AN INTERACTIVE DEBUGGING PACKAGE FOR LISP/MTS

by

PAUL FRIEDMAN
B. S. University of California Irvine

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENT FOR THE DEGREE OF
MAJOR OF SCIENCE

in the Department
of
COMPUTER SCIENCE

We accept this thesis as conforming to the
required standard.

THE UNIVERSITY OF BRITISH COLUMBIA
10-03-73
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the Head of my Department or by his representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver 8, Canada

Date October 5, 1973
ABSTRACT

Most large artificial intelligence programs are written in LISP. These systems are generally so complex, that at any given point, the programmer himself cannot tell what the system is doing. To work with these systems, the programmer must be able to take an active role in the program's execution. This means, he must be able to monitor the program's evaluation, watch for incorrect data generation, and be notified when program errors occur. This requires a truly interactive language.

This thesis presents a LISP program which converts LISP to an interactive language. The model for this system is the BBN-LISP programming system. Included here is a LISP editor and BREAK and ERROR packages. Through function calls and commands, the user is put in full control over the execution of his program and can correct errors while in LISP.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDITOR</td>
<td>2</td>
</tr>
<tr>
<td>COMMANDS THAT CHANGE THE EDITOR'S ATTENTION</td>
<td>3</td>
</tr>
<tr>
<td>-n, -n, 0</td>
<td>3</td>
</tr>
<tr>
<td>UP</td>
<td>5</td>
</tr>
<tr>
<td>!</td>
<td>6</td>
</tr>
<tr>
<td>!0</td>
<td>7</td>
</tr>
<tr>
<td>NX, BK</td>
<td>8</td>
</tr>
<tr>
<td>!NX</td>
<td>9</td>
</tr>
<tr>
<td>F</td>
<td>10</td>
</tr>
<tr>
<td>COMMANDS WHICH CHANGE THE EDITED STRUCTURE</td>
<td>11</td>
</tr>
<tr>
<td>INSERT, I</td>
<td>12</td>
</tr>
<tr>
<td>DELETE, D</td>
<td>13</td>
</tr>
<tr>
<td>EMBED, EM</td>
<td>14</td>
</tr>
<tr>
<td>EXTRACT, EX</td>
<td>15</td>
</tr>
<tr>
<td>COMMANDS THAT &quot;MOVE PARENTHESIS&quot;</td>
<td>16</td>
</tr>
<tr>
<td>ML, MR</td>
<td>17</td>
</tr>
<tr>
<td>BI</td>
<td>18</td>
</tr>
<tr>
<td>BO</td>
<td>19</td>
</tr>
<tr>
<td>COMMANDS THAT UNDO</td>
<td>20</td>
</tr>
<tr>
<td>??</td>
<td>21</td>
</tr>
<tr>
<td>UNDO</td>
<td>22</td>
</tr>
<tr>
<td>!UNDO</td>
<td>23</td>
</tr>
<tr>
<td>TEST, UNBLOCK</td>
<td>24</td>
</tr>
<tr>
<td>COMMANDS THAT PRINT</td>
<td>25</td>
</tr>
<tr>
<td>P, ? , PP</td>
<td>25</td>
</tr>
<tr>
<td>¬P</td>
<td>26</td>
</tr>
<tr>
<td>MISCELLANEOUS COMMANDS</td>
<td>27</td>
</tr>
<tr>
<td>S</td>
<td>27</td>
</tr>
<tr>
<td>E</td>
<td>27</td>
</tr>
<tr>
<td>OK</td>
<td>27</td>
</tr>
<tr>
<td>HELP</td>
<td>27</td>
</tr>
<tr>
<td>DEFAULTS</td>
<td>28</td>
</tr>
<tr>
<td>A SAMPLE EDITING RUN</td>
<td>29</td>
</tr>
<tr>
<td>EDITOR FUNCTIONS</td>
<td>34</td>
</tr>
<tr>
<td>EDITOR INTERNAL FUNCTIONS</td>
<td>35</td>
</tr>
<tr>
<td>DEBUGGING FACILITIES</td>
<td>37</td>
</tr>
<tr>
<td>WHAT YOU CAN DO IN A BREAK</td>
<td>39</td>
</tr>
<tr>
<td>GO</td>
<td>39</td>
</tr>
<tr>
<td>OK</td>
<td>39</td>
</tr>
<tr>
<td>EVAL</td>
<td>39</td>
</tr>
<tr>
<td>RETURN</td>
<td>39</td>
</tr>
<tr>
<td>FROM</td>
<td>39</td>
</tr>
<tr>
<td>USE</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ARGS</td>
<td>40</td>
</tr>
<tr>
<td>HELP</td>
<td>40</td>
</tr>
<tr>
<td>CONTEXT COMMANDS</td>
<td>41</td>
</tr>
<tr>
<td>F</td>
<td>41</td>
</tr>
<tr>
<td>TOP</td>
<td>42</td>
</tr>
<tr>
<td>EDIT</td>
<td>42</td>
</tr>
<tr>
<td>AT</td>
<td>42</td>
</tr>
<tr>
<td>FROM</td>
<td>42</td>
</tr>
<tr>
<td>BACKTRACE COMMANDS</td>
<td>43</td>
</tr>
<tr>
<td>BKF</td>
<td>43</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>BKE</td>
<td>43</td>
</tr>
<tr>
<td>BK</td>
<td>43</td>
</tr>
<tr>
<td>BREAK PACKAGE</td>
<td>44</td>
</tr>
<tr>
<td>BREAKF</td>
<td>44</td>
</tr>
<tr>
<td>TRACE</td>
<td>44</td>
</tr>
<tr>
<td>UNBREAK</td>
<td>44</td>
</tr>
<tr>
<td>UNTRACE</td>
<td>45</td>
</tr>
<tr>
<td>BREAKO</td>
<td>46</td>
</tr>
<tr>
<td>ERROR PACKAGE</td>
<td>47</td>
</tr>
<tr>
<td>BREAK PACKAGE COMMAND SUMMARY</td>
<td>48</td>
</tr>
<tr>
<td>A SAMPLE DEBUGGING SESSION</td>
<td>50</td>
</tr>
<tr>
<td>BREAK PACKAGE INTERNAL FUNCTIONS</td>
<td>58</td>
</tr>
<tr>
<td>PRETTYPRINTING AND PROGRAM FILES</td>
<td>60</td>
</tr>
<tr>
<td>PRETTYPRINT</td>
<td>60</td>
</tr>
<tr>
<td>DSKIN</td>
<td>60</td>
</tr>
<tr>
<td>DSKOUT</td>
<td>60</td>
</tr>
<tr>
<td>UPDATE</td>
<td>61</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>62</td>
</tr>
<tr>
<td>APPENDEX</td>
<td>63</td>
</tr>
</tbody>
</table>
INTRODUCTION

Writing and debugging LISP programs can either be a dream or nightmare. The LISP language contains both aids (its functional nature, the fact that programs are data) and obstacles (the "parentheses matching" problem and the ease in which the programmer can create recursive rats' nests) to good programming. Presented here is a collection of LISP functions which, together, create a more friendly environment for programming.

The READ-EVAL-PRINT loop in LISP is actually a fast turnaround batch system which, by itself, provides very little in the way of debugging aids. If no LISP errors occur, than a value is printed, which may be correct or incorrect (of course, side effects may occur). If the value is incorrect, the programer is left pondering over his listings, or inserting calls to PRINT in the program file, altering and rereading s-expressions, and trying again. If LISP errors do occur, a (generally unreadable) backtrace of function calls is printed.

Since LISP is executed interpretively, much information can be gathered during the evaluation of LISP functions. Also, since LISP functions are themselves list structures, they can be changed to provide even more information about the execution of the program.

The functions presented here (following the BBN-LISP model, the break package documentation is from UCI) attempt to make LISP a truly interactive programming language by allowing the user to take corrective action when LISP errors are detected and to monitor the evaluation of (user) selected functions. When errors are detected (either by the interpreter or the user), the function definitions may be changed (within LISP) and execution continued. The parentheses problem, which new LISP users complain about so much is eased, with a prettyprinter (a function which prints LISP structures in an indented form) and by the fact that it is harder to make parentheses errors inside LISP because, once the parentheses match, it is impossible for them to be made not to match.

No amount of "programming aids" will cushion the blow of learning LISP. It is very different than most other languages. Most new users tend to write FORTRAN or ALGOL-like LISP programs missing many of the advantages of the LISP language. Similarly, users accustomed to a batch type language (as most are), sometimes find it difficult to convert to a truly interactive one. Consequently, many LISP users still run in the essentially batch READ-EVAL-PRINT mode, not taking advantage of the debugging aids possible. Such users are encouraged to experiment with the debugging package described in this manual, if only to discover for themselves the power, flexibility and elegance of a genuinely interactive LISP system.
A LISP EDITOR

INTRODUCTION

The power of LISP as an interactive language is reduced when the user is unable to change his program while in the LISP environment. For this reason, many LISP editors have been written, including one at BBN. The BBN LISP editor has the advantage of being both simple to use and very powerful. It has the disadvantage of being very large. The editor presented here is a somewhat scaled-down version of the BBN editor, which with the prettyprinter, requires 10 pages of memory. The editor is written in LISP/MTS.

The editor operates by being called by one of two top level functions. For example, to edit function definitions, one uses \((\text{EDITF F00 FIE})\) where F00 and FIE are the functions to be edited. The other top level function is \(\text{EDITE}\). \(\text{EDITE}\) evaluates its argument passing it to the editor. Once in the editor, the user directs operations through commands. These commands fall into three broad classes, those which change the editor's attention, those which actually modify the structure being edited, and miscellaneous commands.
COMMANDS THAT CHANGE THE EDITOR'S ATTENTION

In any editor, there must be a way to specify where changes are to be made. In the MTS editor, this is done both by the use of line numbers, and by searching for strings. Since there are no lines in a LISP structure, the commands here will need to specify by structural context where to look. The reader is reminded that what is typed to the function (READ) and what he is returned from (PRINT) does not exist as a character string in memory. There is no

\[(A \ B \ C \ D)\]
inside LISP, but rather something more like

```
```

When the user types (EDITF FOO), the editor's attention is on the list structure of the function FOO. At any time, the editor's attention will be on some substructure of the function FOO. This substructure will hereafter be called the **current expression**, and when we say that the current expression is changed, we mean that the editor is looking somewhere else, **NOT THAT THE STRUCTURE OF THE FUNCTION IS CHANGED**. The current expression can always be seen by typing the command ? . Now, suppose, at the top level of LISP, we enter

```
* (DEFUN FOO (X)
*    (COND ((NULL X) 'NIL)
*    (T (CONS X X)))
> FOO
* (EDITF FOO)
:EDITING FOO
:(LAMBDA (X) (COND & 6)
:.?
:LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))
*
```

The input prefix character for the editor is "." and the output prefix character is ":". When the editor is first entered, it responds with the name of the function it is editing and the current expression as printed by the P command (see P command) which at the beginning is the entire list structure. As the editor descends into the structure, it must know how to get out. It keeps this "backup" information on a list called the EDITCHAIN. Thus, changing the editor's attention amounts to changing the EDITCHAIN. We can now look at commands which descend into and ascend from the structure being edited.
**COMMAND: n, -n**

Typing a positive integer tells the editor to make the new current expression the nth element of the current expression.

Example:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
.3 ?
:(COND ((NULL X) NIL) (T (CONS X X)))
.2 1 ?
:(NULL X)
```

Note the use of several commands on a line. The editor executes all commands on a line unless one causes an error in which case the rest of the line is ignored. A P is performed after every line unless the last command on the line was a P, PP, TEST, UNBLOCK, E, HELP, or ?.

A positive integer makes the current expression the nth from the left, a negative integer makes the current expression the nth from the right.

Example:

```
.? 
:(COND ((NULL X) NIL) (T (CONS X X)))
.-1 ?
:(T (CONS X X))
.2 -1 ?
:X
```

A zero pops the EDITCHAIN once ascending one level.

Example:

```
.? 
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
.2 ?
:(X)
.1 ?
:X
.0 ?
:(X)
.0 ?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
.3 2 ?
:((NULL X) NIL)
.0 ?
:(COND ((NULL X) NIL) (T (CONS X X)))
```
**COMMAND: UP**

A tail of a list is defined as being 0 or more CDR's down the list. Thus in the structure

```
  1  2  3  4
  A ---> B ---> C ---> D /
```

boxes 1, 2, 3 and 4 are all tails of FOO. PRINT would say (B C D) is a tail of (A B C D). The UP command is used to look at tails of lists. A tail is indicated by the ... printed by the editor.

Example:

```
> ? : (LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
> 3 ? : (COND ((NULL X) NIL) (T CONS X X))
> UP ? : : (COND ((NULL X) NIL) (T CONS X X)))
> ?
```

What has happened is that the current expression is ascended exactly one CAR. Continuing with the example:

```
> ? : : (COND ((NULL X) NIL) (T CONS X X)))
> 1 ? : (COND ((NULL X) NIL) (T CONS X X))
> 2 UP ? : : ( (NULL X) NIL) (T CONS X X))
> 
```

After an UP, the old current expression is the first element of the new current expression. If the current expression happens to be the first element of the next higher expression, then an UP and 0 do the same thing.
**COMMAND: 1**

This command directs the editor to "go to the top" of the structure.

Example:

```lisp
?(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.3 2 1 ?
:(NULL X)
.1 ?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))

..
COMMAND: _10

This command (usually pronounced bang-zero) insures that the current expression is not a tail of the next higher expression. It always goes back to the next higher current expression.

Example:

`*?:(A B C D E F G)
.*3 UP ?
.*.*. C D E F G)
.*3 UP ?
.*.*. E F G)
.*0 ?
.*.*. C D E F G)
.*3 UP ?
.*.*. E F G)
.*!0 ?
.*!(A B C D E F G)
.*`
COMMAND: NK, BK

These two commands direct the editor to go to the "next" element or to go "back" one element. The simplest way to explain their operation is by example.

Example:

```
NX ?
:((LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
1 ?
:LAMBDA
NX ?
:(X)
NX ?
:(COND ((NULL X) NIL) (T (CONS X X)))
1 ?
:COND
NX ?
:((NULL X) NIL)
NX ?
:(T (CONS X X))
BK ?
:((NULL X) NIL)
BK ?
:COND
```

NX effectively does an UP followed by a 2. It can be useful when stepping through COND, PROG and SELECT clauses.
COMMAND: _NX

The NX command will fail if the current expression is the last element of the next higher expression. The !NX command makes the current expression the next expression at a higher level. It will go through any number of right parentheses to get the next expression.

Example:

```
.?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
.3 2 1 ?
:(NULL X)
.NX ?
:NIL
.NX ?
:YOU'RE AT THE END
.!NX ?
:(T (CONS X X))
.
```

Here, the NX command failed because NIL is the last expression in ((NULL X) NIL) but !NX will succeed and go to the (T (CONS X X)). To continue with the example:

```
.?
:(T (CONS X X))
.0 0 ?
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
.2 1 ?
:X
.!NX
:(COND ((NULL X) NIL) (T (CONS X X)))
.
```
**COMMAND: F**

This command causes the editor to search the current expression for the next "thing" in the command stream. Usually this "thing" is an atom. Because there is nothing that can be done with an atom except look at it, the F command searches for a list whose CAR is the "thing". If the current expression is an atom after an F command, the editor does an UP.

Example:

```
?.
:(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
.F X ?
:(X)
.
```

The editor is asked to find the first occurrence of the atom X (in PRINT order). It finds a list whose CAR is X and that list becomes the current expression. Continuing with the example:

```
.| F NULL ?
:(NULL X)
.| F COND ? F X ?
):(COND ((NULL X) NIL) (T (CONS X X)))
|:... X)
| 0 ?
):(NULL X)
.UP UP ? F CONS ?
|:... ((NULL X) NIL) (T (CONS X X)))
):(CONS X X)
.| F (NULL X) ?
):(NULL X)
.
```

While searching, the F command looks for

1. Something EQUAL to its argument
2. Something whose CAR is EQ to its argument.

The F command does no CONSes unless it finds what it is looking for, and then only enough to build the EDITCHAIN.

If F fails, the EDITCHAIN is not changed.
COMMANDS WHICH CHANGE THE EDITED STRUCTURE

This section will discuss the commands which change the structure being edited. These commands use RPLACA's and RPLACD's to perform their modifications and are thus destructive. This is useful when the editor is called while in a BREAK. If a function is changed, the change will affect even previous entries on the stack.

All structural changes are undoable. Since the act of undoing is itself a structural change, UNDO's are undoable. This should allow the user to experiment with the editor with some degree of safety. He can always UNDO his mistakes.

All structure changing commands use the same routine to find the position to perform the modification (called the location specification). These routines operate in the following manner.

1. If its argument is an editor command, execute it. These must be commands which take no arguments (NX, but not P).
2. If its argument is not an editor command, perform an F (find) on it.

Structure modifying commands NEVER change the editors attention. If the entire current expression is deleted, the operation will not appear by typing a ?.
COMMAND: INSERT, I

I is the abbreviated form of the INSERT command. The general form if the INSERT command is

AFTER
INSERT S1 S2 S3 ... SN BEFORE P1 P2 P3 ... PN <::>
FOR

Because the INSERT command takes an arbitrary number of arguments, a : or end of line is used to indicate the end of command.

The S's are the "things' to be inserted.
The P's are the location specification.

Example:

• ?
  :(LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))
  *INSERT (A B C) AFTER X : ?
  :(LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
  *

Since the X is not an editor command, a F X is performed which finds the list (X).

• ?
  :(LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
  *INSERT B BEFORE X 1 : ?
  :(LAMBDA (B X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
  *

Here, the X again causes an F X to be performed finding the (X), the 1 is an editor command which made the current expression X and the B was inserted before it.

• F NULL ? INSERT B FOR X : ?
  :(NULL X)
  :(NULL B)
  *INSERT ABC D AFTER B : ?
  :(NULL B ABC D)
  *

This time, a B is searched for in the substructure (NULL B) and is found, the current expression becomes ... B). The INSERT inserts the ABC D after the B. Instead of typing F NULL INSERT B FOR X, we could have used INSERT B FOR NULL X which would do a F NULL and a F X.
COMMAND: DELETE, D

The D command is the abbreviated form of the DELETE. The general form is:

DELETE P1 P2 P3 ... PN <::>

where the P's are the location specification of the item to be deleted.

Example:

: (LAMBDA (Q X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
  DELETE Q 1 : ?
: (LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))

In this example, the Q found the list (Q X) and the 1 made the current expression Q, which is deleted.

: (LAMBDA (X) (A B C) (COND ((NULL X) NIL) (T (CONS X X))))
  DELETE (A B C) : ?
: (LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X))))

We could have used D A, D (A B C), D 2, D 1 NX, or any way we want to find the (A B C). In the example, we can specify the entire structure if we want.

: (LAMBDA (X) (COND ((NULL X) NIL) (T CONS X X))))
  DELETE NIL ?
: (LAMBDA (X) (COND ((NULL X)) (T (CONS X X))))
COMMAND: EMBED, EM

The EM command is the abbreviated form of EMBED. The general form is:

EMBED P1 P2 P3 ... PN IN I

where the P's are the location specification of the item to be embedded and the I is the structure to contain it.

In LISP, because of its nested function structure, special commands (the EMBED and EXTRACT) are included to insert functions inside other functions. It is not uncommon to simply want to embed a structure in a QUOTE or PRINT.

Example:

.?
: (COND ((NULL X) NIL) (T (CONS X X)))
. EMBED NULL IN (PRINT *) ?
: (COND ((PRINT (NULL X)) NIL) (T (CONS X X)))
.

The EMBED operates by replacing every occurrence of "*" in the I by the current expression and performing an INSERT FOR on the current expression.

. EMBED CONS IN (CONS * *) ?
: (COND ((PRINT (NULL X)) NIL) (T (CONS (CONS X X) (CONS X X))))
.

Note, there is no need for the : to terminate the command (as in the INSERT and DELETE) because there can be only one item following the IN.
**COMMAND: EXTRACT, EX**

EX is the abbreviated form of the EXTRACT command. The general form is:

```
EXTRACT P1 P2 P3 ... PM FROM Q1 Q2 Q3 ... QN <:>
```

where the P's are the location specification of the item to replace the item specified by the Q's. Where the EMBED command is used to nest one function within another, the EXTRACT removes a nesting. The item specified by the Q's must be within the item specified by the P's.

Example.

```
: (COND ((NULL (CAR X)) NIL) (T (CONS (CAR X) (CADR X)))))
*EXTRACT CAR FROM NULL : ?
: (COND ((CAR X) NIL) (T (CONS (CAR X) (CADR X)))))
*EXTRACT CONS FROM COND : ?
: (CONS (CAR X) (CADR X))
```

The EXTRACT command replaces a structure L by a substructure of L.
COMMANDS THAT "MOVE PARENTHESES"

Since parentheses are simply notations to (READ) and (PRINT), it is impossible to add or remove one left or right parenthesis. These commands are used to change the structure being edited so that when the structure is printed, it looks as though a parenthesis was moved, or a pair of parentheses was added or removed.
COMMANDS: ML, MR

These commands move a left parenthesis or move a right parenthesis. Their general form is:

ML
  _integer P1 P2 P3 ... Pn <:>

MR

For both commands, a positive integer indicates the parenthesis is to be moved to the right, a negative integer indicates the parenthesis is to be moved to the left. The P's are the location specification for the list whose parenthesis is to be "moved".

Example:

```
  .?
  :(A B C (D E F) G H I)
  .ML -1 4 : ?
  :(A B (C D E F) G H I)
```

It appears as if the left parenthesis of the 4th element has been moved 1 place to the left (-1 places to the right).

```
  .?
  :(A B (C D E F) G H I)
  .ML 3 C : ?
  :(A B C D E (F) G H I)
```

The left parenthesis of the list whose CAR is C is moved 3 places to the right.

```
  .?
  :(A B C D E (F) G H I)
  .MR 2 F : ?
  :(A B C D E (F G H) I)
```

The move right command is used to move the right parenthesis of the list whose CAR is F 2 places to the right.

```
  .?
  :(A B C D E (F G H) I)
  .MR -1 F : ?
  :(A B C D E (F G) H I)
```
**COMMAND: BI**

General form:

BI P1 P2 P3 ... Pm THRU Q1 Q2 Q3 ... Qn <::>

where the P's and Q's are location specifications.
This command is used to insert a pair of left and right parentheses
before the element specified by the P's and after the element
specified by the Q's.

Example:

```plaintext
.: (? (A B C D E F)
.: BI B THRU E : ?
.: (A (B C D E) F)
.: BI C THRU D : ?
.: (A (B (C D) E) F)
.: BI 1 NX THRU -1 BK : ?
.: (A (( B (C D) E)) F)
```

-
COMMAND: BO

This command is used to remove a pair of parentheses (or take "both out"). The general form is:

BO P1 P2 P3 ... Pn <;>

where the P's are the location specification of the list whose parentheses are to be removed.

Example:

. ?
. (A ((B (C D) E)) F)
. BO C : ?
. (A ((B C D E)) F)
. BO 2 : ?
. (A (B C D E) F)
. BO 1 NX : ?
. (A B C D E F)
. 
COMMANDS THAT UNDO

For each structural change made, an entry is also made on a list called the undolist. There are 4 commands which affect undoing. UNDO and !UNDO actually undo the changes. TEST and UNBLOCK help to determine what is to be undone. The undo facilities of the editor permit the edited structure to be restored to its state at any prior point in the editing session. These features should encourage the user to experiment with the editor; he can always painlessly recover from his mistakes.
**COMMAND: ?**

This command causes the entries on the undolist to be listed. Example:

```
.?  
:(A B C D E F)  
*INSERT (W X Y) AFTER C : ?  
:(A B C (W X Y) D E F)  
*DELETE D : ?  
:(A B C (W X Y) E F)  
*MR 1 W : ?  
:(A B C (W X Y E) F)  
??  
:MR DELETE INSERT  
.*
```

There are three "things" on the undolist and they are listed (and undone) in the inverse order of their entry.
**COMMAND: UNDO**

This command causes the last structural change to be undone.

Example:

```
.?
: (A B C D E F)
.DELETE C : DELETE D : ?
: (A B E F) L .UNDO 0
:DELETE UNDONE L .? ?
: (A B D E F) .UNDO NDO
:DELETE UNDONE L .? ?
: (A B C D E F) .UNDO NDO
:NOTHING SAVED .?? ??
: UNDO UNDO
.
```

Note UNDO's and !UNDO's are skipped by the UNDO command unless they are specifically requested.

```
.UNDO UNDO
: UNDO UNDONE
.?
: (A B D E F)
.
```
COMMAND: !UNDO

The !UNDO command undoes all changes back to the beginning or to the first UNDO block (see TEST and UNBLOCK).

Example:

```plaintext
.?
:(A B C (D E F) G H I)
:INSERT (W X Y) Z AFTER B : ?
:(A B (W X Y) Z C (D E F) G H I)
:DELETE D : ?
:(A B (W X Y) Z C G H I)
:MR 5 W : ?
:(A B (W X Y Z C G H I)) .!UNDO NDO
:MR UNDONE
:DELETE UNDONE
:INSERT UNDONE
.?
:(A B C (D E F) G H I) L .?? ?
:!UNDO .UNDO !UNDO UNDO
:!UNDO UNDONE
.?
:(A B (W X Y Z C G H I)) L .?? ?
:UNDO
```
COMMAND: TEST, UNBLOCK

A block may be placed on the undolist in two ways. If the TEST command is entered, then a block is placed on the front of the undolist. Also, after each editing session, all UNDO information is saved on the atom of the function name with a block on the front so that editing sessions are divided. The command UNBLOCK removes the first undo block on the undolist.

Example:

```
  .?  
  :(A B C (D E F) G H I)  
  .DELETE D : DELETE C : INSERT (W W W W) AFTER H : MR 1 W : ? ??  
  :(A B G H (W W W W I))  
  :MR INSERT DELETE DELETE .TEST ?? T ??  
  :BLOCK MR INSERT DELETE DELETE L .? ?  
  :(A B G H)  
  :DELETE W : DELETE B : ? ??  
  :(A G H)  
  :DELETE DELETE BLOCK MR INSERT DELETE DELETE  
  .!UNDO  
  :DELETE UNDONE  
  :DELETE UNDONE  
  .? ??  
  :(A B G H (W W W W I))  
  :!UNDO BLOCK MR INSERT DELETE DELETE  
```

The !UNDO was stopped by the block on the undolist. The TEST command is useful for making tentative changes which are easily removed by a !UNDO. We can remove the undo block by typing:

```
  .UNBLOCK ??  
  :!UNDO MR INSERT DELETE DELETE  
  .!UNDO  
  :!UNDO UNDONE  
  :MR UNDONE  
  :INSERT UNDONE  
  :DELETE UNDONE  
  :DELETE UNDONE  
  .? ??  
  :(A B C (D E F) G H I)  
  .!UNDO
```
COMMANDS THAT PRINT

COMMAND: P, ?, PP

P is the standard print command of the editor. It prints the current expression up to depth 2. If followed by an integer n, it prints up to depth n.
Example:

?  
: (LAMBDA (X) (COND ((NULL X) NIL) (T (CONS X X)))))
. P
: (LAMBDA (X) (COND & &))
. P 3
: (LAMBDA (X) (COND (& NIL) (T &)))

Usually the entire current expression is too big and only P is used. The command ? has been used for examples here so that the entire current expression could be seen.

The ? Command is the same as P 1000.

The PP command prettyprints the current expression.
Example:

PP
: (LAMBDA (X)
  : (COND ((NULL X) NIL)
  : (T (CONS X X)))))
COMMAND: \texttt{\textasciitilde P}

Strictly speaking, this is not a print command but an attention changing command. It restores the EDITCHAIN to where it was as of two prints ago. It is useful for switching back and forth between EDITCHAINS.

Example:

\begin{verbatim}
  .P
  :(LAMBDA (X) (COND & &))
  .F NULL P \texttt{\textasciitilde P}
  :(NULL X)
  :(LAMBDA (X) (COND & &))
\end{verbatim}

In this example, the \texttt{P NULL} found the first clause of the \texttt{COND}. The \texttt{\textasciitilde P} sends us back to the \texttt{(LAMBDA (X) (COND & &))}. Another \texttt{\textasciitilde P} would make the current expression \texttt{(NULL X)} again.
MISCELLANEOUS COMMANDS

COMMAND: S

This command saves the current expression on the property list of the next "thing" in the input stream. This value can later be used by an INSERT command by using a ## before the "thing" in the list of items to be inserted. The saved current expression can also be found by EVALing (GET "THING" *EDITFLAG*).

Example:

```lisp
(defun null-p (x) (cond ((null x) nil) (t (cons x x)))

(insert ## QREG FOR CONS : ?

(defun null-p (x) (cond ((null x) nil) (t (cons x x)))
```

In the example, (null X) was saved on the atom QREG (actually, in order to protect against circular lists, a copy is made). The INSERT command instructs the editor to get that value (by preceding QREG by the atom ## and insert it for the CONS. If, at the top level of LISP we EVALed (GET 'QREG *EDITFLAG*), (null X) would be returned.

COMMAND: E

This command EVALs the next "thing" in the input stream. It is not "errorset" protected, that is, it will possibly destroy saved information (EDITCHAIN, undolist, etc.) if a bad form is EVALed. If the stack information is retained (through the error package, for example), then the editor may be reentered at EDITONE or EDITNEXT by EVALing (UNEVAL 'EDITONE NIL) or (UNEVAL 'EDITNEXT NIL).

COMMAND: OK

This command exits the editor saving the undolist on the property list of the function name if the structure being edited was a function.

COMMAND: HELP

This command causes a brief description of the editor commands to be listed on the user's terminal.
The commands INSERT, MR, ML, BO, BI, EXTRACT may have their last argument left out. If the command was an INSERT, then FOR is assumed and the S's will be inserted FOR the current expression. For the others, the current expression is assumed.
A SAMPLE EDITING RUN

A sample editing run is presented here to show the use of several commands. The functions which will be edited are supposed to print the names of all atoms which don't have a system function on their property lists. It is assumed that the editor is in memory.

```lisp
*<DEFUN SYSFIND (X) (MAPC 'SYSPRONT ((OBLIST> SYSFIND
*<DEFUN SYSPRINT (X) (CONS ((GETL X ' (SUBR NSUBR) NIL) (T (PRINT Y> SYSPRINT
*(EDITF SYSFIND SYSPRINT)
:EDITING SYSFIND
:(LAMBDA (X) (MAPC & 5))
:PP
:(LAMBDA (X)
: (MAPC (QUOTE SYSPRONT) ((OBLIST))))
:OK
:EDITING SYSPRINT
:(LAMBDA (X) (CONS &))
:PP
:(LAMBDA (X)
: (CONS
: ((GETL X (QUOTE (SUBR NSUBR)) NIL)
: (T (PRINT Y))))
:OK
> NIL

We prettyprint the functions to see how they look, then try them. We could use the E command to try the functions, but that would require some knowledge of the break package which we do not assume here.

*(SYSFIND)
+  ***4 TOO MANY ARGUMENTS
+  SYSFIND
?NIL
*(EDITF SYSFIND)
:EDITING SYSFIND
:(LAMBDA (X) (MAPC &))
:INSERT NIL FOR X
:(LAMBDA NIL (MAPC &))
:OK
> NIL

The (X) was wrong here, we remove it and try again.

*(SYSFIND)
+  ***17 UNDEFINED FUNCTION
+  110
?NIL
*(EDITF SYSFIND)
The 110 just happened to be the first atom on the oblist and was EVALed because of the extra set of parentheses. We remove them and try again.

*(SYSFIND)
+   ***17 UNDEFINED FUNCTION
+  SYSPRONT
NIL
*(EDIT SYSFIND)
:EDITING SYSFIND
:(LAMBDA NIL (MAPC &E))
*PP
:(LAMBDA NIL
    (MAPC (QUOTE SYSPRONT) ((OBLIST))))
*BO OBLIST : PP
:(LAMBDA NIL
    (MAPC (QUOTE SYSPRONT) (OBLIST)))
*OK
> NIL

SYSPRINT was typed incorrectly so we change it with an insert command. Note, there is no need to search for the SYSPRONT, the insert will do that for us.

*(SYSFIND)
+   ***17 UNDEFINED FUNCTION
+  NIL
NIL
*(EDIT SYSPRINT)
:EDITING SYSPRINT
:(LAMBDA (X) (CONS &E))
*PP
:(LAMBDA (X)
    (CONS
        : ((GETL X (QUOTE (SUBR NSUBR)) NIL)
        : (T (PRINT Y)))))
*INSERT COND FOR CONS : PP
:(LAMBDA (X)
    : COND)
*UNDO
:INSERT UNDONE
*PP
:(LAMBDA (X)
    : (CONS
        : ((GETL X (QUOTE (SUBR NSUBR)) NIL)
        : (T (PRINT Y)))))
*INSERT COND FOR CONS 1 : PP
In the above try, we discover that there is an error in SYSFIND so we prettyprint it and see that we typed CONS for COND. We use the insert command to change it but forget that it will find the list (CONS & &) instead of the atom CONS. The insert doesn't do what we want to do. We undo the insert and try again, this time succeeding. We try again.

* (SYSFIND)
+ *(**17 UNDEFINED FUNCTION*)
+ T

?NIL
* (EDITF SYSPRINT)
: EDITING SYSPRINT
: (LAMBDA (X) (COND &))
: PP
: (LAMBDA (X)
: (COND
: (GETL X (QUOTE (SUBR NSUBR))) NIL)
: (T (PRINT Y))))

3
: (COND (& &))
: MR -1 2
: (COND (&) (T &))
: PP
: (COND ((GETL X (QUOTE (SUBR NSUBR))) NIL)
: (T (PRINT Y)))
: MR -1 GETL : PP
: (COND ((GETL X (QUOTE (SUBR NSUBR))) NIL)
: (T (PRINT Y)))
: P
: (COND (& NIL) (T &))
: OK

> NIL

In the above test, we discover that parentheses errors were made when the functions were typed in. We use both the PP and P commands to show the nestings.

* (SYSFIND)
+ *(**16 UNDEFINED ATOM*)
+ Y

?NIL
* (EDITF SYSFIND)
: EDITING SYSFIND
: (LAMBDA NIL (MAPC & &))
: INSERT X FOR Y : PP
: Y NOT FOUND
: OK
Above, we try to change the atom Y to the atom X but we call the editor on the wrong function. We exit and type the right function name in. We change the Y to X, prettyprint it to see that all is ok, and try again.

```lisp
*(SYSFIND)
> -4
> 110
> 68
> 26
.

+ 16 UNDEFINED ATOM
+ X
?NIL
*(EDITF SYSPRINT)
:EDITING SYSPRINT
:(LAMBDA (X) (COND & E))
:PP
:(LAMBDA (X)
: (COND ((GETL X (QUOTE (SUBR NSUBR))) NIL)
: (T (PRINT X))
).
:EDITING SYSPRINT
: (LAMBDA (X)
: (COND ((GETL X (QUOTE (SUBR NSUBR))) NIL)
: (T (PRINT X)))
).
OK
> NIL
```

The above case is more subtle. When we do a GETL on the atom *UNDEF*, we create an error condition. We cannot EVAL the atom *UNDEF*. To solve this problem, we first check the value (CAR) of X to see if it is the undefined atom *UNDEF*, and if so, we stop there.

```lisp
*(SYSFIND)
> -4
> 110
> 68
> 26
.
```
>DISPLAY

*:EDITF SYSPRINT
:EDITING SYSPRINT
:(LAMBDA (X) (COND & E))
:INSERT FSUBR AFTER NSUBR : OK
> NIL

We see from the output, that some functions are sneaking through because we forgot to check for FSUBRs. We change SYSPRINT (we are brave and don't print the change) and our function will work.

In the following example, we edit a PLANNER theorem with EDITE. PLANNER theorems are stored on property lists under the indicator THEOREM.

*/<DEFPROP F THEOREM (THCONSE (IN HOUSE $?X) (THGOAL $?X))
> F
*/(EDIT (GET F *THEOREM))
:(THCONSE (IN HOUSE $?X) (THGOAL $?X))
:EMBED THGOAL 2 IN (RABBIT *) PP
:(THCONSE (IN HOUSE $?X)
: (THGOAL (RABBIT $?X)))
:OK
> NIL

In the above example, we call the editor on a structure which is a PLANNER theorem. We make a change and exit.
EDITOR FUNCTIONS

The following are the top level functions of the editor.

EDITF  This is an FEXPR. It looks for the property EXPR
       MACRO, or BUG on the property list of its atomic
       arguments.

EDITE  This is an EXPR. It passes its EVALed argument to the
       editor.

EDITEXCISE  This is a function of no arguments. EVALing
               it removes the editor function definitions
               freeing storage of the editor. The resulting
               free core is then available as work space.

All these functions are found in the file CS:MINIED.
EDITOR INTERNAL FUNCTIONS

The following is a list of the major editor functions and their operation.

EDITF is the top level function for editing function definitions. EDITF opens the necessary buffers and performs the status calls. It then uses GETL to find the function definition passing it to EDITSTART.

EDITNEXT is the editor input function. Together with EDITINTERCEPT, it returns the next item in the input stream. It takes as argument a code which directs the intercept function to either signal an error condition if there is no more data (code T), return *EDITFLAG* on end of line (code of A) or read a new line (code of NIL).

EDITSMASH performs the structure changes for the INSERT, DELETE, EMBED, and EXTRACT commands as well as the parenthesis moving commands. It takes as arguments any number of triplets. Each triplet describes how a particular CONS cell is to be changed (the cell, its new CAR, and its new CDR). An entry is made on the undolist for each call to EDITSMASH.

EDITSTART is the READ EVAL loop function of the editor.

EDITONE contains the code for most of the editor commands. It is really one big SELECTQ.

EDITFIND is the recursive searching function used by the F command as well as the location specification operations. It builds the EDITCHAIN to the item it found.

EDITCOLLECT is used when input items are to be scanned up to a delimiter (THRU, AFTER, IN, etc). It returns a list of the items it collected.

PRINLEV is the function used by the P and ? commands. It takes as arguments the expression to be printed and the level to stop at.

EDITEVAL performs the location specification operations. It checks to see if an item gathered by EDITCOLLECT is a command and if so, calls EDITONE on it, else it calls EDITFINDIT (the top level find function) on the item.

EDITUNDO is the undoing function. It takes as argument the item on the undolist to be undone and the
type of undo (UNDO or !UNDO).
DEBUGGING FACILITIES

Introduction

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data are being generated and transmitted. There are three facilities which aid the user in monitoring his program. One of these is the Error Package which takes control whenever an error occurs in a program and which allows the user to examine the state of the world. The other two facilities (BREAKF and TRACE) allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs. All of these facilities use the same system function, BREAKFUNCTION, as the user interface.

BREAKF and TRACE together are called the Break Package.

BREAKF modifies the definition of a function FN, so that if a break condition (defined by the user) is satisfied, the evaluation is halted temporarily on a call to FN. The user can then interrogate the state of the world, perform any computations, and continue or return from the call.

TRACE modifies a definition of a function FN so that whenever FN is called, its arguments (or some other values specified by the user) are printed. When the value of FN is computed it is printed also.

The heart of the debugging package is a function called BREAKFUNCTION. BREAKF and TRACE redefine functions in terms of BREAKFUNCTION. When an error occurs control is passed to BREAKFUNCTION.

Whenever LISP types a message of the form — BROKEN followed by 'n :• the user is then 'talking to' BREAKFUNCTION, and he is 'in a break'. BREAKFUNCTION allows the user to interrogate the state of the world and affect the course of the computation. It uses the prompt character '=" to indicate it is ready to accept input(s) for evaluation, in the same way as the top level of LISP uses "*". The n before the ':' is the level number which indicates how many levels of BREAKFUNCTION are currently open. The user may type in an expression for evaluation and the value will be printed out, followed by another '=". Or the user can type in one of the commands described below which are specifically recognized by BREAKFUNCTION (for summary of commands see Table I). Since BREAKFUNCTION puts all of the power of LISP at the user's command, he can do anything he could do at the top level of LISP. For example, he can define new functions or edit existing ones, set breaks, or trace functions. The user may evaluate an expression, see that the value was incorrect, call the editor, change a function, and evaluate the expression again, all without leaving the break.

It is important to emphasize that once a break occurs, the user is in complete control of the flow of the computation, and the computation will not proceed without specific instruction from him. Only if the user gives one of the commands that exits from the break (GO, OK, RETURN, FROM) will the computation continue. The computation
can also be aborted (using | or ||).

Note that BREAKFUNCTION is just another LISP function, not a special system feature like the interpreter or the garbage collector. It has arguments and returns a value, the same as any other function. A call to BREAKFUNCTION has the form

\[(\text{BREAKFUNCTION BREAKEXPR BREAKWHEN BREAKFN BREAKCMDS BREAKTYPE})\]

The arguments to BREAKFUNCTION follow: BREAKWHEN is a LISP function which is evaluated to determine if a break will occur. If BREAKWHEN returns NIL, BREAKEXPR is evaluated and returned as the value of the break. Otherwise a break occurs. BREAKFN is the name of the function being broken and is used to print an identifying message. BREAKCMDS is a list of command lines which are executed as if they had been typed in. The command lines on BREAKCMDS are executed before commands are accepted from the terminal, so that if one of the commands on BREAKCMDS causes a return, a break occurs without the need for user interaction. BREAKTYPE identifies the type of the break. It is used primarily by the error package and in all cases the user can use BREAK for this argument.

The value returned by BREAKFUNCTION is called 'the value of the break'. The user can specify this value explicitly by using the RETURN command described below. In most cases, however, the value of the break is given implicitly, via a GO or OK command, and is the result of evaluating 'the break expression', BREAKEXPR.

BREAKEXPR is, in general, an expression equivalent to the computation that would have taken place had no break occurred. In other words, one can think of BREAKFUNCTION as a fancy EVAL, which permits interaction before and after evaluation. The break expression then corresponds to the argument to EVAL. For BREAKF and TRACE, BREAKEXPR is a form equivalent to that of the function being traced or broken. For errors, BREAKEXPR is the form which caused the error.
WHAT YOU CAN DO IN A BREAK

Break Commands

Once in a break, in addition to evaluating expressions, the user can ask BREAKFUNCTION to perform certain useful actions by giving it atomic items as "break commands". The following commands can be typed in by the user or may be put on the list BREAKCMDS. TABLE I is a summary of these commands.

GO

Releases the break and allows the computation to proceed. BREAKFUNCTION evaluates BREAKEXPR, its first argument, prints the value, and returns it as the value of the break. BREAKEXPR is the expression set up by the function that called BREAKFUNCTION. For BREAKF or TRACE, BREAKEXPR is equivalent to the body of the definition of the broken function. For the error package, BREAKEXPR is the expression in which the error occurred.

OK

Same as GO except that the value of BREAKEXPR is not printed.

EVAL

Causes BREAKEXPR to be evaluated. The break is maintained and the value of the evaluation is printed and bound on the variable BREAKVALUE. Typing GO or OK will not cause reevaluation of BREAKEXPR following EVAL, but another EVAL will. EVAL is a useful command when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.

RETURN form

The form is evaluated and its value is returned as the value of the break. For example, one might RETURN (REVERSE BREAKVALUE).

FROM form

This permits the user to release the break and return to a previous context with form to be evaluated. For details see context commands.

USE expr

For use either with UNDEFINED ATOM error or UNDEFINED FUNCTION error. Replaces the expression (using RPLACA, the change is permanent) containing the error with expr (not the value of expr) e.g.,

+ ***16 UNDEFINED ATOM
+ Q
+ 
* 1: CAR BROKEN
=USE XX
Changes Q to XX in the form (CAR Q), which, in the above example, caused the error.

Aborts the break. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL and RETURN commands, maintain the break.

This returns control directly to the top level of LISP.

ARGS
Prints the names and the current values of the arguments of the function at BREAKPOINTER. In most cases, these are the arguments of the broken function.

HELP
Causes a brief description of the break package commands to be listed on the user's terminal.

<FORM>
is EVALed if not a break command.
Context Commands

All information pertaining to the evaluation of forms in LISP is kept on the push down stack. Whenever a form is evaluated, the form is placed on the push down stack. Whenever a variable is bound, the old binding is saved on the push down stack. The context (the bindings of free variables) of a function is determined by its position in the stack. When a break occurs, it is often useful to explore the contexts of other functions on the stack. BREAKFUNCTION allows this by means of a context pointer, BREAKPOINTER, which is a pointer into the push down stack. BREAKFUNCTION contains commands to move the context pointer and to evaluate atoms or expressions relative to their positions in the stack. For the purpose of this document, when moving through the stack, "backward" is considered to be toward the top level or, equivalently, towards the older function calls on the stack.

F arg1 arg2 ... argN
Reset the variable BREAKPOINTER, which establishes a context for the commands USE, ARGS, AT, FROM and the backtrace commands described below. BREAKPOINTER is the position of a function call on the push down list. It is initialized to the function just before the call to BREAKFUNCTION.

F takes the rest of the line as its list of arguments.

Numbers
If positive, move BREAKPOINTER back (i.e. towards the top level) that number of calls. If negative, forward.

Example:
If the push-down stack looks like

<table>
<thead>
<tr>
<th>BREAKFUNCTION</th>
<th>(13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUN</td>
<td>(1)</td>
</tr>
<tr>
<td>FIE</td>
<td>(2)</td>
</tr>
<tr>
<td>PROG</td>
<td>(3)</td>
</tr>
<tr>
<td>COND</td>
<td>(4)</td>
</tr>
<tr>
<td>PIE</td>
<td>(5)</td>
</tr>
<tr>
<td>COND</td>
<td>(6)</td>
</tr>
<tr>
<td>PIE</td>
<td>(7)</td>
</tr>
<tr>
<td>COND</td>
<td>(8)</td>
</tr>
<tr>
<td>PROG</td>
<td>(9)</td>
</tr>
<tr>
<td>SETQ</td>
<td>(10)</td>
</tr>
<tr>
<td>SETQ</td>
<td>(11)</td>
</tr>
<tr>
<td>PROG</td>
<td>(12)</td>
</tr>
<tr>
<td>FIE</td>
<td>(13)</td>
</tr>
</tbody>
</table>

Then
F FIE COND will set BREAKPOINTER to (7)
F COND will then set BREAKPOINTER to (5)
F 2 will then move BREAKPOINTER to (3)
TOP will reset BREAKPOINTER to (12)
F can be used on BREAKCMDS. In that case, the next element of the list is treated as the list of arguments to F, (e.g. (F (FOO FIE FOO))

TOP

TOP repositions BREAKPOINTER to a stack position just prior to BREAKFUNCTION

EDIT arg1 arg2 ... argn

EDIT uses its arguments to reset BREAKPOINTER in the same manner as the F command. The form at BREAKPOINTER is then given to EDITF. This command can often save the user the trouble of calling EDITF and finding the expression that he needs to edit. If this command is not used, the editor need not be in memory. This command will cause an error (a new break level) if the editor is not in core when it is entered.

AT arg1 arg2 ... argn

This command is used to display the values of variables at position BREAKPOINTER. If the user types:

AT X (CAR Y)

he will see the value of X, and the value of (CAR Y). The difference between using AT and typing X and (CAR Y) directly into BREAKFUNCTION is that AT evaluates its inputs as of BREAKPOINTER. This provides a way of examining variables or forms at a particular point on the stack. For example,

F FOO 1 FOO
AT X

will allow the user to examine the value of X in an earlier call to FOO.

AT can also be used on BREAKCMDS, in which case the next element of the list on BREAKCMDS is treated as the list of arguments. For example, if BREAKCMDS is (EVAL AT (X (CAR Y)) GO), BREAKEXPR will be evaluated, the values of X and (CAR Y) printed, and then the function exited with its value being printed.

FROM [form]

FROM exits the break by undoing the push down stack back to BREAKPOINTER. If form is missing, re-evaluation continues with the form on the push down stack at BREAKPOINTER. If form is not missing, the function call on the push down stack at BREAKPOINTER is replaced by form and evaluation continues with form. Form is evaluated in the context of
BREAKPOINTER. There is no way of recovering the break because the push down stack has been undone. FROM, among other things, allows the user to return a particular value as the value of any function call on the stack. To return 1 as the value of the previous call to FOO:

```
:F FOO
:FROM 1
```

**Backtrace commands**

The backtrace commands print information about function calls on the push down list. The information is printed in reverse order to that in which calls were made. All backtraces start at BREAKPOINTER.

**BKF**

BKF gives a backtrace of the CARs of forms that are still pending.

**BKF**

BKE gives a backtrace of the expressions which called functions still pending (i.e. it prints the function calls themselves instead of only the names as in BKF).

**BK**

BK gives a full backtrace of all expressions still pending. It evaluates the form (DUMP 0). Output is in the form:

- function name
- list of arguments
- function name
- list of arguments

BKF and BKE may be followed by an integer. If the integer is included, it specifies how many blocks are to be printed. The limiting point of a block is a function call. This form is useful when working on a 3270. Using the integer feature in conjunction with the F command, which moves BREAKPOINTER, the user can display any contiguous part of the backtrace.

Example:

BKF would print the names of the functions called before BREAKPOINTER.
BREAK PACKAGE

How to set a Break

The following functions are useful for setting and unsetting breaks and traces.

Both BREAKF and TRACE use a function BREAKO to do the actual modification of function definitions.

BREAKF

BREAKF is an FLAMBDA. For each atomic argument, it breaks the function named each time it is called. For each list of the form (FN1 IN FN2), it breaks only those occurrences of FN1 which appear in FN2. This feature is very useful for breaking a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN FIE), etc. For each list not in this form, it assumes that the CAR is a function to be broken; the CADR is the break condition; (when the function is called, the break condition is evaluated. If it returns a non-NIL value, the break occurs. Otherwise computation continues without break.) and the CADDR is a list of command lines to be performed before an interactive break is made (see BREAKWHEN and BREAKCMDS of BREAKFUNCTION) or NIL. For example,

(BREAKF FOO1 (FOO2 (GREATERP N 5) (ARGS)))

will break all calls to FOO1 and all calls to FOO2 when N is greater than 5 after first printing the arguments of FOO2.

(BREAKF ((FOO4 IN FOO5) (MINUSP X) NIL))

will break all calls to FOO4 made from FOO5 when X is negative.

Examples:

(BREAKF FOO)
(BREAKF ((GET IN FOO) T (GO)))
(BREAKF ((SETQ IN FIE) (EQ N 1) ((PRINT (QUOTE N=1)) AT (M))))

TRACE

TRACE is an FLAMBDA. For each atomic argument, it traces the function named each time it is called. For each list in the form (FN1 IN FN2), it traces only those calls to FN1 that occur within FN2.

For example, (TRACE FOO1 (SETQ IN FOO3)) will cause both FOO1 and SETQ in FOO3 to be traced.

Note: The user can always call BREAKO himself to obtain combinations of options of BREAKFUNCTION not directly available with BREAKF and TRACE (see section on BREAKO below). These functions merely provide convenient ways of calling BREAKO, and will serve for most uses.
UNBREAK

UNBREAK is an FLAMBDA. It takes a list of functions modified by BREAKF or TRACE and restores them to their original state. Its value is NIL.

(UNBREAK T) will unbreak the function most recently broken.

(UNBREAK) will unbreak all of the functions currently broken (i.e. all those on BROKENFN).

If one of the functions fn is not broken, UNBREAK prints fn NOT BROKEN for that function and no changes are made to fn.

UNTRACE

UNTRACE is an FLAMBDA. It is the similar to UNBREAK.
BREAKO [FN WHEN COMS]

BREAKO is an EXPR. It sets up a break on the function FN by redefining FN as a call to BREAKFUNCTION with BREAKEXPR a form equivalent to the definition of FN, and WHEN, FN and COMS as BREAKWHEN, BREAKFN, and BREAKCMDS, respectively (see BREAKFUNCTION). BREAKO also adds FN to the front of the list BROKENFNS. Its value is FN.

If FN is non-atomic and of the form (fn1 IN fn2), BREAKO first calls a function which changes the name of fn1 wherever it appears inside of fn2 to that of a new function, fn1-IN-fn2, which is initially defined as fn1. Then BREAKO proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN PIE), etc. This only works in interpreted functions.
ERROR PACKAGE

Introduction

When an error occurs during the evaluation of a LISP expression, control is turned over to the Error Package. The idea behind the error package is that it may be possible to 'patch up' the form in which the error occurred and continue. Or, at least, that you can find the cause of the error more easily if you can examine the state of the world at the time of the error. Basically, what the Error Package does is call BREAKFUNCTION with BREAKEXPR set to the form in which the error occurred. This puts the user 'in a break' around the form in which the error occurred. BREAKFUNCTION acts just like the top level of the interpreter with some added commands (see section on BREAKFUNCTION). The main difference when in the Error Package is that the variable bindings that were in effect when the error occurred are still in effect. Furthermore, the expressions that were in the process of evaluation are still pending. While in the Error Package, variables may be examined or changed, and functions may be defined or edited just as if you were at the top level. In addition, there are several ways in which you can abort or continue from the point of error. In particular, if you can patch up the error, you can continue by typing OK. If you can't patch the error, will get you out of the break. When you are in the error package, the prompt character is '=' and is preceded by a level number. Note: if for some reason you don't want the error package invoked, it can be turned off by evaluating (ERRORPACKAGE OFF). Similarly (ERRORPACKAGE ON) will turn the error package back on.
<table>
<thead>
<tr>
<th>Command</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GO</strong></td>
<td>Evaluates BREAKEXPR, prints its value, and continues with this value</td>
</tr>
<tr>
<td><strong>OK</strong></td>
<td>Same as GO but no print of value</td>
</tr>
<tr>
<td><strong>EVAL</strong></td>
<td>Re-evaluate BREAKEXPR and print its value. Its value is bound to BREAKVALUE</td>
</tr>
<tr>
<td><strong>RETURN xx</strong></td>
<td>Evaluate xx and continue with its value</td>
</tr>
<tr>
<td><strong>l</strong></td>
<td>Escape one level of BREAKFUNCTION</td>
</tr>
<tr>
<td><strong>ll</strong></td>
<td>Escape to the top level</td>
</tr>
<tr>
<td><strong>USE expr</strong></td>
<td>After an error, use expr for the erring atom</td>
</tr>
<tr>
<td><strong>FROM form</strong></td>
<td>Continues by re-evaluating form at BREAKPOINTER</td>
</tr>
<tr>
<td><strong>F a1..aN</strong></td>
<td>Resets BREAKPOINTER (stack context)</td>
</tr>
<tr>
<td><strong>TOP</strong></td>
<td>Sets BREAKPOINTER to the function just before BREAKFUNCTION.</td>
</tr>
<tr>
<td><strong>EDIT a1..AN</strong></td>
<td>Resets BREAKPOINTER and gives the form at BREAKPOINTER to the LISP Editor</td>
</tr>
<tr>
<td><strong>AT f1..FN</strong></td>
<td>Evaluates form fi as of BREAKPOINTER</td>
</tr>
<tr>
<td><strong>ARGS</strong></td>
<td>Prints arguments of the function at BREAKPOINTER</td>
</tr>
<tr>
<td><strong>BKF</strong></td>
<td>Backtrace Function Names</td>
</tr>
<tr>
<td><strong>BKE</strong></td>
<td>Backtrace Function Calls</td>
</tr>
</tbody>
</table>
Note: BKE and BKF can be followed by an integer. The integer will limit the trace to that number of blocks. For example, BK, BKE, BKF 5 and BKE 4 are all legitimate commands.
A SAMPLE DEBUGGING SESSION

The following is a somewhat clumsy attempt to write a program which returns a list of all the atoms which are not system functions in a given structure. The error package, break package, and editor are used to debug the program.

*<DEFUN ATMS (X) (ATOMCOLLECT (GET XX EXPR)
> ATMS
*<DEFUN ATOMCOLLECT (MAPC '<LAMBDA (X) (COND < ((ATOM X)
*(COND ((GETL X (FSUBR NSUBR)) NILL) ((T (SETQ ATOMS
*(CONS X ATOMS) (T (ATOMCOLLECT X) X)
> ATOMCOLLECT
*(PRETTYPRINT ATMS ATOMCOLLECT)
> (LAMBDA (X)
> (ATOMCOLLECT (GET XX EXPR)))
> (LAMBDA
> (MAPC
> (QUOTE
> > (LAMBDA (X)
> > (COND
> > (((ATOM X)
> > (COND ((GETL X (FSUBR NSUBR)) NILL)
> > (T
> > (SETQ ATOMS (CONS X ATOMS)))))
> > (T (ATOMCOLLECT X))))
> > X))
> > NIL

We enter the program and prettyprint it. The error package is on so we try the program.

*(ATMS ATOMCOLLECT)
+ + ***16 UNDEFINED ATOM + ATOMCOLLECT
1: ATMS BROKEN

There is an error. Because ATMS is an EXPR, its argument is evaluated. We use the error package to look at the state of the world and decide to use the editor to change ATMS to an NEXPR.

= BKF
1: ATMS
?? ** TOP ** *(EDITF ATMS)
:EDITING ATMS:
:(LAMBDA (X) (ATOMCOLLECT &)) INSERT NLAMBDA FOR LAMBDA 1
:(NLAMBDA (X) (ATOMCOLLECT &)) OK
> NIL
We have changed the function ATMS with the editor and we try it again.

(*) (ATMS ATOMCOLLECT)
+  + ***16 UNDEFINED ATOM
+  XX

>1 : GET BROKEN
=BEK
>1 : (GET XX EXPR)
>1 : (ATOMCOLLECT (GET XX EXPR))
>1 : (ATMS ATOMCOLLECT)
=USE X
=BEK 1
>1 : (GET X EXPR)
=FROM

We encounter an error in the function ATMS. X was accidentally entered as XX. We use the BKE command to find out where we are in the execution, and the USE command to change the erring atom. We don't need to use the editor here. The BKE 1 is used to verify the change and the FROM command will restart the execution of the GET.

+  + ***16 UNDEFINED ATOM
+  EXPR

>1 : GET BROKEN
=USE 'EXPR
=BEK
>1 : (GET X (QUOTE EXPR))
>1 : (ATOMCOLLECT (GET X (QUOTE EXPR)))
>1 : (ATMS (ATOMCOLLECT)
=FROM

This time, we find that we forgot to quote the EXPR. The error message tells us that EXPR is being EVALed. Again, we can make use of the USE command to change the function and avoid having to use the editor. Note, the argument to USE need not be atomic. The FROM will restart us at the GET.

+  + ***3 TOO FEW ARGUMENTS
>1 : ATOMCOLLECT BROKEN
=ARGS
>1 : MAPC = (LAMBDA (MAPC (QUOTE (LAMBDA (X) (COND (((
>1 :ATOM X) (COND ((GETL X (FSUBR NSUBR)) NILL) ((T SETQ
>1 :ATOMS (CONS X ATOMS))))) )))) (T (ATOMCOLLECT X))) X))
>1 : (QUOTE (LAMBDA (X) (COND (((ATOM X) (COND ((GETL X (FIN:
>1 :FSUBR NSUBR)) NILL) ((T (SETQ ATOMS (CONS X ATOMS)))))
>1 :))) ) (T (ATOMCOLLECT X))) ) = (LAMBDA (X) (COND (((ATOM
>1 :X) (COND ((GETL X (FSUBR NSUBR)) NILL) ((T (SETQ ATOMS (AT:
>1 :CONS X ATOMS))))))) ) (T ATOMCOLLECT X)))
>1 : X = ATOMCOLLECT

Something is very wrong! We try to print the arguments to ATOMCOLLECT and get garbage. We use the editor again to fix the
After we insert the (X) into ATOMCOLLECT, we need to find a safe place to restart the computation. It would probably be ok to restart at the call to ATOMCOLLECT, but to be sure, we reposition BREAKPOINTER with the F command at the call to ATMS and use the FROM command to restart us there.

The parentheses in ATOMCOLLECT are very wrong. We try to use the BO command to fix them but use it incorrectly. We undo our mistake and try again.

```lisp
(EDITF ATOMCOLLECT)
:EDITING ATOMCOLLECT
:(LAMBDA (MAPC & X))
: (LAMBDA (X) (MAPC & X))
:OK
>1 : NIL
=BF
>1 : ATOMCOLLECT
>1 : ATMS
=F ATMS
=FROM

(EDITF ATOMCOLLECT)
:EDITING ATOMCOLLECT
:(LAMBDA (X) (MAPC & X))
: (LAMBDA (X) (MAPC & X))
:COND
: (COND (S) (T S))
:2
: (COND & 8))
:PP
: (COND (ATOM X)
: ((GETL X (FSUBR NSUBR)) NIL)
: (T (SETQ ATOMS (CONS X ATOMS))))))
:BO ATOM : PP
: (((ATOM X)
: (COND ((GETL X (FSUBR NSUBR)) NIL)
: (T (SETQ ATOMS (CONS X ATOMS))))))
:UNDO
:BO UNDONE

(EDITF ATOMCOLLECT)
:EDITING ATOMCOLLECT
:(LAMBDA (MAPC & X))
: (LAMBDA (X) (MAPC & X))
:OK
>1 : NIL
=BF
>1 : ATOMCOLLECT
>1 : ATMS
=F ATMS
=FROM

(EDITF ATOMCOLLECT)
:EDITING ATOMCOLLECT
:(LAMBDA (X) (MAPC & X))
: (LAMBDA (X) (MAPC & X))
:COND
: (COND (S) (T S))
:2
: (COND & 8))
:PP
: (COND (ATOM X)
: ((GETL X (FSUBR NSUBR)) NIL)
: (T (SETQ ATOMS (CONS X ATOMS))))))
```

The parentheses in ATOMCOLLECT are very wrong. We try to use the BO command to fix them but use it incorrectly. We undo our mistake and try again.
We figure out how to use the BO command and fix ATOMCOLLECT.

After fixing ATOMCOLLECT, we try to perform a BKF command but type it incorrectly. Since the break package EVALs all non-command input, KBF is EVALed and, of course, causes a new error break. The level number is now 2. Since this is only a typing mistake, we can pop back to the break we were working at (level 1) with the | command.

There is a great deal of garbage on the stack. We again need to find a safe place to restart the computation (we could always start from the beginning). We decide that the COND is a safe place so we reposition the BREAKPOINTER pointer there and return.
We forgot to quote the list (FSUBR NSUBR) so FSUBR is EVALed. We cannot use the USE command here because that would replace the CAR of the list (FSUBR NSUBR) which is not what we want. We use the editor to make the change. We cannot simply use the FROM command because the top of the stack has the form (FSUBR NSUBR) on it so we must restart from the GETL.

```
+ ***16 UNDEFINED ATOM
+ NIL
>1 : COND BROKEN
=USE NIL
=BKE
>1 : (COND ((GETL X (QUOTE (FSUBR NSUBR))) NIL) ((T (SETQ
>1 : ATOMS (CONS X ATOMS)))))
>1 : (COND ((ATOM X) (COND ((GETL X (QUOTE (FSUBR NSUBR)))
>1 : NIL) ((T (SETQ ATOMS (CONS X ATOMS))))) ) (T (n
>1 : ATOMCOLLECT X)))
+
+ ***1 ATTN
>2 : PRINT BROKEN
=
```

The above error was caused by incorrectly typing NIL. We change it with the USE command and look at it with the BKE command. We decide that the BKE will print out more information than we want to see so we interrupt it with the attention button. This causes a new break level. We pop back to the original error level with the | command. The break is on the COND and we can use the FROM command to restart.

```
=FROM
+ ***17 UNDEFINED FUNCTION
+ T
>1 : T BROKEN
=BKE 1
>1 : (T (SETQ ATOMS (CONS X ATOMS)))
=EDITF ATOMCOLLECT
:EDITING ATOMCOLLECT
:(LAMBDA (X) (COND &))
:PP
:(LAMBDA (X)
  (MAPC
We use the editor again to fix the parenthesis problem in ATOMCOLLECT. We use the BKF command to find a safe place to restart (note, ATOMCOLLECT is recursive, there are 2 calls on the stack). The COND is a safe place. We restart there.

=F COND
=FROM
+
+ ***16 UNDEFINED ATOM
+ ATMS
>1 : CONS BROKEN

This time, the problem is that in the function ATMS, we should have set ATOMS to NIL. We make the change to ATMS and set ATOMS to NIL and continue.

=(EDITF ATMS)
:EDITING ATMS
:(NLISTA NIL (MAPC & X))
.I (SETQ ATOMS NIL) AFTER 2 : PP
:(NLISTA (X)
: (SETQ ATOMS NIL)
We have fought our way through the program and found many bugs. The program terminates but does not give the right answer. The function CONS should not be on the list but it is. To find out why, we set a breakpoint on the call to GETL inside ATOMCOLLECT. We want execution to stop when GETL has argument CONS.

\[(\text{BREAKF (\text{GETL IN ATOMCOLLECT) (EQ X 'CONS) NIL))})\]

When the breakpoint is reached, execution is stopped. We evaluate X to be sure that we are at the place we want to be (it has the value CONS which is what we wanted) and we evaluate the GETL. It returns NIL which is not correct. The reason is that we forgot to check for SUBRs. We use the editor to fix the program. We can now remove the breakpoint and continue.

\[(\text{UNBREAK})\]

The list is still wrong but we suspect that the incorrect atoms were placed on it before the break we set. We evaluate the functions again.

\[(\text{ATOMS ATOMCOLLECT} \text{ ATOMS})\]
All seems well. To make sure, we decide to trace the program on a small structure so we can follow its execution.

*(TRACE ATOMCOLLECT)
> DONE
*(ATMS ATMS)
>1 : -> ATOMCOLLECT
>1 : X = (NLAMBDA (X) (SETQ ATOMS NIL) (ATOMCOLLECT (GET X (QUOTE EXPR))))
>2 : -> ATOMCOLLECT
>2 : X = (X)
>2 : <- ATOMCOLLECT = NIL
>2 : -> ATOMCOLLECT
>2 : X = (SETQ ATOMS NIL)
>2 : <- ATOMCOLLECT = NIL
>2 : -> ATOMCOLLECT
>2 : X = (ATOMCOLLECT (GET X (QUOTE EXPR)) )
>3 : -> ATOMCOLLECT
>3 : X = (GET X (QUOTE EXPR))
>4 : -> ATOMCOLLECT
>4 : X = (QUOTE EXPR)
>4 : <-ATOMCOLLECT = NIL
>3 : <-ATOMCOLLECT = NIL
>2 : <-ATOMCOLLECT = NIL
>1 : <-ATOMCOLLECT = NIL
> NIL

The program works now.
BREAKPACKAGE INTERNAL FUNCTIONS

The following is a description of the major functions which comprise the break package.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK0</td>
<td>See description of BREAK0 under BREAK PACKAGE.</td>
</tr>
<tr>
<td>BREAKONE</td>
<td>performs one break command. The code for most of the break commands can be found in the SELECTQ which is the body of this function.</td>
</tr>
<tr>
<td>BREAKBT</td>
<td>performs the operations for the backtrace commands BKF and BKE.</td>
</tr>
<tr>
<td>BREAKFUNCTION</td>
<td>performs the setup operations for the break and error package. It can be thought of as the top level function when a break is in effect. It is also the READ EVAL PRINT loop of the break package.</td>
</tr>
<tr>
<td>BREAKEVAL</td>
<td>is the break package EVAL function for the EVAL and EVALT commands.</td>
</tr>
<tr>
<td>BREAKLEVEL</td>
<td>used by TOP and when the break package is first entered. Sets BREAKPOINTER.</td>
</tr>
<tr>
<td>BREAKFF</td>
<td>performs the F command operations moving BREAKPOINTER.</td>
</tr>
<tr>
<td>BREAKEVALAT</td>
<td>used both by the AT command and by ARGS. It simulates the evaluation of forms using the values from the function DISPLAY instead of the variable values.</td>
</tr>
<tr>
<td>BREAKARGS</td>
<td>performs the ARGS command operations.</td>
</tr>
<tr>
<td>ERRORPACKAGE</td>
<td>sets the error forms to the call to BREAKFUNCTION and saves the old error forms on BREAK.</td>
</tr>
<tr>
<td>BREAKSUB</td>
<td>used by BREAKF, TRACE, UNBREAK and UNTACE for modifying the function definitions.</td>
</tr>
<tr>
<td>UNBREAK</td>
<td>together with UNBREAK1 reverses the actions of BREAKF and TRACE. It uses BREAKSUB to undo the structural changes made to the function definitions.</td>
</tr>
<tr>
<td>TRACE</td>
<td>like BREAKF changes the function definitions to be calls to BREAKFUNCTION. TRACE includes items to be evaluated which print trace information and continue.</td>
</tr>
<tr>
<td>BREAKEXCISE</td>
<td>removes the break and error package functions from memory making the space available for</td>
</tr>
</tbody>
</table>
other uses.

The break and error package function may be found in the file CS:BREAK.
PRETTYPRINTING AND PROGRAM FILES

Four functions are provided for prettyprinting and for reading and writing files which contain user functions.

PRETTYPRINT

This is an FLAMBDA which looks on the property list of its atomic arguments for an EXPR, MACRO, or BUG and prettyprints them on *SOURCE*.

Example:

*(PRETTYPRINT FOO FIE)*

>(LAMBDA (X))

>(CONS X X))

>(LAMBDA (Y))

>(CAR (CADR Y))

> NIL

The two functions FOO and FIE are prettyprinted.

DSKIN

This function is an FLAMBDA. Its arguments are MTS file names. For each argument, all s-expressions are read from that file and EVALed. For each function definition, (DEFUN, DEFPFUNC, or DEFINE) the function name is placed on the property list of the file name. For each s-expression which is not a function definition, the s-expression is also placed on the property list of the file name. After a DSKIN there is under the property FUNCTIONS the list of functions and s-expressions read so that UPDATE can rebuild the file.

Example:

*(DSKIN FOOFILE)*

> DONE

*(GET 'FOOFILE 'FUNCTIONS)*

> (FOO FIE (SETQ C NIL) F0027).

In the above example, 3 functions and an s-expression were read from the file FOOFILE. All were EVALed.

DSKOUT

DSKOUT is an FLAMBDA. It takes as arguments a file name and a list of function names and s-expressions. DSKOUT causes the functions named and the s-expressions to be prettyprinted into the named file.

Example:

*(DSKOUT FOOFILE FIE FOO (PUT 'A 'B 'C) BAR (SETQ 'A NIL))*

> DONE

The file FOOFILE will contain

1. The definition of FIE.
2. The definition of FOO.
3. The PUT.
4. The definition of BAR.
5. The SETQ.

UPDATE

This function is used when a file was read by DSKIN and the functions or s-expressions have been changed or extended. It takes as arguments a file specified in a DSKIN and a list of function names and s-expressions to be included with the original file. For file protection, the functions and s-expressions are first prettyprinted to the file -LISPTEMP then copied to the specified file.

Example

*(UPDATE POOFILE (SETQ A NIL))
#EMPTY -LISPTEMP
#DONE
#EMPTY POOFILE
#FILE "POOFILE" IS TO BE EMPTIED, PLEASE CONFIRM.
#OK
#COPY -LISPTEMP TO POOFILE
> DONE

Poofile now contains the updated functions and the SETQ.

These functions are found in CS:DSKIO and are read in with the editor.
BIBLIOGRAPHY

(PROGN
  (DEFPROP BREAK IN (BREAK ARG)
    (QUOTE)
    (LAMBDA (BREAK ARG)
      (COND
        (OR (ATOM BREAK ARG) (EQ (CADR BREAK ARG) (QUOTE IN)))
        (BREAK ARG T NIL))
        (T (BREAK ARG (CADR BREAK ARG) (CADDR BREAK ARG)))))
  (BREAK ARG)
  (QUOTE DONE))
  (DEFPROP BREAK OUT (BREAK ARG)
    (LAMBDA (BREAK ARG BREAK WHEN BREAK CMD)
      (PROG (BREAK A BREAK B BREAK C BREAK D)
        (COND
          ((HEAQ BREAK FN (GET (QUOTE BROKEN FN) (QUOTE BROKEN FN)))
            (PRIN1 BREAK FN)
            (PRIN1 (QUOTE "ALREADY BROKEN"))
            (RETURN (TERPRI)))
          (PUT (QUOTE BROKEN FN)
            (QUOTE BROKEN FN)
            (CONS BREAK FN (GET (QUOTE BROKEN FN) (QUOTE BROKEN FN))))
          (ATOM BREAK FN)
          (SETQ BREAK A
            (GETL BREAK FN
              (QUOTE (EXPR MACRO BUG))
              (QUOTE
                (PROGN (PRIN1 BREAK FN)
                  (PRIN1 (QUOTE "NOT A FUNCTION"))
                  (RETURN (TERPRI)))))
            (RPLACA (CDDADR BREAK A)
              (LIST (QUOTE BREAK FUNCTION)
                (CONS (QUOTE PROGN)
                  (CONS (CAR (CDDADR BREAK A))
                    (CDR (CDDADR BREAK A))))
                BREAK WHEN
                BREAK CMD)
            (QUOTE BREAK))
          (RPLACD (CDDADR BREAK A) NIL))
          (EQ (CADR BREAK FN) (QUOTE IN))
          (SETQ BREAK C
            (FADOK (CAR BREAK FN) (QUOTE IN) (CADDR BREAK FN))
            (CDDADR BREAK A)
            (CAR BREAK FN))
          (COND
            ((NULL BREAK ARG)
              (PRIN1 (QUOTE NO))
              (PRIN1 (CAR BREAK ARG))))
        (COND
          ((ATOM BREAK ARG) (EQ (CADR BREAK ARG) (QUOTE IN))
            (BREAK ARG T NIL))
          (T (BREAK ARG (CAD R BREAK ARG) (CADDR BREAK ARG)))))))
  (BREAK ARG)
  (QUOTE DONE))
  (DEFPROP BREAK OUT (BREAK ARG)
    (LAMBDA (BREAK ARG BREAK WHEN BREAK CMD)
      (PROG (BREAK A BREAK B BREAK C BREAK D)
        (COND
          ((HEAQ BREAK FN (GET (QUOTE BROKEN FN) (QUOTE BROKEN FN)))
            (PRIN1 BREAK FN)
            (PRIN1 (QUOTE "ALREADY BROKEN"))
            (RETURN (TERPRI)))
          (PUT (QUOTE BROKEN FN)
            (QUOTE BROKEN FN)
            (CONS BREAK FN (GET (QUOTE BROKEN FN) (QUOTE BROKEN FN))))
          (ATOM BREAK FN)
          (SETQ BREAK A
            (GETL BREAK FN
              (QUOTE (EXPR MACRO BUG))
              (QUOTE
                (PROGN (PRIN1 BREAK FN)
                  (PRIN1 (QUOTE "NOT A FUNCTION"))
                  (RETURN (TERPRI)))))
            (RPLACA (CDDADR BREAK A)
              (LIST (QUOTE BREAK FUNCTION)
                (CONS (QUOTE PROGN)
                  (CONS (CAR (CDDADR BREAK A))
                    (CDR (CDDADR BREAK A))))
                BREAK WHEN
                BREAK CMD)
            (QUOTE BREAK))
          (RPLACD (CDDADR BREAK A) NIL))
          (EQ (CADR BREAK FN) (QUOTE IN))
          (SETQ BREAK C
            (FADOK (CAR BREAK FN) (QUOTE IN) (CADDR BREAK FN))
            (CDDADR BREAK A)
            (CAR BREAK FN))
          (COND
            ((NULL BREAK ARG)
              (PRIN1 (QUOTE NO))
              (PRIN1 (CAR BREAK ARG))))
        (COND
          ((ATOM BREAK ARG) (EQ (CADR BREAK ARG) (QUOTE IN))
            (BREAK ARG T NIL))
          (T (BREAK ARG (CAD R BREAK ARG) (CADDR BREAK ARG)))))))
(PRIN1 (QUOTE IN))
(PRIN1 (CADDR BREAKFN))
(RETURN (TERPRI))))
(SELECTQ
(CAR
(SETO BREAKA
(GETL (CAR BREAKFN)
(QUOTE (EXPR SUBR MACRO BUG FSUBR NSUBR))
(QUOTE
(PROGN (TERPRI)
(PRIN1 (CAR BREAKFN))
(PRIN1 (QUOTE " NOT FOUND")
(RETURN (TERPRI)))))))

((EXPR MACRO BUG)
(PUT BREAKC
(CAR BREAKA)
(CONS (CAADR BREAKA)
(CONS (CADDR BREAKA)
(CONS (CAR (CDADDR BREAKA)) (CDR (CDADDR BREAKA))))))

(SUBR BREAKD
(SETO BREAKA
(LIST (QUOTE LAMBDA)
(QUOTE BREAKDUMMY)
(CONS (COND ((ATOM BREAKB) BREAKB) (T (CAR BREAKB))) NIL))

(REPEAT
(QUOTE
(PROGN (SETQ BREAKFN (ADD1 BREAKFN))
(KCONC (CADR BREAKB) (LIST (LIST (QUOTE ARG) BREAKFN))))
(SUB1 (LENGTH BREAKB))))

(T
(SETO BREAKCMDS
(CONS
(QUOTE
(PROGN (SETQ BREAKA (CADR BREAKEXPR) BREAKB 0)
(REPEAT
(QUOTE
(PROGN (SETQ BREAK (ADD1 BREAKB))
(NCONC BREAKA
(LIST (LIST (QUOTE ARG) BREAKB))))
(SUB1 BREAKDUMMY))))

(BREAKCMDS)))))

(PUT BREAKC (QUOTE EXPR) BREAKD))

((SUBR FSUBR)
(PUT BREAKC
(QUOTE EXPR)
(LIST (QUOTE FLAMBDA)
(QUOTE BREAKDUMMY)
(LIST (QUOTE APPLY)
(LIST (QUOTE QUOTE) (CAR BREAKFN)
(QUOTE BREAKDUMMY)))))

(WIL)
(BREAKO BREAKC BREAKWHEN BREAKCMDS))

(T
(KAPC
(QUOTE (LAMBDA (BREAKARG) (BREAKO BREAKARG BREAKWHEN BREAKCMDS))
BREAKFN)))))
```lisp
(lselect (car breakcmds)
  (cond ((null (cdr breakcmds))
    (print (quote "RETURN WHAT?") breakout))
    (t (setq breaka (eval (car breakcmds)))))
  (uneval (quote breakfunction) (list (quote quote) breaka))))
(uneval (quote breakfunction) (quote
  (cond ((null (uneval (quote breakone)))
    (uneval -1 (quote (quote "** TOP **")))))
  (t (uneval -1 (quote (quote "** TOP **")))))
  (from (cond ((null (cdr breakcmds))
    (print (quote " USE WHAT?") breakout))
    (t (setq breakcmd (cdr breakcmds))))))
(usecond `(null (cdr breakcmds))
  (print (quote "USE LEGAL WITH UNDEFINED ATOM OR UNDEFINED FUNCTION ONLY")
    breakout)))))
(setq breakcmds (cdr breakcmds))
(ARGS (setq breaka (uneval breakpointer))
  (cond ((atom breaka) (breakargs breaka))
    (t (print (quote "FUNCTION NAME NOT ATOMIC") breakout))))
(f (breaks)
  (cond (breakcheck (setq breakcmds nil))
    (t (setq breakcmds (cdr breakcmds)))
    (top (breaklevel))
    (edit (breakff))
    (edit (uneval breakpointer)))
(cond (breakecho (setq breakcmds nil))
  (t (setq breakcmds (cdr breakcmds))))

(quote
  (lambda (breakarg)
    (print (breakevalat breakarg) breakout)))

(cond (breakecho (cdr breakcmds))
  (t ((setq breakcmds (cdr breakcmds)))))

(quote
  (lambda (breakarg)
    (print (breakevalat breakarg) breakout)))

(cond (breakecho (setq breakcmds nil))
  (t (setq breakcmds (cdr breakcmds)))

(mapc (quote
  (lambda (breakarg)
    (print (breakevalat breakarg) breakout))

  (cond (breakecho (cdr breakcmds))
    (t (setq breakcmds (cdr breakcmds))))

(bkf)

(cosd
  (setq breakcmds (cdr breakcmds))
  (breakbt (cadr breakcmds) (quote car))
  (setq breakcmds (cdr breakcmds)))

(t (breakbt (sub 0 breakpointer) (quote car))))

(bkf)

(cosd
  (setq (numberp (cadr breakcmds))
    (breakbt (cadr breakcmds) (quote nconc))
    (setq breakcmds (cdr breakcmds)))

(t (breakbt (sub 0 breakpointer) (quote nconc))))

(bk (dump 0))

(help (quote "COPY CS:break.help")

(cond (breakecho (print (eval (car breakcmds)) breakout)))

(defprop breakbt expr
  (lambda (breakarg breakdo)
    (prog nil
      (setq breaka breakpointer)
      (repeat
        (quote
          (prog
            (cond
              (atom (setq breakb (uneval breaka)))
              (print breakb breakout))
            (t (print (apply 1 breakdo breakarg) breakout)))
            (setq breaka (add 1 breaka))))
            breakarg))

(defprop breakfunction expr
  (lambda (breakexpr breakexpr breaktype)
    (and (eq breaktype (quote break))
      (not (eval breakvalue))
      (return (eval breakexpr)))
      (prog
        (breaklevel breaka breakb breakc breaktab breakecho breakvalue breakflag breakpointer)
        (setq breaktab 1)
        (and (eq breaktype (quote expr)) (dump 1))
        (setq breaklevel
          (cond
            ((eq (setq breaka (display breakfunction expr breaklevel))
              (quote *undef*))))

(deffunc breakexpr expr
  (lambda (breakarg breakdo)
    (prog nil
      (setq breaka breakpointer)
      (repeat
        (quote
          (prog
            (cond
              (atom (setq breakb (uneval breaka)))
              (print breakb breakout))
            (t (print (apply 1 breakdo breakarg) breakout)))
            (setq breaka (add 1 breaka))))
            breakarg))

(defprop breaktype expr
  (lambda (breakarg breakdo)
    (prog nil
      (setq breaka breakpointer)
      (repeat
        (quote
          (prog
            (cond
              (atom (setq breakb (uneval breaka)))
              (print breakb breakout))
            (t (print (apply 1 breakdo breakarg) breakout)))
            (setq breaka (add 1 breaka))))
            breakarg))

(deffunc breakfunc expr
  (lambda (breakarg breakdo)
    (prog nil
      (setq breaka breakpointer)
      (repeat
        (quote
          (prog
            (cond
              (atom (setq breakb (uneval breaka)))
              (print breakb breakout))
            (t (print (apply 1 breakdo breakarg) breakout)))
            (setq breaka (add 1 breaka))))
            breakarg))
(DEFPROP BREAKLEVEL EXPR
  (LAMBDA NIL
    (COND
      ((EQ (SETQ BREAKA (DISPLAY BREAKFUNCTION BREAKPOINTER))
         (QUOTE *UNDEFINED*))
       (SETQ BREAKA 0))
      ((NOT (NUMBERP BREAKA))
       (SETQ BREAKA 0)))
      (REPEAT (QUOTE
        (PROGN (SETQ BREAKA (SUB1 BREAKA))
          (CAR (UNEVAL (SUB1 BREAKA))))
        1000)
      (SETQ BREAKPOINTER (COND ((EQ BREAKTYPE (QUOTE ERROR))
                      (ADD1 BREAKA))
                      (T BREAKA))))))
(DEFPROP BREAKFF EXPR
  (LAMBDA NIL
    (SETQ BREAKA BREAKPOINTER)
    (TAPC
      (QUOTE
        (LAMBDA (BREAKARG)
          (COND
            ((NUMBERP BREAKARG)
             (COND
              ((NOT (MINUSP (SETQ BREAKB (ADD BREAKPOINTER BREAKARG))))
               (PRINT (QUOTE "NUMBER TAKES YOU BEYOND THE TOP LEVEL")
                      BREAKOUT)
              (UNEVAL (QUOTE BREAKFF) NIL))
              (T (SETQ BREAKA BREAKARG))))))
            (T
             (PROG NIL
                (COND
                  ((NULL (UNEVAL BREAKA))
                   (PRINT BREAKARG BREAKOUT)
                   (PRINT (QUOTE "NOT FOUND") BREAKOUT)
                   (TERPRI BREAKOUT)
                   (UNEVAL (QUOTE BREAKFF) NIL))))))
((EQ BREAKARG (UNEVAL BREAKA)) (RETURN NIL)))

((EQ BREAKARG (CAR (UNEVAL BREAKA))) (RETURN NIL)))

(SETQ BREAKA (ADD1 BREAKA))

(GO LOOP)))

(DEFPROP BREAKEVALAT EXPR)

(LAMBDA (BREAKARG)

(SETQ BREAKINTER BREAKA)))

(DEFPROP BREAKEVALAT EXFR

(LAMBDA (BREAKARG)

(COND ((BREAKECHO (CDR BEXACKCMDS))

(T (CADR BEXACKCMDS)))))

(DEFPROP BREAKARGS EXPR

(LAMBDA (BREAKARG)

(SETQ BREAKB

(CADADS

(GETL BREAKARG

(QUOTE (EXPR  MACRO BOG))

(QUOTE

(PSIN1  (QUOTE "NO EXPR  MACRO OB BUG FOR") BREAKOUT)

(PR IN 1 BREAKARG BREAKOUT)

(TERPRI  BREAKOUT)

(UNEVAL (QUOTE BREAKONE)  NIL))))))))

(COND ((ATOM BEXAKN)

(SETQ BEXAKC O)

(REPEAT

(QUOTE

(PROG (SETQ BEXAKC (ADD1 BEXAKC))

(TAG BREAKTAB BREAKOUT)

(PRIN1 (QUOTE "ARG #") BREAKOUT)

(PRIN1 BREAKA BREAKOUT)

(PRIN1 (QUOTE "=" ) BREAKOUT)

(PRIN1 (BREAKEVALAT (LIST (QUOTE ARG) BREAKC BREAKS))

(BREAKOUT)

(TERMPI BREAKOUT)))))))

(UNEVAL (QUOTE BREAKONE)  NIL))))

(BREAKEVALAT BREAKA)))

(T (MAPC

(PROG (SETQ BEXAKC (ADD1 BEXAKC))

(TAG BREAKTAB BREAKOUT)

(PRIN1 (QUOTE "ARG #") BREAKOUT)

(PRIN1 BREAKA BREAKOUT)

(PRIN1 (QUOTE "=" ) BREAKOUT)

(PRIN1 (BREAKEVALAT (LIST (QUOTE ARG) BREAKC BREAKS))

(BREAKOUT)

(TERMPI BREAKOUT)))))))

(SETQ BEXACKCMDS)

(T (CADR BEXACKCMDS)))))

(DEFPROP BREAKEVALAT EXPR)

(LAMBDA (BREAKARG)

(SETQ BREAKINTER BREAKA)))

(DEFPROP BREAKEVALAT EXFR

(LAMBDA (BREAKARG)

(COND ((BREAKECHO (CDR BEXACKCMDS))

(T (CADR BEXACKCMDS)))))

(DEFPROP BREAKARGS EXPR

(LAMBDA (BREAKARG)

(SETQ BREAKB

(CADADS

(GETL BREAKARG

(QUOTE (EXPR  MACRO BOG))

(QUOTE

(PSIN1  (QUOTE "NO EXPR  MACRO OB BUG FOR") BREAKOUT)

(PR IN 1 BREAKARG BREAKOUT)

(TERPRI  BREAKOUT)

(UNEVAL (QUOTE BREAKONE)  NIL))))))))

(COND ((ATOM BEXAKN)

(SETQ BEXAKC O)

(REPEAT

(QUOTE

(PROG (SETQ BEXAKC (ADD1 BEXAKC))

(TAG BREAKTAB BREAKOUT)

(PRIN1 (QUOTE "ARG #") BREAKOUT)

(PRIN1 BREAKA BREAKOUT)

(PRIN1 (QUOTE "=" ) BREAKOUT)

(PRIN1 (BREAKEVALAT (LIST (QUOTE ARG) BREAKC BREAKS))

(BREAKOUT)

(TERMPI BREAKOUT)))))))

(UNEVAL (QUOTE BREAKONE)  NIL))))

(BREAKEVALAT BREAKA)))

(T (MAPC
(QUOTE
  (LAMBDA (BREAKARG)
    (PRIN1 BREAKARG BREAKOUT)
    (PRINT (QUOTE " = ") BKFAKOUT)
    (PRIN1 (BREAKLEVEL BREAKARG) BREAKOUT)
    (TERPRI BREAKOUT)))

(DEFPROP BREAKIO EXPR
  (LAMBDA NIL
    (STATUS (8 BREAKOUT NIL) (13 BREAKIN =))
    (TAB 1 BREAKOUT)
    (PRINT BREAKOUT)
    (PRINT 1 BREAKOUT)
    (PRIN1 BREAKLEVEL BREAKOUT)
    (PRIN 1 (QUOTE BREAKINTERCEPT) BREAKOUT)
    (DEFPROP BREAKINTERCEPT EXPR
      (LAMBDA (BREAKARG)
        (QUOTE BREAKFLAC*))
      (DEFPROP BREAKREAD EXPR
        (LAMBDA NIL
          (PROG (BREAKA BREAKB)
            (LOOP (COND
                ((EQ (SETQ BREAKA (READ BREAKIN)) (QUOTE *BREAKFLAG*))
                 (RETURN BREAKB))
                (T (SETQ BREAKB (NCONC BREAKB (LIST BREAKA))) (GO LOOP))))))
      (DEFPROP ERRORPACKAGE EXPR
        (FLAMBDA (BREAKA)
          (PROG (BREAKB)
            (COND
              ((EQ (CAR BREAKA) (QUOTE ON))
                (SETQ BREAKA 0)
                (REPEAT
                  (QUOTE (SETQ BREAKB (CONS (APPLY 1 (QUOTE STATUS) (LIST 1 (SETQ BREAKA (SD01 BREAKA)) (QUOTE BREAKERROR)) BREAKB))))
                  39)
                (PUT (QUOTE BREAKF) (QUOTE BREAKF) BREAKB) NIL)
              ((EQ (CAR BREAKA) (QUOTE OFF))
                (SETQ BREAKA 10)
                (MAPC (QUOTE (LAMBDA (BREAKB)
                                (APPLY 1 (QUOTE STATUS) (LIST 1 (SETQ BREAKA (SUB1 BREAKA)) BREAKB))))
                      (GET (QUOTE BREAKF) (QUOTE BREAKF))))
              (T (PRINT (QUOTE "WHAT?"))))))))
      (DEFPROP BREAKSUB EXPR
        (LAMBDA (BREAKTO BREAK1NX BREAKFROM)
          (COND
            ((ATOM BREAKINX) NIL)
            ((EQ (CAR BREAKINX) BREAKFROM)
              (COND
                ((NULL BREAKTO) (CONS (CAR BREAKINX) (CDR BREAKINX)))
                ((ATOM BREAKTO)
                  (CONS (LENGTH BREAKINX) (LENGTH BREAKINX))
                  (T (SETQ BREAKTO BREAKIFROM))))))
          (PRINT BREAKOUT)
          (PRINT BREAKOUT)
          (PRINT BREAKOUT)
          (PRINT BREAKOUT)))
  )
(DEFPROP UNBREAK1 EXPR
    (LAMBDA (BREAKARG)
        (PROG (BREAKA BREAKB)
            (COND
                ((ATOM BREAKARG)
                    (AND
                        (KEQ (QUOTE BREAKFUNCTION)
                            (CAAR BREAKA)
                            (SETQ BREAKA
                                (CDR BREAKA))
                        (GETL BREAKARG
                            (QUOTE (EXPR MACRO BUG))
                            (QUOTE
                                (PROGN (PR1 BREAKARG)
                                    (PRINT (QUOTE " NOT A FUNCTION")
                                    (TESPRI)
                                    (RETURN NIL))))))
                (PRIH1 BREAKARG)
                (PRIH1 (QUOTE " NOT BROKEN"))
                (TESPRI)
                (RETURN NIL))
                (EQ (CADR BREAKARG) (QUOTE IN))
                (BREAKSUB (CAR BREAKARG)
                    (GETL (CADDR BREAKARG)
                        (QUOTE (EXPR MACRO BUG))
                        (QUOTE
                            (PROGN (PRI1 (CADDR BREAKARG))
                                (PRINT (QUOTE " NOT A FUNCTION")
                                    (TESPRI)
                                    (RETURN NIL))))
                        (MKTORH (CAR BREAKARG) (QUOTE -IN-) (CADDR BREAKARG))))
                (T (PRI1 BREAKARG) (PRIH1 (QUOTE ?)) (TESPRI)))
            )))
    (DEFPROP UNBREAK EXPR
        (FLAMBDA (BREAKARG)
            (PROG NIL
                (COND
                    ((NULL BREAKARG)
                        (MAPC (QUOTE UNBREAK1)
                            (GET (QUOTE BROKENFN) (QUOTE BROKENPK))))
                    (EQ (CADDR BREAKARG) T)
                        (UNBREAK (CADDR BREAKARG) (QUOTE BROKENFN))))
                (PUT (QUOTE BROKENFN)
                    (QUOTE BROKENFN)
                    (CDR (GET (QUOTE BROKENFN) (QUOTE BROKENFN))))
                (T
                    (MAPC (QUOTE UNBREAK1)
                        (LABEL UNBREAK1
                            (LAMBDA (BREAKARG)
                                (GET (QUOTE BROKENFN)
                                    (QUOTE BROKENFN)
                                    (QUOTE)
                                    (PROG (PR1 BREAKARG)
                                        (PRINT (QUOTE " NOT BROKEN")
                                            (TESPRI)
                                            (RETURN NIL))))))))
            )))
    )
(defprop trace expr
  (lambda (breakarg)
    (prog (breaka)
      (open (breakin 60 #resource*) (breakout 80))
      (setq breaka
        (quote (((setq breaktau
            (add (remain (times (sub1 breaklevel) 2) 30) 4))
          (tab (sub breaktab 3) breakout)
          (print (quote "->") breakout)
          (print breakfn breakout)
          (terpri breakout)
          args
          evalt
          (tab (sub breaktab 3) breakout)
          (print (quote "<-") breakout)
          (print breakfn breakout)
          (print (quote "=") breakout)
          (print breakvalue breakout)
          (terpri breakout)
          ok))
        (mapc
          (quote
            (lambda (breakarg)
              (cond ((atom breakarg) (breako breakarg t breaka))
                ((eq (cadr breakarg) (quote in))
                  (breako breakarg t breaka))
                (t
                  (breako (car breakarg)
                    (cadr breakarg)
                    (nconc (caddr breakarg) breaka))))))))
      breakarg)
    (quote done))))
  (lambda (exprs)
    (eval (cons (quote unbreak) breakarg))
    (quote done)))
  (lambda (expr)
    (errorpackage off)
    (rem (quote
      (break breako
        breakone
        break
        breakfunction
        breakval
        breaklevel
        breakff
        breakevalat
        breakarg)
      (quote
        (print (quote "not broken"))
        (tempri)
        (return nil))))))
  (quote done)))
(ERRORPACKAGE ON)
"*BREAK LOADED"

(QUOTE BREAKERROR)
(QUOTE (£VAL (LIST  (QUOTE BREAKFUNCTION)
 (PROGN (OPEN  (BREAKIN 80 *KSOURCE*) (BREAKOUT 80))
 (STATUS 53))
 T
 (COND ({ATOM (UNEVAL 5)) (UNEVAL 9})
 [T (CIR (UNEVAL 5))]))
NIL
(QUOTE ERROR)))))

(QUOTE (QUOTE (QUOTE BREAKFUNCTION)
 (PROGN (OPEN  (BREAKIN  80 *SOURCE*) (BREAKOUT 80))
 (STATUS 53))
 T
 (COND ({ATOM (UNEVAL 5)) (UNEVAL 9})
 [T (CIR (UNEVAL 5))]))
NIL
(QUOTE ERROR)))))

(DEFPROP EDITF  EXPR
 (FLAMBDA (EDITLIST)
 (OPEN (EDITIN 255 *SOURCE*)
 (EDITOUT 70 *SINK*)
 (PRETTYBUFFER 70 *SINK*)
 (STATUS (11 EDITIN EDITINTERCEPT)
  (5 EDITOUT 2)
  (13 EDITIN ".")
  (13 EDITOUT :)
  (13 PRETTYBUFFER :))
 (MAPC
 (QUOTE
 (LAMBDA (EDITEL)
 (TERPRI EDITOUT)
 (PRIH1 (QUOTE "EDITING ") EDITOUT)
 (PRIK1 EDITEL EDITOUT)
 (TERPRI EDITOUT)
 (EDITSTART
 (CADR
 (GETL EDITELE
 (QUOTE (EXPR MACRO BUG))
 (QUOTE
 (PROGN (TERPRI EDITOUT)
 (PRIH1 (QUOTE " NOT FOUND") EDITOUT)
 (TERPRI EDITOUT)
 (UNEVAL (QUOTE EDITSTART) NIL1)))))))
 EDITLIST)))

(DEFPROP EDITNEXT EXPR
 (FLAMBDA (EDITFLAG)
 (READ EDITIN))

(DEFPROP EDITINTERCEPT EXPR
 (LAMBDA (EDITDEY)
 (COND ((EQ EDITFLAG (QUOTE A)) (QUOTE *EDITFLAG*))
(EDITFLAG (EDITERN "MISSING SOMETHING"))
(SETQ EDITLASTCOMMAND NIL)
(EDITONE (QUOTE P))
(READLINE EDITIN NIL)))

(DEFFPROP EDITSTART EXP
(LAMBDA (EDITLIST)
(PROG (EDITCHAIN LASTAIL EDITLASTCOMMAND PRETTYB PRETTYC PRETTYRP EDITDOT EDITDOT# EDITDOTS EDITAMP PRETTYLP PRETTYRP EDITLASTAIL EDITNEWUNPRINT EDITUNPRINT EDITUNDOLIST)
(SETQ EDITCHAIN (CONS EDITLIST NIL))
(SETQ EDITUNDOLIST (GET EDITTURE (QUOTE *EDITFLAG*))
(PRETTYUUP PRETTYBUFFER)
(SETQ EDITDOT (COPY (QUOTE " . "))
EDITDOT# (COPY (QUOTE " ...#"))
EDITDOTS (COPY (QUOTE " ... "))
EDITAMP (COPY (QUOTE " &"))
(SETQ PRETTYB 60)
(EDITONE (QUOTE P))
(EDITLOOP (EDITTOP (SETQ EDITLASTCOMMAND (EDITNEXT NIL)))
(GO EDITLOOP)))

(DEFFPROP EDITONE EXP
(LAMBDA (COMMAND)
(PROG (EDITA EDITB EDITC EDITD EDITE EDITF)
(EDITCHAIN LASTAIL EDITLASTCOMMAND PRETTYB PRETTYC PRETTYRP EDITDOT EDITDOT# EDITDOTS EDITAMP PRETTYLP PRETTYRP EDITLASTAIL EDITNEWUNPRINT EDITUNPRINT EDITUNDOLIST)
(SETQ EDITCHAIN (CONS EDITLIST NIL))
(SETQ EDITUNDOLIST (GET EDITTURE (QUOTE *EDITFLAG*))
(PRETTYUUP PRETTYBUFFER)
(SETQ EDITDOT (COPY (QUOTE " . "))
EDITDOT# (COPY (QUOTE " ...#"))
EDITDOTS (COPY (QUOTE " ... "))
EDITAMP (COPY (QUOTE " &"))
(SETQ PRETTYB 60)
(EDITONE (QUOTE P))
(EDITLOOP (EDITTOP (SETQ EDITLASTCOMMAND (EDITNEXT NIL)))
(GO EDITLOOP)))

(DEFFPROP EDITNEW EXP
(LAMBDA (COMMAND)
(PROG (EDITA EDITB EDITC EDITD EDITE EDITF))
(cond ((null command) nil)
  ((numberp command)
    (cond ((zerop command)
            (cond ((cadr editchain) (editerr "YOU'RE AT THE TOP")))
          (setq editchain (cadr editchain)))
      ((minusp command)
       (and (greaterp (minusp command) (length (cadr editchain)))
            (editerr "TOO FEW ELEMENTS"))
       (editone (add 1))
       (command
         (cond ((atom (car editchain))
                (editerr "CURRENT EXPRESSION IS AN ATOM")
                (setq editb (nth -1 command))
                editb)
              (editerr "TOO FEW ELEMENTS")
              (t (setq editchain (cadr editchain)))
              (setq lastail * )
              (setq edita (car editchain))
              (or (eq edita (car editchain))
                   (editone (quote !0))
                   (or
                    (setq editb
                       (and (tailp edita (car editchain))
                            (editerr "YOU'RE AT THE END"))
                    (editerr "YOU'RE AT THE BEGINNING")
                    (cadr editchain)
                    (editone (quote !0))
                    (and (eq edita (cadr editchain))
                         (editerr "YOU'RE AT THE END")
                         (cadr editchain))
                    (cadr editchain))
                    editchain)))
    (selectq command
      (| (setq editchain (last editchain)))
      (!0
       (or (cadr editchain) (editerr "YOU'RE AT THE TOP"))
       (prog nil
         (loop (setq editchain (cadr editchain))
            (and (tailp (cadr editchain) (cadr editchain))
                 (go loop))))
      (nx
       (setq editchain
         (lambda (editchain)
          (editone (quote up))
          (setq edita (cadr editchain))
          (or (eq edita (cadr editchain))
               (editone (quote 10))
               (or
                (setq editb
                   (and (tailp edita (cadr editchain))
                        (editerr "YOU'RE AT THE END")
                        (cadr editchain))
                (editerr "YOU'RE AT THE BEGINNING")
                (cadr editchain)
                (cadr editchain))
                editchain)))
      (bk
       (setq editchain
         (lambda (editchain)
          (editone (quote op))
          (setq edita (cadr editchain))
          (or (eq edita (cadr editchain))
               (editone (quote 10))
               (and (eq (cadr editb) edita)
                    (editerr "YOU'RE AT THE BEGINNING")
                    (setq editb (cadr editchain))
                    (prog nil
                      (loop (setq (cadr editb) edita)
                        (return
                         (cons (car (setq lastail editb))
                              editchain))))
               editchain)))
EDITCHAIN))
(OR (SETQ EDITB (CDR EDITB))
 (EDITERR "EXPRESSION IS LAST EXPRESSION")
 (GO LOOP)))

(EDITCHAIN))
(SETQ EDITCHAIN
((LAMBDA (EDITCHAIN)
 (PROG NIL
 LOOP (COND
 (NULL (SETQ EDITCHAIN (CDR EDITCHAIN)))
 (EDITERR "YOU'RE AT THE END")
 . (NULL (CDR (HEMQ (CAR EDITCHAIN) (CADR EDITCHAIN))))
 (GO LOOP))))
 (EDITONE (QUOTE NX)))))

EDITCHAIN))
(F (SETQ EDITA (EDITNEXT T)) (EDITPEDIT))
(UP (PROG NIL
 (SETQ EDITA (CAR EDITCHAIN))
 LOOP (COND
 (NULL (SETQ EDITB (CDR EDITCHAIN)))
 (EDITERR "NOT A SUBEXPRESSION")
 . (NOT (SETQ EDITC (KCONQ EDITA (CAR EDITB)))))
 (EDITERR "NOT A TAIL")
 . (OR (EQ EDITC LASTAIL) (RETURN NIL).)
 . (NOT (SETQ EDITD (MEBQ EDITA (CDR EDITC)))))
 (AND (EQ EDITA (CAR LASTAIL)) (TAILP LASTAIL EDITD))
 (EDITEDIT LASTAIL)
 (T (TERMQ EDITOUT))
 (PRINT (QUOTE "LOCATION UNCERTAIN") EDITOUT)))
 (COND ((EQ EDITC (CAR EDITB)) (SETQ EDITCHAIN EDITB))
 (T (SETQ EDITCHAIN (CONS EDITC EDITB))))))

((EXCEPT 1)
 (EDITCOLLECT (QUOTE (AFTER EEFORE FOR »EDITFLAG*)))
 . (LAMBDA (EDITCHAIN)
 (SETQ EDITE EDITA EDITB EDITB)
 (COND
 (MEMO EDITE (QUOTE (: * EDITFLAG*)))
 (EDIT EVAL (QUOTE (*EC1TFLAG*  :))))
 (SELECTQ EDITE
 AFTER
 (EDITSMASH (CAR EDITCHAIN)
 (CAR EDITCHAIN)
 (CONS EDITD (CDAR EDITCHAIN))))
 BEFORE
 (COND
 ((NULL (CDR EDITCHAIN))
 (EDITSMASH (CAR EDITCHAIN)
 (CAR EDITCHAIN)
 (CONS (CDR EDITD)
 (CONS (CDAR EDITCHAIN)
 (CDAR EDITCHAIN)))))
 . (TAILP (CAR EDITCHAIN) (CADR EDITCHAIN))))}
(EDITONE (QUOTE BK))
(EDITONE (QUOTE UP))
(EDITSMASH (CAR EDITCHAIN)
(CAR EDITCHAIN)
(CONC EDITD (CDAR EDITCHAIN))
(T (EDITONE (QUOTE UP)))
(EDITSMASH (CAR EDITCHAIN)
(CONC EDITD (CAR EDITCHAIN))
(CAR EDITCHAIN)
(CONC EDITD (CAR EDITCHAIN))
(EDITONE (QUOTE BK))
(EDITONE (QUOTE UP))
(EDITSMASH (CAR EDITCHAIN)
(CAR EDITCHAIN)
(CONC EDITD (CAR EDITCHAIN))
(EDITERR "NOT BEFORE, AFTER, OR FOR")
(EDITCHAIN))
((EMBED EM)
(LAMBDA (EDITCHAIN)
(EDITCHAIN)
(EDITCHAIN)
(EDITCHAIN)
(EDITCHAIN))
((LAMBDA (EDITCHAIN)
(EDITCHAIN))
((LAMBDA (EDITCHAIN)
(EDITCHAIN))
((LAMBDA (EDITCHAIN)
(EDITCHAIN))
((DELETE D)
(LAMBDA (EDITCHAIN)
(EDITCHAIN))
((EDIT D)
(LAMBDA (EDITCHAIN)
(EDITCHAIN))))
((EDITCHAIN)
(BR)
(LAMBDA (EDITCHAIN)
(EDITCHAIN)))
(EDITCHAIN)

(BO)

(EDITCHAIN)

(BI)

(EDITCHAIN)

(EV)

(EDITCHAIN)

(UNDO (SETQ EDITA (EDITNEXT A)))
(OR EDITUNDOLIST (EDITERR "NOTHING SAVED"))

(COND
  ((EQ EDITA (QUOTE *EDITFLAG*))
   (PROG NIL
     (SETQ EDITB EDITUNDOLIST)
     LOOP (COND ((NULL EDITB) (EDITERR "UNDO WHAT?"))
               ((EQ (CAAR EDITB) (QUOTE BLOCK))
                (EDITERR "UNDO BLOCKED"))
               ((MEMO (CAAR EDITB) (QUOTE (UNDO !UNDO NIL)))
                (SETQ EDITB (CDR EDITB))
                (GO LOOP))
               (T (EDITUNDOP (CAAR EDITB))
                (EDITUNDO EDITB (QUOTE UNDO))))))))

(T
  (PROG NIL
    (SETQ EDITC EDITUNDOLIST)
    LOOP (COND ((NULL EDITC) (EDITERR "THERE ISN'T ONE")
                 ((EQ (CAAR EDITC) EDITA)
                  (EDITUNDOP (CAAR EDITC))
                  (EDITUNDO EDITC (QUOTE UNDO)))
                 (T (SETQ EDITC (CDR EDITC)) (GO LOOP)))))))

(UNDO
  (OR (SETQ EDITB NIL EDITA EDITUNDOLIST)
      (EDITERR "NOTHING SAVED"))
  (PROG NIL
    LOOP (COND ((OR (NULL EDITA) (EQ (CAAR EDITA) (QUOTE BLOCK)))
                 (EDITUNDOP (CONS (CONS (QUOTE UNDO) EDITB) NIL)
                 (QUOTE UNDO)))
               ((NOT (CAR EDITA)) (SETQ EDITB (CDR EDITA)) (GO LOOP))
               (T (EDITUNDOP (CAAR EDITA))
                (EDITUNDO EDITB (CDR EDITA)))
                (NCONC EDITA NIL)
                (SETQ EDITA (CDR EDITA))
                (GO LOOP))))))

("-P" (OR EDITUNPRINT (EDITERR "NOWHERE TO GO"))
  (SETQ EDITCHAIN EDITUNPRINT)

(S
  (PUT (EDITNEXT T)
       (QUOTE *EDITFLAG*)
       (COPY (CAR EDITCHAIN)))
  (P "?"
    (SETQ EDITUNPRINT EDITUNPRINT)
    (EDITUNPRINT EDITUNPRINT)
    EDITCHAIN
    DOTFLAG
    (FAILP (CAR EDITCHAIN) (CADDR EDITCHAIN))
    (COND ((EQ COMMAND (QUOTE "?")) (PRINTLEV (CAR EDITCHAIN) 1000))
           ((NUMBERP (SETQ EDITA (EDITNEXT A)))
            (PRINTLEV (CAR EDITCHAIN) EDITA))
           ((PRINTLEV (CAR EDITCHAIN) 2) (EDITUNPRINT EDITCHAIN)))
    (SPRINT (CAR EDITCHAIN)
             (COND ((EQ (CAAR EDITCHAIN) (QUOTE PROC)) 10)
                    (T 1))
             NIL)
    (TERPRI PRETTYBUFFER)
    (E (PRINT (EVAL (EDITNEXT T))))
    (OK (EDITONE (QUOTE TEST)))

)
(PUT EDITLELE (QUOTE *EDITFLAG*) EDITUNDOLIST)
(UNEVAL (QUOTE EDITSTART) (QUOTE (QUOTE NIL)))
(TTEST
(SETQ EDITUNDOLIST
(CONS (CONS (QUOTE BLOCK) NIL) EDITUNDOLIST)))
(UNBLOCK
(MAP
(QUOTE
(LAMBDA (EDITLELE)
(COND
((EQ (CAAR EDITLELE) (QUOTE BLOCK))
(REPLCA EDITLELE NIL)
(UNEVAL (QUOTE ZAP) NIL)))))
EDITUNDOLIST))
("???" (TEMPRI EDITOUT)
(MAPC
(QUOTE
(LAMBDA (EDITLE)
(COND
(EDITLE (PRIN1 (CAR EDITLE) EDITOUT)
(PRIN1 (QUOTE " WHAT?") EDITOUT)
(TEMPRI EDITOUT)
(EDITERR NIL)))))
EDITUNDOLIST)
(TEMPRI EDITOUT))
(HELP (MTS (QUOTE "COPY CS:MINIEC.HELP")))
(PROG (PRIN1 COMMAND EDITOUT)
(PRIN1 "WHAT?" EDITOUT)
(TEMPRI EDITOUT)
(EDITERR NIL)))))
(DEFPROP EDITFIND EXPR
(LAMBD (EDIT)
(COND
((EQUAL EDITA EDITB) (CONS EDITB NIL))
((EQ (EDITA) (QUOTE BLOCK))
((CONS EDITB NIL))
((PROG (EDITU)
(SETQ EDIT EDITB)
LOOP (COND
((NULL (SETQ EDITLASTAIL EDITB)) (RETURN NIL))
((SETQ EDIT EARL (CAR EDITB))
((RETURN (NCONC EDITC (CONS (SETQ EDIT A) EDITSTOPPER))
GO LOOP)))))
(EDITNEXT A))  EDITSTOPPER))
((EQ EDITB (QUOTE T))
(EDITNEXT T)
(PRIN1 "ECITFLAG")
(EDITERR "NOTHING SAVED")))))
(DEFPROP EDITCOLLECT EXPR
(LAMBD (EDITSTOPPER)
(PROG NIL
(SETQ EDITA NIL)
LOOP (COND
((EQ (SETQ EDIT (EDITSTART A)) (EDITSTOPPER))
(EDITA EDITA (NCONC EDITA)
(EDITA (CONS
((AND
((EQ (EDITA (QUOTE #))
(EDITA COMMAND (QUOTE I INSERT)))
(GET (EDITA T)
(QUOTE *EDITFLAG*)
(QUOTE (EDITERR "NOTHING SAVED")))
(EDITA)
(NIL)))))
(GO LOOP))
(Return EDITA))))))
(DEFPROP EDITERR EXPR

(DEFPROP PRINTLEV EXPR
  (LAMBDA (EDITX EDITN)
    (COND ((ATOM EDITX) (PRIN1 EDITX PRETTYBUFFER))
          ((ZEROP EDITN) (PRIN1 EDITAMP PRETTYBUFFER))
          (T (PROG (EDITA EDITB)
                   (PRIN1 (COND (DOTFLAG (SETQ DOTFLAG NIL) EDITDOTS) (T PRETTYLP))
                           PRETTYBUFFER)
                      (SETQ EDITA EDITX)
                   LOOP (COND ((MEMCDR EDITX EDITA) (COND (EDITC (PRIN1 EDITDOT* PRETTYBUFFER) (RETURN NIL))
                                               (T (SETQ EDITB T) ) )
                           (SETQ EDITA (CDR EDITA))
                      (COND ((NULL EDITA) (PRIH1 PRETTYRP PRETTYBUFFER) (RETURN NIL))
                             (T (PRIN1 (CAR EDITA) PRETTYBUFFER)))
                   (GO LOOP))))))

(DEFPROP EDITFINDEDIT EXPR
  (LAMBDA NIL
    (SETQ EDITB (CAR EDITCHAIN))
    (COND ((SETQ EDITLASTAIL LASTAIL EDITP (EDITFIND EDITP))
           (SETQ EDITLASTAIL EDITCHAIN)
           (NCNC EDITP (CAR EDITCHAIN)))
          (AND (ATOM (CAR EDITCHAIN)) (EDITHE (QUOTE UP))))))
(DEFPROP EDITEVAL EXPR
  (LAMBDA (EDITTEST)
    (EDITCOLLECT EDITTEST)
    (MAPC (QUOTE (LAMBDA (EDITA)
        (COND ((OR (NUMBERP EDITA) (MEMQ EDITA ED1TCOKM AN DS))
                  (EDITONE EDITA))
            (EDITFINDIT))
            EDITA)
        (OR (NOT (CDR EDITCHAIN))
            (TAILP (CAR EDITCHAIN) (CADR EDITCHAIN))
            (EDITONE (QUOTE UP)))))))

(DEFPROP EDITUNDO EXPR
  (LAMBDA (EDITLIST EDITTYPE)
    (PROG (EDITA)
      (SETQ EDITUNDOLIST (CONS EDITTYPE (REVERSE (CDAR EDITLIST)))
        EDITUNDOLIST))
      (SETQ EDITA (CDAR EDITLIST))
      LOOP (OR EDITA (RETURN (KPLACA EDITLIST NIL)))
      (RPLACA (CAAR EDITA) (CADAR EDITA))
      (RPLACD (CAAR EDITA) (CDDAR EDITA))
      (RPLACA (CDAR EDITA) EDITA)
      (RPLACD (CDAR EDITA) EDITF)
      (SETQ EDITA (CDR EDITA))
      (GO LOOP))))

(DEFPROP EDITTOP EXPR
  (LAMBDA (EDITLIST)
    (EDITONE EDITLIST)))

(DEFPROP EDITUNDOP EXPR
  (LAMBDA (EDITLIST)
    (EDITOR EDITLIST)
    (EDITOR EDITLIST))))

(DEFPROP EDITE EXPR
  (LAMBDA (EDITLIST)
    (OPEN (EDITIN 100 *SOURCE*)
      (EDITOUT 70 *SINK*)
      (PRETTYBUFFER 80 *SINK*)
      (STATUS (11 EDITIN EDITINTERCEPT)
        (5 EDITOUT T)
        (13 EDITIN ".")
        (13 EDITOUT :)
        (13 PRETTYBUFFER :)
        (SETO EDITLIST (QUOTE EDITUNDOLIST))
        (REM (QUOTE EDITUNDOLIST) (QUOTE *EDITFLAG*.)
        (EDITSTART EDITLIST)))

(DEFPROP EDITEXCISE EXPR
  (LAMBDA NIL
    (REM (QUOTE
(EDITF EDITNEXT
EDITINTERCEPT
EDITSMASH
EDITSTART
EDITONE
EDITFIND
EDITCOLLECT
EDITERR
PRINTLEV
REDCEDR
TAILP
PRINLEV
EDITFINDIT
EDITVALUE
EDITUNDO
EDITTOP
EDITUNDO
EDITE
EDITEXCISE)

(QUOTE EXPRI)))

(setq editcommands
(quote (NX OK P PP TEST UNBLOCK UNDO UP UNDO O "?" "??" "??")

"*EDITOR LOADED")

(defprop prettysetup expr
(declare (prettybuffername)
(setq prettyyl
  (copy (quote " "))
  prettyyl
  (copy (quote ")")
  prettyyl
  (copy (quote ")")
  prettyyl
  (open (prettytry 60))
  (status (12 prettytry prettybuffer)
    5 prettytry t)
  (status (12 prettytry prettybuffer)
    6 prettytry t))

(defprop prettyprint expr
(declare (prettylist)
  (defprop prettybuffer prettybuffer)
  (open (prettybuffer 100))
  (setq prettybuffer 60)
  (prettysetup prettybuffer)
  (mapc
    (quote
      (lambdareturn)
    )
    (quote (TERPRI)
      (GET PRETTYELE
        (QUOTE EXPR)
      )
      (QUOTE PRETTYELE)
      (QUOTE EXPR)
      (QUOTE (TERPRI 1)
        (PRINT PRETTYELE)
        (PRINT (QUOTE " NOT FOUND"))
        (TERMpri)
      )
    )
    1
    nil)
  )
  prettylist)
(DEFPROP TERRPRI PRETTYBUFFER))))

(DEFPROP DSKOUT EXPR
(FLAMEDA (FILE)
(PROG (PRETTYM PRETTYIPM PRETTYLP PRETTYBP PRETTYC)
(AND (GET (QUOTE UNBREAK) (QUOTE EXPR)))
(SETO PRETTYC (COPY (QUOTE ""))
(APPLY (QUOTE OPEN)
(LIST (QUOTE PRETTYBUFFER)
(ADD (SETO PRETTYB 100) 20)
(CAR FILE))
(PRETTYSIOUP PRETTYBUFFER)
(MAP (QUOTE PRETTYDEF) (CAR FILE))
(QUOTE DONE))))

(DEFPROP PRETTYLEN EXPR
(LAMBDA (PRETTYA)
(PLEN PRETTYA)))

(DEFPROP SPRINT EXPR
(LAMBDA (PRETTYA PRETTYINDENT PRETTYPROCFLAG)
(COND
((ATOM PRETTYA)
(COND
((EQ PRETTYPROCFLAG (QUOTE PROG))
(TERRPRI PRETTYBUFFER)
(TAB (SUB PRETTYINDENT (PRETTYLEN PRETTYA) 1)
PRETTYBUFFER)
(PRIN1 PRETTYA PRETTYBUFFER)
(SETO PRETTYC T))
(T (TAB PRETTYINDENT PRETTYBUFFER)
(PRIN1 PRETTYA PRETTYBUFFER)))))

(MEQQ (CAR PRETTYA)
(QUOTE (COND PROG LAMBDA SELECT FLAMBDA NILAMEDA))
(SPRINTIT))

((GREATERP (SUB PRETTYB (PRETTYLEN (TAB PRETTYINDENT PRETTYBUFFER)))
(PRETTYLEN PRETTYA)))
(PRIN1 PRETTYA PRETTYBUFFER))

((SPRINTIT))))

(DEFPROP SPRINTIT EXPR
(LAMBDA NIL
(TAB PRETTYINDENT PRETTYBUFFER)
(PRIN1 PRETTYA PRETTYBUFFER)
(COND
((ATOM (CAR PRETTYA))
(PRIN1 (CAR PRETTYA) PRETTYBUFFER)
(PRIN1 PRETTYUL PRETTYBUFFER)
(PRIN1 PRETTYA PRETTYBUFFER)
(COND
((GREATERP PRETTYB
(ADD (PRETTYLEN (CARA PRETTYA))
(PRETTYLEN (SKIP 0 PRETTYBUFFER))))
(SETO PRETTYINDENT
(ADD (PRETTYLEN (SKIP 0 PRETTYBUFFER) 1))
(PRIN1 (CARA PRETTYA) PRETTYBUFFER)
MPC
(QUOTE
(LAMBDA (PRETTYA)
(COND (PRETTYM (SETO PRETTYC NIL))
((TEMPRI PRETTYBUFFER))
(SPRINT PRETTYA PRETTYINDENT (CAR PRETTYA)))
(CEOR PRETTYA)))))
(T
(MAPC
(QUOTE
  (LAMBDA (PRETTYFNQ)
    (COND (PRETTY (SETQ PRETTY NIL))
          ((TERM PRETTYBUFFER))
          (PRINT PRETTYFNQ (ADD PRETTY 2) (CAR PRETTYFN))
          (CDR PRETTYFN))
    (T (PRINT (CAR PRETTYFN) (ADD PRETTY 1) NIL))
    (MAPC
      (QUOTE
        (LAMBDA (PRETTYFNQ)
          (TERM PRETTYBUFFER)
          (PRINT PRETTYFNQ (ADD PRETTY 1) NIL)))
      (CDR PRETTYFN))
    (PRINT PRETTYRP PRETTYBUFFER))))

DEFPROP PRETTYLEN EXPR
  (LAMBDA (PRETTYX)
    (TAB 1 PRETTYTRY)
    (PBIN1 PRETTYX PRETTYTRY)
    (PRETTYLEN (SKIP 0 PRETTYTRY))
  )

DEFPROP PRETTYBB EXPR
  (LAMBDA (X)
    (UNEVAL (QUOTE PRETTYLEN) (QUOTE 1000)))

DEFPROP DSKIN EXPR
  (FLAMBDA (FILE)
    (PROG (FUNCTIONS IN)
      (MAPC
        (QUOTE
          (LAMBDA (PRETTYFNQ)
            (TERM PRETTYBUFFER)
            (PRINT PRETTYFNQ (ADD PRETTY 1) NIL)))
        (CDR PRETTYFN))
      (PRINT PRETTYRP PRETTYBUFFER))))

DEFPROP PRETTYDSK EXPR
  (LAMBDA (PRETTYEB)
    (COND
      (ATOM PRETTYEB)
      (TERM PRETTYBUFFER)
      (PRINT PRETTYDEP PRETTYBUFFER)
      (PRINT PRETTYEB PRETTYBUFFER)
      (PRINT PRETTYDEP PRETTYBUFFER)
      (PRINT)
      (CAR)
      (SETQ PRETTYQ (GETL PRETTYEB (QUOTE (EXPR MACRO BUG)))
        (QUOTE)
      ))
    (QUOTE DONE))))

85
n .................................................. 3
NX .................................................. 8
OK .................................................. 27, 39
P .................................................. 25
PP .................................................. 25
PRETTYPRINT ..................................... 60
RETURN ............................................. 39
S .................................................. 27
TEST ............................................... 24
TOP ................................................ 42
TRACE ........................................... 37, 44
UNBLOCK .......................................... 24
UNBREAK .......................................... 44
UNDO .............................................. 11, 22
UNTRACE .......................................... 45
UP .................................................. 5
UPDATE ........................................... 61
USE ................................................ 39
0 .................................................. 3