Berkeley
Lisp [illegible]

PRELIMINARY GUIDE TO THE LISP EDITOR

P. Deutsch

University of California, Berkeley

Document No. W-21

Issued April 18, 1967

Contract No. SD-185
Office of Secretary of Defense
Advanced Reserach Projects Agency
Washington 25, D. C.

W-21
April 18, 1967

An On-Line LISP Editor

I. Motivation

LISP 1.5 (and local variants thereof) is being used increasingly as an interactive language. Adaptations of the original LISP language are available to users at teletype consoles in the Q-32, PDP-1 (dedicated system), and SDS 940 time-sharing systems [1,2,3]. In the course of composing and running LISP programs in this kind of environment, certain needs arise naturally for utility functions within the LISP system that are not necessary within batch processing systems such as the original LISP 1.5 for the IBM 7090 [4].

The principal use of interaction in LISP, as in most other languages originally designed for operation on batch process machines, is to greatly reduce the time required for program debugging. Thus two new classes of software services are required: debugging aids, and tools for rapid modification of already existing LISP programs within the LISP system. Packages of the former kind exist in all three systems mentioned above and will not be discussed further here (they are originally related to the TRACE facility of 7090 LISP). However, little has been written about editing facilities within on-line LISP systems.

The editor described here is implemented within the PDP-1 and SDS 940 time-sharing LISP sytems, but can be used with minor changes within any LISP system which includes the capabilities of LISP 1.5 It was begun by the author in 1965 and later extended by Bobrow and Teitelman at BBN. The object has not been to produce an especially elegant editor (although the command language is simple and the implementation fairly straightforward). In fact, this editor is considered in the nature of a stopgap until a much more elegant editor based on FLIP [5], a very general format-oriented language embedded in LISP, has been completed. However, even the present crude capabilities can cut the time required to make changes in LISP functions dramatically compared to the time required to type the function in again or even perform the emendation with SUBST.

II. Editor language structure

Let us take a concrete example of a list (not necessarily a function definition) to be edited. Suppose we are editing the following incorrect definition of the APPEND function:

```
    (LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR)
          (APPEND (CDR X Y)))))).
```

At any given moment, the editor's attention is confined to a single list (generally a subcomponent of the original list being edited), which it will print when given the command P. To avoid printing of confusing detail, sublists of sublists will be printed simply as &. Thus:

```
    *P
     (LAMBDA (X) Y (COND & &)).
```

Only the list on which attention is currently focused may be changed. Commands thus fall naturally into four classes: moving around the list structure; making changes in the current list; printing parts of the list being edited; and entering and leaving the editor.

Many commands use the convention that an integer designates a sublist of the current lists. For example, if an integer alone is typed, attention is focused on the designated sublist of the current list.

Thus:

```
    *2
    *P
     (X)
```

The converse command is the number ∅, which causes the current list to revert to its former state. For example, starting again with the list at the beginning of the section:

```
    *3 P
     Y
    *∅ P
     (LAMBDA (X) Y (COND & &)).
```

Note the use of several commands on a single line. This is possible in the three time-shared systems mentioned earlier but it may not be in others.

In the remaining examples, unless mentioned specifically, it is assumed that the state of the edit is that which existed at the end of the previous example. As above, lines typed by the user are prefixed with an asterisk.

III. Attention commands

The two fundamental commands for moving around the structure have already been mentioned: a positive integer _n_, to examine the _n_th sublist, and 0, to revert to the superlist. If _n_ is a positive integer, then -_n_ examines the _n_th sublist of the current list starting from the end and counting backwards, i.e. -1 examines the last sublist of the current list.

A more drastic command is ↑, which clears the editor's memory of descent through the structure and reestablishes the top level of the entire list structure being edited as current. Thus:

```
*4 2 1↑ P
 (LAMBDA (X) Y (COND & &)).
```

A command similar to n is (NTH n) which caused the list starting with the nth sublist of the current list to become current. Thus:

```
*(NTH 3)
*P
 (Y (COND & &)).
*ϕ P
 (LAMBDA (X) Y (COND & &)).
```

The command (F e), where e is any S-expression, searches for an instance of e in the current list, and then acts like NTH, so that for example:

```
*(F Y)
*P
 (Y (COND & &)).
```

A more thorough (and time-consuming) search is provided by (F e T) which searches through the entire structure. Thus:

```
* ↑(F Z T)
*P
 (Z)
*ϕ P
 ((NUL X) Z)
*ϕ P
(COND (& Z) (T &))
*ϕ P
 (LAMBDA (X) Y (COND & &)).
```

The argument e of the F commands need not be a literal S-expression. The symbol & will match any element of a list; the symbol -- as the last element of a list to be searched for will match any list. Thus:

```
* ↑ (F (NUL &) T)
*P
 (NUL X)
* ↑ (F (CDR --) T)
*P
 (CDR X Y)
* ↑ (F (CDR &) T)
 ?
```

The question mark which followed the last command is the editor's all-purpose error comment: it simply means something was wrong with the last command. The commands are simple enough that it is rarely difficult to ascertain the nature of the error. A problem may arise if several commands were stacked on a single line, since no indication is given of which one caused the error: in this case the state of the edit can always be discovered by using P.

One more facility is available for changing the attention of the editor. At any stage in the edit, a mark can be made and later returned to. The commands are MARK, which marks the current state for future reference; ← , which returns to the last mark without destroying it; and ← , which returns to the last mark and forgets it. For example:

```
* ↑ 4 2 P
((NUL X) Z)
```

```
*MARK ↑ (F CONS T)
*P
 (CONS (CAR) (APPEND &))
* ↑ P
 ((NUL X) Z)
*  ← P
 ((NUL X) Z)
* ← P
 ?
```

This last example demonstrates another facet of the error recovery mechanism: to avoid further confusion when an error occurs, all commands on the line beyond the one which caused the error are forgotten.

IV. Modification commands

Just as most general text editors contain INSERT, REPLACE, and APPEND commands, the LISP editor provides facilities for these three basic operations. To insert the S-expressions $\underline{e}_1 \ldots \underline{e}_m$ before sublist $\underline{n}$ of the current list, one simply gives the command $(-\underline{n}\ \underline{e}_1 \ldots \underline{e}_m)$, thus:

```
* ↑ (F CAR T)
*P
 (CAR)
*(-1 CRR)
 P
 (CRR CAR).
```

To replace the $\underline{n}$th sublist with $\underline{e}_1 \ldots \underline{e}_m$, one gives the command $(\underline{n}\ \underline{e}_1 \ldots \underline{e}_m)$, for example:

```
*↑(F NUL T)
*P
 (NUL X)
*(1 NULL IS)
*P
 (NULL IS X).
```

And to append at the end of the current list, one writes (N $\underline{e}_1 \ldots \underline{e}$), thus:

```
*(N THIS LIST)
*P
 (NULL IS X THIS LIST).
```

Deletions may be accomplished by using the replace operation with no new S-expressions specified: to restore the list we have just created to the state in which we presumably want it, we can say:

```
*(5)
*(4)
*(2)
*P
 (NULL X).
```

Deletions should generally be made from back to front, since otherwise the indices of later sublists will change as earlier ones are deleted, e.g. the above sequence of commands given in front to back order would have been

```
*(2)
*(3)
*(3).
```

A more rarely useful, but occasionally convenient, facility is provided by (I n̲ e̲), which is equivalent to (n̲ e̲^{1}) where e^1=eval[e]. For example, if the current value of J is (A B C), then we have the following result:

```
*↑4 2 P
 ((NUL X) Z)
*(I 1 J)
*P
 ((A B C) Z).
```

V. Structure changing commands

The commands presented in the last section do not allow convenient alteration of the list structure itself, as opposed to components thereof. Consider, for example, the list (A B (C D E) F G). We can remove the parenthesis around (C D E), which is the third sublist, by (LO 3) (this stands for take Left paren Out). This produces the list (A B C D E F G). Conversely, if we want to take the partial list beginning at B and subordinate it one level, making (A (B (C D E) F G)), we can say (LI 2), i.e. put a Left parenthesis in before sublist 2 (and a matching right parenthesis at the end of the list).

Two other operations of this sort are also possible. If we wanted to bring only the D and E up to the level of the A B F G, and leave (C) as a sublist, we can use (RI 1 3), namely move the Right paren on the end of sublist 3 In to after sublist 1 (of sublist 3). This will produce (A B (C) D E F G). A related operation is (RO 3), which means move the Right parenthesis of sublist 3 Out to the end of the list, producing (A B (C D E F G)). Finally, if it is desired to move a right parenthesis only partway out, for example to produce (A B (C D E F) G), this can be accomplished by (RO 3) followed by (RI 4 3).

VI. Printing commands

We have already encountered the command P, which prints the current list showing only one level of nesting. To print a selected sublist in the same way without changing the state of the edit, (P _n_) is used: for example,

```
* ↑ P
 (LAMBDA (X) Y (COND & &))
*(P 2)
 (X).
```

Furthermore, one may examine the _n_th sublist (or, if _n_=0, the current list) to _m_ levels of nesting by using (P _n_ _m_). The convention is that _m_=3 yields the usual format: several illustrations are given below.

```
*(P Ø 1)
 &
*(P Ø 2)
 (LAMBDA & Y &)
*(P Ø 3)
(LAMBDA (X) Y (COND & &))
*(P 4 2)
 (COND & &)
*(P 4 4)
 (COND ((NUL X) Z) (T (CONS & &))).
```

Another command which is available for examining the environment during editing is (E _e_), which simply prints the value of _e_ without disturbing the state of the edit. This is done under ERRORSET, so that one can actually try to run the function which one is editing. It should be mentioned that changes are made as soon as they are typed in, so that the state of the definition of a function (which is what is usually being edited) is always exactly what one expects.

VII. Using the editor

As presently interfaced to the outside world, the editor consists of a basic function for editing S-expressions, EDITE, and three special functions for editing values, definitions, and property lists, respectively EDITV, EDITF, and EDITP. Thus,

```
*EDITF(APPEND)
 EDIT
```

would be used to begin the edit which has been used as the example. When editing is complete, NIL will cause EDITE to exit with the edited list as value. The three interface functions all return as value the atom being edited. A complete example, starting with the erroneous definition given at the beginning of section 2 and ending with the correct definition of APPEND, is given below.

```
*EDITF(APPEND)
 EDIT
*(P ∅ 1∅∅)
 (LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND (CDR X Y))))))
*(3)
*(2 (X Y))
*P
```

```
 (LAMBDA (X Y) (COND & &))
*3 2 P
 ((NUL X) Z)
*1 (1 NULL)
*ø (2 Y)
*P
 ((NULL X) Y)
* ↑ (F CAR T)
*(N X)
* ↑ (F CONS T)
*3 (RI 2 2)
*P
 (APPEND (CDR X) Y)
* ↑ (P ø 1øø)
 (LAMBDA (X Y) (COND ((NULL X) Y) (T (CONS (CAR X) (APPEND
 (CDR X) Y)))))
*NIL
 APPEND.
```

In all fairness, it should be admitted that in this particular instance it probably would have been faster to type the function in again. However, LISP functions are typically three times as big as APPEND and have only one or two errors. It has been found, after over a year of use at BBN and Berkeley, that the editor just described does materially decrease the amount of time required to produce working LISP programs.

VII. Summary

A simple LISP editor has been found to be of substantial value in an interactive environment. It does not require more than an hour's training to learn to use enough of the editor for its use to be profitable. With the aid of the listings in the appendix to this paper, a competent LISP programmer should be able to adapt it for any LISP 1.5-based LISP system in less than a week.

It is the feeling of the author that there is a point of investment in support software (such as editors and debuggers) beyond which further work within the same executive system (LISP 1.5 in this case) does not pay sufficiently to justify the expenditure of effort. The FLIP editor will be worthwhile because it will have required relatively little effort to implement <u>within the FLIP executive</u> (FLIP itself is a gigantic piece of work). Large amounts of further work on the simple editor described here do not seem to be justified.

References

1. Weissman, Clark, A Self-Tutor for Q-32 LISP 1.5, System Development Corporation technical memorandum TM-2337/010/00 (June 1965).

2. Bobrow, Daniel, et al., The BBN-LISP System.

3. Deutsch, L. Peter, Reference Manual 930 LISP, Project GENIE (ARPA) document R-9, University of California at Berkeley (in preparation).

4. McCarthy, John, et al., LISP 1.5 Programmer's Manual.