3. LISTS

One of the most useful datatypes in Lisp is the list cell, a data structure that contains pointers to two other objects, called the CAR and the CDR of the list cell. You can build very complicated structures out of list cells, including lattices and trees, but most often they're used to represent simple linear lists of objects.

The following functions are used to manipulate individual list cells:

(**CONS** *X* *Y*) [Function]

CONS is the primary list construction function. It creates and returns a new list cell containing pointers to *X* and *Y*. If *Y* is a list, this returns a list with *X* added at the beginning of *Y*.

(**LISTP** *X*) [Function]

Returns *X* if *X* is a list cell, e.g., something created by CONS; NIL otherwise.

```
(LISTP NIL) = NIL
```

(**NLISTP** *X*) [Function]

The same as (NOT (LISTP *X*)). Returns T if *X* is not a list cell, NIL otherwise. However, (NLISTP NIL) = T

(**CAR** *X*) [Function]

Returns the first element of the list *X*. CAR of NIL is always NIL. For all other nonlists (e.g., symbols, numbers, etc.), the value returned is controlled by CAR/CDRERR (below).

(**CDR** *X*) [Function]

Returns all but the first element of the list *X*. CDR of NIL is always NIL. The value of CDR for other nonlists is controlled by CAR/CDRERR (below).

CAR/CDRERR [Variable]

The variable CAR/CDRERR controls the behavior of CAR and CDR when they are passed non-lists (other than NIL).

If CAR/CDRERR = NIL (the current default), then CAR or CDR of a non-list (other than NIL) return the string "{car of non-list}" or "{cdr of non-list}". If CAR/CDRERR = T, then CAR and CDR of a non-list (other than NIL) causes an error.

If CAR/CDRERR = ONCE, then CAR or CDR of a string causes an error, but CAR or CDR of anything else returns the string "{car of non-list}" or "{cdr of non-list}" as above. This catches loops which repeatedly take CAR or CDR of an object, but it allows one-time errors to pass undetected.

If CAR/CDRERR = CDR, then CAR of a non-list returns "{car of non-list}" as above, but CDR of a non-list causes an error. This setting is based on the observation that nearly all infinite loops involving non-lists occur from taking CDRs, but a fair amount of careless code takes CAR of something it has not tested to be a list.

(**CAAR** *X*) (**CADR** *X*) (**CDDR** *X*) etc. [Function]

Often, combinations of CAR and CDR are used to extract parts of complex list structures. Functions of the form C...R may be used for some of these combinations:

```
(CAAR X) ==> (CAR (CAR X))
(CADR X) ==> (CAR (CDR X))
(CDDDDR X) ==> (CDR (CDR (CDR (CDR X))))
```

All 30 combinations of nested CARs and CDRs up to 4 deep are included in the system.

(**RPLACD** *X Y*) [Function]

Replaces the CDR of the list cell *X* with *Y*. This physically changes the internal structure of *X*, as opposed to CONS, which creates a new list cell. You can make a circular list by using RPLACD to place a pointer to the beginning of a list at the end of the list.

The value of RPLACD is *X*. An attempt to RPLACD NIL will cause an error, Attempt to RPLACD NIL (except for (RPLACD NIL NIL)). An attempt to RPLACD any other non-list will cause an error, Arg not list.

(**RPLACA** *X Y*) [Function]

Like RPLACD, but replaces the CAR of *X* with *Y*. The value of RPLACA is *X*. An attempt to RPLACA NIL will cause an error, Attempt to RPLACA NIL, (except for (RPLACA NIL NIL)). An attempt to RPLACA any other non-list will cause an error, Arg not list.

(**RPLNODE** *X A D*) [Function]

Performs (RPLACA *X A*), (RPLACD *X D*), and returns *X*.

(**RPLNODE2** *X Y*) [Function]

Performs (RPLACA *X* (CAR *Y*)), (RPLACD *X* (CDR *Y*)) and returns *X*.

(**FRPLACD** *X Y*) [Function]
(**FRPLACA** *X Y*) [Function]
(**FRPLNODE** *X A D*) [Function]
(**FRPLNODE2** *X Y*) [Function]

Faster versions of RPLACD, etc.

Usually, you don't use list cells alone, but in structures called "lists". A list is represented by a list cell whose CAR is the first element of the list, and whose CDR is the rest of the list. That's normally another list cell (with another element of the list) or the "empty list," NIL, marking the list's end. List elements may be any Lisp objects, including other lists.

You type in a list as a sequence of Lisp data objects (symbols, numbers, other lists, etc.) enclosed in parentheses or brackets. Note that () is read as the symbol NIL.

Sometimes, you won't want your list to end in `NIL`, but just with the final element. To indicate that, type a period (with spaces on both sides) in front of the final element. This makes `CDR` of the list's final cell be the element immediately following the period, e.g. `(A . B)` or `(A B C . D)`. Note that a list needn't end in `NIL`. It is simply a structure composed of one or more list cells. The input sequence `(A B C . NIL)` is equivalent to `(A B C)`, and `(A B . (C D))` is equivalent to `(A B C D)`. Note, however, that `(A B . C D)` will create a list containing the five symbols `A`, `B`, `%.`, `C`, and `D`.

Lists are printed by printing a left parenthesis, and then printing the first element of the list, a space, the second element, etc., until the final list cell is reached. The individual elements of a list are printed by `PRIN1`, if the list is being printed by `PRIN1`, and by `PRIN2` if the list is being printed by `PRINT` or `PRIN2`. Lists are considered to terminate when `CDR` of some node is not a list. If `CDR` of this terminal node is `NIL` (the usual case), `CAR` of the last node is printed followed by a right parenthesis. If `CDR` of the terminal node is *not* `NIL`, `CAR` of the last node is printed, followed by a space, a period, another space, `CDR` of the last node, and the right parenthesis. A list input as `(A B C . NIL)` will print as `(A B C)`, and a list input as `(A B . (C D))` will print as `(A B C D)`. `PRINTLEVEL` affects the printing of lists (see the `PRINTLEVEL` section of Chapter 25), and that carriage returns may be inserted where dictated by `LINELENGTH` (see the Output Functions section of Chapter 25).

Note: Be careful when testing the equality of list structures. `EQ` will be true only when the two lists are the *exact* same list. For example,

```
←  (SETQ A '(1 2))
    (1 2)
←  (SETQ B A)
    (1 2)
←  (EQ A B)
    T
←(SETQ C '(1 2))
    (1 2)
←(EQ A C)
    NIL
←(EQUAL A C)
    T
```

In the example above, the values of `A` and `B` are the exact same list, so they are `EQ`. However, the value of `C` is a totally different list, although it happens to have the same elements. `EQUAL` should be used to compare the elements of two lists. In general, one should notice whether list manipulation functions use `EQ` or `EQUAL` for comparing lists. This is a frequent source of errors.

Creating Lists

(**LIST** *X1 X2 ... XN*) [NoSpread Function]

Returns a list of its arguments, e.g.

```
(LIST 'A 'B '(C D))  =>  (A B (C D))
```

(**LIST*** *X1 X2 ... XN*) [NoSpread Function]

Returns a list of its arguments, using the last argument for the tail of the list. This is like an iterated `CONS`: `(LIST* A B C) == (CONS A (CONS B C))`. For example,

```
(LIST* 'A 'B 'C)  =>  (A B . C)
```

```
(LIST* 'A 'B '(C D))  =>  (A B C D)
```

(**APPEND** X_1 X_2 ... X_N) [NoSpread Function]

Copies the top level of the list X_1 and appends this to a copy of the top level of the list X_2 appended to ... appended to X_N, e.g.,

```
(APPEND '(A B) '(C D E) '(F G))  =>  (A B C D E F G)
```

Only the first N−1 lists are copied. However N = 1 is treated specially; (APPEND X) copies the top level of a single list. To copy a list to all levels, use COPY.

The following examples illustrate the treatment of non-lists:

```
(APPEND '(A B C) 'D)  =>  (A B C . D)
(APPEND 'A '(B C D))  =>  (B C D)
(APPEND '(A B C . D) '(E F G))  =>  (A B C E F G)
(APPEND '(A B C . D))  =>  (A B C . D)
```

(**NCONC** X_1 X_2 ... X_N) [NoSpread Function]

Returns the same value as APPEND, but modifies the list structure of X_1 ... X_{n-1}.

NCONC cannot change NIL to a list:

```
←(SETQ FOO NIL)
   NIL
←(NCONC FOO '(A B C))
   (A B C)
←FOO
   NIL
```

Although the value of the NCONC is (A B C), FOO has *not* been changed. The "problem" is that while it is possible to alter list structure with RPLACA and RPLACD, there is no way to change the non-list NIL to a list.

(**NCONC1** *LST* *X*) [Function]

Adds *X* to the end of *LST*: (NCONC *LST* (LIST *X*))

(**ATTACH** *X* *L*) [Function]

"Attaches" *X* to the front of *L* by doing a RPLACA and RPLACD. The value is EQUAL to (CONS *X* *L*), but EQ to *L*, which it physically changes (except if *L* is NIL). (ATTACH X NIL) is the same as (CONS X NIL). Otherwise, if *L* is not a list, an error is generated, Arg not list.

(**MKLIST** *X*) [Function]

"Make List." If *X* is a list or NIL, returns *X*; Otherwise, returns (LIST *X*).

Building Lists From Left to Right

(**TCONC** *PTR X*) [Function]

TCONC is similar to NCONC1; it is useful for building a list by adding elements one at a time at the end. Unlike NCONC1, TCONC does not have to search to the end of the list each time it is called. Instead, it keeps a pointer to the end of the list being assembled, and updates this pointer after each call. This can be considerably faster for long lists. The cost is an extra list cell, *PTR*. (CAR *PTR*) is the list being assembled, (CDR *PTR*) is (LAST (CAR *PTR*)). TCONC returns *PTR*, with its CAR and CDR appropriately modified.

PTR can be initialized in two ways. If *PTR* is NIL, TCONC will create and return a *PTR*. In this case, the program must set some variable to the value of the first call to TCONC. After that, it is unnecessary to reset the variable, since TCONC physically changes its value. Example:

```
←(SETQ FOO (TCONC NIL 1))
    ((1) 1)
←(for I from 2 to 5 do (TCONC FOO I))
    NIL
←FOO
    ((1 2 3 4 5) 5)
```

If *PTR* is initially (NIL), the value of TCONC is the same as for *PTR* = NIL. but TCONC changes *PTR*. This method allows the program to initialize the TCONC variable before adding any elements to the list. Example:

```
←(SETQ FOO (CONS))
    (NIL)
←(for I from 1 to 5 do (TCONC FOO I))
    NIL
←FOO
    ((1 2 3 4 5) 5)
```

(**LCONC** *PTR X*) [Function]

Where TCONC is used to add *elements* at the end of a list, LCONC is used for building a list by adding *lists* at the end, i.e., it is similar to NCONC instead of NCONC1. Example:

```
←(SETQ FOO (CONS))
    (NIL)
←(LCONC FOO '(1 2))
    ((1 2) 2)
←(LCONC FOO '(3 4 5))
    ((1 2 3 4 5) 5)
←(LCONC FOO NIL)
    ((1 2 3 4 5) 5)
```

LCONC uses the same pointer conventions as TCONC for eliminating searching to the end of the list, so that the same pointer can be given to TCONC and LCONC interchangeably. Therefore, continuing from above,

```
←(TCONC FOO NIL)
    ((1 2 3 4 5 NIL) NIL)
```

```
←(TCONC FOO '(3 4 5))
   ((1 2 3 4 5 NIL (3 4 5)) (3 4 5))
```

The functions DOCOLLECT and ENDCOLLECT also let you build lists from left-to-right like TCONC, but without the overhead of an extra list cell. The listis kept as a circular list. DOCOLLECT adds items; ENDCOLLECT replaces the tail with its second argument, and returns the full list.

(**DOCOLLECT** *ITEM LST*) [Function]

"Adds" *ITEM* to the end of *LST*. Returns the new circular list. Note that *LST* is modified, but it is not EQ to the new list. The new list should be stored and used as *LST* to the next call to DOCOLLECT.

(**ENDCOLLECT** *LST TAIL*) [Function]

Takes *LST*, a list returned by DOCOLLECT, and returns it as a non-circular list, adding *TAIL* as the terminating CDR.

Here is an example using DOCOLLECT and ENDCOLLECT. HPRINT is used to print the results because they are circular lists. Notice that FOO has to be set to the value of DOCOLLECT as each element is added.

```
←(SETQ FOO NIL]
   NIL
←(HPRINT (SETQ FOO (DOCOLLECT 1 FOO]
   ↑(1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 2 FOO]
   ↑(2 1 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 3 FOO]
   ↑(3 1 2 . {1})
←(HPRINT (SETQ FOO (DOCOLLECT 4 FOO]
   ↑(4 1 2 3 . {1})
←(SETQ FOO (ENDCOLLECT FOO 5]
   (1 2 3 4 . 5)
```

The following two functions are useful when writing programs that reuse a scratch list to collect together some result(s) (both of these compile open):

(**SCRATCHLIST** *LST* X_1 X_2 ... X_N) [NLambda NoSpread Function]

SCRATCHLIST sets up a context in which the value of *LST* is used as a "scratch" list. The expressions X_1, X_2, ... X_N are evaluated in turn. During the course of evaluation, any value passed to ADDTOSCRATCHLIST will be saved, reusing CONS cells from the value of *LST*. If the value of *LST* is not long enough, new CONS cells will be added onto its end. If the value of *LST* is NIL, the entire value of SCRATCHLIST will be "new" (i.e., no CONS cells will be reused).

(**ADDTOSCRATCHLIST** *VALUE*) [Function]

For use under calls to SCRATCHLIST. *VALUE* is added on to the end of the list of things being collected by SCRATCHLIST. When SCRATCHLIST returns, its value is a list containing all of the things added by ADDTOSCRATCHLIST.

Copying Lists

(**COPY** *X*) [Function]

Creates and returns a copy of the list *X*. All levels of *X* are copied down to non-lists, so that if *X* contains arrays and strings, the copy of *X* will contain the same arrays and strings, not copies. `COPY` is recursive in the `CAR` direction only, so very long lists can be copied.

To copy just the *top level* of *X*, do `(APPEND X)`.

(**COPYALL** *X*) [Function]

Like `COPY`, but it copies down to atoms. Arrays, hash-arrays, strings, user data types, etc., are all copied. Analagous to `EQUALALL` (see the Equality Predicates section of Chapter 9). This will not work if given a data structure with circular pointers; in this case, use `HCOPYALL`.

(**HCOPYALL** *X*) [Function]

Like `COPYALL`, but it will work even if the data structure contains circular pointers.

Extracting Tails of Lists

(**NTH** *X N*) [Function]

Returns the tail of *X* beginning with the *N*th element. Returns `NIL` if *X* has fewer than *N* elements. This is different from Common Lisp's `NTH`. Examples:

```
(NTH '(A B C D) 1)  =>  (A B C D)
(NTH '(A B C D) 3)  =>  (C D)
(NTH '(A B C D) 9)  =>  NIL
(NTH '(A . B) 2)  =>  B
```

For consistency, if `N = 0`, `NTH` returns `(CONS NIL X)`:

```
(NTH '(A B) 0)  =>  (NIL A B)
```

(**FNTH** *X N*) [Function]

Faster version of `NTH` that terminates on a null-check.

(**LAST** *X*) [Function]

Returns the last list cell in the list *X*. Returns `NIL` if *X* is not a list. Examples:

```
(LAST '(A B C))   =>  (C)
(LAST '(A B . C))  =>  (B . C)
(LAST 'A)  =>  NIL
```

(**FLAST** *X*) [Function]

Faster version of `LAST` that terminates on a null-check.

(**NLEFT** *L N TAIL*) [Function]

`NLEFT` returns the tail of *L* that contains *N* more elements than *TAIL*. If *L* does not contain *N* more elements than *TAIL*, `NLEFT` returns `NIL`. If *TAIL* is `NIL` or not a tail of *L*, `NLEFT` returns the last *N* list cells in *L*. NLEFT can be used to work backwards through a list. Example:

```
←(SETQ FOO '(A B C D E))
   (A B C D E)
←(NLEFT FOO 2)
   (D E)
←(NLEFT FOO 1 (CDDR FOO))
   (B C D E)
←(NLEFT FOO 3 (CDDR FOO))
   NIL
```

(**LASTN** *L N*) [Function]

Returns (CONS X Y), where Y is the last *N* elements of *L*, and X is the initial segment, e.g.,

```
(LASTN '(A B C D E) 2)  =>  ((A B C) D E)
(LASTN '(A B) 2)  =>  (NIL A B)
```

Returns NIL if *L* is not a list containing at least *N* elements.

(**TAILP** *X Y*) [Function]

Returns *X*, if *X* is a *tail* of the list *Y*; otherwise NIL. *X* is a tail of *Y* if it is EQ to 0 or more CDRs of *Y*.

Note: If *X* is EQ to 1 or more CDRs of *Y*, *X* is called a "proper tail."

Counting List Cells

(**LENGTH** *X*) [Function]

Returns the length of the list *X*, where "length" is defined as the number of CDRs required to reach a non-list. Examples:

```
(LENGTH '(A B C))  =>  3
(LENGTH '(A B C . D))  =>  3
(LENGTH 'A)  =>  0
```

(**FLENGTH** *X*) [Function]

Faster version of LENGTH that terminates on a null-check.

(**EQLENGTH** *X N*) [Function]

Equivalent to (EQUAL (LENGTH *X*) *N*), but more efficient, because EQLENGTH stops as soon as it knows that *X* is longer than *N*. EQLENGTH is safe to use on (possibly) circular lists, since it is "bounded" by *N*.

(**COUNT** *X*) [Function]

Returns the number of list cells in the list *X*. Thus, COUNT is like a LENGTH that goes to all levels. COUNT of a non-list is 0. Examples:

```
(COUNT '(A))  =>  1
(COUNT '(A . B))  =>  1
(COUNT '(A (B) C))  =>  4
```

In this last example, the value is 4 because the list `(A X C)` uses three list cells for any object *X*, and `(B)` uses another list cell.

(**COUNTDOWN** *X N*) [Function]

Counts the number of list cells in *X*, decrementing *N* for each one. Stops and returns *N* when it finishes counting, or when *N* reaches 0. `COUNTDOWN` can be used on circular structures since it is "bounded" by *N*. Examples:

```
(COUNTDOWN '(A) 100)  =>  99
(COUNTDOWN '(A . B) 100)  =>  99
(COUNTDOWN '(A (B) C) 100)  =>  96
(COUNTDOWN (DOCOLLECT 1 NIL) 100)  =>  0
```

(**EQUALN** *X Y DEPTH*) [Function]

Like `EQUAL`, for use with (possibly) circular structures. Whenever the depth of `CAR` recursion plus the depth of `CDR` recursion exceeds *DEPTH*, `EQUALN` does not search further along that chain, and returns the symbol `?`. If recursion never exceeds *DEPTH*, `EQUALN` returns `T` if the expressions *X* and *Y* are `EQUAL`; otherwise `NIL`.

```
(EQUALN '(((A)) B) '(((Z)) B) 2)  =>  ?
(EQUALN '(((A)) B) '(((Z)) B) 3)  =>  NIL
(EQUALN '(((A)) B) '(((A)) B) 3)  =>  T
```

Set Operations

(**INTERSECTION** *X Y*) [Function]

Returns a list whose elements are members of both lists *X* and *Y* (using `EQUAL` to do compares).

Note that `(INTERSECTION X X)` gives a list of all members of `X` without duplicates.

(**UNION** *X Y*) [Function]

Returns a (new) list consisting of all elements included on either of the two original lists (using `EQUAL` to compare elements). It is more efficient for *X* to be the shorter list.

The value of `UNION` is *Y* with all elements of *X* not in *Y* `CONS`ed on the front of it. Therefore, if an element appears twice in *Y*, it will appear twice in `(UNION X Y)`. Since `(UNION '(A) '(A A)) = (A A)`, while `(UNION '(A A) '(A)) = (A)`, `UNION` is non-commutative.

(**LDIFFERENCE** *X Y*) [Function]

"List Difference." Returns a list of the elements in *X* that are not members of *Y* (using `EQUAL` to compare elements).

Note: If *X* and *Y* share no elements, `LDIFFERENCE` returns a copy of *X*.

(**LDIFF** *LST TAIL ADD*) [Function]

TAIL must be a tail of *LST*, i.e., `EQ` to the result of applying some number of `CDR`s to *LST*. `(LDIFF LST TAIL)` returns a list of all elements in *LST* up to *TAIL*.

If *ADD* is not `NIL`, the value of `LDIFF` is effectively `(NCONC ADD (LDIFF LST TAIL))`, i.e., the list difference is added at the end of *ADD*.

If *TAIL* is not a tail of *LST*, `LDIFF` generates an error, `LDIFF: not a tail`. `LDIFF` terminates on a null-check, so it will go into an infinite loop if *LST* is a circular list and *TAIL* is not a tail.

Example:

```
←(SETQ FOO '(A B C D E F))
   (A B C D E F)
←(CDDR FOO)
   (C D E F)
←(LDIFF FOO (CDDR FOO))
   (A B)
←(LDIFF FOO (CDDR FOO) '(1 2))
   (1 2 A B)
←(LDIFF FOO '(C D E F))
   LDIFF: not a tail
   (C D E F)
```

Note that the value of `LDIFF` is always new list structure unless *TAIL* `= NIL`, in which case the value is *LST* itself.

Searching Lists

(**MEMB** *X Y*) [Function]

Determines if *X* is a member of the list *Y*. If there is an element of *Y* `EQ` to *X*, returns the tail of *Y* starting with that element. Otherwise, returns `NIL`. Examples:

```
(MEMB 'A '(A (W) C D))  =>  (A (W) C D)
(MEMB 'C '(A (W) C D))  =>  (C D)
(MEMB 'W '(A (W) C D))  =>  NIL
(MEMB '(W) '(A (W) C D))  =>  NIL
```

(**FMEMB** *X Y*) [Function]

Faster version of `MEMB` that terminates on a null-check.

(**MEMBER** *X Y*) [Function]

Identical to `MEMB` except that it uses `EQUAL` instead of `EQ` to check membership of *X* in *Y*. Examples:

```
(MEMBER 'C '(A (W) C D)) => (C D)
(MEMBER 'W '(A (W) C D)) => NIL
(MEMBER '(W) '(A (W) C D)) => ((W) C D)
```

(**EQMEMB** *X Y*) [Function]

Returns `T` if either *X* is `EQ` to *Y*, or else *Y* is a list and *X* is an `FMEMB` of *Y*.

Substitution Functions

(**SUBST** *NEW OLD EXPR*) [Function]

Returns the result of substituting *NEW* for all occurrences of *OLD* in the expression *EXPR*. Substitution occurs whenever *OLD* is `EQUAL` to `CAR` of some subexpression of *EXPR*, or when *OLD* is atomic and `EQ` to a non-`NIL` `CDR` of some subexpression of *EXPR*. For example:

```
(SUBST 'A 'B '(C B (X . B))) => (C A (X . A))
(SUBST 'A '(B C) '((B C) D B C)) => (A D B C) not (A D . A)
```

`SUBST` returns a copy of *EXPR* with the appropriate changes. Furthermore, if *NEW* is a list, it is copied at each substitution.

(**DSUBST** *NEW OLD EXPR*) [Function]

Like `SUBST`, but it does not copy *EXPR*, but changes the list structure *EXPR* itself. Like `SUBST`, `DSUBST` substitutes with a copy of *NEW*. More efficient than `SUBST`.

(**LSUBST** *NEW OLD EXPR*) [Function]

Like `SUBST,` but *NEW* is substituted as a segment of the list *EXPR* rather than as an element. For instance,

```
(LSUBST '(A B) 'Y '(X Y Z))  =>  (X A B Z)
```

If *NEW* is not a list, `LSUBST` returns a copy of *EXPR* with all *OLD*'s deleted:

```
(LSUBST NIL 'Y '(X Y Z))  =>  (X Z)
```

(**SUBLIS** *ALST EXPR FLG*) [Function]

ALST is a list of pairs:

((*OLD*$_1$. *NEW*$_1$) (*OLD*$_2$. *NEW*$_2$) ... (*OLD*$_N$. *NEW*$_N$))

Each *OLD*$_i$ is an atom. `SUBLIS` returns the result of substituting each *NEW*$_i$ for the corresponding *OLD*$_i$ in *EXPR*, e.g.,

```
(SUBLIS '((A . X) (C . Y)) '(A B C D))  =>  (X B Y D)
```

If *FLG* = `NIL`, new structure is created only if needed, so if there are no substitutions, the value is `EQ` to *EXPR*. If *FLG* = `T`, the value is always a copy of *EXPR*.

(**DSUBLIS** *ALST EXPR FLG*) [Function]

Like `SUBLIS`, but it changes the list structure *EXPR* itself instead of copying it.

(**SUBPAIR** *OLD NEW EXPR FLG*) [Function]

Like `SUBLIS`, but elements of *NEW* are substituted for corresponding atoms of *OLD* in *EXPR*, e.g.,

```
(SUBPAIR '(A C) '(X Y) '(A B C D))  =>  (X B Y D)
```

As with `SUBLIS`, new structure is created only if needed, or if *`FLG`* `= T`, e.g., if *`FLG`* `= NIL` and there are no substitutions, the value is `EQ` to *`EXPR`*.

If *`OLD`* ends in an atom other than `NIL`, the rest of the elements on *`NEW`* are substituted for that atom. For example, if *`OLD`* `= (A B . C)` and *`NEW`* `= (U V X Y Z)`, `U` is substituted for `A`, `V` for `B`, and `(X Y Z)` for `C`. Similarly, if *`OLD`* itself is an atom (other than `NIL`), the entire list *`NEW`* is substituted for it. Examples:

```
(SUBPAIR '(A B . C) '(W X Y Z) '(C A B B Y)) => ((Y Z) W X X
Y)
```

`SUBST`, `DSUBST`, and `LSUBST` all substitute copies of the appropriate expression, whereas `SUBLIS`, and `DSUBLIS`, and `SUBPAIR` substitute the identical structure (unless *`FLG`* `= T`). For example:

```
← (SETQ FOO '(A B))
   (A B)
← (SETQ BAR '(X Y Z))
   (X Y Z)
← (DSUBLIS (LIST (CONS 'X FOO)) BAR)
   ((A B) Y Z)
← (DSUBLIS (LIST (CONS 'Y FOO)) BAR T)
   ((A B) (A B) Z)
← (EQ (CAR BAR) FOO)
   T
← (EQ (CADR BAR) FOO)
   NIL
```

Association Lists and Property Lists

It is often useful to associate a set of property names (*`NAME1`*, *`NAME2`*, etc.), with a set of property values (*`VALUE1`*, *`VALUE2`*, etc.). Two list structures commonly used to store such associations are called “property lists” and “association lists.” A list in “association list” format is a list where each element is a call whose `CAR` is a property name, and whose `CDR` is the value:

`( (`*`NAME1`*` . `*`VALUE1`*`) (`*`NAME2`*` . `*`VALUE2`*`) ...)`

A list in “property list” format is a list where the first, third, etc. elements are the property names, and the second, forth, etc. elements are the associated values:

`( `*`NAME1 VALUE1 NAME2 VALUE2`*` ...)`

Another data structure that offers some of the advantages of association lists and property lists is the hash array (see the first page of Chapter 6).

The functions below provide facilities for searching and changing lists in property list or association list format.

Note: Property lists are used in many Medley system datatypes. There are special functions that can be used to set and retrieve values from the property lists of symbols (see the Property Lists section of Chapter 2), from properties of windows (see the Window Properties section of Chapter 28), etc.

(**ASSOC** *`KEY ALST`*) [Function]

`ALST` is a list of lists. `ASSOC` returns the first sublist of *`ALST`* whose `CAR` is `EQ` to *`KEY`*. If such a list is not found, `ASSOC` returns `NIL`. Example:

```
(ASSOC 'B '((A . 1) (B . 2) (C . 3)))  =>  (B . 2)
```

(**FASSOC** *KEY ALST*) [Function]

Faster version of ASSOC that terminates on a null-check.

(**SASSOC** *KEY ALST*) [Function]

Same as ASSOC, but uses EQUAL instead of EQ when searching for *KEY*.

(**PUTASSOC** *KEY VAL ALST*) [Function]

Searches *ALST* for a sublist CAR of which is EQ to *KEY*. If one is found, the CDR is replaced (using RPLACD) with *VAL*. If no such sublist is found, (CONS *KEY VAL*) is added at the end of *ALST*. Returns *VAL*. If *ALST* is not a list, generates an error, Arg not list.

The argument order for ASSOC, PUTASSOC, etc. is different from that of LISTGET, LISTPUT, etc.

(**LISTGET** *LST PROP*) [Function]

Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If one is found, returns the next element of *LST*, otherwise NIL. Returns NIL if *LST* is not a list. Example:

```
(LISTGET '(A 1 B 2 C 3) 'B)  =>  2
(LISTGET '(A 1 B 2 C 3) 'W)  =>  NIL
```

(**LISTPUT** *LST PROP VAL*) [Function]

Searches *LST* two elements at a time, by CDDR, looking for an element EQ to *PROP*. If *PROP* is found, replaces the next element of *LST* with *VAL*. Otherwise, *PROP* and *VAL* are added to the end of *LST*. If *LST* is a list with an odd number of elements, or ends in a non-list other than NIL, *PROP* and *VAL* are added at its beginning. Returns *VAL*. If *LST* is not a list, generates an error, Arg not list.

(**LISTGET1** *LST PROP*) [Function]

Like LISTGET, but searches *LST* one CDR at a time, i.e., looks at each element. Returns the next element after *PROP*. Examples:

```
(LISTGET1 '(A 1 B 2 C 3) 'B)  =>  2
(LISTGET1 '(A 1 B 2 C 3) '1)  =>  B
(LISTGET1 '(A 1 B 2 C 3) 'W)  =>  NIL
```

(**LISTPUT1** *LST PROP VAL*) [Function]

Like LISTPUT, but searches *LST* one CDR at a time. Returns the modified *LST*. Example:

```
←(SETQ FOO '(A 1 B 2))
    (A 1 B 2)
←(LISTPUT1 FOO 'B 3)
    (A 1 B 3)
←(LISTPUT1 FOO 'C 4)
    (A 1 B 3 C 4)
←(LISTPUT1 FOO 1 'W)
    (A 1 W 3 C 4)
←FOO
```

```
(A 1 W 3 C 4)
```

If *LST* is not a list, no error is generated. However, since a non-list cannot be changed into a list, *LST* is not modified. In this case, the value of `LISTPUT1` should be saved. Example:

```
←(SETQ FOO NIL)
   NIL
←(LISTPUT1 FOO 'A 5)
   (A 5)
←FOO
   NIL
```

Sorting Lists

(**SORT** *DATA COMPAREFN*) [Function]

DATA is a list of items to be sorted using *COMPAREFN*, a predicate function of two arguments which can compare any two items on *DATA* and return `T` if the first one belongs before the second. If *COMPAREFN* is `NIL`, `ALPHORDER` is used; thus `(SORT` *DATA*`)` will alphabetize a list. If *COMPAREFN* is `T`, `CAR`'s of items that are lists are given to `ALPHORDER`, otherwise the items themselves; thus `(SORT A-LIST T)` will alphabetize an assoc list by the `CAR` of each item. `(SORT X 'ILESSP)` will sort a list of integers.

The value of `SORT` is the sorted list. The sort is destructive and uses no extra storage. The value returned is `EQ` to *DATA* but elements have been switched around. There is no safe way to interrupt `SORT`. If you abort a call to `SORT` by any means, you may loose elements from the list beeing sorted. The algorithm used by `SORT` is such that the maximum number of compares is $N * \log_2 N$, where N is `(LENGTH` *DATA*`)`.

> Note: If `(`*COMPAREFN* `A B) = (`*COMPAREFN* `B A)`, then the ordering of `A` and `B` may or may not be preserved.

For example, if `(FOO . FIE)` appears before `(FOO . FUM)` in `X`, `(SORT X T)` may or may not reverse the order of these two elements.

(**MERGE** *A B COMPAREFN*) [Function]

A and *B* are lists which have previously been sorted using `SORT` and *COMPAREFN*. Value is a destructive merging of the two lists. It does not matter which list is longer. After merging both *A* and *B* are equal to the merged list. (In fact, `(CDR` *A*`)` is `EQ` to `(CDR` *B*`)`).

(**ALPHORDER** *A B CASEARRAY*) [Function]

A predicate function of two arguments, for alphabetizing. Returns a non-`NIL` value if its arguments are in lexicographic order, i.e., if *B* does not belong before *A*. Numbers come before literal atoms, and are ordered by magnitude (using `GREATERP`). Literal atoms and strings are ordered by comparing the character codes in their print names. Thus `(ALPHORDER 23 123)` is `T`, whereas `(ALPHORDER 'A23 'A123)` is `NIL`, because the character code for the digit 2 is greater than the code for 1.

Atoms and strings are ordered before all other data types. If neither *A* nor *B* are atoms or strings, the value of ALPHORDER is always T.

If *CASEARRAY* is non-NIL, it is a casearray (see the Random Access File Operations section of Chapter 25) that the characters of *A* and *B* are translated through before being compared. Numbers are not passed through *CASEARRAY*.

Note: If either *A* or *B* is a number, the value returned in the "true" case is T. Otherwise, ALPHORDER returns either EQUAL or LESSP to discriminate the cases of *A* and *B* being equal or unequal strings/atoms.

Note: ALPHORDER does no UNPACKs, CHCONs, CONSes or NTHCHARs. It is several times faster for alphabetizing than anything that can be written using these other functions.

(**UALPHORDER** *A B*) [Function]

Defined as (ALPHORDER *A B* UPPERCASEARRAY). UPPERCASEARRAY maps every lowercase character into the corresponding uppercase character. For more information on UPPERCASEARRAY see Chapter 25.

(**MERGEINSERT** *NEW LST ONEFLG*) [Function]

LST is NIL or a list of partially sorted items. MERGEINSERT tries to find the "best" place to (destructively) insert *NEW*, e.g.,

```
(MERGEINSERT 'FIE2 '(FOO FOO1 FIE FUM)) => (FOO FOO1 FIE
   FIE2 FUM)
```

Returns *LST*. MERGEINSERT is undoable.

If *ONEFLG* = T and *NEW* is already a member of *LST*, MERGEINSERT does nothing and returns *LST*.

MERGEINSERT is used by ADDTOFILE (see the Functions for Manipulating File Command Lists section of Chapter 17) to insert the name of a new function into a list of functions. The algorithm is essentially to look for the item with the longest common leading sequence of characters with respect to *NEW*, and then merge *NEW* in starting at that point.

Other List Functions

(**REMOVE** *X L*) [Function]

Removes all top-level occurrences of *X* from list *L*, returning a copy of *L* with all elements EQUAL to *X* removed. Example:

```
(REMOVE 'A '(A B C (A) A))  =>  (B C (A))
(REMOVE '(A) '(A B C (A) A))  =>  (A B C A)
```

(**DREMOVE** *X L*) [Function]

Like REMOVE, but uses EQ instead of EQUAL, and actually modifies the list *L* when removing *X*, and thus does not use any additional storage. More efficient than REMOVE.

DREMOVE cannot *change* a list to NIL:

```
←(SETQ FOO '(A))
```

```
    (A)
←(DREMOVE 'A FOO)
    NIL
←FOO
    (A)
```

The DREMOVE above returns NIL, and does not perform any CONSes, but the value of FOO is *still* (A), because there is no way to change a list to a non-list. See NCONC.

(**REVERSE** *L*) [Function]

Reverses (and copies) the top level of a list, e.g.,

```
(REVERSE '(A B (C D)))  =>  ((C D) B A)
```

If *L* is not a list, REVERSE just returns *L*.

(**DREVERSE** *L*) [Function]

Value is the same as that of REVERSE, but DREVERSE destroys the original list *L* and thus does not use any additional storage. More efficient than REVERSE.

(**COMPARELISTS** *X Y*) [Function]

Compares the list structures *X* and *Y* and prints a description of any differences to the terminal. If *X* and *Y* are EQUAL lists, COMPARELISTS simply prints out SAME. Returns NIL.

COMPARELISTS prints a terse description of the differences between the two list structures, highlighting the items that have changed. This printout is not a complete and perfect comparison. If *X* and *Y* are radically different list structures, the printout will not be very useful. COMPARELISTS is meant to be used as a tool to help users isolate differences between similar structures.

When a single element has been changed for another, COMPARELISTS prints out items such as (*A* -> *B*), for example:

```
←(COMPARELISTS '(A B C D) '(X B E D))
    (A -> X) (C -> E)
    NIL
```

When there are more complex differences between the two lists, COMPARELISTS prints *X* and *Y*, highlighting differences and abbreviating similar elements as much as possible. "&" is used to signal a single element that is present in the same place in the two lists; "--" signals an arbitrary number of elements in one list but not in the other; "-2-," "-3-", etc. signal a sequence of two, three, etc. elements that are the same in both lists. Examples:

```
(COMPARELISTS '(A B C D) '(A D))
    (A B C --)
    (A D)
←(COMPARELISTS '(A B C D E F G H) '(A B C D X))
    (A -3- E F --)
    (A -3- X)
←(COMPARELISTS '(A B C (D E F (G) H) I) '(A B (G) C (D E F
    H) I))
    (A &     & (D -2- (G) &) &)
```

```
(A & (G) & (D -2-      &) &)
```

(**NEGATE** *X*) [Function]

For a form *X*, returns a form which computes the negation of *X*. For example:

```
(NEGATE '(MEMBER X Y)) => (NOT (MEMBER X Y))
(NEGATE '(EQ X Y)) => (NEQ X Y)
(NEGATE '(AND X (NLISTP X))) => (OR (NULL X) (LISTP X))
(NEGATE NIL) => T
```

[This page intentionally left blank]