIFILEINDEX is e ected by the following variables:

MULTIFILEINDEXMAPFLG [Variable]

If T, indicates that you want the le index output. Initially T.

MULTIFILEINDEXFILESFLG [Variable]

If T, indicates that you want the le listings output to DESTINA TIONFILE . Initially T.

PRINTER [Variable]

If the MAPFILE argument to MULTIFILEINDEX is NIL, it defaults to the value of PRINTER. Initially {LPT} in Interlisp- D, LPT: in Interlisp- 10.

LINESPERPAGE [Variable]

The value of LINESPERPAGE determines the number of lines per page. Initially 65 in Interlisp- D, 58 in Interlisp- 10.

FONTCHANGEFLG [Variable]

If NIL, page headings and the index numbers that preceed the de nition of each function are printed bold; that is, overprinted; otherwise, they are printed using the BOLDFONT (PRETTYCOMFONT if BOLDFONT doesn't exist) in the current FONTPROFILE (see page 6.55).

FILELINELENGTH [Variable]

The value of FILELINELENGTH determines the width of the page.

The following four parameters a ect how the columns are placed:

MULTIFILEINDEXCOLS [Variable]
MULTIFILEINDEXNAMECOL [Variable]
MULTIFILEINDEXFILECOL [Variable]
MULTIFILEINDEXTYPECOL [Variable]

The value of MULTIFILEINDEXCOLS indicates how the other three are to be interpreted. If MULTIFILEINDEXCOLS is the atom FLOATCOLS (its initial value), then an attempt is made to t the columns onto the page in a way that maximizes the amount of space for the type information (the amount of space allocated for the type eld must be at least 45% of FILELINELENGTH in this case). If MULTIFILEINDEXCOLS is either T or FIXCOLS, then the value of the other variables are treated as absolute column positions on the page. If MULTIFILEINDEXCOLS is either NIL or FIXFLOATCOLS, the columns will be oated, but will not be any smaller than the column positions de ned by the other variables.

The initial values of these four variables are FLOATCOLS, 0, 26 and 41, respectively.

MULTIFILEINDEX has an interface to Masterscope. If the value of either of the next two variables is T, then MULTIFILEINDEX assumes that the source les have already been analyzed by Masterscope, and calls UPDATECHANGED.

MULTIFILEINDEXFNSMSFLG [Variable]

If T, indicates that you want the Masterscope information about each function output. This includes who calls each function, who this function calls, and what variables are set or referred to either locally or freely. Initially NIL.

MULTIFILEINDEXVARSMSFLG [Variable]

If T, indicates that all variables used in the les should have some information output about them at the end of the listing. The list of variables to look at is obtained by e ectively asking Masterscope the question: ‘‘WHO IS USED BY ANY AND WHO IS SET BY ANY’’. The listing will include information about who binds, uses freely or locally, or smashes freely or locally each variable. The variable map is case-independently sorted by the name of the variable. Initially NIL.

In order to make the index, or map, of the les, the lecoms for all the les being listed need be loaded (see page 11.21); MULTIFILEINDEX does a GETDEF on each le (le names are obtained using FINDFILE) to obtain its lecoms. As other indirections are noted, they also are obtained using GETDEF. For example, if you have a le TEST, and its lecoms is ((FNS * TESTFNS)), just doing a GETDEF on TESTCOMS will not su ce; as the expression (FNS * TESTFNS) is parsed, a GETDEF is also done to obtain the value of TESTFNS.

MULTIFILEINDEXLOADVARSFLG [Variable]

If T, then a LOADVARS of all the VARS on a particular le is performed before the lecoms is loaded with GETDEF. Initially NIL.

MULTIFILEINDEXGETDEFFLG [Variable]

If T, MULTIFILEINDEX will inform the user when it does GETDEFs. Initially NIL.

23.4 DATABASEFNS

Note: Databasefns is a LispUsers package that is contained on the le DATABASEFNS.DCOM.

Databasefns is a very small package whose purpose is to make the construction and maintenance of MASTERSCOPE databases an essentially automatic process. It modi es MAKEFILE, LOAD, and LOADFROM to behave in the following way:

A database will be maintained automatically for any le (containing functions) whose le name has the property DATABASE with value YES. Whenever such a le is dumped via MAKEFILE, MASTERSCOPE will analyse any new or changed functions on the le, and a database for all of the functions on the le will be written on a separate le whose name is of the form FILE.DATABASE. Whenever a le which has a DATABASE property with value YES is loaded via LOAD or LOADFROM, then the corresponding .DATABASE le, if any, is also loaded. The database will not be dumped or loaded if the value of the DATABASE property for the le is NO. The DATABASE property is considered to be NO if the le is loaded with LDFL G= SYSLOAD.

If the DATABASE property is not YES or NO, then for MAKEFILE, LOAD, and LOADFROM will ask the user whether he wants automatic database maintenance. The user's answer will be stored on the DATABASE property so that he will not be asked again. Thus when a le is dumped for the rst time, the user will

be asked ‘‘Do you want a Masterscope Database for this le?’’. Similarly, if the user loads a le which has an associated database, the user will be asked ‘‘load database for FILE?’’.

The above interactions may be controlled via the global variables SAVEDBFLG and LOADDBFLG. When a le which has neither a YES or NO database property is being dumped, MAKEFILE will assume (and store) a YES value if the value of SAVEDBLFG is YES, and a NO value if SAVEDBFLG is NO. The user will be queried only if SAVEDBFLG is ASK (its initial value). Similarly, if LOADDBFLG is YES, LOAD and LOADFROM will automatically load an existing .DATABASE le for a le which does not have a YES or NO value for its DATABASE property. The database will not be loaded if LOADDBFLG is NO, and the user will be interrogated as described above if LOADDBFLG is ASK (its initial value).

The user can dump and restore databases explicitly via the following functions:

(DUMPDB FILE) [Function]

Dumps a database for FILE then sets the DATABASE property to YES, so that database maintenance for FILE will subsequently be automatic.

(LOADDB FILE) [Function]

Loads the le FILE.DATABASE if one exists. After the database is loaded, the DATABASE property for FILE is set to YES, so that maintenance will thereafter be automatic.

Database les include the date and full lename of the le to which they correspond. LOADDB will print out a warning message if it loads a database that does not correspond to the in-core version of the le.

Note that LOADDB is the only approved way of loading a database: Attempting to load a database le will cause an error.

23.5 LAMBDATRAN

Note: Lambdatran is a LispUsers package that is contained on the le LAMBDATRAN.DCOM.

The purpose of this package is to facilitate de ning new LAMBDA words in such a way that a variety of other system packages will respond to them appropriately. A LAMBDA word is a word that can appear as CAR of a function de nition, like LAMBDA and NLAMBDA. New LAMBDA words are useful because they enable the user to de ne his own conventions about such things as the interpretation of arguments, and to build in certain defaults about how values are returned. For example, the DECL package (page 23.18) de nes DLAMBDA as a new LAMBDA word with unconventional arguments such as the following:

```
(DLAMBDA ((A FLOATP) (B FIXP) (RETURNS SMALLP)) (FOO A B))
```

In order for such an expression to be executable and compilable, a mechanism must be provided for translating this expression to an ordinary LAMBDA or NLAMBDA, with the special behavior associated with the arguments built into the function body. The lambdatran package accomplishes this via an appropriate entry on DWIMUSERFORMS (see page 15.10) that computes the translation.

Besides executing and compiling, Interlisp applies a number of other operations to function de nitions (e.g. breaking, advising), many of which depend on the system being able to determine certain properties

of the function, such as the names of its arguments, their number, and the type of the function (EXPR, FEXPR, etc.). The lambdatran package also provides new definitions for the functions FNTYP, ARGLST, NARGS, and ARGTYPE which can be told how to compute properties for the user's LAMBDA-words.

A new LAMBDA-word is defined in the following way:

1. Add the LAMBDA-word itself (e.g. the atom DLAMBDA) to the list LAMBDASPLST. This suppresses attempts to correct the spelling of the LAMBDA-word.

2. Add an entry for the LAMBDA-word to the association-list LAMBDATRANFNS, which is a list of elements of the form: (LAMBDA-WORD TRANFN FNTYP ARGLIST), where

LAMBDA-WORD is the name of the LAMBDA-word (e.g. DLAMBDA).

TRANFN is a function of one argument that will be called whenever a real definition is needed for the LAMBDA-word definition. Its argument is the LAMBDA-word definition, and its value should be a conventional LAMBDA or NLAMBDA expression which will become the translation of the LAMBDA-word form. The free variable FAULTFN is bound to the name of the function in which the LAMBDA-word form appeared (or TYPE-IN if the form was typed in).

FNTYP determines the function-type of a definition beginning with LAMBDA-WORD. It is consulted if the definition does not already have a translation from which the function type may be deduced. If FNTYP is one of the atoms EXPR, FEXPR, EXPR*, FEXPR*, then all definitions beginning with LAMBDA-word are assumed to have that type. Otherwise, FNTYP is a function of one argument that will be applied to the LAMBDA-word definition. Its value should be one of the above four function types.

ARGLIST determines the argument list of the definition if it has not already been translated (if it has, the ARGLIST is simply the ARGLIST of the translation). It is also a function of one argument, the LAMBDA-word definition, and its value should be the list of arguments for the function (e.g. (A B) in the DLAMBDA example above). If the LAMBDA-word definition is ill-formed and the argument list cannot be computed, the function should return T. If an ARGLIST entry is not provided in the LAMBDATRANFNS element, then the argument list defaults to the second element of the definition.

As an example, the LAMBDATRANFNS entry for DLAMBDA is (DLAMBDA DECL EXPR DLAMARGLIST), where DECL and DLAMARGLIST are functions of one argument.

Note: if the LAMBDA-word definition has an argument list with argument names appearing either as literal atoms or as the first element of a list, the user should also put the property INFO with value BINDS on the property list of the LAMBDA-word in order to inform DWIMIFY (page 16.14) to take notice of the names of the arguments when DWIMIFYing.

23.6 PERMSTATUS

Note: Permstatus is a LispUsers package that is contained on the file PERMSTATUS.COM.

The function PERMSTATUS defined in this package can be used in conjunction with WHENCLOSE (page 6.11) to make a file ‘‘permanently’’ open in the sense that as much of its status as possible will be restored when a SYSOUT is resumed. This includes its access mode, file-pointer position, bytesize, and any pages mapped in by the page mapping facility (page 14.17). The desired effect is achieved by saying

(WHENCLOSE FILENAME 'STATUS 'PERMSTATUS) after the file has been opened.

Note that the permanency of files is not guaranteed in that files may be deleted or renamed, or their contents changed, despite their permanent attribute in some SYSOUT. When restarting a SYSOUT, a warning message will be printed if the file cannot be found or restored. However, PERMSTATUS will not be able to detect that the contents of a file have been modified since the SYSOUT was created. Note also that ''permanent'' files will still be closed by CLOSEF, and will not be immune to CLOSEALL or to closing on end-of-file errors unless the appropriate WHENCLOSE attributes for CLOSEALL and EOF are also established.

23.7 THE DECL PACKAGE

Note: Decl is a LispUsers package that is contained on the file DECL.DCOM*. The Decl package requires the* LAMBDATRAN *package (section 23.5), so* LAMBDATRAN.DCOM *will automatically be loaded with Decl if it is not already present.*

The Decl package extends Interlisp to allow the user to declare the types of variables and expressions appearing in functions. It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing, or inefficient.

The Decl package provides a simple language for declarations, and augments the interpreter and the compiler to guarantee that these declarations are always satisfied. The declarations make programs more readable by indicating the type, and therefore something about the intended usage, of variables and expressions in the code. They facilitate debugging by localizing errors that manifest themselves as type incompatibilities. Finally, the declaration information is available for other purposes: compiler macros can consult the declarations to produce more efficient code; coercions for arguments at user interfaces can be automatically generated; and the declarations will be noticed by the Masterscope function analyzer.

The declarations interpreted by the Decl package are in terms of a set of declaration types called *decltypes*, each of which specifies a set of acceptable values and also (optionally) other type specific behavior. The Decl package provides a set of facilities for defining decltypes and their relations to each other, including type valued expressions and a comprehensive treatment of union types.

The following description of the Decl package is divided into three parts. First, the syntactic extensions which permit the concise attachment of declarations to program elements are discussed. Second, the mechanisms by which new decltypes can be defined and manipulated are covered. Finally, some additional capabilities based on the availability of declarations are outlined.

23.7.1 Using Declarations in Programs

Declarations may be attached to the values of arbitrary expressions and to LAMBDA and PROG variables throughout (or for part of) their lexical scope. The declarations are attached using constructs that resemble the ordinary Interlisp LAMBDA, PROG, and PROGN, but which also permit the expression of declarations. The following examples illustrate the use of declarations in programs.

Consider the following definition for the factorial function (FACT N):

```
[LAMBDA (N)
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

Obviously, this function presupposes that `N` is a number, and the run-time checks in `ITIMES` and `SUB1` will cause an error if this is not so. For instance, `(FACT T)` will cause an error and print the message `NON-NUMERIC ARG T`. By defining `FACT` as a `DLAMBDA`, the Decl package analog of `LAMBDA`, this presupposition can be stated directly in the code:

```
[DLAMBDA ((N NUMBERP))
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

With this definition, `(FACT T)` will *not* result in a `NON-NUMERIC ARG T` error when the body of the code is executed. Instead, the `NUMBERP` declaration will be checked when the function is first entered, and a *declaration fault* will occur. Thus, the message that the user will see will not dwell on the offending value `T`, but instead give a symbolic indication of what variable and declaration were violated, as follows:

```
DECLARATION NOT SATISFIED
((N NUMBERP) BROKEN)
:
```

The user is left in a break from which the values of variables, e.g. `N`, can be examined to determine what the problem is.

The function `FACT` also makes other presuppositions concerning its argument, `N`. For example, `FACT` will go into an infinite recursive loop if `N` is a number less than zero. Although the user could program an explicit check for this unexpected situation, such coding is tedious and tends to obscure the underlying algorithm. Instead, the requirement that `N` not be negative can be succinctly stated by declaring it to be a subtype of `NUMBERP` which is restricted to non-negative numbers. This can be done by adding a `SATISFIES` clause to `N`'s type specification:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

The predicate in the `SATISFIES` clause will be evaluated after `N` is bound and found to satisfy `NUMBERP`, but before the function body is executed. In the event of a declaration fault, the `SATISFIES` condition will be included in the error message. For example, `(FACT -1)` would result in:

```
DECLARATION NOT SATISFIED
((N NUMBERP (SATISFIES (NOT (MINUSP N))) BROKEN)
:
```

The `DLAMBDA` construct also permits the type of the value that is returned by the function to be declared by means of the pseudo-variable `RETURNS`. For example, the following definition specifies that `FACT` is to return a positive integer:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
```

```
          [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

After the function body is evaluated, its value is bound to the variable VALUE and the RETURNS declaration is checked. A declaration fault will occur if the value is not satisfactory. This prevents a bad value from propagating to the caller of FACT, perhaps causing an error far away from the source of the difficulty.

Declaring a variable causes its value to be checked not only when it is first bound, but also whenever that variable is reset by SETQ within the DLAMBDA. In other words, the type checking machinery will not allow a declared variable to take on an improper value. An iterative version of the factorial function illustrates this feature in the context of a DPROG, the Decl package analog of PROG:

```
(DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
          [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
   [DPROG ([TEMP 1 FIXP (SATISFIES (IGREATERP TEMP 0]
           [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
      LP  (COND ((EQ N 0) (RETURN TEMP)))
          (SETQ TEMP (ITIMES N TEMP))
          (SETQ N (SUB1 N))
          (GO LP]
```

DPROG declarations are much like DLAMBDA declarations, except that they also allow an initial value for the variable to be specified. In the above example, TEMP is declared to be a positive integer throughout the computation and N is declared to be non-negative. Thus, a bug which caused an incorrect value to be assigned by one of the SETQ expressions would cause a declaration failure. Note that the RETURNS declaration for a DPROG is also useful in detecting the common bug of omitting an explicit RETURN.

23.7.2 DLAMBDAs

The Decl package version of a LAMBDA expression is an expression beginning with the atom DLAMBDA. Such an expression is a function object that may be used in any context where a LAMBDA expression may be used. It resembles a LAMBDA expression except that it permits declaration expressions in its argument list, as illustrated in the examples given earlier. Each element of the argument list of a DLAMBDA may be a literal atom (as in a conventional LAMBDA) or a list of the form (NAME TYPE . EXTRAS).[5]

NAME fulfills the standard function of a parameter, i.e. providing a name to which the value of the corresponding argument will be bound.

TYPE is either a Decl package type name or type expression. When the DLAMBDA is entered, its arguments will be evaluated and bound to the corresponding argument names, and then, after *all* the argument names

[5]Strictly, this would require a declaration with a SATISFIES clause to take the form (N (NUMBERP (SATISFIES --)) --) (page 23.27). However, due to the frequency with which this construction is used, it may be written without the inner set of parentheses, e.g. (N NUMBERP (SATISFIES --) --).

have been bound, the declarations will be checked. The type checking is delayed so that SATISFIES predicates can include references to other variables bound by the same DLAMBDA. For example, one might wish to define a function whose two arguments are not only both required to be of some given type, but are also required to satisfy some relationship (e.g., that one is less than the other).

EXTRAS allows some additional properties to be attached to a variable. One such property is the accessibility of NAME outside the current lexical scope. Accessibility specifications include the atoms LOCAL or SPECIAL, which indicate that this variable is to be compiled so that it is either a LOCALVAR or a SPECVAR, respectively. This is illustrated by the following example:

```
[DLAMBDA ((A LISTP SPECIAL)
          (B FIXP LOCAL))
           ]
```

A more informative equivalent to the SPECIAL key word is the USEDIN form, the tail of which can be a list of the other functions which are expected to have access to the variable: [6]

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE))
          (B FIXP LOCAL))
           ]
```

EXTRAS may also include a comment in standard format, so that descriptive information may be given where a variable is bound:

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE)   (* This is an important variable))
          (B FIXP LOCAL))
           ]
```

As mentioned earlier, the value returned by a DLAMBDA can also be declared, by means of the pseudo-variable RETURNS. The RETURNS declaration is just like other DLAMBDA declarations, except (1) in any SATISFIES predicate, the value of the function is referred to by the distinguished name VALUE; and (2) it makes no sense to declare the return value to be LOCAL or SPECIAL.

23.7.3 DPROG

Just as DLAMBDA resembles LAMBDA, DPROG is analogous to PROG. As for an ordinary PROG, a variable binding may be specified as an atom or a list including an initial value form. However, a DPROG binding also allows TYPE and EXTRAS information to appear following the initial value form. The format for these augmented variable bindings is (NAME INITIALVALUE TYPE . EXTRAS). The only difference between a DPROG binding and a DLAMBDA binding is that the second position is interpreted as the initial value for the variable. Note that if the user wishes to supply a type declaration for a variable, an initial value *must* be specified. The same rules apply for the interpretation of the type information for DPROGs as for DLAMBDAs, and the same set of optional EXTRAs can be used. DPROGs may also declare the type of the value they return, by specifying the pseudo-variable RETURNS.

---

[6] USEDIN is mainly for documentation purposes, since there is no way for such a restriction to be enforced.

Just as for a DLAMBDA, type tests in a DPROG are not asserted until *after* all the variables have been bound, thus permitting predicates to refer to other variables being bound by this DPROG. If NIL appears as the initial value for a binding (i.e. the atom NIL actually appears in the code, not simply an expression which evaluates to NIL) the initial type test will be suppressed, but subsequent type tests, e.g. following a SETQ, will still be performed.

A common construct in Lisp is to bind and initialize a PROG variable to the value of a complicated expression in order to avoid recomputing it, and then to use this value in initializing other PROG variables, e.g.

```
[PROG ((A EXPRESSION ))
      (RETURN (PROG ((B (    A    ))
                     (C (    A    )))
                      ]
```

The ugliness of such constructions in conventional Lisp often tempts the programmer to loosen the scoping relationships of the variables by binding them all at a single level and using SETQ's in the body of the PROG to establish the initial values for variables that depend on the initial values of other variables, e.g.

```
[PROG ((A EXPRESSION ) B C)
      (SETQ B (    A    ))
      (SETQ C (    A    ))
        ]
```

In the Decl package environment, this procedure undermines the protection o ered by the type mechanism by encouraging the use of uninitialized variables. Therefore, the DPROG o ers a syntactic form to encourage more virtuous initialization of its variables. A DPROG variable list may be segmented by occurrences of the special atom THEN, which causes the binding of its variables in stages, so that the bindings made in earlier stages can be used in later ones, e.g.

```
[DPROG ((A (LENGTH FOO) FIXP LOCAL)
        THEN (B (SQRT A) FLOATP)
        THEN (C (CONS A B) LISTP))
         ]
```

Each stage is carried out as a conventional set of DPROG bindings (i.e., simultaneously, followed by the appropriate type testing). This layering of the bindings permits one to gradually descend into a inner scope, binding the local names in a very structured and clean fashion, with initial values type-checked as soon as possible.

23.7.4 Declarations in Iterative Statements

The CLISP iterative statement (page 16.1) provides a very useful facility for specifying a variety of PROGs that follow certain widely used formats. The Decl package allows declarations to be made for the scope of an iterative statement via the DECLARE CLISP i.s.opr. DECLARE can appear as an operator anywhere in an iterative statement, followed by a list of declarations, for example:

```
(for J from 1 to 10 declare (J FIXP) do
```

Note that DECLARE declarations do not *create* bindings, but merely provide declarations for existing bindings. For this reason, an initial value cannot be speci ed and the form of the declaration is the same

as that of DLAMBDAs, namely (NAME TYPE . EXTRAS).

Note that variables bound *outside* of the scope of the iterative statement, i.e. a variable used freely in the i.s, can also be declared using this construction. Such a declaration will only be in e ect for the scope of the iterative statement.

23.7.5 Declaring a Variable for a Restricted Lexical Scope

The Decl package also permits declaring the type of a variable over some restricted portion of its existence. For example, suppose the variable X is either a xed or oating number, and a program branches to treat the two cases separately. On one path X is known to be xed, whereas on the other it is known to be oating. The Decl package DPROGN construct can be used in such cases to state the type of the variable along each path. DPROGN is exactly like PROGN, except that the second element of the form is interpreted as a list of DLAMBDA format declarations. These declarations are added to any existing declarations in the containing scope, and the composite declaration (created using the ALLOF type expression, page 23.26) is considered to hold throughout the lexical scope created by the DPROGN. Thus, our example becomes:

```
(if (FIXP X)
  then (DPROGN ((X FIXP))    )
  else (DPROGN ((X FLOATP))    ))
```

Like DPROG and DLAMBDA, the value of a DPROGN may also be declared, using the pseudo- variable RETURNS.

DPROGN may be used not only to restrict the declarations of local variables, but also to declare variables which are being used freely. For example, if the variable A is used freely inside a function but is known to be FIXP, this fact could be noted by enclosing the body of the function in (DPROGN ((A FIXP FREE)) BOD Y). Instead of FREE, the more speci c construction (BOUNDIN FUNCTION$_1$ FUNCTION$_2$) can be used. This not only states that the variable is used freely but also gives the names of the functions which might have provided this binding.[7]

Since the DPROGN form introduces another level of parenthesization, which results in the enclosed forms being prettyprinted indented, the Decl package also permits such declarations to be attached to their enclosing DLAMBDA or DPROG scopes by placing a DECL expression, e.g. (DECL (A FIXP (BOUNDIN FUM)), before the rst executable form in that scope. Like DPROGN's, DECL declarations use DLAMBDA format.

23.7.6 Declaring the Values of Expressions

The Decl package allows the value of an arbitrary form to be declared with the Decl construct THE. A THE expression is of the form (THE TYPE . FORMS), e.g. (THE FIXP (FOO X)). FORMS are evaluated in order, and the value of the *last* one is checked to see if it satis es TYPE , a type name or type expression. If so, its value is returned, otherwise a declaration fault occurs.

[7]Like USEDIN declarations, FREE and BOUNDIN declarations cannot be checked, and are provided for documentation purposes only.

23.7.7 Assertions

The Decl package also allows for checking that an arbitrary predicate holds at a particular point in a program's execution, e.g. a condition that must hold at function entry but not throughout its execution. Such predicates can be checked using an expression of the form `(ASSERT` FORM$_1$ FORM$_2$ `)`, in which each FORM$_i$ is either a list (which will be evaluated) or a variable (whose declaration will be checked). Unless all elements of the `ASSERT` form are satisfied, a declaration fault will take place.

`ASSERT`ing a variable provides a convenient way of verifying that the value of the variable has not been improperly changed by a lower function. Although a similar effect could be achieved for predicates by explicit checks of the form `(OR` PREDICATE `(SHOULDNT))`, `ASSERT` also provides the ability both to check that a variable's declaration is currently satisfied and to remove its checks at compile time without source code modification (see page 23.25).

23.7.8 Using Type Expressions as Predicates

The Decl package extends the Record package `TYPE?` construct so that it accepts decltypes, as well as record names, e.g. `(TYPE? (FIXP (SATISFIES (ILESSP VALUE 0)))` EXPR `)`. Thus, a `TYPE?` expression is exactly the same as a `THE` expression except that, rather than causing a declaration fault, `TYPE?` is a predicate which determines whether or not the value satisfies the given type.

23.7.9 Enforcement

The Decl package is a ''soft'' typing system - that is, the data objects themselves are not inherently typed. Consequently, declarations can only be enforced within the lexical scope in which the declaration takes place, and then only in certain contexts. In general, changes to a variable's value such as those resulting from side effects to embedded structure (e.g., `RPLACA`, `SETN`, etc.) or free variable references from outside the scope of the declaration cannot be, and therefore are not, enforced.

Declarations *are* enforced i.e. checked, in three different situations: when a declared variable is bound to some value or rebound with `SETQ` or `SETQQ`, when a declared expression is evaluated, and when an `ASSERT` expression is evaluated. In a binding context, the type check takes place *after* the binding, including any user-defined behavior specified by the type's binding function. Any failure of the declarations causes a break to occur and an informative message to be printed. In that break, the name to which the declaration is attached (or VALUE if no name is available) will be bound to the offending value. Thus, in the `(FACT T)` example above, `N` would be bound to `T`. The problem can be repaired either by returning an acceptable value from the break via the `RETURN` command, or by assigning an acceptable value to the offending name and returning from the break via an `OK` or `GO` command. The unsatisfied declaration will be reasserted when the computation is continued, so an unacceptable value will be detected.[8]

The automatic enforcement of type declarations is a very flexible and powerful aid to program development. It does, however, exact a considerable run-time cost because of all the checking involved. Factors of two to ten in running speed are not uncommon, especially where low level, frequently used functions employ type declarations. As a result, it is usually desirable to remove the declaration enforcement code when

[8]With this exception, assignments to variables from within the break are not considered to be in the scope of the declarations that were in effect when the break took place, and so are not checked.

the system is believed to be bug-free and performance becomes more central. This can be done with the variable COMPILEIGNOREDECL:

COMPILEIGNOREDECL [Variable]

Setting the value of the variable COMPILEIGNOREDECL to T (initially NIL) instructs the compiler not to insert declaration enforcement tests in the compiled code. More selective removal can be achieved by setting COMPILEIGNOREDECL to a list of function names. Any function whose name is found on this list is compiled without declaration enforcement.

(IGNOREDECL . VAL) [File Package Command]

Declaration enforcement may be suppressed selectively by file using the IGNOREDECL file package command. If this appears in a file's file commands, it redefines the value of COMPILEIGNOREDECL to VAL for the compilation of this file only.

23.7.10 Decltypes

A Decl package type, or decltype, specifies a subset of data values to which values of this type are restricted. For example, a ‘‘positive number’’ type might be defined to include only those values that are numbers and greater than zero. A type may also specify how certain operations, such as assignment or binding (see page 23.28), are to be performed on variables declared to be of this type.

The inclusion relations among the sets of values which satisfy the different types define a natural partial ordering on types, bound by the universal type ANY (which all values satisfy) and the empty type NONE (which no value satisfies). Each type has one or more *supertypes* (each type has at least ANY as a supertype) and one or more *subtypes* (each type has at least NONE as a subtype). This structure is important to the user of Decl as it provides the framework in which new types are defined. Typically, much of the definition of a new type is defaulted, rather than specified explicitly. The definition will be completed by inheriting atttributes which are shared by all its immediate supertypes.

An initial set of decltypes which defines the Interlisp built-in datatypes and a few other commonly used types is provided. Thereafter, new decltypes are created in terms of existing ones using the type expressions described below. For conciseness, such new types can be associated with literal atoms using the function DECLTYPE (page 23.28).

23.7.11 Predefined Types

Some commonly used types, such as the Interlisp built-in data types, are already defined when the Decl package is loaded. These types, indented to show subtype-supertype relations, are:

```
ANY
  ATOM          LST[9]      ARRAYP      STRINGP     FUNCTION    STACKP
    LITATOM       ALIST[10]   HARRAYP
      NIL         LISTP        READTABLEP
    NUMBERP
      FIXP
        LARGEP
        SMALLP
      FLOATP
```

NONE

Note that the definition of LST causes NIL to have multiple supertypes, i.e. LITATOM and LST, reflecting the duality of NIL as an atom and a (degenerate) list.

In addition, declarations made using the Record package (page 3.1) also define types which are attached as subtypes to an appropriate existing type (e.g., a TYPERECORD declaration defines a subtype of LISTP, a DATATYPE declaration a subtype of ANY, etc.) and may be used directly in declaration contexts.

23.7.12 Type Expressions

Type expressions provide convenient ways for defining new types in terms of modifications to, or compositions of one or more, existing types.

(MEMQ VALUE$_1$ VALUE$_N$) [Decl Type Expression]

Specifies a type whose values can be any one of the fixed set of elements {VALUE$_1$ VALUE$_N$}. For example, the status of a device might be represented by a datum restricted to the values BUSY and FREE. Such a ''device status'' type could be defined via (MEMQ BUSY FREE). The new type will be a subtype of the narrowest type which all of the alternatives satisfy (e.g., the ''device status'' type would be a subtype of LITATOM). The membership test uses EQ if this supertype is LITATOM; EQUAL otherwise. Thus, lists, floating point numbers, etc., can be included in the set of alternatives.

(ONEOF TYPE$_1$ TYPE$_N$) [Decl Type Expression]

Specifies a type which is the union of two or more other types. For example, the notion of a possibly degenerate list is something that is either LISTP or NIL. Such a type can be (and the built-in type LST in fact is) defined simply as (ONEOF NIL LISTP). A union data type becomes a supertype of all of the alternative types specified in the ONEOF expression, and a subtype of their lowest common supertype. The type properties of a union type are taken from its alternative types if they all agree, otherwise from the supertype.

(ALLOF TYPE$_1$ TYPE$_N$) [Decl Type Expression]

Specifies a type which is the intersection of two or more other types. For example, a variable may be required to satisfy both FIXP and also some type which is defined as (NUMBERP (SATISFIES PREDICATE)). The latter type will admit numbers that are not FIXP, i.e. floating point numbers; the former does not include PREDICATE. Both restrictions can be obtained by using the type (ALLOF (NUMBERP (SATISFIES PREDICATE)) FIXP).[11]

---

[9]LST is defined as either LISTP or NIL, i.e. a list or NIL. The name LST is used, because the name LIST is treated specially by clisp.

[10]ALIST is defined as either NIL, or a list of elements each of which is of type LISTP.

[11]When a value is tested, the component type tests are applied from left to right.

(AGGREGATE OF ELEMENT) [Decl Type Expression]

Specifies a type which is an aggregate of values of some other type (e.g., list of numbers, array of strings, etc.). AGGREGATE must be a type which provides an EVERYFN property (page 23.28). The EVERYFN is used to apply an arbitrary function to each of the elements of a datum of the aggregate type, and check whether the result is non-NIL for each element. ELEMENT may be any type expression. For example, the type ''list of either strings or atoms'' can be defined as (LISTP OF (ONEOF STRINGP ATOM)). The type test for the new type will consist of applying the type test for ELEMENT to each element of the aggregate type using the EVERYFN property. The new type will be a subtype of its aggregate type.[12]

(TYPE (SATISFIES $FORM_1$ $FORM_N$)) [Decl Type Expression]

Specifies a type whose values are a subset of the values of an existing type. The type test for the new type will first check that the base type is satisfied, i.e. that the object is a member of TYPE, and then evaluate $FORM_1$ $FORM_N$. If each form returns a non-NIL value, the type is satisfied.

The value that is being tested may be referred to in $FORM_1$ $FORM_N$ by either (a) the variable name if the type expression appears in a binding context such as DLAMBDA or DPROG (b) the distinguished atom ELT for a SATISFIES clause on the elements of an aggregate type, or (c) the distinguished atom VALUE, when the type expression is used in a context where no name is available (e.g., a RETURNS declaration). For example, one might declare the program variable A to be a negative integer via (FIXP (SATISFIES (MINUSP A))), or declare the value of a DLAMBDA to be of type ((ONEOF FIXP FLOATP) (SATISFIES (GREATERP VALUE 25))). Note that more than one SATISFIES clauses may appear in a single type expression attached to different alternatives in a ONEOF type expression, or attached to both the elements and the overall structure of an aggregate. For example,

```
[LISTP OF [FIXP (SATISFIES (ILEQ ELT (CAR VALUE]
          (SATISFIES (ILESSP (LENGTH VALUE) 7]
```

specifies a list of less than 7 integers each of which is no greater than the first element of the list.

(SHARED TYPE) [Decl Type Expression]

Specifies a subtype of TYPE with default binding behavior, i.e. the binding function (see page 23.28), if any, will be suppressed.[13] For example, if the type FLOATP were redefined so that DLAMBDA and DPROG bindings of variables that were declared to be FLOATP copied their initial values (e.g., to allow SETNs to be free of side effects), then variables declared (SHARED FLOATP) would be initialized in the normal fashion, without copying their initial values.

[12] The built-in aggregate types are ARRAYP, LISTP, LST, and STRINGP (and their subtypes).

[13] As no predefined type has a binding function, this is of no concern until the user defines or redefines a type to have a binding function.

23.7.13 Named Types

Although type expressions can be used in any declaration context, it is often desirable to save the definition of a new type if it is to be used frequently, or if a more complex specification of its behavior is to be given than is convenient in an expression. The ability to define a named type is provided by the function DECLTYPE.

(DECLTYPE TYPENAME TYPE $PROP_1$ VAL_1 … $PROP_N$ VAL_N) [NLambda NoSpread Function]

Nlambda, nospread function. TYPENAME is a literal atom, TYPE is either the name of an existing type or a type expression, and $PROP_1$, VAL_1, …, $PROP_N$, VAL_N is a specification (in property list format) of other attributes of the type. DECLTYPE derives a type from TYPE, associates it with TYPENAME, and then defines any properties specified with the values given.

The following properties are interpreted by the Decl package.[14] Each of these properties can have as its value either a function name or a LAMBDA expression.

TESTFN — will be used by the Decl package to test whether a given value satisfies this type. The type is considered satisfied if FN applied to the item is non-NIL. For example, one might define the type INTEGER with TESTFN FIXP.[15]

EVERYFN — specifies a mapping function which can apply a functional argument to each ‘‘element’’ of an instance of this type, and which will return NIL unless the result of every such application was non-NIL. FN must be a function of two arguments: the aggregate and the function to be applied. For example, the EVERYFN for the built-in type LISTP is EVERY. As described on page 23.27, the Decl package uses the EVERYFN property of the aggregate type to construct a type test for aggregate type expressions. In fact, it is the presence of an EVERYFN property which allows a type to be used as an aggregate type.[16][17]

BINDFN — is used to compute from the initial value supplied for a DLAMBDA or DPROG variable of this type, the value to which the variable will actually be initialized. FN must be a function of one argument which will be applied to the initial value,[18] and which should produce another value which is to be used to make the binding. For example, a BINDFN could be used to bind variables of some type so that new

---

[14] Actually, any property can be attached to a type, and will be available for use by user functions via the function GETDECLTYPEPROP, described below.

[15] Typically, the TESTFN for a type is derived from its type expression, rather than specified explicitly. The ability to specify the TESTFN is provided for those cases where a predicate is available that is much more efficient than that which would be derived from the type expression. For example, the type SMALLP is defined to have the function SMALLP as its TESTFN, rather than (LAMBDA (DATUM) (AND (NUMBERP DATUM) (FIXP DATUM) (SMALLP DATUM))) as would be derived from the subtype structure.

[16] Note that a type's EVERYFN is *not* used in type tests for that type, but only in type tests for types defined by OF expressions which used this type as the aggregate type. For example, EVERY is not used in determining whether some value satisfies the type LISTP.

[17] The Decl package never applies the EVERYFN of a type to a value without first verifying that the value satisfies that type.

[18] For a DPROG binding, FN will be applied to no arguments if the initial value is lexically NIL.

bindings are copies of the initial value. Thus, if FLOATP were given the BINDFN FPLUS, any variable declared FLOATP would be initialized with a new floating box, rather than sharing with that of the original initial value.[19]

SETFN is used for performing a SETQ or SETQQ of variables of this type. FN is a function of two arguments, the name of the variable, and its new value. A SETFN is typically used to avoid the allocation of storage for intermediate results. Note that the SETFN is *not* the mechanism for the enforcement of type compatibility, which is checked *after* the assignment has taken place. Also note that not all functions which can change values are affected: in particular, SET and SETN are not.

23.7.13.1 Manipulating Named Types

DECLTYPE is a file package type (page 11.1). Thus all of the operations relating to file package types, e.g. GETDEF, PUTDEF, EDITDEF, DELDEF,[20] SHOWDEF, etc., can be performed on decltypes.

The file package command, DECLTYPES, is provided to dump named decltypes symbolically. They will be written as a series of DECLTYPE forms which will specify only those fields which differ from the corresponding field of their supertype(s). If the type depends on any unnamed types, those types will be dumped (as a compound type expression), continuing up the supertype chain until a named type is found. Care should be exercised to ensure that enough of the named type context is dumped to allow the type definition to remain meaningful.

The functions GETDECLTYPEPROP and SETDECLTYPEPROP, defined analogously to the property list functions for atoms, allow the manipulation of the properties of named types. Setting a property to NIL with SETDECLTYPEPROP removes it from the type.

23.7.14 Relations Between Types

The notion of equivalence of two types is not well defined. However, type equivalence is rarely of interest. What is of interest is type *inclusion*, i.e. whether one type is a supertype or subtype of another. The predicate COVERS can be used to determine whether the values of one type include those of another.

(COVERS HI LO) [Function]

is T if HI can be found on some (possibly empty) supertype chain of LO; else NIL. Thus, (COVERS 'FIXP (DECLOF 4))= T, even though the DECLTYPE of 4 is SMALLP, not FIXP. The extremal cases are the obvious identities: (COVERS 'ANY ANYTYPE) = (COVERS ANYTYPE 'NONE) = (COVERS X X) for any type X = T.

COVERS allows declaration based transformations of a form which depend on elements of the form being of a certain type to express their applicability conditions in terms of the weakest type to which they

---

[19] The BINDFN, if any, associated with a type may be suppressed in a declaration context by creating a subtype with the type expression operator SHARED, as described on page 23.27.

[20] Deleting a named type could possibly invalidate other type definitions that have the named type as a subtype or supertype. Consequently, the deleted type is simply unnamed and left in the type space as long as it is needed.

apply, without explicit concern for other types which may be subtypes of it. For example, if a particular transformation is to be applied whenever an element is of type NUMBERP, the program which applies that transformation does not have to check whether the element is of type SMALLP, LARGEP, FIXP, FLOATP, etc., but can simply ask whether NUMBERP COVERS the type of that element.

The elementary relations among the types, out of which arbitrary traversals of the type space can be constructed, are made available via:

(SUBTYPES TYPE) [Function]

Returns the list of types which are *immediate* subtypes of TYPE.

(SUPERTYPES TYPE) [Function]

Returns the list of types which are *immediate* supertypes of TYPE.

23.7.15 The Declaration Database

One of the primary uses of type declarations is to provide information that other systems can use to interpret or optimize code. For example, one might choose to write all arithmetic operations in terms of general functions like PLUS and TIMES and then use variable declarations to substitute more efficient, special purpose code at compile time based on the types of the operands. To this end, a data base of declarations is made available by the Decl package to support these operations.

(DECLOF FORM) [Function]

Returns the type of FORM in the current declaration context.[21] If FORM is an atom, DECLOF will look up that atom directly in its database of current declarations. Otherwise, DECLOF will look on the property list of (CAR FORM) for a DECLOF property, as described below. If there is no DECLOF property, DECLOF will check if (CAR FORM) is one of a large set of functions of known result type (e.g., the arithmetic functions). Failing that, if (CAR FORM) has a MACRO property, DECLOF will apply itself to the result of expanding (with EXPANDMACRO, page 5.19) the macro definition. Finally, if FORM is a Lisp program element that DECLOF "understands" (e.g., a COND, PROG, SELECTQ, etc.), DECLOF applies itself recursively to the part(s) of the contained form which will be returned as value.

DECLOF [Property Name]

Allows the specification of the type of the values returned by a particular function. The value of the DECLOF property can be either a type, i.e. a type name or a type expression, or a list of the form (FUNCTION FN), where FN is a function object. FN will be applied (by DECLOF) to the form whose CAR has this DECLOF property on its property list. The value of this function application will then be considered to be the type of the form.

---

[21]The "current declaration context" is defined by the environment at the time that DECLOF is called. Code reading systems, such as the compiler and the interpreter, keep track of the lexical scope within which they are currently operating, in particular, which declarations are currently in effect. Note that (currently) DECLOF does *not* have access to any global data base of declarations. For example, DECLOF does not have information available about the types of the arguments of, or the value returned by, a particular function, unless it is currently "inside" of that function. However, the DECLOF property (described below) can be used to inform DECLOF of the type of the value returned by a particular function.

As an example of how declarations can be used to automatically generate more efficient code, consider an arithmetic package. Declarations of numeric variables could be used to guide code generation to avoid the inefficiencies of Interlisp's handling of arithmetic values. Not only could the generic arithmetic functions be automatically specialized, as suggested above, but by redefining the BINDFN and the SETFN properties for the types FLOATP and LARGEP to re-use storage in the appropriate contexts (i.e., when the new value can be determined to be of the appropriate type), tremendous economies could be realized by not allocating storage to intermediate results which must later be reclaimed by the garbage collector. The Decl package has been used as the basis for several such code optimizing systems.

23.7.16 Declarations and Masterscope

The Decl package notifies MASTERSCOPE about type declarations and defines a new MASTERSCOPE relation, TYPE, which depends on declarations. Thus, the user can ask questions such as "WHO USES MUMBLE AS A TYPE?," "DOES FOO USE FIXP AS A TYPE?," and so on.

23.8 TRANSOR

Note: TRANSOR is a LispUsers package contained on the file TRANSOR.DCOM.

TRANSOR is a LISP-to-LISP translator intended to help the user who has a program coded in one dialect of LISP and wishes to carry it over to another. The user loads TRANSOR along with a file of transformations. These transformations describe the differences between the two LISPs, expressed in terms of Interlisp editor commands needed to convert the old to new, i.e. to edit forms written in the source dialect to make them suitable for the target dialect. TRANSOR then sweeps through the user's program and applies the edit transformations, producing an object file for the target system. In addition, TRANSOR produces a file of translation notes, which catalogs the major changes made in the code as well as the forms that require further attention by the user. Operationally, therefore, TRANSOR is a facility for conducting massive edits, and may be used for any purpose which that may suggest.

Since the edit transformations are fundamental to this process, let us begin with a definition and some examples. A transformation is a list of edit commands associated with a literal atom, usually a function name. TRANSOR conducts a sweep through the user's code, until it finds a form whose CAR is a literal atom which has a transformation. The sweep then pauses to let the editor execute the list of commands before going on. For example, suppose the order of arguments for the function TCONC must be reversed for the target system. The transformation for TCONC would then be: ((SW 2 3)). When the sweep encounters the form (TCONC X (FOO)), this transformation would be retrieved and executed, converting the expression to (TCONC (FOO) X). Then the sweep would locate the next form, in this case (FOO), and any transformations for FOO would be executed, etc.

Most instances of TCONC would be successfully translated by this transformation. However, if there were no second argument to TCONC, e.g. the form to be translated was (TCONC X), the command (SW 2 3) would cause an error, which TRANSOR would catch. The sweep would go on as before, but a note would appear in the translation listing stating that the transformation for this particular form failed to work. The user would then have to compare the form and the commands, to figure out what caused the problem. One might, however, anticipate this difficulty with a more sophisticated transformation: ((IF (## 3) ((SW 2 3)) ((-2 NIL)))), which tests for a third element and does (SW 2 3) or (-2 NIL) as appropriate. It should be obvious that the translation process is no more sophisticated than the

transformations used.

This documentation is divided into two main parts. The rst describes how to use TRANSOR assuming that the user already has a complete set of transformations. The second documents TRANSORSET, an interactive routine for building up such sets. TRANSORSET contains commands for writing and editing transformations, saving one's work on a le, testing transformations by translating sample forms, etc.

Two transformations les presently exist for translating programs into Interlisp. <LISP>SDS940.XFORMS is for old BBN LISP (SDS 940) programs, and <LISP>LISP16.XFORMS is for Stanford AI LISP 1.6 programs. A set for LISP 1.5 is planned.

23.8.1 Using TRANSOR

The rst and most exasperating problem in carrying a program from one implementation to another is simply to get it to read in. For example, SRI LISP uses / exactly as Interlisp uses %, i.e. as an escape character. The function PRESCAN exists to help with these problems: the user uses PRESCAN to perform an initial scan to dispose of these di culties, rather than attempting to TRANSOR the foreign source les directly.

PRESCAN copies a le, performing character-for-character substitutions. It is hand-coded and is much faster than either READC's or text-editors.

(PRESCAN FILE CHARLST) [Function]

> Makes a new version of FILE, performing substitutions according to CHARLST. Each element of CHARLST must be a dotted pair of two character codes, (OLD-CHAR-CODE . NEW-CHAR-CODE).

For example, SRI les are PRESCANed with CHARLST = ((37 . 47) (47 . 37)), which exchanges slash (47) and percent-sign (37).

The user should also make sure that the treatment of double quotes by the source and target systems is similar. In Interlisp, an unmatched double-quote (unless protected by the escape character) will cause the rest of the le to read in as a string.

Finally, the lack of a STOP at the end of a le is harmless, since TRANSOR will suppress END OF FILE errors and exit normally.

23.8.2 Translating

TRANSOR is the top-level function of the translator itself, and takes one argument, a le to be translated. The le is assumed to contain a sequence of forms, which are read in, translated, and output to a le called {FILE}.TRAN. The translation notes are meanwhile output to {FILE}.LSTRAN. Thus the usual sequence for bring a foreign le to Interlisp is as follows: PRESCAN the le; examine code and transformations, making changes to the transformations if needed; TRANSOR the le; and clean up remaining problems, guided by the notes. The user can now make a pretty le and proceed to exercise and check out his program. To export a le, it is usually best to TRANSOR it, then PRESCAN it, and perform clean-up on the foreign system where the le can be loaded.

(TRANSOR FILE) [Function]

Translates FILE. Prettyprints translation on {FILE}.TRAN; translation listing on {FILE}.LSTRAN.

(TRANSORFORM FORM) [Function]

FORM is a LISP form. Returns the (destructively) translated form. The translation listing is dumped to the primary output le.

(TRANSORFNS FNLST) [Function]

FNLST is a list of function names whose interpreted de nitions are destructively translated. Listing to primary output le.

TRANSORFORM and TRANSORFNS can be used to translate expressions that are already in core, whereas TRANSOR itself only works on les.

23.8.3 The Translation Notes

The translation notes are a catalog of changes made in the user's code, and of problems which require, or may require, further attention from the user. This catalog consists of two cross-indexed sections: an index of forms and an index of notes. The rst tabulates all the notes applicable to any form, whereas the second tabulates all the forms to which any one note applies. Forms appear in the index of forms in the order in which they were encountered, i.e. the order in which they appear on the source and output les. The index of notes shows the name of each note, the entry numbers where it was used, and its text, and is alphabetical by name. The following sample was made by translating a small test le written in SRI LISP.

```
LISTING FROM TRANSORING OF FILE TESTFILE.;7
     DONE ON 1-NOV-71 20:10:47

                                                                 INDEX OF FORMS
1. APPLY/EVAL at
  [DEFINEQ
      (FSET (LAMBDA &
            (PROG ...3...
                (SETQ Z (COND
                    ((ATOM (SETQ --))
                     (COND
                       ((ATOM (SETQ Y (NLSETQ "(EVAL W)")))
                        --)
                     --))
                    --))
            --  ]
2. APPLY/EVAL at
  [DEFINEQ
     (FSET (LAMBDA &
            (PROG ...3...
                 (SETQ Z (COND
                    ((ATOM (SETQ --))
                      (COND
                    ((ATOM (SETQ --))
```

```
                         "(EVAL (NCONS W))")
                        --))
                    --))
            --  ]
3. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
            (PROG ...3...
                 (COND
                   ...2...
                   ((NOT (EQUAL (SETQ X2 "(OPENR (MAKNUM & -))"
                               )
                             --))
                    --))
            --   ]
4. MACHINE-CODE at
  [DEFINEQ
     (LESS1 (LAMBDA &
            (PROG ...3...
                 (COND
                   ...2...
                   ((NOT (EQUAL & (SETQ Y2
                               "(OPENR (MAKNUM & --))")))
                    --))
              --   ]

                                                        INDEX OF NOTES
APPLY/EVAL at 1, 2.
     TRANSOR will translate the arguments of the APPLY or EVAL expression, but
the user must make sure that the run-time evaluation of the arguments returns
a BBN-compatible expression.
MACHINE-CODE at 3, 4.
     Expression dependent on machine-code.  User must recode.
```

The translation notes are generated by the transformations used, and therefore re ect the judgment of their author as to what should be included. Straightforward conversions are usually made without comment; for example, the DEFPROPs in this le were quietly changed to DEFINEQs. TRANSOR found four noteworthy forms on the le, and printed an entry for each in the index of forms, consisting of an entry number, the name of the note, and a printout showing the precise location of the form. The form appears in double-quotes and is the last thing printed, except for closing parentheses and dashes. An ampersand represents one non-atomic element not shown, and two or more elements not shown are represented as ...N..., where N is the number of elements. Note that the printouts describe expressions on the output le rather than the source le; in the example, the DEFPROPs of SRI LISP have been replaced with DEFINEQs.

23.8.4 Errors and Messages

TRANSOR records its progress through the source le by terminal printouts which identify each expression as it is read in. Progress within large expressions, such as a long DEFINEQ, is reported every three minutes

by a printout showing the location of the sweep.

If a transformation fails, TRANSOR prints a diagnostic to the teletype which identi es the faulty transformation, and resumes the sweep with the next form. The translation notes will identify the form which caused this failure, and the extent to which the form and its arguments were compromised by the error.

If the transformation for a common function fails repeatedly, the user can type control-H. When the system goes into a break, he can use TRANSORSET to repair the transformation, and even test it out (see TEST command, page 23.36). He may then continue the main translation with OK.

23.8.5 TRANSORSET

To use TRANSORSET, type (TRANSORSET) to Interlisp. TRANSORSET will respond with a + sign, its prompt character, and await input. The user is now in an executive loop which is like EVALQT with some extra context and capabilities intended to facilitate the writing of transformations. TRANSORSET will thus progress APPLY and EVAL input, and execute history commands just as EVALQT would. Edit commands, however, are interpreted as additions to the transformation on which the user is currently working. TRANSORSET always saves on a variable named CURRENTFN the name of the last function whose transformation was altered or examined by the user. CURRENTFN thus represents the function whose transformation is currently being worked on. Whenever edit commands are typed to the + sign, TRANSORSET will add them to the transformation for CURRENTFN. This is the basic mechanism for writing a transformation. In addition, TRANSORSET contains commands for printing out a transformation, editing a transformation, etc., which all assume that the command applies to CURRENTFN if no function is speci ed. The following example illustrates this process.

```
_TRANSORSET()
+FN TCONC                    [1]
TCONC
+(SW 2 3)                    [2]
+TEST (TCONC A B)            [3]
P
(TCONC B A)
+TEST (TCONC X)              [4]
TRANSLATION ERROR: FAULTY TRANSFORMATION
TRANSFORMATION: ((SW 2 3))    [5]
OBJECT FORM:        (TCONC X)

1. TRANSFORMATION ERROR AT    [6]
  "(TCONC X)"

(TCONC X)
+(IF (## 3) ((SW 2 3)) ((-2 NIL]    [7]
+SHOW
TCONC
  [(SW 2 3)
   (IF (## 3)                 [8]
       ((SW 2 3))
       ((-2 NIL]
TCONC
```

```
+ERASE                          [9]
TCONC
+REDO IF                        [10]
+SHOW
TCONC
  [(IF (## 3)
       ((SW 2 3))
       ((-2 NIL]
TCONC
+TDST
=TEST                           [11]
(TCONC NIL X)
+
```

In this example, the user begins by using the FN command to set CURRENTFN to TCONC *[1]*. He then adds to the (empty) transformation for TCONC a command to switch the order of the arguments *[2]* and tests the transformation *[3]*. His second TEST *[4]* fails, causing an error diagnostic *[5]* and a translation note *[6]*. He writes a better command *[7]* but forgets that the original SW command is still in the way *[8]*. He therefore deletes the entire transformation *[9]* and redoes the IF *[10]*. This time, the TEST works *[11]*.

23.8.6 TRANSORSET Commands

The following commands for manipulating transformations are all Prog. Asst. commands which treat the rest of their input line as arguments. All are undoable.

FN [Transorset Command]

Resets CURRENTFN to its argument, and returns the new value. In e ect FN says you are done with the old function (as least for the moment) and wish to work on another. If the new function already has a transformation, the message (OLD TRANSFORMATIONS) is printed, and any editcommands typed in will be added to the end of the existing commands. FN followed by a carriage return will return the value of CURRENTFN without changing it.

SHOW [Transorset Command]

Command to prettyprint a transformation. SHOW followed by a carriage return will show the transformation for CURRENTFN, and return CURRENTFN as its value. SHOW followed by one or more function names will show each one in turn, reset CURRENTFN to the last one, and return the new value of CURRENTFN.

EDIT [Transorset Command]

Command to edit a transformation. Similar to SHOW except that instead of prettyprinting the transformation, EDIT gives it to EDITE. The user can then work on the transformation until he leaves the editor with OK.

ERASE [Transorset Command]

Command to delete a transformation. Otherwise similar to SHOW.

TEST [Transorset Command]

Command for checking out transformations. TEST takes one argument, a form

for translation. The translation notes, if any, are printed to the teletype, but in an abbreviated format which omits the index of notes. The value returned is the translated form. TEST saves a copy of its argument on the free variable TESTFORM, and if no argument is given, it uses TESTFORM, i.e. tries the previous test again.

DUMP [Transorset Command]

Command to save your work on a le. DUMP takes one argument, a lename. The argument is saved on the variable DUMPFILE, so that if no argument is provided, a new version of the previous le will be created.

The DUMP command creates les by MAKEFILE. Normally FILEFNS will be unbound, but the user may set it himself; functions called from a transformation by the E command may be saved in this way. DUMP makes sure that the necessary command is included on the FILEVARS to save the user's transformations. The user may add anything else to his FILEVARS that he wishes. When a transformation le is loaded, all previous transformations are erased unless the variable MERGE is set to T.

EXIT [Transorset Command]

Exits TRANSORSET, returning NIL.

23.8.7 The REMARK Feature

The translation notes are generated by those transformations that are actually executed via an edit macro called REMARK. REMARK takes one argument, the name of a note. When the macro is executed, it saves the appropriate information for the translation notes, and adds one entry to the index of forms. The location that is printed in the index of forms is the editor's location when the REMARK macro is executed.

To write a transformation which makes a new note, one must therefore do two things: de ne the note, i.e. choose a new name and associate it with the desired text; and call the new note with the REMARK macro, i.e. insert the edit command (REMARK NAME) in some transformation. The NOTE command, described below, is used to de ne a new note. The call to the note may be added to a transformation like any other edit command. Once a note is de ned, it may be called from as many di erent transformations as desired.

The user can also specify a remark with a new text, without bothering to think of a name and perform a separate de ning operation, by calling REMARK with more than one argument, e.g. (REMARK TEXT-OF-REMARK). This is interpreted to mean that the arguments are the text. TRANSORSET notices all such expressions as they are typed in, and handles naming automatically; a new name is generated[22] and de ned with the text provided, and the expression itself is edited to be (REMARK GENERA TED-NAME). The following example illustrates the use of REMARK.

```
_TRANSORSET()
+NOTE GREATERP/LESSP (BBN'S GREATERP AND LESSP ONLY TAKE TWO ARGUMENTS, WHEREAS
SRI'S FUNCTIONS TAKE AN INDEFINITE NUMBER. AT THE PLACES NOTED HERE, THE SRI
CODE USED MORE THAN TWO ARGUMENTS, AND THE USER MUST RECODE.]    [1]
```

[22]The name generated is the value of CURRENTFN su xed with a colon, or with a number and a colon.

```
GREATERP/LESSP
+FN GREATERP
GREATERP
+(IF (IGREATERP (LENGTH (##))3) NIL ((REMARK GREATERP/LESSP]    [2]
+FN LESSP
LESSP
+REDO IF    [3]
+SHOW
LESSP
  [(IF (IGREATERP (LENGTH (##))
                  3)
       NIL
       ((REMARK GREATERP/LESSP]
LESSP
+FN ASCII
(OLD TRANSFORMATIONS)
ASCII
+(REMARK ALTHOUGH THE SRI FUNCTION ASCII IS IDENTICAL TO THE BBN FUNCTION CHARACTER,
THE USER MUST MAKE SURE THAT THE CHARACTER BEING CREATED SERVES THE SAME PURPOSE
ON BOTH SYSTEMS, SINCE THE CONTROL CHARACTERS ARE ALL ASSIGNED DIFFRENTLY.]    [4]

+SHOW    [5]
ASCII
  ((1 CHARACTER)
   (REMARK ASCII:))
ASCII
+NOTE ASCII:    [6]
EDIT
*NTH -2
*P
... ASSIGNED DIFFRENTLY.)
*(2 DIFFERENTLY.)
OK
ASCII:
+
```

In this example, the user defines a note named GREATERP/LESSP by using the NOTE command *[1]*, and writes transformations which call this note whenever the sweep encounters a GREATERP or LESSP with more than two arguments *[2]* and *[3]*. Next, the implicit naming feature is used *[4]* to add a REMARK command to the transformation for ASCII, which has already been partly written. The user realizes he mistyped part of the text, so he uses the SHOW command to find the name chosen for the note *[5]*. Then he uses the NOTE command on this name, ASCII:, to edit the note *[6]*.

NOTE [Transorset Command]

First argument is note name and must be a literal atom. If already defined, NOTE edits the old text; otherwise it defines the name, reading the text either from the rest of the input line or from the next line. The text may be given as a line or as a list. Value is name of note.

The text is actually stored.[23] as a comment, i.e. a * and %% are added in front when the note is rst de ned. The text will therefore be lower-cased the rst time the user DUMPs (see page 6.52).

DELNOTE [Transorset Command]

Deletes a note completely (although any calls to it remain in the transformations).

23.8.8 Controlling the Sweep

TRANSOR's sweep searches in print-order until it nds a form for which a transformation exists. The location is marked, and the transformation is executed. The sweep then takes over again, beginning from the marked location, no matter where the last command of the transformation left the editor. User transformations can therefore move around freely to examine the context, without worrying about confusing the translator. However, there are many cases where the user wants his transformation to guide the sweep, usually in order to direct the processing of special forms and FEXPRs. For example, the transformation for QUOTE has only one objective: to tell the sweep to skip over the argument to QUOTE, which is (presumably) not a LISP form. NLAM is an edit macro that permits this.

NLAM [Transorset Command]

An atomic edit macro which sets a ag which causes the sweep to skip the arguments of the current form when the sweep resumes.

Special forms such as COND, PROG, SELECTQ, etc., present a more di cult problem. For example, (COND (A B)) is processed just like (FOO (A B)): i.e. after the transformation for COND nishes, the sweep will locate the ``next form,'' (A B), retrieve the transformation for the function A, if any, and execute it. Therefore, special forms must have transformations that preempt the sweep and direct the translation themselves. The following two atomic edit macros permit such transformations to process their forms, translating or skipping over arbitrary subexpressions as desired.

DOTHIS [Transorset Command]

Translates the editor's current expression, treating it as a single form.

DOTHESE [Transorset Command]

Translates the editor's current expression, treating it as a list of forms.

For example, a transformation for SETQ might be (3 DOTHIS).[24] This translates the second argument to a SETQ without translating the rst. For COND, one might write (1 (LPQ NX DOTHESE)), which locates each clause of the COND in turn, and translates it as a list of forms, instead of as a single form.

The user who is starting a completely new set of transformations must begin by writing transformations for all the special forms. To assist him in this and prevent oversights, the le <LISP>SPECIAL.XFORMS contains a set of transformations for LISP special forms, as well as some other transformations which should also be included. The user will probably have to revise these transformations substantially, since they merely perform sweep control for Interlisp, i.e. they make no changes in the object code. They are provided chie y as a checklist and tutorial device, since these transformations are both the rst to be written and the most di cult, especially for users new to the Interlisp editor.

---

[23] On the global list USERNOTES.

[24] Recall that a transformation is a list of edit commands. In this case, there are two commands, 3 and DOTHIS.

When the sweep mechanism encounters a form which is not a list, or a form CAR of which is not an atom, it retrieves one of the following special transformations.

NLISTPCOMS [Variable]

Global value is used as a transformation for any form which is not a list.

For example, if the user wished to make sure that all strings were quoted, he might set NLISTPCOMS to ((IF (STRINGP (##)) ((ORR ((_ QUOTE))((MBD QUOTE)))) NIL)).

LAMBDACOMS [Variable]

Global value is used as a transformation for any form, CAR of which is not an atom.

These variables are initialized by <LISP>SPECIAL.XFORMS and are saved by the DUMP command. NLISTPCOMS is initially NIL, making it a NO-OP. LAMBDACOMS is initialized to check rst for open LAMBDA expressions, processing them without translation notes unless the expression is badly formed. Any other forms with a non-atomic CAR are simply treated as lists of forms and are always mentioned in the translation notes. The user can change or add to this algorithm simply by editing or resetting LAMBDACOMS.

23.9 WHEREIS PACKAGE

Note: The WHEREIS is a LispUsers package that is contained on the le WHEREIS.COM*. WHEREIS requires the hash le package (page 23.41). Loading* WHEREIS.COM *will also load* HASH.COM*, if it has not already been loaded.*

This package extends the function WHEREIS (page 11.10) such that, when asked about a given name as a function, WHEREIS will consult not only the commands of les that have been noticed by the le package (page 11.1) but also a hash le database (page 23.41) that associates function names with lenames.

(WHEREIS NAME TYPE FILES FN) [Function]

Behaves exactly like the de nition on page 11.10 unless TYPE = FNS (or NIL) and FILES= T. In this case, WHEREIS will consult, in addition to the les on FILELST, the hash le that is the value of WHEREIS.HASH (initially <LISPUSER>WHEREIS.HASH).

Note: Most system functions call WHEREIS with FILES= T, so loading this package automatically makes the information contained in the WHEREIS database available throughout the system.

Information may be added to a WHEREIS hash le by explicitly calling the following function:

(WHEREISNOTICE FILEGROUP NEWFLG) [Function]

Inserts the information about all of the functions on the les in FILEGROUP into the WHEREIS data base contained on (the value of) WHEREIS.HASH. FILEGROUP is given as a legroup argument to DIRECTORY (page 14.6), so &, $, etc. may be used. If NEWFLG = T, a new version of WHEREIS.HASH will be created containing the database for the functions speci ed in FILEGROUP.

23.10 HASH FILES

Note: The hash le facility is a LispUsers package that is contained on the le HASH.COM. *It currently only works in Interlisp-10.*

The hash le facility permits information associated with string or atom ‘‘keys’’ to be stored on and retrieved from les. The information (or ‘‘values’’) associated with the keys in a le may be numbers, strings, or arbitary Interlisp expressions. The associations are maintained by a hashing scheme that minimizes the number of page-maps it takes to access a value from its key.

A hash le may contain information other than key-value associations. The user may print on the le using ordinary printing functions (e.g. PRIN1, PRINTDEF), and he may also store non-character information (e.g. binary data) formatted to suit his particular applications. This information is stored in regions of the le distinct from the hash index. The hash index can be used to locate non-hash information, if the necessary le addresses are stored as hash values.

A hash le is created by the function CREATEHASHFILE:

(CREATEHASHFILE FILE VALUETYPE ITEMLENGTH qENTRIES) [Function]

A new version of FILE is opened and initialized as a hash le. VALUETYPE is an atom interpreted as follows:

NUMBER The values are 24-bit unsigned integers.

STRING The values are strings with less than 128 characters.

EXPR The values are arbitrary Interlisp expressions. The values are stored by printing them in the le with readtable HASHFILERDTBL, initially ORIG.

SMALLEXPR
The values are arbitrary Interlisp expressions such that (NCHARS VALUE T HASHFILERDTBL) is less than 128. Storing and retrieving is more e cient than if VALUETYPE = EXPR .

SYMBOLTABLE
The values are 24-bit unsigned integers, as when VALUETYPE = NUMBER, except that the numbers are treated as the addresses of ‘‘symbols’’ located on non-hash pages in the le. See the discussion of symbol-tables below.

The other arguments to CREATEHASHFILE are optional. ITEMLENGTH is the user’s estimate of the average number of characters in the entries he expects to store in the hash le (= the average key length plus the average number of characters in the values for VALUETYPE STRING or SMALLEXPR). qENTRIES is an estimate of the the total number of key-value associations he is likely to store. These two arguments determine how many pages in the le will be initially allocated as hash-pages; accurate estimates can reduce the number of times that the le must be rehashed as information is stored in it. If these arguments are not given, reasonable defaults are supplied.

After being initialized, FILE is left open and CREATEHASHFILE returns as its value

a ‘‘hash le datum,’’ a handle on the hash le that may be used as an argument for most of the functions described below.

(OPENHASHFILE FILE ACCESS) [Function]

Re-opens the previously existing hash le FILE. ACCESS may be INPUT (or NIL), in which case FILE is opened for reading only, or BOTH, in which case FILE is open for both input and output. Causes an error NOT A HASHFILE, if FILE is not recognized as a hash le.

If ACCESS is BOTH and FILE is a hash le open for reading only, OPENHASHFILE attempts to close it and re-open it for writing. Otherwise, if FILE designates an already open hash le, OPENHASHFILE is a no-op.

OPENHASHFILE returns a hash le datum.

(HASHFILEP X) [Function]

Returns X if X is a hash le datum (i.e., a value returned by CREATEHASHFILE or OPENHASHFILE). If X is NIL, returns SYSHASHFILE if it is a hash le datum. If X is the name of an open hash le, returns the corresponding hash le datum. Otherwise, returns NIL.

The following functions require an open hash le as an argument, i.e. an object for which HASHFILEP is non-NIL.

(PUTHASHFILE KEY VALUE HASHFILE) [Function]

Puts VALUE in HASHFILE, indexed under KEY. If VALUE is NIL, any previous entry for KEY is deleted.

(GETHASHFILE KEY HASHFILE) [Function]

Returns the value corresponding to KEY in HASHFILE. For les where VALUETYPE is STRING, NUMBER, or SYMBOLTABLE, the value returned by GETHASHFILE is temporary in that any subsequent calls to a hash le or page mapping function may smash it. CONCAT or MKATOM must be applied if the value is a string, or IPLUS if it is a number, in order to make the value permanent.

(HASHFILEPROP HASHFILE PROP) [Function]

Returns the value of the PROP property of HASHFILE. The recognized PROPs and the values returned are:

VALUETYPE
: One of NUMBER, STRING, EXPR, SMALLEXPR, or SYMBOLTABLE.

NAME
: The full name of the le.

ACCESS
: BOTH if le is open for writing, INPUT if it is read-only.

(HASHFILENAME HASHFILE) [Function]

Same as (HASHFILEPROP HASHFILE 'NAME).

(CLOSEHASHFILE HASHFILE) [Function]

Same as (CLOSEF (HASHFILEPROP HASHFILE 'NAME)).

The function HASHSTATUS can be used as a STATUS function for WHENCLOSE (page 6.11) to restore

the state of a hashfile when a SYSOUT is resumed. If HASHSTATUS is used, the PERMSTATUS package (page 23.17) must also be loaded.

(MAPHASHFILE HASHFILE MAPFN) [Function]

For each entry in HASHFILE, performs (MAPFN KEY (GETHASHFILE KEY HASHFILE)). If MAPFN is a function of only one argument, performs (MAPFN KEY) thereby avoiding the call to GETHASHFILE needed to obtain the value. KEY is temporary, as for GETHASHFILE. VALUE is also temporary, for STRING, NUMBER, and SYMBOLTABLE files.

(REHASHFILE HASHFILE) [Function]

After many insertions and deletions much of the space in a hashfile may be unusable. REHASHFILE reclaims that space by rehashing all the keys. The information on non-hash pages in the file is not altered or moved, except that the print name pointers in a SYMBOLTABLE file are updated (see below).

(COPYHASHFILE HASHFILE NEWNAME FN VTYPE) [Function]

Calls CREATEHASHFILE to open NEWNAME as a hashfile, with VALUETYPE, ITEMLENGTH and q ENTRIES determined by examining the open hashfile HASHFILE. Then maps through all the keys in HASHFILE, doing the equivalent of:

```
(PUTHASHFILE
   KEY
   (GETHASHFILE KEY HASHFILE)
   NEWHASHFILE)
```

for each key KEY. In essence, COPYHASHFILE copies the hash portion of HASHFILE to NEWNAME.

If FN is given, then it is applied to the successive values of HASHFILE, the old HASHFILE, and the new hashfile, and the value returned is used as the value in the new file. In effect,

```
(PUTHASHFILE
   KEY
   (FN (GETHASHFILE KEY HASHFILE)
        HASHFILE
        NEWHASHFILE)
   NEWHASHFILE)
```

is evaluated for each key. Thus, the user can intervene as each key is processed in order to copy information associated with the key that resides on non-hash pages.

For example, an EXPR file could be implemented by printing the full expressions in a NUMBER file's printing region (see below) and storing their byte-positions as hash values. Instead of reading an expression into internal data structures before writing it out to the new file, a FN could be given that transferred the expression to the new file more efficiently, via COPYBYTES. The function would return the byte-position on the new file where the expression ended up. (Actually, this is the way EXPR files are copied if FN is not specified.)

If FN is given, then VTYPE, if specified, is a temporary valuetype (NUMBER,

STRING, etc.) to be used during copying. This permits the user to force the valuetype of both les to one more suited for FN, e.g. SMALLEXPR to STRING or EXPR to NUMBER, as in the example. VTYPE does not a ect the permanent valuetype of either le.

(HASHFILESPLST HASHFILE) [Function]

Returns a ‘‘generator’’ for the keys in HASHFILE that is acceptable as an argument to FIXSPELL (page 15.18). Thus, (FIXSPELL BADWORD 70 (HASHFILESPLST HASHFILE)) will spelling correct a word using the keys in HASHFILE.

(LOOKUPHASHFILE KEY VALUE HASHFILE CALLTYPE) [Function]

A generalized entry for inserting and retrieving values; provides certain options not available with GETHASHFILE or PUTHASHFILE. LOOKUPHASHFILE looks up KEY in HASHFILE. CALLTYPE is an atom or a list of atoms. These keywords are interpreted as follows:

RETRIEVE
: If KEY is found, then if CALLTYPE is or contains RETRIEVE, the old value is returned from LOOKUPHASHFILE; otherwise returns T.

DELETE
: If CALLTYPE is or contains DELETE, the value associated with KEY is deleted from the le.

REPLACE
: If CALLTYPE is or contains REPLACE, the old value is replaced with VALUE.

INSERT
: If CALLTYPE is or contains INSERT, LOOKUPHASHFILE inserts value as the value associated with KEY.

If KEY is not found, LOOKUPHASHFILE returns NIL.

Examples:

To either return an old value or insert a new value in the le if one does not already exist, perform (LOOKUPHASHFILE KEY NEWVALUE HASHFILE '(INSERT RETRIEVE)). The value returned will be NIL if NEWVALUE was inserted, or the old value if KEY was found.

To merely check whether KEY exists in the le without actually retrieving its value (which may be expensive for the more general valuetypes), perform (LOOKUPHASHFILE KEY NIL HASHFILE NIL).

The function PUTHASHFILE is de ned as:

```
(LAMBDA (KEY VALUE HASHFILE)
    (if VALUE=NIL
      then (LOOKUPHASHFILE KEY NIL HASHFILE 'DELETE)
      else (LOOKUPHASHFILE KEY VALUE HASHFILE '(INSERT REPLACE))
           VALUE))
```

And GETHASHFILE is de ned as:

```
(LAMBDA (KEY HASHFILE)
    (LOOKUPHASHFILE KEY NIL HASHFILE 'RETRIEVE))
```

23.10.1 Unstructured Pages and Symbol Tables

The non-hash information in a hash-le may be formatted as printed character strings or binary data. Printed information resides in a le's ‘‘printing region’’, while binary data is stored on ‘‘unstructured pages’’.

Unstructured pages in a le are allocated and deallocated by the hash package so that they do not encroach on hash or printing pages. Other than that, the user has complete freedom to map them in for arbitrary reading and writing. The primitive operations are:

(GETPAGE HASHFILE N) [Function]

Returns the page number of a free page in HASHFILE. If N is given, then the user is guaranteed that the page returned is the rst of N contiguous pages all of which are free.

(DELPAGE PAGEq HASHFILE) [Function]

Removes page PAGEq from HASHFILE. PAGEq should be the number of an unstructured page, either a value of GETPAGE or within the block of free pages guaranteed by GETPAGE. The contents of the page in the le are lost, and the page itself becomes available for re-allocation either by GETPAGE or internally as a hash page. If PAGEq happens to be the number of a hash page, the hashing information will be destroyed.

Unstructured pages are available on hash les so that the user can link hash keys to data in special formats. For example, the user might associate lists of properties with a key by writing the properties on an unstructured page, and then storing the le address of the properties as the value of the key in a NUMBER le.

A SYMBOLTABLE hash le provides an additional feature that makes it possible to implement arbitrary le-resident symbol processing systems. The user may store the data to be associated with a key on unstructured pages, and he can then link the le address to the key via PUTHASHFILE, as described above. The dierence between a NUMBER and SYMBOLTABLE le is that for a SYMBOLTABLE, the hash package also stores the reverse link from the le address to the key. This makes it possible to obtain a ‘‘print-name’’ for an address on an unstructured page, via the function GETPNAME:

(GETPNAME FILEADR HASHFILE) [Function]

Returns a temporary string containing the characters of the key whose hash value is the 24-bit unsigned FILEADR. Causes an error if HASHFILE is not a SYMBOLTABLE le.

The hash package automatically updates the print-name information for the le address if the key is relocated by rehashing, and it destroys the back-link if the value for the key is deleted. A SYMBOLTABLE le imposes one restriction on the way unstructured pages are treated: If a le address is stored as a hash-value for some key, then the right-most 24 bits of the word at that location in the le are reserved for the use of the hash mechanism.[25] The user must not write into it.

With these primitives, a list-processing system with a 24-bit non-resident address space is easy to build. The user is responsible for allocating ‘‘atoms’’ on unstructured pages, and updating the ‘‘atom hash table’’

[25]The left-most 12 bits are available and can be used for a number of applications, e.g. to store type-bits.

with PUTHASHFILE. The second (and subsequent) words after an atom address may be used to store the atom's ''property list'', containing other atom addresses, or other addresses interpreted as pointers to ''cons'' cells. These can also be allocated on unstructured pages. It is a simple matter to implement the equivalent of CAR, CDR, RPLACA, and RPLACD.

23.10.2 The Printing Region

Hash files are organized so that it is always permissible to print at the end of the file with ordinary Interlisp output functions. That is, the file is arranged so that the hash and unstructured pages are always located before the end-of-file for sequential reading and writing. This is accomplished by creating the file with the end-of-file some number of free pages past the last hash or unstructured page. When all free pages below the end-of-file have been used, the end-of-file is moved so that there are again a reservoir of free pages before it.

Thus, the printing region may shift as a result of calls to GETPAGE or PUTHASHFILE, and the user cannot rely on the output from two different printing operations being located at adjacent positions in the file. The expressions printed cannot be retrieved by successive calls to standard reading functions. Instead, the user should record the byte position of each printed expression as a hash value or on an unstructured page so that he may use SETFILEPTR to position the file properly. If he does change the file's byte-pointer, he must be sure to reset it to the end-of-file (e.g. (SETFILEPTR FILE -1)) before more printing is done.

23.11 EDITA

Note: EDITA is a LispUsers package contained on the file EDITA.COM. *That portion of EDITA relating to compiled code may not be available in implementations of Interlisp other than Interlisp-10. EDITA also has a* FILEDEF *property so that the user can simply call* EDITA *and the file will be automatically loaded.*

EDITA is an editor for arrays. However, its most frequent application is in editing compiled functions (which are also arrays in Interlisp-10), and a great deal of effort in implementing EDITA, and most of its special features, are in this area. For example, EDITA knows the format and conventions of Interlisp-10 compiled code, and so, in addition to decoding instructions a la DDT (one of the oldest debugging systems still around), EDITA can fill in the appropriate COREVALS, symbolic names for index registers, references to literals, linked function calls, etc. The following output shows a sequence of instructions in a compiled function first as they would be printed by DDT, and second by EDITA.

```
466716/  PUSH 16,LISP&KNIL           3/   PUSH PP,KNIL
466717/  PUSH 16,LISP&KNIL           4/   PUSH PP,KNIL
466720/  HRRZ 1,-12(16)              5/   HRRZ 1,-10(PP)
466721/  CAME 1,LISP&KNIL            6/   CAME 1,KNIL
466722/  JRST 466724                 7/   JRST 9[26]
466723/  HRRZ 1,@467575              8/   HRRZ 1,@'BRKFILE
466724/  PUSH 16,1                   9/   PUSH PP,1
466725/  LISP&IOFIL,,467576          10/  PBIND 'BRKZ
466726/  -3,,-3                      11/  -524291
466727/  HRRZ 1,-14(16)              12/  HRRZ 1,-12(PP)
466730/  CAMN 1,467601               13/  CAMN 1,'OK
466731/  JRST 466734                 14/  JRST 17
466732/  CAME 1,467602               15/  CAME 1,'STOP
466733/  JRST 466740                 16/  JRST 21
466734/  PUSH 16,467603              17/  PUSH PP,'BREAK1
466735/  PUSH 16,467604              18/  PUSH PP,'(ERROR!)
466736/  LISP&FILEN,,467605          19/  CCALL 2,'RETEVAL
466737/  JRST 467561                 20/  JRST 422
466740/  CAME 1,467606               21/  CAME 1,'GO
466741/  JRST 466754                 22/  JRST 33
466742/  HRRZ 1,@-12(16)             23/  HRRZ 1,@-10(PP)
466743/  PUSH 16,1                   24/  PUSH PP,1
```

Therefore, rather than presenting EDITA as an array editor with some extensions for editing compiled code, we prefer to consider it as a facility for editing compiled code, and point out that it can also be used for editing arbitrary arrays.

23.11.1 Overview

EDITA is envoked by calling the function EDITA:

(EDITA FN COMS) [Function]

Envokes EDITA to edit the function FN. To the user, EDITA looks very much like DDT with Interlisp-10 extensions. If COMS is given, it should be a list of commands for EDITA. These are then executed exactly as though they had been typed. EDITA can be exited with the command OK.

Individual registers or cells in the function may be examined by typing their address followed by a slash, e.g.

[26] Note that EDITA prints the addresses of cells contained *in* the function relative to the origin of the function.

```
6/  HRRZ 1,-10(PP)
```

The slash is really a command to EDITA to open the indicated register.[27] Only one register at a time can be open, and only open registers can be changed. To change the contents of a register, the user first opens it, types the new contents, and then closes the register with a carriage-return,[28] e.g.

```
7/  CAME 1,'^    CAMN 1,'^cr
```

If the user closes a register without specifying the new contents, the contents are left unchanged. Similarly, if an error occurs or the user types control-E, the open register, if any, is closed without being changed.

23.11.2 Input Protocol

EDITA processes all inputs not recognized as commands in the same way. If the input is the name of an instruction (i.e., an atom with a numeric OPD property), the corresponding number is added to the input value being assembled,[29] and a flag is set which specifies that the input context is that of an instruction.

The general form of a machine instruction is (OPCODE AC , @ ADDRESS (INDEX)) as described on page 22.15. Therefore, in instruction context, EDITA evaluates all atoms (if the atom has a COREVAL property, the value of the COREVAL is used), and then if the atom corresponds to an AC,[30] shifts it left 23 bits and adds it to the input value, otherwise adds it directly to the input value, but performs the arithmetic in the low 18 bits.[31] Lists are interpreted as specifying index registers, and the value of CAR of the list (again COREVALs are permitted) is shifted left 18 bits. Examples:

```
PUSH PP, KNIL
HRRZ 1,-10(PP)
CAME 1, 'GO
JRST 33 ORG
```

EDITA cannot in general know whether an address field in an instruction that is typed in is relative or absolute. Therefore, the user must add ORG, the origin of the function, to the address field himself. Note that EDITA would *print* this instruction, JRST 53 ORG, as JRST 53.

The user can also specify the address of a literal via the ' command, see page 23.50. For example, if the literal " UNBROKEN" is in cell 85672, HRRZ 1,'" UNBROKEN" is equivalent to HRRZ 1, 85672.

---

[27]EDITA also converts absolute addresses of cells within the function to relative address on input. Thus, if the definition of FOO begins at 85660, typing 6/ is *exactly* the same as typing 85666/.

[28]Since carriage-return has a special meaning, EDITA indicates the balancing of parentheses by typing a space.

[29]The input value is initially 0.

[30]i.e., if a ‘‘,’’ has not been seen, *and* the value of the atom is less than 16, *and* the low 18 bits of the input value are all zero.

[31]If the absolute value of the atom is greater than 1000000Q, full word arithmetic is used. For example, the indirect bit is handled by simply binding @ to 20000000Q.

When the input context is *not* that of an instruction, i.e., no OPD has been seen, all inputs are evaluated (the value of an atom with a COREVAL property is the COREVAL.) Then numeric values are simply added to the previous input value; non-numeric values *become* the input value.[32]

The only exception to the entire procedure occurs when a register is open that is in the pointer region of the function, i.e., literal table. In this case, atomic inputs are *not* evaluated. For example, the user can change the literal FOO to FIE by simply opening that register and then typing FIE followed by carriage-return, e.g.

```
'FOO/   FOO     FIE cr
```

Note that this is equivalent to

```
'FOO/   FOO     (QUOTE FIE) cr
```

23.11.3 EDITA Commands and Variables

cr (carriage-return) — If a register is open and an input was typed, store the input in the register and close it.[33]

If a register is open and nothing was typed, close the register without changing it.

If a register is not open and input was typed, type its value.

ORG — Has the value of the address of the first instruction in the function. i.e., LOC of GETD of the function.

/ — Opens the register specified by the low 18 bits of the quantity to the left of the /, and types its contents. If nothing has been typed, it uses the last thing typed by EDITA, e.g.,

```
35/  JRST 53    /    CAME 1,'RETURN    /    RETURN
```

If a register was open, / closes it without changing its contents.

After a / command, EDITA returns to that state of no input having been typed.

tab (control-I) — Same as carriage-return, followed by the address of the quantity to the left of the tab, e.g.,

```
35/   JRST 53 <tab>
53/   CAME 1,'RETURN
```

Note that if a register was open and input was typed, tab will change the open register before closing it, e.g.,

[32] Presumably there is only one input in this case.

[33] If the register is in the unboxed region of the function, the unboxed value is stored in the register.

```
35/    JRST 53    JRST 54 TAB
54/    JRST 70 cr
35/    JRST 54
```

`.` (period) — Has the value of the address of the current (last) register examined.

line-feed — Same as carriage-return followed by `(ADD1 .)/` i.e. closes any open register and opens the *next* register.

`^` — Same as carriage-return followed by `(SUB1 .)/`

`$Q` (<esc>Q) — Has as its value the last quantity typed by EDITA e.g.

```
35/   JRST 53    $Q 1 cr
./    JRST 54
```

`LITS` — Has as value the (relative) address of the rst literal.

`BOXED` — Same as `LITS`

`$` (dollar) — Has as value the relative address of the last literal in the function.

`=` — Sets `RADIX` (page 6.19) to -8 and types the quantity to the left of the = sign, i.e., if anything has been typed, types the input value, otherwise, types `$Q`, e.g.

```
35/    JRST 54 =254000241541Q
JRST 54=254000000066Q
```

Following =, `RADIX` is restored and EDITA returns to the no input state.

`OK` — Exits EDITA.

`?` — Returns to ''no input'' state. `?` is a ''weak'' control-E, i.e., it negates any input typed, but does not close any registers.

ADDRESS$_1$, ADDRESS$_2$/ — Prints the contents of registers ADDRESS$_1$ through ADDRESS$_2$. `.` is set to ADDRESS$_2$ after the completion.

Output goes to `FILE`, initially set to `T`. The user can also set `FILE` (while in EDITA) to the name of a disc le to redirect the output. (The user is responsible for opening and closing `FILE`.) Note that `FILE` only a ects output for the ADDRESS$_1$, ADDRESS$_2$/ command.

`'X` — Corresponds to the `'` in LAP. The next expression is read, and if it is a small number, the appropriate o set is added to it. Otherwise, the literal table is searched for `X`, and the value of `'X` is the (absolute) address of that cell. An error is generated if the literal is not found, i.e., `'` cannot be used to *create* literals.

`:ATOM` — De nes ATOM to an address: (1) the value of `$Q` if a register is open, (2) the input if any input was typed, otherwise (3) the value of ''.'' (Only the low 18 bits are used and converted to a relative address whenever possible). For example:

```
35/    JRST 54     :FOO cr
```

```
:FIE cr
FIE/  JRST FOO .=35
```

EDITA keeps its symbol tables on two free variables, USERSYMS and SYMLST. USERSYMS is a list of elements of the form (NAME . VALUE) and is used for *encoding* input, i.e., all variables on USERSYMS are bound to their corresponding values during evaluation of any expression inside EDITA. SYMLST is a list of elements of the form (VALUE . NAME) and is used for *decoding* addresses. USERSYMS is initially NIL, while SYMLST is set to a list of all the COREVALS. Since the : command adds the appropriate information to both these two lists, new de nitions will remain in e ect even if the user exits from EDITA and then reenters it later.

Note that the user can e ectively de ne symbols without using the : command by appropriately binding USERSYMS and/or SYMLST before calling EDITA. Also, he can thus use di erent symbol tables for di erent applications.

$W (<esc>W) Search command.

Searching consists of comparing the object of the search with the contents of each register, and printing those that match, e.g.,

```
HRRZ @ $W cr
8/   HRRZ 1,@'BRKFILE
23/  HRRZ 1,@-10(PP)
28/ HRRZ 1,@-12(PP)
```

The $W command can be used to search either the unboxed portion of a function, i.e., instructions, or the pointer region, i.e., literals, depending on whether or not the object of the search is a number. If any input was typed before the $W, it will be the object of the search, otherwise the next expression is read and used as the object.[34] The user can specify a starting point for the search by typing an address followed by a ‘‘,’’ before calling $W, e.g., 1, JRST $W. If no starting point is speci ed, the search will begin at 0 if the object is a number, otherwise at LITS, the address of the rst literal.[35] After the search is completed, ‘‘.’’ is set to the address of the last register that matched.

If the search is operating in the unboxed portion of the function, only those elds (i.e., INSTRUCTION , AC, INDIRECT , INDEX , and ADDRESS) of the object that contain one bits are compared.[36] For example, HRRZ @ $W will nd all instances of HRRZ indirect, regardless of AC, INDEX , and ADDRESS elds. Similarly, 'PRINT $W will nd all instructions that reference the literal PRINT.[37]

---

[34]Note that inputs typed before the $W will have been processed according to the input protocol, i.e., evaluated; inputs typed after the $W will not. Therefore, the latter form is usually used to specify searching the literals, e.g., $W FOO is equivalent to (QUOTE FOO) $W.

[35]Thus the only way the user can search the pointer region for a number is to specify the starting point via ‘‘,’’.

[36]Alternately, the user can specify his own mask by setting the variable MASK (while in EDITA), to the appropriate bit pattern.

[37]The user may need to establish instruction context for input without giving a speci c instruction. For example, suppose the user wants to nd all instructions with AC=1 and INDEX = PP. In this case, the user can give & as a pseudo- instruction, e.g., type & 1, (PP).

If the search is operating in the pointer region, a ‘‘match’’ is as defined in the editor. For example, `$W (&)` will find all registers that contain a list consisting of a single expression.

`$C` (<esc>C) Like `$W` except only prints the first match, then prints the number of matches when the search finishes.

23.11.4 Editing Arrays

`EDITA` is called to edit a function by giving it the name of the function. `EDITA` can also be called to edit an array by giving it the array as its first argument, [38] in which case the following differences are to be noted:

1. decoding - The contents of registers in the unboxed region are boxed and printed as numbers, i.e., they are never interpreted as instructions, as when editing a function.

2. addressing convention - Whereas 0 corresponds to the first instruction of a function, the first element of an array by convention is element number 1.

3. input protocols - If a register is open, lists are evaluated, atoms are not evaluated (except for `$Q` which is always evaluated). If no register is open, all inputs are evaluated, and if the value is a number, it is added to the ‘‘input value’’.

4. left half - If the left half of an element in the pointer region of an array is not all 0’s or `NIL`, it is printed followed by a ‘‘`;`’’, e.g.

```
10/   (A B) ; T
```

Similarly, if a register is closed, either its left half, right half, or both halves can be changed, depending on the presence or absence, and position of the ‘‘`;`’’ e.g.

```
10/  (A B) ; T    B;^cr        [changes left]
./    B ; T       NIL^cr       [changes right]
./    B ; NIL     A ; C^cr     [changes both]
./    A ; C
```

If ‘‘`;`’’ is used in the unboxed portion of an array, an error will be generated.

[38] the array itself, *not* a variable whose value is an array, e.g., `(EDITA FOO)`, not `(EDITA 'FOO)`.

The $W command will look at both halves of elements in the pointer region, and match if either half matches. Note that $W A ; B is not allowed.

23.12 CJSYS

Note: Cjsys is a LispUsers package that is contained on the file CJSYS.COM. *It only works with Interlisp-10.*

This package provides assistance to Interlisp-10 users who wish to make direct calls on the operating system (via JSYSes). It also makes the coding of certain common ASSEMBLE constructions more convenient. The package defines the following functions:

(JS JSYSNAME AC1 AC2 AC3 RESULT) [NLambda Function]

All arguments are evaluated except for JSYSNAME. Like JSYS (see page 22.6), loads the unboxed values of AC1, AC2, and AC3 into the appropriate registers, and executes the JSYS JSYSNAME. JS differs from JSYS in that the JSYS may be indicated by its *symbolic* name, not just by its number. JS also generates slightly cleaner code than JSYS. JS also differs from JSYS in that:

(a) if any argument is supplied as NIL, then it is not loaded at all, i.e. the corresponding AC will contain garbage. (JSYS loads the AC with 0.)

(b) if RESULT is NIL, then no value is loaded (interpreted, JS returns the string "garbage result from JS").

(c) RESULT can be T, meaning return T if the JSYS skips, NIL if not.

Because of these differences, caution must be exercised in turning JSYS calls into JS calls.

The symbolic JSYS name is looked up on the list JSYSES, an association-list with elements of the form (JSYSNAME JSYSNUMBER qSKIPS). If no entry is found, then the file STENEX.MAC (or SYS:MONSYMS.MAC for Tops-20) is scanned.

Examples: (JS BIN (OPNJFN FILE) NIL NIL 2) returns the value of AC2 after doing a BIN from the JFN of FILE. (JS BOUT (OPNFJN FILE) 3) sends a control-C to FILE. The value of this JS call is garbage.

(XWD N_1 N_2) [Function]

Returns (LOGOR (LLSH N_1 18) (LOGAND N_2 777777Q)), i.e. the word with N_1 in the left half and N_2 in the right.

(BIT BITq WORD) [NoSpread Function]

If WORD is not specified, BIT simply returns a number with bit BITq set to 1 and all other bits 0. If WORD is given, then BIT is a predicate that returns T if BITq is set in WORD. Bits are numbered from left to right.

Examples: (BIT 32) is 8 (=10Q), (BIT 32 8) is T.

(JSYSERROR ERRORN) [NLambda Function]

Returns the TENEX/TOPS-20 error number for ERRORN. For example, (JSYSERROR GJFX23) is 600103Q. JSYSERROR compiles open as a constant.

This package also defines the following ASSEMBLE macros:

(JS JSYSNAME) Can be used in ASSEMBLE statements instead of (JSYS JSYSNUMBER).

(CV EXPR) Expands to (CQ (VAG (FIX EXPR))), which unboxes EXPR to AC1.

(CV2 EXPR) Expands to (CQ2 (VAG (FIX EXPR))), which unboxes EXPR to AC2, saving AC1.

23.13 NOBOX

Note: Nobox is a LispUsers package that is contained on the file NOBOX.COM. *It only works with Interlisp-10.*

This package contains facilities for subverting the normal manner of dynamically allocating and collecting CONS cells, large integer boxes, and floating boxes in Interlisp-10 by using static, compile-time allocation. Storage allocation is controlled by allocating the memory for temporary results (e.g. a list that will be thrown away or a floating number that will not exist outside a local computational context) at compile-time or load-time. This ''static'' storage will be reused whenever the given line of code is re-executed. Because functions which use these facilities may exhibit bizarre behaviour if they are called recursively or if values escape outside of them, these facilities must be used with extreme caution, and should be reserved for those cases where the normal method of storage allocation and garbage collection is not workable or practical. Note: compiled functions need no run-time support for these facilities, i.e. NOBOX does not have to be loaded to execute compiled code.

23.13.1 CONS Cells

The function CBOX may be used to avoid allocation of CONS cells. When run interpreted, CBOX is exactly equivalent to the function CONS. Compiled, CBOX operates like CONS, except that the CONS cell returned is constructed (once) at compile or load time. New values for CAR and CDR are smashed into the cell at each execution.

The function LBOX performs an analagous role for LIST. When run interpreted, LBOX is exactly equivalent to LIST. Compiled, the corresponding CONS cells are allocated at compile or load time. For example, (LBOX A B C) will cause a 3-element static list to be included with a compiled function's literals. Each time the corresponding compiled code is executed, those three cells will be returned containing the current values of the variables A, B, and C.

LBOX allocates as many cells as there are arguments in the corresponding form, i.e. the number of scratch cells is determined at compile time. The iterative statement operator SCRATCHCOLLECT enables avoiding CONSes when the length of a list is not known at compile-time. SCRATCHCOLLECT is used in iterative statements exactly as COLLECT. Each time it is executed, it reuses the cells that it returned on previous executions, which it remembers as an internal scratch list. The length of this scratch list is always the

length of the longest value that was ever returned; new cells are allocated whenever the scratch list runs out, and they are permanently remembered.

The SCRATCHCOLLECT i.s.opr and the function SCRATCHLIST (page 14.2) have similar applications. With SCRATCHLIST, the user makes explicit the origin of the list getting smashed, while with the SCRATCHCOLLECT i.s.opr, the scratch list is hidden (and there is a di erent scratch-list for each occurence of the i.s.opr).

23.13.2 Number Boxes

The functions IBOX, FBOX, and NBOX, and the record declarations IBOX and FBOX are provided to improve the e ciency of arithmetic computations. They permit information to be given to the Interlisp-10 compiler that will inhibit the allocation (and subsequent collection) of number boxes needed for holding temporary results of numeric computations.[39] In addition, access time to variable-values that are known to be large integers or oating point numbers is improved.

The records IBOX and FBOX essentially describe the structure of large integer and oating point boxes respectively. IBOX consists of a single eld, called I, which corresponds to the actual contents of the large integer box. FBOX consists of a single eld, called F, which corresponds to the contents of the oating point box. For example, the user can create a large integer box containing a given value and assign it to X by saying (SETQ X (create IBOX I _ FORM)). Even if the value of FORM is a *small* integer, the result will be stored in a new, large number box. This seeming ine ciency is important because if *some* values of FORM might be large, making *all* values large means that the compiler can be told how to treat all references to X without generating run-time tests to discover how to do the unboxing. Thus, wherever the value of X is to be referenced, the user simply writes (fetch I of X). In compiling this expression, the compiler generates a single MOVE instruction without any type-testing whatsoever. The user can reuse that number box by saying (replace I of X with (FOO)), which is equivalent to, but much more e cent than, (SETN X (FOO)). In other words, once it is known that X is bound to a large integer, (replace I of) can be used in all number-contexts to inform the compiler of that fact.

The facilities described so far do nothing to suppress the creation of unnecessary boxes; indeed, the (create IBOX --) will produces boxes for small numbers that would not be allocated otherwise. The functions (not records) IBOX, FBOX, and NBOX are used to suppress unnecessary boxing of temporaries. E ectively, they cause ''constant'' or ''static'' boxes of the appropriate type to be allocated and stored in a function's literals when a function is compiled or loaded. Those boxes can be used (and reused) to hold temporary results.

IBOX and FBOX can be called with 0 or 1 arguments. If no arguments are speci ed (as opposed to a single argument whose value is NIL), then the value of the function is a large-integer or oating number box which is allocated statically. For example, these might be used to construct an initial binding for a variable into which temporary values will be stored using the I or F assignments. For example:

```
(PROG ((X (IBOX))) (replace I of X with (FOO))    )
```

---

[39] In the latter respect, these duplicate some of what SETN (page 22.5) does, except that they are more convenient to use and are executed with less run-time checking (i.e. SETN will never smash random memory locations).

.

If an argument is speci ed for IBOX or FBOX, then a static box of the appropriate type will be allocated at compile- or load-time, and the value of the argument will be stored in that box whenever the IBOX statement is executed. For example, suppose the user wanted to set a le pointer to 1 past a given byte position. The expression

```
(SETFILEPTR FILE (ADD1 POS))
```

would generate a new number box on each execution for which POS happened to be a large number. That box would be passed into SETFILEPTR and then returned as its value. Since the value is not saved, the box would be thrown away, to be collected later. The expression

```
(SETFILEPTR FILE (IBOX (ADD1 POS)))
```

would store the desired position in a constant box, and no allocations would take place.

As another example, consider a complicated integer expression whose value must be saved in a variable to be used a little further down in a program:

```
(SETQ X (IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5))))
.
.
.
(SETQ Z (IPLUS X (GETFILEPTR FILE)))
```

The Interlisp-10 compiler is smart enough to suppress the boxing inside the (IPLUS 2000 &) expression, but it will generate a box when it comes to do the SETQ. This box can be suppressed by writing

```
(SETQ X (IBOX (IPLUS 2000 (ITIMES FOO (IQUOTIENT FUM 5)))))
```

Furthermore, since it is known that X is bound to a large integer, the Z assignment can be speeded up by writing

```
(SETQ Z (IPLUS X:I (GETFILEPTR FILE)))
```

The function FBOX behaves the same as IBOX, except that it uses constant oating boxes. Note that if the argument of IBOX is FLOATP, then it will be FIXed; if the argument of FBOX is FIXP, it will be FLOATed.

The function NBOX is a generic function for copying unknown values into constant number boxes. It allocates two constant boxes, one integer and one oating, and stores the value of its argument in the one compatible with the value's type. NBOX is useful if the argument value is a constant number box (but one of unknown type) that needs to be copied (see caution (2) below).

23.13.3 Cautions

There are some dangers in using these facilities. The user of this package should be particularly aware of the following:

(1) The F and I elds aim at e ciency more than validity. This means that they *do not check the type* of

the pointer that they smash into. For example, if X is bound to NIL, the expression (replace I of X with Z will clobber CAR and CDR of NIL! The user must be very careful that the arguments given for replacing do indeed point to cells that unboxed numbers can be smashed into. Note: the DECL package (page 23.18) can be used to generate the replaces, IBOXes, FBOXes automatically in a safe and e cient way.

(2) CBOX, LBOX, SCRATCHCOLLECT, IBOX, and FBOX all allocate constant boxes, and those boxes will be reused (i.e. smashed with new values) every time the code containing that function call is executed. If that box is saved in a variable or data-structure (e.g. by a SETQ) as a way of preserving the value it contains, and then the code is re-executed, the value that was saved will be smashed. Thus, the user must beware of using constant boxes to save information in loops or recursions that can get back to the same statement. In these situations, the values must be copied into other cells, perhaps a constant associated with some other line of code, or into cells allocated in the ordinary way. The user must also be careful about returning a constant box as the value of a function, since the caller might unknowingly save the value and re-invoke the box-returner.

(3) Because the constant boxes are allocated only in compiled code, *these functions will work quite di erently compiled and interpreted*. Side e ects which occur because of inadvertent smashing of shared structures will only occur when running compiled de nitions and will not be detectable when running interpreted.

23.14 DATEFORMAT

Note: Dateformat is a LispUsers package that is contained on the le DATEFORMAT.COM. *It only works in Interlisp-10.*

Dateformat is a small le (one function) which provides assistance for constructing format bits for the ODTIM JSYS (output date/time) as required by DATE and GDATE (page 14.9).

(DATEFORMAT KEY$_1$ KEY$_N$) [NLambda NoSpread Function]

KEY$_1$ KEY$_N$ are a set of keywords (unevaluated). DATEFORMAT returns a number suitable as a parameter to DATE and GDATE. The variable DATEFORMAT.DEFAULT is the number used as the initial value to work with. Therefore, to switch any of the defaults, set the variable DATEFORMAT.DEFAULT to be the value of a call to DATEFORMAT with the appropriate keys.

The keywords are given below (usually in pairs) and can be thought of as switches (i.e. turn on or o a particular format feature). If no keyword is given for a particular pair, the default is used.

The variable DATEFORMAT.KEYS is a list of the keywords used for spelling correction.

DATE (default)
NO.DATE Do/don't include the date information.

NAME.OF.MONTH (default)
NUMBER.OF.MONTH
Show the month as a name (NAME.OF.MONTH) or a number (NUMBER.OF.MONTH).

MONTH.LONG

`MONTH.SHORT` (default)
: If the name of the month was requested, spell it out (`MONTH.LONG`) or abbreviate it (`MONTH.SHORT`).

`YEAR.LONG`
`YEAR.SHORT` (default)
: Print four digit year, e.g. 1978 (`YEAR.LONG`) or two digit year, e.g. 78 (`YEAR.SHORT`).

`DAY.OF.WEEK`
`NO.DAY.OF.WEEK` (default)
: Do/Don't include the day of the week in the date information.

`DAY.LONG`
`DAY.SHORT` (default)
: If the day of the week was included, spell it out (`DAY.LONG`) or abbreviate it (`DAY.SHORT`).

`DASHES` (default)
`SLASHES`
`SPACES`
: Separate the <day>, <month>, and <year> elds with dashes/slashes/spaces.

`USA.FORMAT`
`EUROPE.FORMAT` (default)
: Print the date in the order <month> <day> <year> (`USA.FORMAT`) or in the order <day> <month> <year> (`EUROPE.FORMAT`).

`LEADING.SPACES` (default)
`NO.LEADING.SPACES`
: If `LEADING.SPACES` is speci ed, the <day> eld will always be two characters long. If `NO.LEADING.SPACES`, the <day> eld can be one character for dates earlier than the 10th.

`TIME` (default)
`NO.TIME`
: Do/Don't include the time information.

`TIME.ZONE`
`NO.TIME.ZONE` (default)
: Do/Don't include the time zone in the time speci cation.

`SECONDS` (default)
`NO.SECONDS`
: Do/Don't include the seconds.

`CIVILIAN.TIME`
`MILITARY.TIME` (default)
: Use 12 hour time with AM or PM (`CIVILIAN.TIME`) or 24 hour time

(MILITARY.TIME).

23.15 EXEC

Note: The Exec package is a LispUsers package that is contained on the file EXEC.COM*. The Exec package uses the passwords package (see page 23.62). Loading* EXEC.COM *will load* PASSWORDS.COM *if it has not already been loaded. Note: some of the facilities described below will work correctly only on TENEX systems, others only on TOPS-20. The system will inform the user when he attempts to use a facility not supported by his particular operating system.*

This package defines a set of programmer's assistant commands which resemble features of the Tenex EXEC. It also defines functions that provide certain EXEC capabilities for Interlisp programs, e.g. changing the connected directory, detaching the job, etc.

23.15.1 Exec Commands

DA [Exec Command]

Prints out the current time and date.

LD [Exec Command]
SY [Exec Command]
WHE [Exec Command]

Prints SYSTAT information, just like the LD subsystem. Jobs are sorted in inverse order of CPU utilization.

LD USERNAME [Exec Command]

Prints information for the specified user only.

LD ALL [Exec Command]

Like LD, but includes system jobs.

DET [Exec Command]

Detaches the current job.

QU [Exec Command]

Does a (LOGOUT). Does not go on history list.

LINK USER [Exec Command]
TALK USER [Exec Command]

Mimics the exec link command. If USER has multiple jobs logged in, asks which tty to link to.

BR [Exec Command]

Breaks links.

CONN DIR PWD [Exec Command]

Connects to the directory DIR. If the password PWD is not given and is required, CONN will prompt. DIR can be abbreviated; if omitted, it defaults to the user's login

directory. If PWD is given in command line, it is removed from the history list so that ?? will not print it out. Password prompting is handled by GETPASSWORD from the passwords package (page 23.62).

NDIR FILEGR OUP [Exec Command]

Prints the les in FILEGR OUP in a multi-column format.

NDIR FILEGR OUP [Exec Command]

Deletes speci ed les. Uses DIRECTORY (page 14.6). Note that if <esc> is speci ed, *all* les that match will be deleted. This command is undoable.

UND FILEGR OUP [Exec Command]

Undeletes the speci ed les (undoably).

DELVER FILEGR OUP [Exec Command]

Deletes all but 1 version of the legroup speci ed. Uses DIRECTORY (page 14.6), so FILEGR OUP may utilize any of the options allowed for directory legroup speci cations.

EXP DIR [Exec Command]

Expunges directory DIR. If the user does not have access to DIR, a message is printed.

TY FILE OUTFILE BYTESIZE [Exec Command]
SEE FILE OUTFILE BYTESIZE [Exec Command]

Copies FILE to OUTFILE , or to T if OUTFILE is not given. Assumes that the bytes of FILE are BYTESIZE bits wide (BYTESIZE = NIL defaults to 7). Suppresses blank lines and control character sequences used to indicate font changes.

DSK DIR DAYS [Exec Command]

Prints out disk allocation and usage for the directory DIR using DSKSTAT. Also prints total size of les untouched in days DAYS (90 if DAYS not speci ed).

FI [Exec Command]

Like the EXEC FILESTAT command, prints out status of all currently assigned JFNS for the current job.

FI JFN [Exec Command]

Prints information for JFN only.

23.15.2 EXEC Functions

(JOB#) [Function]

Returns the job number for the logged in job.

(TTY#) [Function]

Returns the teletype-number of the current job.

(DETACH) [Function]

Detaches the current job.

(DETACHEDP) [Function]

Returns T if the current program is running detached.

(LINKTOTTY TTY q) [Function]

Generates a two-way link between the controlling terminal of the user's job and TTY q . Returns T if the link was successful, otherwise prints an error message and returns NIL.

(LINKTOUSER USER) [Function]

Links the controlling terminal to a terminal associated with USER . Generates an error if the user is not logged in or not attached. If USER has more than one attached job, then a systat of his jobs is printed, and the user is asked to provide the proper tty number for the job. Returns T if successful.

(BREAKLINKS) [Function]

Breaks all links to the user's controlling terminal.

(CNDIR DIR PASSW ORD) [Function]

Implements the CONN command.

(/DELFILE FILE) [Function]

Undoable version of DELFILE.

(/UNDELFILE FILE) [Function]

Undeletes a single le (undoably).

(EXPUNGE DIR) [Function]

Expunges directory DIR. On TENEX, DIR is ignored. and the connected directory is expunged. On TOPS20, if the user does not have access to DIR, a message is printed.

(COPYALLBYTES FR OMFILE TOFILE BYTESIZE) [Function]

Implements the SEE command.

(DSKSTAT DIR PRINTIF O VER PRINTSYS PRINTDEL PRINTOLD) [Function]

Prints disk usage statistics for directory DIR (either a name or number).

If PRINTIF O VER is NIL, this means always print. If PRINTIF O VER is T, this means only print if DIR is over allocation. If PRINTIF O VER is a number, this means only print if DIR has more than that many pages in use.

If PRINTSYS is T, this means print system disk statistics too.

If PRINTDEL is T, this means print total size of deleted les for DIR (this is slow).

If PRINTOLD is T or a number, this means print total size of les untouched in 90 (or PRINTOLD) days.

(MEMSTAT PG1 PGN FORK) [Function]

Prints the status of the memory pages PG1 (0 if PG1 = NIL) to PGN (the last page of memory if NIL) in fork FORK . FORK is either NIL, meaning the current fork,

or a fork handle.

23.16 PASSWORDS

Note: Passwords is a LispUsers package that is contained on the le PASSWORDS.COM. *It only works with Interlisp-10.*

(GETPASSWORD DIRECTORYNAME) [Function]

Prompts the user for the password for the given directory. The user's response is not echoed. GETPASSWORD remembers the password so that it need not ask again; however, saved information is cleared before SYSOUT, so that the SYSOUT contains no passwords.

23.17 TELNET

Note: Telnet is a LispUsers package that is contained on the le TELNET.COM. *It only works with Interlisp-10. Since the telnet package uses the net package, loading* TELNET.COM *will also load* NET.COM *unless it has already been loaded.*

This package makes it possible to interact with connections created via the net package (page 23.64) without leaving Interlisp. In addition, all typeout is included in the DRIBBLE le. It permits connections to ARPANET hosts (a la TELNET).

(TELNET CONNECTION TYPE SKT _) [Function]

CONNECTION may be an instance of a CONNECTION record (as created by MAKENEWCONNECTION, page 23.64). Alternatively, if CONNECTION is a litatom, TELNET uses (MAKENEWCONNECTION CONNECTION TYPE SKT) for the connection. In any case, TELNET returns the connection as an instance of the CONNECTION record, so that it is possible to TELNET back.

23.18 FTP

Note: Ftp is a LispUsers package that is contained on the le FTP.COM. *It only works with Interlisp-10. Since the Ftp package uses the net and passwords packages, loading* FTP.COM *will also load* NET.COM *and* PASSWORDS.COM *if they are not already loaded.*

The ftp package makes it possible to deal with les at other hosts on the Arpa network almost as if they were les on the user's local machine, i.e. the les can be opened via INFILE, OUTFILE, OPENFILE, read from and printed to by the ordinary reading and printing functions, and closed in the standard way.

Files on remote hosts are designated by including the host name between curly brackets, {}, at the front of the ordinary le name. Since curly brackets are illegal characters in regular le names, a BAD

FILE NAME error is generated. This error is intercepted by an entry on ERRORTYPELST (see page 9.16) which then establishes the appropriate network connections.[40] For example, (INFILE '{BBN-D}<LEWIS>INIT.LISP) will open the le <LEWIS>INIT.LISP on the host BBN-D and make it be the primary input le. The user could then say (READ) to obtain the rst expression on that le. The ftp package extends the functions PACKFILENAME, UNPACKFILENAME, and FILENAMEFIELD so that they will associate the curly bracket syntax with the new le eld HOST. Thus, (PACKFILENAME 'HOST 'BBND 'NAME 'INIT) will return {BBND}INIT.

Remote les have certain properties that limit how they may be used:

(1) RANDACCESSP is NIL for such les, and SETFILEPTR may not be applied to them. This means, for example, that functions and variables may not be loaded from such les via LOADFNS.

(2) The open bytesize of a remote le may not be changed (e.g. by SETFILEINFO). This means that Interlisp- 10 compiled les may not be loaded from remote hosts.

(3) The remote host may close the connection spontaneously (e.g. because of a timeout if the le is not referenced for some length of time, or because of a crash). If this happens, the next attempt at reading or writing on the le will generate FILE DATA ERROR. Note: it is unwise to keep a remote le open for long periods of time.[41]

When the connection for the remote le is rst established, a password for the remote machine/directory may be required. The user will be asked to supply one via the passwords package (page 23.62). Alternatively, if the host name has on its property list the property LOGIN with value of the form (NAME PASSWORD ACCOUNT), then the indicated NAME, PASSWORD, and ACCOUNT will be used to log the user into the remote host.[42]

(FTP HOST FILE ACCESS USER PASSWORD ACCOUNT BYTESIZE) [Function]

Opens a network connection to the ftp server at HOST. If ACCESS = INPUT or OUTPUT, FTP works like OPENFILE: value is a literal atom of the form {HOST}FILE which can then be used as a le name by all Interlisp input and output functions, e.g. READ, PRINT, COPYBYTES, etc.[43] For example, (FTP 'SU-AI 'YUMYUM%[P,DOC%] 'INPUT) will allow the Stanford Restraurant Guide to be read. Note that FILE must satisfy the le name conventions of the remote host.

[40] Note: it is fairly expensive to open a network connection as compared with the time to open a local le, e.g. an order of magnitude slower.

[41] For input les, these limitations may be skirted conveniently in the following way: if a colon appears between the last character of the host name and the right curly bracket (e.g. {BBND:}<LEWIS>INIT.LISP), then the remote le will be copied to a temporary *local* le when it is opened, and all subsequent references will be to that local le.

[42] If the value is of the form (NAME NIL ACCOUNT), then (GETPASSWORD NAME) will be used for the password. If the ACCOUNT eld is NIL, no account will be supllied to the remote host. If no LOGIN property is supplied, ANONYMOUS will be used as the user name.

[43] In reality, this ‘‘ le’’ is a network connection to the host’s ftp server. This ‘‘ le’’ has a WHENCLOSE attribute (page 6.11) associated with it so that when Interlisp closes the le, the correct terminating sequence will be performed.

If ACCESS = DIRECTORY, then FTP will print on the terminal the names of all files which match FILE, e.g. (FTP 'PARC-MAXC2 '<NETLISP>*.SAV 'DIRECTORY).

USER, PASSWORD, and ACCOUNT are used for logging in to the remote host. If not supplied, the values are obtained from the LOGIN property (if any) as described above. BYTESIZE is the byte size in which to open the connection. Byte sizes of 7, 8, 16, 32 and 36 are supported. BYTESIZE = NIL defaults to 7.

23.19 NET

Note: Net is a LispUsers package that is contained on the file NET.COM. *It only works with Interlisp-10.*

This package contains functions for establishing ARPANET connections from an Interlisp-10 job. A connection is described by and is an instance of the record CONNECTION. The only fields of interest to the user in this record are IN and OUT, which are guaranteed to be CAR and CADR, respectively. IN is a file name which can be read from, OUT a file name which can be printed to.

(MAKENEWCONNECTION HOST TYPE SKT SCRATCHCONN WAITFLG) [Function]

Makes a connection to HOST. For TYPE = ARPA, HOST is the name of the host to which the connection is to be made. For SKT = NIL (the normal case), the connection will be to the telnet server of HOST; connections to other servers can be made by supplying the appropriate value for SKT.

The value of MAKENEWCONNECTION is a CONNECTION. If WAITFLG is non-NIL, MAKENEWCONNECTION waits until its request for connection is acknowledged. Otherwise, CHECKCONNECTION must be called on the result before it is used (this allows additional processing to be done while waiting for the remote host to respond).

If SCRATCHCONN is non-NIL, it is a scratch connection which is reused.

For example, (MAKENEWCONNECTION 'BBND) makes an ARPA connection to BBND, (MAKENEWCONNECTION 'SU-AI 'ARPA 'FINGER) makes a connection to the Stanford WHEREIS service.

(CLOSECONNECTION CONNECTION) [Function]

Closes the given CONNECTION and replaces the IN and OUT fields with NIL.

(CHECKCONNECTION CONNECTION) [Function]

Checks to make sure that the given connection is still open (e.g. it hasn't been closed remotely). If the connection is valid, CONNECTION is returned. If the connection is in an in-between state, i.e. in the process of being opened or closed, CHECKCONNECTION waits to see what happens before returning. Otherwise the connection is cleaned up (as if a CLOSECONNECTION were performed) and CHECKCONNECTION returns NIL.

(NETSERVER ARPAq WAITFLG) [Function]

Initiates a ''server'' connection. This is a connection which will talk to a ''user'' connection. If WAITFLG is non-NIL, waits for a user to connect; if WAITFLG = NIL,

returns immediately (and CHECKCONNECTION must be called on the connection before the connection is actually used). ARP Aq defaults to 0.

(NETUSER HOST USER ARP Aq W AITFL G) [Function]

Initiates the other half of an Arpa connection. ARP Aq defaults to 0 and must be the same as the argument given the corresponding call to NETSERVER. USER must be the USERNUMBER (directory number) under which the server job is logged in.

For example, to establish an ARPANET connection between two Interlisp jobs (which can then be written to and read from like les), do (SETQ CONN (NETSERVER)) in one job and (SETQ CONN (NETUSER HOST USER)) in the other job, where HOST is the machine on which the rst job is running and USER is the directory number under which the rst job is logged in (obtainable through the function USERNUMBER). Then, perform (CHECKCONNECTION CONN) in each job; when these return, the connection is ready to be used.

(FORCEOUT CONNECTION/FILE) [Function]

Normally, characters sent to the "OUT" of a connection are bu ered locally. The function FORCEOUT can be used to force partially lled packets of bytes to be sent across the connection. The argument to FORCEOUT can either be the CONNECTION record or the OUT lename.

Net

23.66