MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Project MAC

Artificial Intelligence
Memo No. 84

Memorandum MAC-M-264

# EDIT and BREAK functions for LISP

by Warren Teitelman

## 0.0 Summary

This memo describes some LISP functions which have been found to be extremely useful in easing the often painful process of converting the initial versions of LISP programs into final debugged code. They are part of a much larger system currently being developed but may be used as two independent packages. The break package contains a more sophisticated break function than that in the current CTSS version of LISP, which includes facilities for breaking on undefined functions as well as SUBRS and FEXPRS, plus a selective TRACE feature. The Edit package combines many of the features of the CTSS command "ed" with a knowledge of the structure of LISP (e.g., it knows about balancing parentheses). It eliminates the need to leave a LISP system to edit a function, and therefore one may edit even when track quota is exhausted. Edit will update a user's file upon request by constructing a new file containing all of the latest definitions of the functions in that file, even where some of them may currently be in the machine in compiled form.

## 1.0 Break Package

When a LISP function does not produce the desired result, it is often necessary to examine its operation in close detail in order to find the cause of failure. The TRACE feature of LISP is a concession to this need, but it is often not sufficient. This may occur because TRACE does not give enough information, since it only prints the arguments and value of the function being traced, or because it is undiscriminating, i.e., one may have to trace through many pages of output to find the trouble spot. A break function on the other hand, allows the user to specify whether or not a break will occur by making the break conditional upon the result of some computation, and, in this event, to arrest the operation of a function. He may then interrogate the broken function as to the current value of its arguments or other variables, or perform arbitrary LISP computations, and then either continue with the execution of the broken function, or return with a specified value for it without actually entering it. Another possibility is just to "crack" the function by printing out the result of some computation before executing it and then printing out its final value. Used in this way, break will act like a selective trace.

To give the potential user a feeling for how the break feature might be used, the following hypothetical debugging session is presented.

## 1.1 An example

Suppose a function FOO1, of two arguments x and y, has been been defined, and that FOO1 calls FOO2, a function of x, and MEMBER, as well as other functions. Somewhere in the operation of FOO, which calls FOO1, something is not operating properly; and we suspect it is FOO1.

```
breaklist ((foo1))
    (FOO1)
foo (2 3)       now compute a value of FOO
    (BREAK IN FOO1)
x
    4
y
    NIL
(car a)         a is some prog variable of FOO
    2
stop            everything seems correct, proceed
    (VALUE OF FOO1)
    NIL
```

```
          NIL        the value of FOO
```

Since the value of FOO1 is still not correct, we will break on FOO2 to see whether this is the cause of the trouble.

```
     breaklist ((foo2))
          (FOO2)
     foo (2 3)
          (BREAK IN FOO1)
     stop
          (BREAK IN FOO2)
     x
          3
```

This gives the value of x, the argument of FOO2. To obtain the value of x, the argument of FOO1, one can ask for the entire alist by executing the function ALIST, or by performing (EVAL (QUOTE X) (CDR (ALIST))). (Of course if the arguments of FOO1 had different names than those of FOO2, we could ask for them directly.)

```
     (alist)
          ((X . 3) (X . 4) (Y) (X . 2) (Y . 3) (A 2))
     (eval (quote x (cdr (alist)))
       #INT. 0
          (BREAK IN FOO2)
```

Realizing too late that we forgot to close the parenthesis after QUOTE X, our only course was to hit the interrupt button. However, since any error inside of a break resets the break, and interrupt causes an ERROR*A 1*, the break is reset and no harm is done.

```
     (eval (quote x) (cdr (alist)))
           4
     stop
          (VALUE OF FOO2)
          2
          (VALUE OF FOO1)
          NIL
     NIL
```

FOO2 was correct, but FOO1 is still wrong somewhere.

We are now forced to resort to breaking on MEMBER, but we wish the break to occur only at the time that the first argument of MEMBER is 4, as this is the point of interest for FOO1.

```
    unbreaklist ((foo2 foo))
        (FOO2 (FOO NOT BROKEN))
    break (member (equal a 4) nil)
        (MEMBER IS A SUBR *** NEED ARGS)
    (a b)
        MEMBER
    break (fool t nil)
        (FOO1 ALREADY BROKEN)
    foo (2 3)
        (BREAK IN FOO1)
    stop
        (BREAK IN MEMBER)
    a
        4
    x
   3
```

X should be 2, so we have found our mistake. Now we would like to return to the break in FOO1 and see if FOO is correct except for the error in FOO1. We can do this by inducing an error in the computation of FOO1 as this automatically resets the break. Note that we cannot do this by simply pressing interrupt, since this automatically resets the break in MEMBER, because of the built in safety device in the break feature. However, we can cause an error return by typing "quit".

```
    quit
        ERROR*A  1*          this induces an error return
        MEMBER
        (BREAK IN FOO1)
    return (list a)          the value FOO1 should have
        (VALUE OF FOO1)
        (4)
        *t*       The correct value of FOO
```

Now we could correct FOO by editing.

## 1.2 Function Definitions

The preceding example did not demonstrate all of the features available in the break package, e.g. cracking a function, breaking on an undefined function, etc. nor the various safeguards against error. These are all implicit in the definitions of the operation of the various functions given below, but only through experience will the user be able to fully appreciate them. (The listings of these functions are contained in the Appendix, section 3.2.)

There are two main functions, BREAK and BREAK1. BREAK redefines the function in question using BREAK1 so that at the time the function would have been entered, BREAK1 is entered instead with the definition of the function and information regarding the conditions for breaking. BREAK1 then takes the appropriate action.

### 1.2.1 BREAK

BREAK is a function of three arguments, FN WHEN WHAT. If FN is an EXPR or FEXPR, of the form (LAMBDA (X Y Z ...) FORM), BREAK redefines it to be (LAMBDA (X Y Z ...) (BREAK1 FORM WHEN (FN) WHAT)). If FN is a SUBR, BREAK asks for the names of its arguments on the teletype, and redefines FN as an EXPR of the form, (LAMBDA (ARGS) (BREAK1 (DUMMY ARGS) WHEN (FN) WHAT)), where DUMMY is a name created to reference the SUBR definition of FN. If FN is undefined, BREAK defines it as an FEXPR, (LAMBDA (L A) NIL), and then breaks in the normal way except that where (FN) appeared, (FN (UNDEFINED)) appears to remind the user FN was not originally defined. The value of BREAK is FN, or in the case where FN was undefined, (FN (UNDEFINED)). If FN is an FSUBR, the value of BREAK is (FN ****FSUBR****). If FN is already broken (BREAK can tell this by looking at its definition), the value of BREAK is (FN ALREADY BROKEN).

### 1.2.2 BREAK1

BREAK1 is a function of four arguments, FORM WHEN FN WHAT, and is an FEXPR. If (EVAL WHEN A) is NIL, where A is the alist at the time BREAK1 is entered, the value of BREAK1 is (EVAL FORM A), i.e. no break occurs. If (EVAL WHEN A) is (NIL), a CRACK occurs and (CRACK IN FN) is printed. If WHAT is not NIL, (EVAL WHAT A) is also printed (this is presumably information of interest to the user). (VALUE OF FN) is then printed followed by the value of (EVAL FORM A), which is then returned as the value of BREAK1.

If (EVAL WHEN A) is not NIL or (NIL), a bona fide break occurs, and (BREAK IN FN) is printed. If WHAT is not NIL, (EVAL WHAT A) is also printed at this time. BREAK1 then listens to the teletype for inputs to EVAL. If STOP is input, BREAK1 prints (VALUE OF FN) and prints and returns (EVAL FORM A). If RETURN FOO is input, it prints (VALUE OF FN) and prints and returns (EVAL FOO A). If QUIT is input, it performs (ERROR FN). Any other input is evaluated, its value printed, and BREAK1 listens for more inputs. Note that an input may be evaluated for its effect, e.g. one can break or unbreak functions while inside of a break.

BREAK1 is well protected against user errors. If an error occurs in the computation of an input, even if it is on a STOP or RETURN FOO, the break is reset and the breaking message printed out again. Similarly, interrupt can be pressed during the input process or during printing of the evaluation of an input. This means that one can perform a computation which produces a very long list structure, or even a circular list, and then interrupt the printing process. Similarly, one can interrupt the printing process on the exit from BREAK1 and the correct value will still be returned. BREAK1 prints out "OK" following such an interrupt. Caution: if BREAK1 is merely in the process of printing out a long list before exiting (not a circular list), it may have finished the computation and actually have exited from the function before the printing was completed by CTSS. In this case an interrupt may have an unforeseen effect depending on where the program was when it occurred. However, while inside of a break, the user can do anything with the confidence that the break will not be harmed.

### 1.2.3 UNBREAK

UNBREAK is a function of one argument, FN. If FN is not broken the value of UNBREAK is (FN NOT BROKEN). Otherwise, UNBREAK redefines FN as it was before the break and returns the value FN. If FN was undefined, this means that the FEXPR, (LAMBDA (L A) NIL), will remain on the property list. Also if FN was a SUBR, the EXPR, (LAMBDA (ARGS) (DUMMY ARGS)), will also remain. To remove these the user must do a REMPROP.

### 1.2.4 BREAKLIST

BREAKLIST is a function of one argument, a list of function names. It performs (BREAK FN T NIL) for each function name and returns the list of values of BREAK. Note that (BREAK FN T NIL) will cause FN always to break, and will not print out any message, except, of course, (BREAK IN FN).

### 1.2.5 UNBREAKLIST

Similar to BREAKLIST.

### 1.2.6 ALIST

ALIST is an FEXPR. Its value is the alist at the time it was called.

### 1.2.7 ERSETQ and NLSETQ

ERSETQ and NLSETQ are FEXPRS. They are functions of one argument, FORM. They return value NIL if (EVAL FORM ALIST) causes an error. Otherwise they return (LIST (EVAL FORM ALIST)). ERSETQ prints any error messages while NLSETQ does not. They use the LISP function ERRORSET described in the LISP manual.

## 2.0 Edit Package

The editing functions use FLIP, FORMAT LIST PROCESSOR, which is a distant relative of COMIT-METEOR as described in memo . However the user need only acquaint himself with the most elementary features of this formalism in order to avail himself of the editing features. (1) At this level FLIP is very similar to COMIT-METEOR. We include here a brief description of the language.

This package also includes several functions whose purpose is updating files. They will be described in section 2.5. They are autonomous from the editing functions and may be independently useful.

## 2.1 Basic FLIP

FLIP is a format list processing language derivative from COMIT-METEOR. This means it is designed to be useful for certain types of search procedures, parsings, and string manipulations. For the purposes of editing, however the user need only be familiar with the basic operations of COMIT, plus one new operation related to balancing parentheses.

### 2.1.1 The MATCH feature

The operation of FLIP is divided into two distinct procedures, a MATCH and a CONSTRUCT. The MATCH procedure matches the input list against a pattern, which is a list of elementary patterns. A match occurs if each of the elementary patterns matches a segment of the list structure, and if these segments, taken in order comprise the entire list. The output of the match is then a parsing of the list structure with respect to the pattern, i.e. a list of the segments that were matched.

---

(1) Since the entire FLIP package must be resident in core (in its compiled form it occupies approximately 3500 words of binary program storage, or about two-thirds of the binary program storage available) the user may wish to familiarize himself with FLIP and use it in other contexts. Also, since EDIT merely calls FLIP as a subroutine, the user may then wish to use some of the more sophisticated features of FLIP in editing. However, for most purposes, the five elementary patterns and two elementary formats listed here should suffice.

The elementary patterns adopted from COMIT ARE:

$ which matches any segment including the null segment;

$n where n is a number, matches a segment of length n;

n where n is a number, matches a segment equal to that matched by the nth elementary pattern. (counting from the beginning of the pattern);

x where x is any sexpression, matches a segment of length 1 equal to x.

Thus if the input list were (a b c c d) and the pattern ($ $1 2 $), a match would occur with the first $ matching (a b), the $1 matching (c), the 2 matching (c), and the $ matching (d).

In FLIP, these elementary patterns are extended considerably, e.g. n may be the result of a computation, x may be a variable or a computation, and may be treated as matching a segment of length 1 consisting of the single element x, or a segment equal to x, so that, for example, (a b c) may match either the segment ((a b c)) or (a b c) in the list (x y (a b c) d e a b c f). In addition, various predicates may be associated with each elementary pattern, etc. However, as stated above, for the purposes of editing, these four patterns, plus one additional elementary pattern, will usually be ample.

## 2.1.2 The CONSTRUCT feature

The CONSTRUCT procedure constructs a new list structure using the output of MATCH, and a format, which is a list of elementary formats. As in COMIT, we allow:

n where n is a number, is a segment equal to that matched by the nth elementary pattern of the match;

x which is equal to itself.

These are used to perform the necessary changes, insertions, and deletions which make up editing. For example, to find and delete one of three repeated sexpressions in a list one would use as a pattern (i.e., as input to match) ($ $1 $ 2 $ 2 $) and as a format (i.e. as input to construct) (1 2 3 4 5 7). To change the first CAR after the atom MEMBER to CDR one matches with ($ MEMBER $ CAR $) and constructs with (1 2 3 CDR 5).

## 2.2 Editing - A simple example

The usual procedure for editing involves a call to the function EDIT, specifying the structure to be edited. EDIT listen for inputs from the teletype and performs the corresponding operations until the user indicates he is satisfied. Some variations of this procedure are discussed in section 2.4.

EDIT achieves the features of combining a context editor with that of a list structure editor by dealing with objects that have <u>both</u> the linear properties of text and the structural properties of LISP. This is done by "flattening" the list structure which is being edited into one single list of atoms, substituting the special atoms L* for left parentheses, and R* for right parentheses. (2) These are handled specially by the elementary pattern $Bn, (the mnemonic B stands for balanced string), which is described below. Otherwise, all atoms are treated identically. This means that one can remove and insert parentheses, by removing and inserting L*'s and R*'s, or one can remove and insert entire structures. (3) The user need not worry about this flattening process as all inputs are automatically flattened, and conversely, output is printed in unflattened mode.

---

(2) For nonlists, the atom P* is used - e.g. (A B . C) becomes (L* A B P* C R*), ((A . B) (C . D)) becomes (L* L* A P* B R* L* C P* D R* R*).

(3) Should the resulting list not have balanced parentheses, this would be detected when the user wished to leave EDIT by the function UNFLATTEN. An error message would then be printed and the user would be allowed to correct the parentheses errors.

We present a simple example to illustrate these points before discussing the $Bn feature and the operation of EDIT.

Suppose the definition of FOO is (LAMBDA (X) (PROG NIL (COND ((EQ (CAR X) -1) (RETURN NIL))) (SETQ Y (PLUS (Y CAR X))) (SETQ X (CDR X)) (GO START) )).

There are several mistakes which we might like to correct. Let us confine ourselves for now with merely adding Y to the argument list in FOO and labelling the COND statement.

```
        edit (foo expr nil)
        (match $ x $)
        (form 1 2 y 3)            "cons" has too many other meanings
        (match $ nil $)
        (form 1 2 start 3)
        stop
            FOO         value of EDIT
```

We could perform both changes in a single match and construct if we desired. Also, we could check our intermediate results by examining the output of the matches.

```
        edit  (foo expr nil)
        (match $ x $ nil $)
        (cons 1 2 y 3 4 start 5)
        (match $ prog $ cond $)
        1
            (LAMBDA (X Y) (      refers to last match
        3
             NIL START (
        stop
            FOO         value of EDIT
```

Note that the segments match above by the first and third elementary pattern in the match do not appear to be legitimate sexpressions. This is because "(" and ")" are special atoms due to the flattening procedure. Thus we can actually manipulate individual parentheses. For example to change (SETQ Y (PLUS (Y CAR X)) ) to (SETQ Y (PLUS Y (CAR X))) we could perform (match $ plus 1* y $) (cons 1 2 4 3 5). However, for more sophisticated handling of balanced strings, we need the $Bn pattern discussed below.

## 2.3 Balancing Parentheses

In COMIT one usually thinks of the elementary patterns in terms of the segments that they ultimately match, without regard for the sequential nature of the matching process. This has been encouraged here too until now, but for the next elementary pattern it is better to think of an elementary pattern in terms of its effect on the partial match. For example, $n appends to the partial match a list consisting of the next n items in the unmatched structure and passes the remainder of the unmatched structure together with the new partial match to the next elementary pattern. Similarly, $Bn starts from the beginning of the list structure that has not yet been matched, and works out in both directions until n pairs of matching parentheses are found. This segment is then what $Bn matches. Any other elementary patterns that now have their segments included in that matched by $Bn vanish, and any portions of bordering segments included in the $Bn are deleted from the bordering elementary patterns. In effect, what the $Bn pattern says is "I didn't really want to match with CAR but with the list containing the list containing CAR; however, since it was easier to look for the CAR first and then go back and find the corresponding structure, I now have to make the necessary changes in the partial match."

To return to FOO which was (LAMBDA (X) (PROG NIL (COND ((EQ (CAR X) -1) (RETURN NIL))) (SETQ Y (PLUS (Y CAR X))) (SETQ X (CDR X)) (GO START) )), if we performed (match $ car $b2 $) and asked for the result of the match, the following would be printed:

```
     (LAMBDA (X) (PROG NIL (COND (
     (EQ (CAR X) -1)
     (RETURN NIL))) (SETQ Y (PLUS (Y CAR X))) (SETQ X  (CDR  X))
    (GO START) ))
```

Note that the original segment matched by the $ has been altered, and that the segment matched by CAR has completely disappeared. Also the final $ begins after the right hand side of the $B2.

Similarly, (match $ plus $ car $b3 $) gives

```
     (LAMBDA (X ) (PROG NIL (COND ((EQ  (CAR  X)  -1  )  (RETURN
    NIL)))
     (SETQ Y (PLUS (Y CAR X)))
     (SETQ X (CDR X)) (GO START) ))
```

Note that both the plus, the second $, and the car are gone.

This method for locating structures is much more efficient than performing (match $ (setq x (plus (y car x))) $), which would yield the same match. The latter requires many costly false starts, i.e. the program will try for a match each time a "(" is encountered. In addition, the user is less likely to err using the $Bn pattern than explicitly writing the structure, which may be very complex.

## 2.4 Operation of Edit

EDIT is a function of three variables, NAME VAL CHANGES. If VAL is atomic, EDIT operates on (GET NAME VAL), and restores the edited version on the property list of NAME when through. (This is a convenient way to edit functions or APVAL'S.) Otherwise, EDIT operates on VAL directly and returns the edited version as its value if normal exit occurs. IF CHANGES is NIL, EDIT accepts inputs from the teletype. If CHANGES is not NIL, EDIT executes the operations in CHANGES sequentially, until either an error occurs, an exit is achieved, or CHANGES is exhausted. In the latter case, it then waits for inputs from the teletype. If an error occurs, the effect is the same as when QUIT is input as described below.

### 2.4.1 Basic Commands

EDIT responds to the following commands:

```
(match ....)
(form ....)
n
match
x
quit
stop
```

The first two of these commands cause changes to be made, the next three are requests for information by the user, and the last two refer to termination procedure.

The match command causes the current match to be replaced by the results of the new match. The form command causes the old value of the structure being edited to be replaced by the result of a construct. (If errors occur, messages are printed and nothing is changed.) Until a new match is performed, the old is left intact, and similarly, the latest value of the edited structure remains until one performs a successful construct. This has the effect of giving the user a slight backup, i.e. a mistaken match or construct is not completely ruinous.

X causes the current value of the structure being edited to be printed. Similarly, MATCH will cause the entire current MATCH to be printed, while N causes just the Nth segment of the current match to be printed. (The user can interrupt these processes if he wishes).

QUIT aborts the editing operation and causes EDIT to return the value NIL. No changes are made to any of the list structure of the system in this case.

STOP is the normal exit. The structure being edited is unflattened. (If the parentheses do not count out, a message is printed and EDIT continues, allowing the user to correct the parentheses.) If VAL is atomic, EDIT redefines the property list of NAME, and punches out a call to DEFLIST: DEFLIST (((NAME edited structure)) VAL). (4) The value of EDIT is NAME. If VAL is not atomic, the value of EDIT is the edited list structure. In both cases, it also punches in LISP OUTPUT a record of all changes made under the format: PRINT (EDITED NAME ((MATCH ....) (FORM ....) ... STOP)). Thus by loading LISP OUTPUT, the editing history will be printed out, and any functions or other properties that were edited will be reset to their new value.

---

(4) In the case where VAL is EXPR, EDIT also puts the new definition on the property list of the atom EDITED under the property NAME. This is for use in conjunction with UPDATE, RELOAD, and REDEFINE as described below.

### 2.4.2 The EVAL Feature - Defining New Operations

If the operation, O, is not understood by EDIT, (EVAL O (ALIST)) is performed and the value printed. Since EDIT and EDIT1 are EXPRS, it would be possible for O, as a pseudofunction, to alter the value of some of their arguments, e.g. CHANGES or X. This is a way for the user to define new editing conventions. Before discussing this further, we must clarify the role of EDIT1.

EDIT is a function of three variables, as indicated above. Whether it is operating as a subroutine or accepting input from the teletype, it passes each operation to EDIT1 along with the current value of the structure being edited, the last match, and the alist. (5) It then resets the match and structure from the output of EDIT1, and either exits if EDIT1 outputs STOP or QUIT, or continues in the manner described above.

EDIT1 is a function of four arguments, O X M A. O is the operation to be performed, e. g. (match $ car $), stop, etc. X is the structure being edited, M the last match, and A the alist. The value of EDIT1 is (FOO X M), where FOO is either NIL, STOP, or QUIT. EDIT resets its arguments from X and M.

Since both EDIT and EDIT1 are EXPRS, the values of these variables are all on the alist. Thus by typing M, one could examine the value of the last match as it appears internally to EDIT in its unflattened form. Similarly, A will give the alist, etc. This also means that pseudofunctions can be defined which will have the effect of augmenting EDIT's vocabulary.

---

(5) The alist is used in conjunction wtih the naming feature and evaluation of free variables in the match and construct procedures themselves. See section 2.6.3.

For example, to replace all X's by Y's one might type (REPLACE X Y) having defined REPLACE as (LAMBDA NIL (PROG2 (SETQ X (SUBST (CADR O) (CADDR O) X)) T)). More generally, if REPLACE is to replace an arbitrary structure, one could use EDIT1 in its definition, e.g. (LAMBDA NIL (PROG2 (SETQ X (EDIT1 (LIST (QUOTE CONS) 1 (CADDR O) 3) X (CADDR (EDIT1 (LIST (QUOTE MATCH) (QUOTE $) (CADR O) (QUOTE $)) X M A)) A)) T)). (6) Thus this feature can be used to nest macro's and to build up a complex vocabulary based on earlier defined operations.

Another implication of the EVAL feature is that it allows the user to excercise program control over EDIT when using it as a subroutine, i.e. to make the editing process conditional upon the results of previous editing and/or computations. This can be done by using the EVAL feature to reset the variable CHANGES from which EDIT takes its instructions. For example, if CHANGES were initially ((FOO)), EDIT would submit to EDIT1 the single operation (FOO) which would be evaluated against the alist. FOO could in turn perform an editing operation and then reset CHANGES so that the correct operation would be performed next. Alternatively, FOO could itself take over the control of editing by calling EDIT as a subroutine from inside itself.

---

(6) 1. The reason for the PROG2 is merely to avoid having the entire structure typed out when EDIT1 does the EVAL.

2. The CADR and CADDR are because the actual value of O is (REPLACE X Y).

3. It is true that the latter definition of (REPLACE a b) would <u>not</u> change all a to b, but only the first occurrence. To perform an operation of the former type, we utilize the FLIP repeat feature in section 2.4.3.

### 2.4.3 FLIP Repeat

Certain taks require a repeated application of a FLIP rule. This is especially true in editing. Examples of this type of problem are delete the segments between every A and B, change the first CAR after every MEMBER to CDR, etc. These could all be done by reapplying the same MATCH and CONSTRUCT rule until the MATCH fails, but this is both tedious and inefficient, since the MATCH would start from the beginning of the list sructure each time. FLIPR and FLIPR1 are two functions included in the FLIP package to provide some of the features discussed above.

FLIPR is a function of four arguments, WS PATT FORM REP. Its value is (FLIPR1 WS PATT FORM REP NIL).

FLIPR1 is a function of five arguments, WS PATT FORM REP A. WS is the structure being operated on, PATT is the pattern used by MATCH, FORM is the format used by CONSTRUCT, REP is a format which designates what structure the pattern PATT should be reapplied to, and A is the alist.

FLIPR1 first matches WS with PATT. It then constructs with FORM and saves the result. The construct of the result of the match with REP is used for the next match with PATT. FLIPR1 constructs with FORM again and appends the result to the structure it has been saving. REP is then used again and the process continues. When the match ultimately fails, the list structure that is left is appended to the saved results. CDR of the value for FLIPR1 is this structure. CAR of its value is the number of times the match succeeded.

Thus, to delete the segments between every A and B, PATT should be ($ A $ B $), FORM (1 2 4), and REP (5). Let us trace the operation of FLIPR1 for WS (D E F A X Y G B C D A R W B M M).

The first match gives ((D E F) (A) (X Y G) (B) (C D A R W B M M)). Constructing with FORM gives (D E F A B), and with REP (C D A R W B M M). Matching again yields (C D) (A) (R W) (B) (M M). Constructing with FORM gives (C D A B), and this is appended to (D E F A B) giving (D E F A B C D A B) which is saved. The match with (M M) fails this time, so the final result is (2 D E F A B C D A B M M).

Similarly to change the first CAR after every MEMBER to CDR one uses ($ MEMBER $ CAR $) (1 2 3 CDR) and (5).

For an example of a case where REP was not merely the last segment of the match, consider the problem of removing all repetitions in a list. Here, match would be ($ $1 $ 2 $), FORM (1 2), REP ($ 5), so that if WS were ( A B C D E B X A X D F G), the result would be (A B C D E X F G).

Since FLIPR and FLIPR1 were written primarily for editing, they have also been included in the vocabulary of EDIT1. To specify a

FLIPR operation, one types (flipr patt form rep). EDIT1 then performs (FLIPR1 X PATT FORM REP A), and replaces the current value of the edited structure by (cdr result) (unless there is an error in which case no change takes place). It also prints out (car result), which is the number of times the match succeeded. The last match is not harmed so that the user can recover from an incorrect FLIPR command.

If REP is not present, EDIT1 uses a format which obtains the segment corresponding to the last $. EDIT1 also outputs how many times the match succeeded. 2.5 Updating Files

UPDATE is a function of one argument, NAME. The purpose of UPDATE is to create a new file image corresponding to the file with first name NAME, (7) and containing all of the latest definitions of the functions defined in this file, without changing the status of functions in core, even where some of the edited functions have been compiled without previously writing out their new definitions. UPDATE uses the subroutines FIND, REMEX, SUBR, REDEFINE, RELOAD, OUTFILE. Each of these play the role obvious in their definition. We give their definitions before describing how UPDATE works.

### 2.5.1 FIND

Function of one variable, X. Searches INLIST until it finds file with first name X, or file with function X defined in it. Value is corresponding entry in INLIST, e.g. (X DATA (FOO FIE FUM)) or (FOO DATA (FIE X FUM)). If X is not found, an error occurs.

### 2.5.2 REMEX

Function of two variables, X Y. IF X is atomic, REMEX uses (CADDR (FIND X)) as the functions it operates on, otherwise it operates directly on X. If Y is NIL, REMEX removes the property EXPR from all functions which are both EXPRS and SUBRS. If Y is *T*, it removes the property EXPR from all functions regardless.

(7) NAME can also be the name of a function, in which the file in which NAME is defined is used. See definition of FIND in 2.5.1.

2.5.3 SUBR

Function of one variable, X. Returns list of all functions in x which are SUBRS and only SUBRS.

2.5.4 REDEFINE

Function of one variable, X. X is a list of functions. REDEFINE searches the atom EDITED to see if any of the functions named in X appear there, and if so it redefines them. Its value is all of those functions in x which did not appear on EDITED.

2.5.5 RELOAD

Function of one variable, X. Its purpose is to reload the file specified by X, i.e. whose first name is (CAR (FIND X)). It first removes all of the EXPRS by using REMEX. It then calls LOAD, followed by REDEFINE, to get the latest definitions, and returns the value of NIL. 2.5.6 OUTFILE

Function of two arguments, X Y. X is a list of functions. OUTFILE deletes the file Y DATA and writes a new one using the definitions of the functions in X, which are assumed to be EXPRS.

Some of the above functions, notably FIND, REMEX, and RELOAD, are useful by themselves: FIND, to locate the name of a file which contains a particular function, or the functions defined in that file; REMEX, in case one has loaded a file in which some of the functions have previously been compiled and it is desirable to have them operate as SUBRS; and RELOAD, to reload a large file where one may not have the room to fit it in core if LOAD is used. However, these functions are primarily intended to be used in conjunction with UPDATE.

The operation of UPDATE can now be described. First UPDATE determines whether a RELOAD is necessary by using SUBR and REDEFINE. If all of the functions in the indicated file are present in core in EXPR form, either on their property lists or on the property list of EDITED, no RELOAD is done. UPDATE then uses OUTFILE to write out the function definitions, REMEX to restore any of the SUBRS that were changed to EXPRS in the event a RELOAD was necessary, and finally goes through EDITED and removes any of the functions that appeared there that were written out by the call to UPDATE. Thus EDITED will always contain the latest definitions of functions where they differ from the definitions on the disc (unless of course the user does off line editing), and one can EDIT and compile until the final version is produced and not perform an UPDATE until that time. Used in this fashion UPDATE is as economical as the CTSS command "ed", especially if one were to consider the time necessary to load the LISP system.

## 2.6 Advanced Editing

The Edit package that we have been describing is merely a collection of functions that make it easier to use FLIP for editing. The user could very easily write his own EDIT using FLIP as all of the necessary components for editing are contained in the FLIP functions themselves. The purpose of this section is to supply the user with the details of the operation of FLIP as they pertain to editing so that he may either design his own editing functions, or utilize more of the power of FLIP in using the EDIT package presented here. This section is designed to supplement memo , and should be read in conjunction with it.

### 2.6.1 Special FLIP Functions for Editing

Four functions have been added to the FLIP package specifically for editing. These are FLATTEN, UNFLATTEN, FPRINT, and DOLBNF. FLATTEN and DOLBNF are used by FLIP; UNFLATTEN and FPRINT are provided for the user (EDIT uses them).

FLATTEN is a function of one argument, X. If X is an atom, the value of FLATTEN is (X). Otherwise, it is (flattened version of x), where the flattening procedure has been described previously.

UNFLATTEN is a function of one argument, X. If X does not unflatten correctly, an error occurs, and a message containing relevant information is printed (e.g. 3 RIGHT PARENTHESES MISSING). Otherwise, the value of UNFLATTEN is the unflattened structure.

FPRINT is used to print a flattened structure in an unflattened format. It does not actually unflatten the structure and so can be used on lists which do not balance out. Thus FPRINT applied to (L* A B L* C P* D R* E L*) will cause (A B (C . D) E ( to be printed and return the value of NIL.

DOLBNF is the function corresponding to the $Bn pattern described in section 2.3. It is also discussed in the FLIP memo.

### 2.6.2 Editing Modes

EDIT operates with FAST, QUOTE, and EDIT set to *T*. (It resets them to their original value when it exits.) The effect of these modes is described in memo . We give a brief description here.

FAST is used to make the $ operate more efficiently. The only time this should be set to *T* is in some cases involving the use of predicates where it is necessary to maintain close surveillance over the action of DOLF, the function that performs the $ search. In editing, this seldom arises.

QUOTE informs the translator that all list structures it cannot immediately identify are to be treated as quoted sexpressions instead of subpatterns to be translated. Thus if QUOTE is NIL, in (A B (C D)), (C D) will be translated as a subpattern, as will ($1 $ D $) in ($ A B $ ($1 $ D $)). If QUOTE is *T*, neither will be translated. (To use subpatterns when QUOTE is *T*, one writes ($* X) , where X is to be evaluated to the subpattern. In the above case, one would write ($ A B $ ($* (QUOTE ($) $ D $)))).

EDIT must be *T* during editing. It is used to inform the translator for patterns that all sexpressions are to be flattened and treated as segments, and to inform CONSTRUCT that the value of all MARKS and FORMATS are to be flattened before inserting them into the list structure being assembled.

Thus, in order to edit, a user must set EDIT to *T* (using CSET), and should probably set QUOTE and FAST to *T*. The rest will be handled automatically by FLIP. Of course, if the user uses the function EDIT, this will all be done for him automatially.

## 2.6.3 The Segment-Item Distinction

In FLIP, a list structure can be specified by itself (an sexpression), in terms of a segment matched by a previous element of a pattern (a mark), or as the result of a computation involving sexpressions specified as above (a form). However, specifying a list structure alone is not enough. It is necessary to indicate how this list structure is to be used, i.e. as a segment or as an item. This problem was touched on in section 2.1.1, in conjunction with the MATCH feature. In editing, one usually specifies the list structure directly in the MATCH so the difficulty does not arise here. However, it can cause serious complications in the CONSTRUCT, especially where the naming feature is used. We will attempt to clarify this distinction first in a non-editing framework.

Suppose the list being operated on is (a b c (d e) f g). If we match with ($ c $1 $) and construct with (1 (x y z) 4), the result is (a b (x y z) f g), which is what we expected. Here the marks 1 and 4 have been treated as segments, and the sexpressions (x y z) as an item.

Unless specified otherwise, FLIP always assumes marks refer to segments, and sexpressions and forms refer to items. To specify an item in FLIP, we use the symbol *, and for segments the symbol **. Thus (1 (x y z) 3) is identical to ((** 1) (* (x y z)) (** 3)). Had we constructed instead with ((* 1) (** (x y z)) (* 3)), the result would have been ((a b) x y z (f g)). The value of the mark 1 is still (a b) but here it has been inserted directly since it is being treated as an item. Similarly, the value of the sexpression (x y z) was inserted as a segment.

Usually the interpretaton taken by FLIP is the desired one. However, when using the naming feature care must be taken. The naming feature is a device provided to bind the value of a var (i.e. an sexpression, a mark, or a form), to the name of a variable on the alist. It is discussed in detail in the FLIP memo.)

For example, if at one point we matched with ($ c ($set foo 1) $1 $) using the list (a b c (x y z) f g) as above, the value of FOO on the alist would be (a b) corresponding to the value of 1. If we later performed a match ($ f $) and a construct (1 (= foo) 3), the result would be (a b c (x y z) (a b) g). Here the value of the form (=foo) was treated as an item, exactly as the value of any other form would be. There is no distinction made because of the fact that (a b) originally was the value of a mark. To insert (a b) as a segment in this context, we would have to construct with (1 (** (= foo)) 3).

## 2.6.4 The Operation of CONSTRUCT

CONSTRUCT operates differently in EDIT mode than otherwise. The operation of CONSTRUCT is as though the value of the EFVAR or FORMAT in question were first flattened and then, if treated as an item, appended directly to the structure being assembled, or else appended minus the first L* and last R* if treated as a segment. Thus if we matched (LIST X Y (LIST Z FOO) A B) with ($ foo $b1 $) and constructed with (1 (CAR Z) 3), the value of 1 would be (L* LIST X Y) which is flattened to (L* L* LIST X Y R*) with (L* LIST X Y) being appended, since marks are treated as segments. The value of (CAR Z) is (CAR Z), flattened to (L* CAR Z R*) and appended, etc. (8)

The rationale for this procedure is that the value of 1 (L* LIST X Y) is its value in a nonediting sense. We have to flatten this value to be consistent, and this produces (L* L* LIST X Y R*). Similarly, if we matched with ($ foo $b1 ($set ugh 2) $), then UGH would be bound to (L* LIST FOO Z R*) on the alist. If we were not operating with flattened structures, the value of UGH would have been ((LIST FOO Z)), since the $B1 matches a <u>segment</u> consisting of the <u>single item</u>, (LIST FOO Z). Accordingly, if one constructs with (a b (= foo) c d), one would get (A B ((LIST FOO Z)) C D), therefore in the editing environment, one obtains (A B L* L* LIST FOO Z R* R* C D). To insert (LIST FOO Z), one constructs with (a b (** (= ugh)) c d).

### 2.6.5 Using Other FLIP features

All of the features available in FLIP can be used for editing. We have touched on the naming feature above. It can be useful for saving structures which will be deleted. For example, to replace (LIST X (LIST Y) (CONS (GET X Y) (FOO X)) (LIST Z)), by (CONS (GET X Y) (FOO X)) within a larger structure, one could match with ($ foo $b2 ($set ugh 2) $b1 $) and construct with (1 (** (= ugh)) 3). Note that the $b1 absorbed the $b2, but after it was bound to UGH.

Other features of possible interest for editing are the use of negative numbers for marks to count backwards from the present position (or for consructing, to count from the end of the match), turning TRACE on (or in the case of constructing, to count from the end of the match), turning TRACE on to see how the match is working, turning SPEAK on to see the translation of the patterns and forms, using predicates, etc. All of these are discussed in the FLIP memo and the same treatment can be applied directly to the editing environment.

(8) Actually, FLIP does not use append, nor does it flatten a structure that is already flattened. Similarly it does not put on an L* and R* only to take them off again. This is presented merely as a convenient way to visualize the process.

## 3. Appendix

### 3.1 Loading procedures

Hopefully all of the relevant files will be in public. If not please notify Warren Teitelman, Rm. 815.

#### 3.1.1 BREAK

BREAK is contained in the single file BREAK DATA. BREAK, UNBREAK, BREAKLIST, UNBREAKLIST may all be compiled, if desired.

#### 3.1.2 EDIT

EDIT, UPDATE, OUTFILE, etc. , are all contained in the single file EDIT DATA. RELOAD, OUTFILE1, UPDATE, REMEX, SUBR, and REDEFINE may be compiled. FIND may be compiled if INLIST is made COMMON. OUTFILE may be compiled if Y is made SPECIAL. Do not forget to load FLIP.

#### 3.1.3 FLIP

FLIP comes in 9 files of approximately 3 tracks each. These are FLIP DATA and FLIP1 DATA through FLIP8 DATA. To load, merely load ((FLIP)) which loads and compiles FLIP1 through FLIP8. The load time is approximately 60 seconds. FLIP prints out the file names as it finishes loading them.

## 3.2 BREAK

```
 (BREAK
    (LAMBDA (FN WHEN WHAT) (PROG (TYPE DEF)
               (COND
                  ((SETQ DEF (GET FN (QUOTE EXPR))) (SETQ TYPE
  (QUOTE EXPR)))
                  ((SETQ DEF (GET FN (QUOTE FEXPR))) (SETQ TYPE
  (QUOTE FEXPR)))
                  ((GET FN (QUOTE SUBR)) (PROG (*X *Y)
                     P1     (PRINT (CONS FN (QUOTE (IS A SUBR ***
  NEED ARGS))))
                            (COND
                               ((NULL (SETQ *X (NLSETQ (RDFLX))
 )) (GO P1)))
                            (NCONC (SETQ *Y (GENSYM)) (CDR FN))
                            (SETQ TYPE (QUOTE EXPR))
                            (SETQ DEF (LIST
                               NIL
                               (CAR *X)
                               (CONS *Y (CAR *X))))))
                  ((GET FN (QUOTE FSUBR)) (RETURN (LIST
                     FN
                     (QUOTE ****FSUBR****))))
                  ((DEFLIST (LIST
                     (LIST
                        FN
                        (LIST
                           (QUOTE LAMBDA)
                           (QUOTE (L A))
                           (LIST
                              (QUOTE BREAK1)
                              NIL
                              T
                              (SETQ DEF (LIST
                                 FN
                                 (QUOTE (UNDEFINED))))
                              WHAT)))) (QUOTE FEXPR)) (RETURN
DEF)))
               (COND
                  ((EQ (CAADDR DEF) (QUOTE BREAK1)) (RETURN (CONS
 FN (QUOTE (ALREADY BROKEN))))))
               (DEFLIST (LIST
                  (LIST
                     FN
                     (LIST
                        (QUOTE LAMBDA)
                        (CADR DEF)
                        (LIST
                           (QUOTE BREAK1)
                           (CADDR DEF)
                           WHEN
                           (LIST
                              FN)
                           WHAT)))) TYPE)
               (RETURN FN))))
NIL
```

```
(UNBREAK
   (LAMBDA (FN) (PROG (TYPE DEF)
            (COND
               ((SETQ DEF (GET FN (QUOTE EXPR))) (SETQ TYPE
 (QUOTE EXPR)))
               ((SETQ DEF (GET FN (QUOTE FEXPR))) (SETQ TYPE
 (QUOTE FEXPR)))
               (T (RETURN (CONS FN (QUOTE (IS NOT BROKEN)))
)))
           (COND
               ((EQ (CAADDR DEF) (QUOTE BREAK1)) (RETURN (CAR
 (DEFLIST (LIST
                  (LIST
                     FN
                     (LIST
                        (QUOTE LAMBDA)
                        (CADR DEF)
                        (CADADR (CDR DEF))))) TYPE)))))
            (RETURN (CONS FN (QUOTE (IS NOT BROKEN)))))))

(BREAK1
   (LAMBDA (L A) (PROG (*X)
            (COND
               ((NULL (SETQ *X (EVAL (CADR L) A))) (RETURN
(EVAL (CAR L) A)))
               ((NULL (EQUAL *X (QUOTE (NIL)))) (GO B0)))
            (PRINT (APPEND (QUOTE (CRACK IN)) (CADDR L)))
            (COND
               ((NULL (CADDDR L)) NIL)
               (T (PRINT (EVAL (CADDDR L) A))))
            (GO B3)
      B0    (PRINT (APPEND (QUOTE (BREAK IN)) (CADDR L)))
            (COND
               ((NULL (CADDDR L)) NIL)
               (T (PRINT (EVAL (CADDDR L) A))))
      B1    (COND
               ((NULL (SETQ *X (NLSETQ (RDFLX)))) (GO B0))
               ((EQ (CAR *X) (QUOTE QUIT)) (ERROR (CADDR L)
))

               ((EQ (CAR *X) (QUOTE STOP)) (GO B3))
               ((EQ (CAR *X) (QUOTE RETURN)) (GO B2)))
            (COND
               ((AND
                  (SETQ *X (ERSETQ (EVAL (CAR *X) A)))
                  (NLSETQ (PRINT (CAR *X)))) (GO B1)))
            (GO B0)
      B2    (COND
               ((OR
                  (NULL (SETQ *X (NLSETQ (RDFLX))))
                  (NULL (SETQ *X (ERSETQ (EVAL (CAR *X) A))
))) (GO B0)))
            (GO B4)
      B3    (COND
               ((NULL (SETQ *X (ERSETQ (EVAL (CAR L) A))))
(GO B0)))
      B4    (PRINT (APPEND (QUOTE (VALUE OF)) (CADDR L)))
            (COND
               ((NULL (NLSETQ (PRINT (CAR *X)))) (PRINT (QUOTE
 OK)))
            (RETURN (CAR *X)))))
NIL
```

```
(BREAKLIST
   (LAMBDA (X) (MAPLIST X (FUNCTION (LAMBDA (X) (BREAK (CAR
 X) T NIL))))))

(UNBREAKLIST
   (LAMBDA (X) (MAPLIST X (FUNCTION (LAMBDA (X) (UNBREAK (CAR
 X)))))))

(ERRSETQ
   (LAMBDA (L A) (ERRORSET (CAR L) 100000000 T A)))

(NLSETQ
   (LAMBDA (L A) (ERRORSET (CAR L) 100000000 NIL A)))

(ALIST
   (LAMBDA (L A) A))
NIL
```

## 3.3 EDIT

```
 (EDIT
    (LAMBDA (NAME VAL CHANGES) (PROG (*F *Q *E *I *X *A *Z)
             (SETQ *F FAST)
             (SETQ *Q QUOTE)
             (SETQ *E EDIT)
             (SETQ *I (CONS NIL NIL))
             (CSETQ FAST T)
             (CSETQ QUOTE T)
             (CSETQ EDIT T)
             (SETQ *X (LIST
                (FLATTEN (COND
                   ((ATOM VAL) (GET NAME VAL))
                   (T VAL)))
                NIL))
             (SETQ *A (LIST
                (LIST
                   (GENSYM))))
             (COND
                (CHANGES (GO E4)))
       E1    (COND
                ((SETQ *Z (NLSETQ (RDFLX))) (GO E3)))
       E2    (PRIN1 (QUOTE EDIT))
             (PRINT COLON)
             (GO E1)
       E3    (TCONC (COPY (SETQ *Z (CAR *Z))) *I)
             (COND
                ((NULL (ERSETQ (SETQ *Z (EDIT1 *Z (CAR *X) (
CADR *X) *A)))) (GO E2))
                ((NULL (CAR *Z)) NIL)
                ((EQ (CAR *Z) (QUOTE QUIT)) (GO E7))
                ((SETQ *X (ERSETQ (UNFLATTEN (CADR *Z)))) (GO
 E5))
                ((SETQ *X (CDR *Z)) (GO E2)))
             (SETQ *X (CDR *Z))
             (GO E1)
       E4    (TCONC (COPY (CAR CHANGES)) *I)
             (COND
                ((NULL (ERSETQ (SETQ *Z (EDIT1 (CAR CHANGES)
 (CAR *X) (CADR *X) *A)))) NIL)
                ((NULL (EQ (CAR *Z) (QUOTE STOP))) (GO E4A))
                ((PROG2
                   (SETQ *X (CDR *Z))
                   (SETQ CHANGES (CDR CHANGES))) (GO E4))
                ((SETQ *X (ERSETQ (UNFLATTEN (CAR *X)))) (GO
 E5)))
             (ERROR (APPEND (QUOTE (UNSUCCESSFUL ATTEMPT TO
EDIT)) (LIST
                NAME)))
       E4A   (SETQ *X (CDR *Z))
             (COND
                ((SETQ CHANGES (CDR CHANGES)) (GO E4)))
             (GO E2)
       E5    (SETQ *X (CAR *X))
             (PUNCH (QUOTE PRINT))
             (PUNCH (LIST
                (LIST
                   (QUOTE EDITED)
                   NAME)
                (CAR *I)))
```

```
            (COND
               ((NULL (ATOM VAL)) (GO E6))
               ((EQ VAL (QUOTE EXPR)) (PUT *X (QUOTE EDITED
) NAME)))
            (PUNCH (QUOTE DEFLIST))
            (PUNCH (LIST
               (SETQ *Z (LIST
                  (LIST
                     NAME
                     *X)))
               VAL))
            (DEFLIST *Z VAL)
            (SETQ *X NAME)
      E6    (CSETQ FAST *F)
            (CSETQ QUOTE *Q)
            (CSETQ EDIT *E)
            (RETURN *X)
      E7    (SETQ *X NIL)
            (GO E6))))

(EDIT1
   (LAMBDA (O X M A) (PROG (*Y)
            (RETURN (COND
               ((OR
                  (EQ O (QUOTE STOP))
                  (EQ O (QUOTE QUIT))) (LIST
                  O
                  X
                  M))
               ((EQ O (QUOTE X)) (LIST
                  (FPRINT X)
                  X
                  M))
               ((EQ O (QUOTE MATCH)) (LIST
                  (MAP (CDDR M) (FUNCTION (LAMBDA (X) (FPRINT
 (CAR X)))))
                  X
                  M))
               ((NUMBERP O) (LIST
                  (FPRINT (COND
                     ((NULL M) (QUOTE (NO MATCH YET)))
                     ((MINUSP O) (COND
                        ((GREATERP (MINUS O) (SETQ *Y (LENGTH
 (CDDR M)))) (QUOTE (TOO BIG)))
                        (T (CADR (FIRST$ (CDDR M) (PLUS
                           O
                           *Y)))))))
                     ((OR
                        (NULL (SETQ *Y (FIRST$ (CDR M) O)))
                        (NULL (CDR *Y))) (QUOTE (TOO BIG)))
                     (T (CADR *Y))))
                  X
                  M))
               ((EQ (CAR O) (QUOTE MATCH)) (COND
                  ((NULL (SETQ *Y (FLIP1 X (CDR O) NIL A)))
 (LIST
                     (FPRINT (QUOTE (DIDNT MATCH)))
                     X
                     M))
                  (T (LIST
                        NIL
                        X
                        (CAR *Y)))))
                  ((EQ (CAR O) (QUOTE FORM)) (COND
                     ((NULL M) (LIST
                        (FPRINT (QUOTE (NO MATCH YET)))
                        X
                        M))
                     (T (LIST
                        NIL
                        (CDR (FLIP1 M NIL (CDR O) A))
                        M))))
                  ((EQ (CAR O) (QUOTE FLIPR)) (LIST
                     (FPRINT (CONS (CAR (SETQ *Y (FLIPR1 X (CADR
 O) (CADDR O) (COND
                        ((CDDDR O) (CADDDR O))
                        (T (LIST
                           -1))) A))) (QUOTE (MATCHES OCCURRED
))))
                     (CDR *Y)
                     M))
                  (T (PROG2
                     (PRINT (EVAL O (ALIST)))
                     (LIST
                        NIL
                        X
                        M))))))))
NIL
```

```
(FIND
  (LAMBDA (X) (PROG (*X)
            (SETQ *X INLIST)
      F1    (COND
               ((OR
                  (EQ X (CAAR *X))
                  (MEMBER X (CADDAR *X))) (RETURN (CAR *X))
)
               ((SETQ *X (CDR *X)) (GO F1)))
            (ERROR (CONS X (QUOTE (NOT FOUND)))))))

(REMEX
  (LAMBDA (X Y) (PROG (*X)
            (SETQ *X (CONS NIL NIL))
            (COND
               ((ATOM X) (SETQ X (CADDR (FIND X)))))
      R1    (COND
               ((AND
                  (EQ (CADAR X) (QUOTE EXPR))
                  (OR
                     Y
                     (EQ (CADDR (CDAR X)) (QUOTE SUBR)))) (
PROG2
                 (RPLACD (CAR X) (CDDDAR X))
                  (TCONC (CAR X) *X))))
            (COND
               ((SETQ X (CDR X)) (GO R1)))
            (RETURN (CAR *X)))))

(SUBR
  (LAMBDA (X) (PROG (*X)
            (SETQ *X (CONS NIL NIL))
      S1    (COND
               ((NULL X) (RETURN (CAR *X)))
               ((EQ (CADAR X) (QUOTE SUBR)) (TCONC (CAR X)
*X)))
            (SETQ X (CDR X))
            (GO S1))))

(REDEFINE
  (LAMBDA (X) (PROG (*X *Y)
            (SETQ *X (CONS NIL NIL))
      R1    (COND
               ((NULL X) (RETURN (CAR *X)))
               ((SETQ *Y (GET (QUOTE EDITED) (CAR X))) (DEFLIST
 (LIST
                  (LIST
                     (CAR X)
                     *Y)) (QUOTE EXPR)))
               (T (TCONC (CAR X) *X)))
            (SETQ X (CDR X))
            (GO R1))))
NIL
```

```
(RELOAD
   (LAMBDA (X) (PROG (*X)
            (COND
               ((NULL (SETQ X (FIND X))) (ERROR (CONS X (QUOTE
 (NOT FOUND))))))
            (REMEX (CADDR X) T)
            (LOAD (LIST
               (CAR X)))
            (REDEFINE (CADDR X)))))

(OUTFILE
   (LAMBDA (X Y) (PROG NIL
            (SETQ X (OUTFILE1 X))
            (FILEDELETE Y (QUOTE DATA))
            (FILEWRITE Y (QUOTE DATA) (QUOTE DEFINE (())
            (MAP X (FUNCTION (LAMBDA (X) (FILEAPND Y (QUOTE
 DATA) (CAR X)))))
            (FILEAPND Y (QUOTE DATA) (QUOTE )) STOP)))))

(OUTFILE1
   (LAMBDA (X) (PROG (*X)
      OF1   (SETQ *X (TCONC (LIST
               (CAR X)
               (CADDAR X)) *X))
            (COND
               ((SETQ X (CDR X)) (GO OF1)))
            (RETURN (CAR *X)))))

(UPDATE
   (LAMBDA (X) (PROG (*X)
            (COND
               ((NULL (SETQ X (FIND X))) (ERROR (CONS X (QUOTE
 (NOT FOUND)))))
               ((AND
                  (SETQ *X (SUBR (CADDR X)))
                  (REDEFINE *X)) (RELOAD (CAR X))))
            (OUTFILE (CADDR X) (CAR X))
            (COND
               (*X (REMEX *X NIL)))
            (MAP (CADDR X) (FUNCTION (LAMBDA (X) (REMPROP (
QUOTE EDITED) (CAR X)))))
            (RETURN NIL))))
NIL
```