A User’s Guide to ED110

Lyle Ramshaw

March, 1974

Revision by Forrest Howard, June, 1975

INTRODUCTION

ED110 is and in-core editor for the Harvard instructional Lisp system L110. It allows the user to modify internal pieces of list structure conveniently, and thus provides an alternative to the tedious Pretty-Print, TECO, and LOAD scenario in debugging, when it is essential to be able to insert and remove BREAKs easily. ED110 is esentially a civilized way of organizing the use of RPLACA and RPLACD This particular editor was inspired by Ben Wegbreit, and several of the major design decisions were made with his guidance. This documentation describes how I hope ED110 to perform, but I make the usual disclaimer about possible bugs. ED110 is written in LISP, and uses standard syntax for command input. Thus, the full power of the interpreter is always available for programmed editing, temporary storage, etc. The ED110 commands are LISP functions with short print strings, primarily for convenience during type-in. To avoid name conflicts with user's names, the other ED110 symbols are atoms whose print strings begin with a "*". The user is cautioned to avoid rebinding the ED110 command names, and to generally eschew the use of symbols begining with *. To gain the benefits of easy functional composition, and of a clear convention, all ED110 functions are LAMBDAs and hence take their arguments EVALed. The user is, of course, invited to define his own NLAMBDA version of any of the commands, if he desires the extra convenience. For the purposes of ED110, always think of NIL as spelled "()" rathen than "NIL", when it occurs as an element of some list. The conditions where this distinction is important are noted in the following discussions. The ED110 functions are written to tolerate the presence of non-null, non-atomic CDRs in the buffer, even though many of the AUXFNS are not that polite. Howeverr, there are no commands provided for manipulating them. Such facilities could easily be added. ED110 will also tolerate a buffer which contains dotted pairs with several fathers, i.e., buffers which include list structure sharing. It does insist, however, that the buffer be non-circular.

DEFINIONS

At any point in time, ED110 is editing a particular piece of list structure, called the BUFFER. An ENTRY of the buffer is either any element of the buffer, or any element of any element of the bufffer, etc; i.e., it is the data pointed to by the CAR of some dotted pair in the buffer. A SUBLIST of the buffer is either the entire bufffer, or a non-atomic entry of the buffer, or a null entry of the buffer; equivalently, it is the list structure associated with the print string between any two matching parenteses; recall that NIL is spelled "()". A CHUNK of the buffer is any string of zero or more elements of some sublist of the buffer. The site of active editing in the buffer is called the POINTER, and can be located between any two adjacent elements of any sublist, or before the first element of any sublist, or after the last element of any sublist, or in the middle of a null sublist, i.e. between the "(" and ")" of NIL. Equivalently, the pointer can be located between any two adjacent lexemes in the print string of the buffer. When the buffer is typed out, an asterix is inserted at the position of the pointer, to make locating the pointer easier. For example,

```
(A B (C ((D) * E)) F)
```

would be the result of typing out a buffer which contained the list:

```
(A B (C ((D)E)) F),
```

when the pointer was located between the (D) and E. The CURRENT LIST is the smallest sublist of the buffer which contains the pointer. For instance, in the above example, the current list is ((D) E).

The Sharing Guarantee

The editing functions are implemented destructively in ED110. (Convenient facilities are provided for copying when desirable.) In addition, any insertions or deletions at the beginning of the buffer are performed so that the root dotted pair of the initial buffer will be the root dotted pair of the result. Hence, if there are multiple pointers to a particular list, which is the present ED110 buffer, the editing of that list changes it from everyone's point of view. In more sophisticated language, the ED110 buffer is an R-value, not an L-value. This guarantee generates several important issues of a fairly technical nature, which are discussed below: The Atomic Buffer Issue For convenience, ED110 allows the contents of the buffer to temporarily, but never permanently, degenerate to an atom, usually NIL. In view of the sharing guarantee, this degeneration, is, of course, "impossible"; i.e., it cannot be actually implemented with reasonable cost. Hence, the editor merely simulates the condition of an atomic buffer during type-out commands, selection commands, etc. A delete command which would resullt in an atomic buffer merely places the editor in this simulating state, and is performed as if the buffer really had been atomic.

Thus, the buffer never degenerates to an atom. However, it will behave as if atomic; only external pointers to the buffer will notice the difference, and they will see the buffer's last non-atomic contents. Therefore: the above Sharing Guarantee holds only between two points where the buffer is non-atomic. In particular, it is impossible to change a non-null buffer to NIL, or vice versa, using ED110 commands. The Sharing Algorithms Consider first the issue of inserting a list into the current list at the location of the pointer. Note that, if the pointer is not at the beginning of the buffer, i.e, immediatly after the first "(", it is possible to effect this insertion by CONC'ing the insertion with the rest of the current list, and then RPLAC'ing a pointer to the root of the insertion into the preceding dotted pair, in the appropriate half. This is the algorithm used by ED110, and the behavior in this case will be taken as the standard for other, more questionable cases. If the pointer is located at the beginning of the buffer, however, it is necessary, in keeping with the Sharing Guarantee, to interchange the roles of the root dotted pair of the insertion and the root dotted pair of the initial buffer, and perform CONC. If the insertion and the initial buffer do not share some of the same dotted pairs, the above method produces a standard result, and is used in ED110. However, if there is sharing going on, funny things can happen. For example, consider the implementation of the comands (B)(I (XA)). If the above algorithm were applied, an unnecessary and non-standard circularity would result, due to the fact that the CAR of the insertion EQ's the initial buffer's root. When the roles of the roots are switched, this sharing becomes circularity. To allow commands of the above ilk to operate successfully, ED110 climbs the insertion and handles specially all pointers to the root dotted pair of the initial buffer. This is expensive, but convenient, and consistent. Thus, the above commands will produce a standard result, a buffer with sharing, but without circularity. Of course, the situation has a certain symmetry. A non-standard result result would also insue if the initial buffer contained a pointer to the root dotted pair of the insertion. However, ED110 makes no special effort to handle this case, since both the standard and the non-standard outcomes are circular, and hence, illegal in ED110. If desirable, this type of special handling could be easily added. It was left out only to avoid the resulting inefficiency. Note that precisely similar issues arise in the implementation of the (XD) command; they are handled in the analogous fashion.

The Commands of ED110

Throughout the following command descriptions, small letters standing alone will be used to reepresent arbitrary lists EVALing to an object on the given kind. The letters i, j, and k will stand for integers; s represents an arbitrary list; m represents an atom. Note also that all atom names are shown as upper case for lexographical clarity. However, like all system defined L110 atoms, they are lower case (normal type in mode). Examples of commands' actions will be given in the following format: each will take two successive lines. The first line's left half will be the state of the buffer before the command's execution; its right half will be the text of the command. The next line will contain the resulting state of the buffer and the value returned by the command in its left and right parts, respectively.

The Pointer Transportation Commands

(W 1), Walk

This command walks the pointer past k elements of the current list, i.e., makes the pointer follow k CDRs. If k<0, the pointer is walked backwards. (W 0) is a no-op. An attempt to walk the pointer off an end of the current list walks the pointer as far as possible in the desired direction. Thus, (W 1000000000) will always bring the pointer to the end of the current list. Walk returns as value the number of successful steps walked. If the argument is defaulted or non-numeric, (W 1) is assumed. Examples:

```
(A * (A B) C D)          (W 2)
(A (A B) C * D)          2

(A B C * D)              (W -5)
(* A B C D)              -3
```

(J k), Jump This command jumps the pointer down k levels of list structure, i.e., makes the pointer follow k CARs. In the printout, this corresponds too moving the pointer past k left parentheses. Note that the pointer must be positioned immediatly before a left paren in order to make a (J 1) command successful. If k<0, the pointer is pulled out of levels of list structure. If, at any point in this process, the pointer is not immediatly after a left paren, it is walked backwards in the current list until it is. (J 0) is a no-op. Attempts to jump to far into or out of the buffer will jump the pointer as far as possible. Thus, a (J -10000000) will bring the pointer back to the beginning of the buffer. Jump returns as value the number of successful levels jumped. If the argument is defaulted or non-numeric, (J 1) is assumed. Examples:

```
(A ((A B (* (((C D) E))))))         (J 2)
(A ((A B (((* (C D) E))))))         2

(A * ((A B ((C D) E))))             (J 3)
(A ((* A B ((C D) E)))              2

(((((B) C) ((B (* (E F)))))))       (J -4)
((* (A ((B) C)((D ((E F)))))))      -4
```

(C k), Climb This is a convenience command which, by calling walk and jump, moves the pointer over k lexemes of the print string of the buffer. If k>0, the climbing is to the right; if k<0, to the left. (C 0) is a no-op. An attempt to climb of the far end of the buffer will move the pointer as far as possible in the desired direction. Climb returns as value the number of lexemes successfully climbed. If k is defaulted or non-numeric, (C 1) is assumed. Remember that, for the purposes of ED110, NIL is always spelled "()". Hence, a NIL in the buffer counts as two lexemes to be climbed over, not one. Examples:

```
((A Nil (((C D) E * F))))           (C -3)
((A NIL (((C * D) E F))))           -3
((* A NIL ((C D) E)))               (C 2)
((A (*) ((C D) E)))                 2

((A (*) ((C D) E)))                 (C -5)
(* (A NIL ((C D) E)))               -3
```

The Insertion Commands

(I s) Insert This command takes one argument, an arbitrary piece of list structure, and inserts it as an element into the current list, at the position of the pointer; the pointer is left after the inserted element. Insert uses one new dotted pair to hold the extra element of the current list. The value returned is always T. Example:

```
(A B (C) * D E)                (I '(F H))
(A B (C) (F H) * D E)          T
```

(IC s) Insert Copied (LAMBDA(s)(I (COPY s]

(IB s) Insert leaving Pointer Before This command performs the same function as Insert, but leaves the pointer before the new element of the current list. It returns T.
Examples:

```
(* A B C)                      (IB 's)
(* S A B C)                    T
```

(IBC s) Insert leaving pointer before, Copied. (LAMBDA (s)(IB (COPY s]

(IL s) Insert List This command takes one argument, an arbitrary list, s, and inserts the elements of s into the current list at the position of the pointer. It leaves the pointer after the newly inserted elements. The current list's length will increase by exactly the (LENGTH s). No new Dotted Pairs are used; the value returned is T. Note the difference between (I s) and (IL s) as follows: Examples:

```
(A B (C) * D E)                (IL '(F H))
(A B (C) F H * E)              T
```

(ILC s) Insert List Copied
(ILB s) Insert List leaving pointer Before
(ILBC s) Insert List leaving pointer Before, Copied.

Selection, Deletion, and Typing Commands.

The ED110 commands which deal with chunks of the buffer take three integer arguments, (? i j k); the chunk of the list structure refered to is that from where the pointer would be after execution of a (J k)(W i), to where it would be after the execution of a (J k)(W j). If these arguments are ommitted or non-numeric, they are defaulted to i=1, j=0, and k=0. For examples of the use of this argument structure, which might seem obscure, but is, in fact, very convenient, see the following paragraph.

(X i j k) X-select This command takes the three chunk-selecting arguments, and returns as value the selected chunk. The top level dotted pairs in the returned value are all new, since the selected chunk is not necessarily a complete sublist of the buffer. The X-select command neither modifies the buffer nor moves the pointer. Examples:

```
(A B C (D E) * F G)          (X)
(A B C (D E) * F G)          (F)

(A B C (D E) * F G)          (X 2)
(A B C (D E) * F G)          (F G)

(A B C (D E) * F G)          (X -2)
(A B C (D E) * F G)          (C (D E))

(A B C (D E) * F G)          (X -1 5)
(A B C (D E) * F G)          ((D E) F G)

(A B C (D E) * F G)          (X -17 -2)
(A B C (D E) * F G)          (A B)


(A (* (B C D) E (F)) G)      (X 0 2 1)
(A (* (B C D) E (F)) G)      (B C)

(A (* (B C D) E (F)) G)      (X 0 -3 -1)
(A (* (B C D) E (F)) G)      (A)
```

(XC i j k) X-select, Copied (LAMBDA(i j k)(COPY(X i j k]

(XA) X-select All This command takes no arguments, and returns as value the present contents of the buffer.

(XAC) X-select All, Copied

(T i j k) Type

This command takes the three chunk-selecting arguments, and types out the selected chunk of the buffer using PPL, a pretty-printing routine described below. The output goes to POPORT. If the selected chunk contains the present position of the pointer, it prints the present value of the atom *EPTR at that place in the type-out. *EDTR is initially set to "*". Type never modifies the buffer, or moves the pointer; it returns the value T. Example:

```
(A (B * C) D)           (T -1 1 -1)
(A (B * C) D)           "(A (B*C))";T

(A B C D E * F G)       (T 1)
(A B C D E * F G)       "(* F)";T

(A B C D E * F G)       (T -2 -3)
(A B C D E * F G)       "(C)"; T
```

(TA) Type All This command takes no arguments, and types the contents of the buffer, with the pointer indicated as in Type. The buffer contents and pointer position are unchanged; the value returned is T

(D i j k) Delete This command takes the three chunk-selecting arguments, and deletes the selected chunk of the buffer, leaving the pointer where the chunk used to be. (See the note on the Atomic Buffer Issue, page 4). Examples:

```
(A B (* C D E ) F (G)) H)          (D 2)
(A B (* (G)) H)                    T

(A B (* (C D E) F (G)) H)          (D 0 1 -1)
(A B * H)                          T

(A B (* (C D E) F (G)) H)          (D -2 2 1)
(A B ((* E) F (G)) H)              T
```

(DA) Delete All This command deletes the entire buffer in the nse described on page 4, and returns T.

(XD i j k) X-select and Delete This command takes the three chunk-selecting arguments, and deletes the selected chunk from the buffer, leaving the pointer where the chunk used to be. It returns the deleted chunk as a value. No new dtprs are used. Examples:

```
(A (B)  C * D)          (XD -2 2)
(A *)                   ((B) C D)

(A * (B) C D)           (XD 0 1 1)
(A (*) C D)             (B)
```

(XDC i j k) X-select and Delete Copied (LAMBDA(i j k)(COPY(XD i j k]

(XDA) X-select and Delete All

(XDAC) X-select and Delete All Copied

Pattern-Matching and Search Commands

The pattern-matching commands are predicates which examine the list structure of the buffer immediatelyy to the right of the pointer, and test if it matches the supplied pattern, s. The "matching" is in the sense of a special version of EQUAL, discussed below. These commands do not move the pointer, or alter the buffer.

(V s) equiValent to entry This command examines the element of the current list immediately after the pointer, and tests if it matches s. If so, T is returned; else, NIL. If the pointer is immediately before a ")", it returns NIL. Examples:

```
(A B * (C D) E F)            (V '(C D))
(A B * (C D) E F)            T

(A B * (C) D)                (V 'C)
(A B * (C) D)                NIL
```

(VF s) equiValent to chunk, First This command examines the next (LENGTH s) elements of the current list, and tests if they match the elements of s. It also insists that the pointer be located at the beginning of the current list, i.e., that the matched chunk be an initial segment of the current list. Note that (VF NIL) is a predicate which returns T iff the pointer is immediately after a left paren. Examples:

```
(* A B C D)          (VF '(A B))
(* A B C D)          T

(A B * C D)          (VF '(C D))
(A B * C D)          NIL
```

(VL s) equiValent to chunk, Last This command examines the next (LENGTH s) elements of the current list, and tests if they match the elements of s. It also insists that the last element of the current list be the one to atch the last element of s, i.e., that the matched chunk be the terminal segment of the current list. Note that (VL NIL) is a predicate which tests if the pointer is immediatelyy before a right paren. Example:

```
(A B * C D)          (VL '(C))
(A B * C D)          NIL

(A B * C D)          (VL '(C D))
(A B * C D)          T
```

(VM s) equiValent to chunk, Middle This command looks at the next (LENGTH s) elements of the current list, and tests if they match the elements of s. It makes no extra demands, and thus will match a chunk in any location in a sublist. Note that (VM NIL) always returns T. The matching routine used by these commands does a prefix-walk-order climb over the nodes of the pattern and the buffer, as does EQUAL. However, in order to facilitate more sophisticated pattern-replacement, the routine checks any atomic element of the pattern, to see if it is a member of the top-level list *DUMMIES. If so, it APPLY*s that atom to the corresponding structure in the buffer, and returns the result of this function call. Hence, arbitrary new dummies can be added. For example, after
(SETQ *DUMMIES (CONS 'ZORT *DUMMIES)) and
(DEF ZORT (LAMBDA(L)(ATOM L]
are executed, (V 'ZORT) will return T iff the entry following

the pointer is atomic. Similarly, a (S 'BT) will leave the pointer after the next atom in the print string of the buffer (see S below). It is often desirable to allow different calls on such dummies as functions to interact. For example consider the case of

```
(SETQ *DUMMIES (CONS 'SAME *DUMMIES))
(DEF SAME
        (LAMBDA(L)
         (COND(SAME (EQ L (CAR SAME))
             (T   (SETQ SAME (CONS L NIL)) T))))
(VM '(SAME SAME)),
```

which will test if the next two elements of the current list are identical. Note however, that some method of flushing the effects of old calls on these functions is necessary between succesive calls on the V routine. This is provided by the function,*FLUSH, of no arguments, which is initially defined as a no-op. It is called immediately before any check for a match, and should be changed to a routine to accomplish this reinitialization. In the above example, the proper code is (Def *FLUSH (LAMBDA()(SETQ SAME NIL. The search commands also obey the *FLUSH convention; they scan the buffer in left-to-right or right-to-left print order, checking for matches to a supplied pattern, and positioning the pointer accordingly.

(S s k) Search for entry If k>0 this command begins searching for a match to s in the sense of (V s) at the present position of the pointer. It climbs to the right, and positions the pointer after the k'th match for s, if such exists. If so, it returns T. If not, it returns NIL, and leaves the pointer at the beginning of the buffer. If k<0, this command begins by doing a (C -1). It then begins looking for matches of s in the sense of (V s), and positions the pointer after the -k'th match, if such exists, returning T. Else, it returns NIL, and leaves the pointer at the beginning of the buffer. If k=0, the command is a no-op. Note that this command only searches for entries of the buffer which match; in particular, it does not check the whole buffer against the supplied pattern. Examples:

```
(A * B D D)           (S 'C)
(A B C * D)           T

(A * B C D)           (S 'A)
(* A B C D)           NIL

(A (B C D) E *)       (S 'B -1)
(A (B * C D) E)       T

(* A B)               (S '(A B))
(* A B)               NIL
```

(SB s k) Search for entry, leaving pointer Before This command finds the same match as the above, but leaves the pointer immediately before it, in each case.

(SF s k) Search for chunk First This command behaves like S, but looks for matches in the sense of (VF s). The pointer is left after the last element of the final matching chunk.

(SFB s k) Search for chunk First, leaving pointer Before

(SL s k) Search for chunk Last This command behaves like the above S, except that the matching is done in the sense of (VL s). The pointer is left after the last element of the final matching chunk.

(SLB s k) Search for chunk Last, leaving pointer Before
(SM s k) Search for chunk middle This command behaves like S, except
that the matching is done in the sense of (VM s). The pointer
is left after the last element of the final matching chunk.
(SMB s k) Search for chunk Middle, leaving pointer Before.

Initialization Commands

These commands initialise the editor by setting the contents of the buffer. After initialization, the pointer is always put at the beginning of the buffer, i.e., before its first dotted pair. Note that, due to the Sharing Guarantee, ED110 does not involve an "edit mode", and thus, no termination of an editing job is necessary unless the user desires to free the ED110 buffer for garbage collection, or to use the editor in some other buffer. The later commands in this group deal only with the issue of generating backup copies of functions being edited, in case of editing disasters. They allow the user to choose whether to edit the active or reserve copy of the function definition. The first demands more care in some special cases, but allows the user easier testing of the edited version.

(E s) Edit This command initializes ED110 to the list structure s. If s is not a dotted pair, Edit breaks; the value returned is T.

(EC s) Edit Copied (LAMBDA(s)(E (COPY s]

(E N) Edit Nothing This command takes no arguments, and initializes the editor for creation of a new piece of list structuure. The buffer will appear to be (*) after the execution of this command. (EN) is equivalent to (E (CONS))(DA).

(ENC) Edit Nothing Copied This command is equivalent to EN, and is included for symmetry.

(EF m) Edit Function This command takes one atomic argument, i.e., one argument which much eval to an atom, and initializes ED110 to the present function definition definition of that atom, if it exists. Otherwise, it prepares for the creation of a new function binding, by performing the equivalent of (E (CONS)(PUTD m (XA))(DA). The value returned is m; if the function binding is binary#code, EF breaks.

(EFC) Edit Function Copied This command takes one atomic argument, and returns it. It initializes the editor to a copy of the present function binding of m. If there is no such binding, it is equivalent to (EN). If the function binding is binary#code, EFC breaks.

(GR m) Generate Reserve This command takes one atomic argument, and returns it. It PUTs the present function binding of m on the property list of m under the indicator 'RESERVE. If no such binding exists, GR breaks.

(GRC m) Generate Reserve Copied Places a copy of the function binding under the indicator 'RESERVE on m's property list.

(RR m) Repace Reserve This command takes one atomic argument, and returns it. If no reserve copy of m exists on m's property list, RR breaks. Otherwise, it changes m's function binding to be that reserve copy.

(RRC m) Replace Reserve Copied This command takes one atomic argument, and returns it. If no reserve copy of m exists on m's property list, RRC breaks. Otherwise, it changes m's function definition to be a copy of that reserve copy.

(ER m) Edit Reserve This command takes one atomic argument, and returns it. If no reserve copy of m exists, ER breaks. Otherwise, it initializes the editor to that reserve copy.

(ERC m) Edit Reserve Copied

Thus, the command sequence (EF (GRC m)) generates a copy of the present definition of m, and initializes the editor to the active definition. An editing disaster could then be remedied by a (RRC m). If the user desired to edit the non-active version of the function, he should instead use the sequence (ER (GRC m)): the disaster recovery is then to repeat the above. However, at the conclusion of the edit, the user must (RR m) to install the edited reserve copy as the active version of the function.

State-Handling Commands

The state of ED110 is merely the present buffer and the present position of the pointer within the buffer. Although the values of several top-level atoms, such as *EDTR and *DUMMIES, affect the actions of the editor, they are not includedd in the state descriptors handled by the following commands.

(Y) copY This command alters the state of the editor so that it is at precisely the same point in editing a copy of the original buffer. Y returns T.

(YS s)copY State This command takes one argument,and editor state descriptor, and constructs a descriptor for the state that the editor would be in after a (rS s)(Y). This new state descriptor is the returned value. The state of the editor is not affected.

(GS) Generate State This command takes no arguments, and returns as value an editor state descriptor for the present state of the editor.

(GSC)Generate State Copied (LAMBDA()(YS (GS]

(RS s) Replace State This command takes one argument, an editor state descriptor, and restores the editor to be in that state. RS returns T. If the argument is not in proper form, RS breaks.

(RSC s) Replace State Copied This comand takes one argument, and editor state descriptor, and performs a (RS s)(Y). The argument descriptor is unaffected, and the editor assumes the condition of editing a copy of s's buffer, with the pointer in the corresponding place.

Parenthesis-Handling Commands

(P i j k) Parenthesize

This command takes the three chunk-selecting arguments, and inserts parens around the specified chunk. One new dotted pair is used. The pointer is left after the newly inserted right paren. P returns T.

(PB i j k) Parenthesize, leaving pointer Before.

(PA) Parenthesize All This command takes no arguments, and inserts parentheses around the entire buffer. The pointer is left after the new right paren; T is returned.

(PAB) Parenthesize All, leaving pointer Before

(K) Kill Parenthesize If the pointer is positioned immediately before a left paren, (K) deletes that paren and its matching right, and leaves the pointer where the right paren was. If so, the value returned is T. Otherwise, (k) does nothing and returns NIL.

(KB) Kill parentheses, leaving pointer before.

(M x y) Move This command is a nice adjusting command for the many occasions where things that are typed in end up in the wrong place. It works as follows: Package up the lexemes between where the pointer is now, and where it would be after a (C y). Consider this new item as a super-pointer, and walk it past x elements of the remaining structure. Then, leave the real pointer on the same side of the super-pointer as it was at the beginning. If either argument is omitted or non-numeric, it is defaulted to 1. If either argument is 0, the command is a no-op. The value returned is always T. Note that the super pointer can contain parentheses as well as atoms. Examples:

```
(A (B C *) D E)             (M)
(A (B C D *) E)             T

(A B ((* C D )))            (M 1 -1)
(A B (C (* D )))            T

(A B (C (* D)))             (M -5 -3)
((C (* A B D)))             T
```

Pretty-Printing Commands

(PPL s) Pretty Print List

This takes one argument, and calls on PP's subroutines to print it to POPORT nicely formated. The argument is taken EVALED. The value of PPL is NIl.

(PPV x) Pretty Print Value This is a NLAMBDA, which does the same thing that PP does, except for top-level bindings. If x is atomic, it is evaluated to produce a list of atoms; otherwise, x is used directly as the list of atoms. The values of these atoms are pretty-printed to POPORT in the format:
(SETQ <atom> (QUOTE <value>))

(PPF in out funlist vallist) Pretty-Print File This command is a LAMBDA analogous to ECL's UNPARSF. The file in is opened for input, and the file out is opened for output. Then, forms are successively parsed from in, and pretty-printed to out. If in is NIL, no input file is used. If out is NIL, output is to the teletype. The third argument is an update list for functional bindings. While in is being read, any statement of the form (DEF <atom> <>) for any atom in the list funlist, will result in the present function binding to that atom being pretty printed to out, rather that the binding given in the input file. If any elements of funlist are not DEFed in in, they are pretty-printed to out at the end of the file. The fourth argument, vallist, is an equivalent update list for top-level bindings of atoms, with the same conventions vis-a-vis SETQs in the input file.

Convenience Commands

(Z) Z-see This command is short for (T 0 1 -1), and thus types the current list (surrounded with one extra pair of parens).

(A p q) Alter This command, similar to a TECO FS, does a (S p); if this fails, it returns NIL. If the search succeds, (D -1)(I q) is performed, and T is returned.

(AA p q) Alter All Like a TECO <FSp$q$;>$$. The command loops to perform the above alteration throughout the buffer. The pointer is left at the beginning of the buffer, and T is returned.

(L p l) Loop This command is for use in programmed editing with ED110. The first argument is interpreted as and editing program, the second as a list of function names. L iterates through l, initializing the editor to edit these functions, and then EVALing p.