DECUS
PROGRAM LIBRARY

DECUS NO.	8-341
TITLE	LISP-8
AUTHOR	William Neal
COMPANY	Submitted by: Ernest Hayden Speech Communications Research Laboratory Santa Barbara, California
DATE	December 16, 1970
SOURCE LANGUAGE	PAL III

LISP-8 USER'S MANUAL

DECUS Program Library Write-up

DECUS No. 8-341

This manual describes the implementation and usage of a LISP-like list processing language for the PDP-8.

Motivation. The system was written in response to a need to do some fairly sophisticated list processing in connection with research on natural languages. The only other list processing system available, DECUS 8-102a, was a fully interpretative version of LISP 1.5. While this was a full scale implementation, the operating system took up much of core, and storage representation was such that only relatively small programs could be written. As a result, the system described herein was implemented. The operating system is relatively small, and programs require less space since a distinction is made between program and data. This is less flexible and leads to irreconciliable differences between LISP 1.5 and LISP-8, but it is more efficient for a small computer such as a PDP-8.

Minimum requirements. PDP-8 with 4K and EAE.

Requirement note. The EAE is used to do shifting in LISTIN and to do multiplication and division in BINDEC and DECBIN. The shifting in LISTIN may be done by rotates and masks. BINDEC is called by the primitives PLUS, MINUS, TIMES. DECBIN is called by the primitives PLUS, MINUS, TIMES, NUMBER, GREATP. If the user wishes to make use of BINDEC or DECBIN either directly or indirectly without using EAE, he should add subroutines to accomplish this.

Required experience. The user is assumed to be familiar with basic list processing concepts, preferably a LISP-like language. For a good introduction to list processing and recursion, see Foster, J. M. List Processing, Mac Donald and Co., London, 1967.

STORAGE REPRESENTATION

Cells. The storage allocated by the user (see OPERATING PROCEDURES) is organized into cells. Each cell occupies two words. The first word of each cell is at an even address. The two kinds of cells distinguished, list cells and atom cells, are discussed below.

List cells. The first word of a list cell is referred to as the down pointer; the second word is called the next or right pointer.

Since the address of each cell is even (where the address of a cell is that of its first word), bit 11 of an address is always zero and hence may be used to store additional information. The following convention has therefore been adopted: in a list cell, if bit 11 of the down pointer is Ø, then the down pointer points to an atom cell. If bit 11 of the down pointer is 1, then the down pointer points to a list cell.

Atom cells. The first word of an atom cell is called the value of the cell; the second word is as for list cells.

The value of an atom cell consists of two 6 bit ASCII characters, where the left 6 bits are referred to as the left character and the remainder as the right character. The character $ØØ_8$ is considered null.

An atom string (or simply atom) is made up of atom cells. For each cell, the right pointer points to the next cell in the string. The right pointer of the last cell in the string contains Ø. For each cell, bit 11 of the right pointer is always Ø.

A list string is made up of list cells. For each cell, the down pointer points to an atom string (bit 11 equals Ø) or to a list string (bit 11 equals 1). The right pointer points to the next cell in the string. The right pointer of the last cell in the string contains Ø. For each cell, bit 11 of the right pointer is indeterminate.

A list consists of a list string and those lists and atoms pointed to by the string's down pointers.

A sub-list is a list which is pointed to by the down pointer of some other list.

A null list and a null atom are special types. A pointer to a null list consists of a word of all zeros except bit 11 which equals 1. A pointer to a null atom is a word of all zeros.

A numeric atom is an atom which contains only numeric characters (Ø-9).

THE INTERPRETER

Argument types. There are three types of arguments:

(1) Primitive arguments. These are assembled as numbers 0 through 37 and correspond to a table in the interpreter giving the addresses of machine language routines which perform the indicated function.

(2) Subroutine arguments. These are assembled as numbers greater than 37 (8) which are specified in the INIT command (see OPERATING PROCEDURES).

(3) Simple arguments. A simple argument is one which is neither a primitive nor a subroutine. Normally, a simple argument is a number which points to a constant (which has been assembled with the user's LISP-8 program) or to a location in the collection shelf (see OPERATING PROCEDURES).

Argument values. These are:

(1) Primitive argument values. The value of a primitive argument is a word, the value of which depends on the primitive and its arguments (see below under primitive descriptions). This value is returned in the AC. Primitives are divided into classes on the basis of the type of values they return. These are:

(a) Logical primitives, which return logical values. These are NULL, ATOM, NUMBER, EQ, GREATP, AND, OR. The value returned to the AC by a logical primitive is either 0 (False) or 1 (True).

(b) List Primitives, which return list values. These are primitives not listed in (a) except QUOTE (see below). The value returned by a list primitive is a pointer to a list or an atom.

(2) Subroutine argument values. As indicated in the section on subroutines, a subroutine consists of a single argument with its arguments. When a subroutine argument is recognized, the corresponding subroutine is evaluated with the appropriate parameter substitution. The value of a subroutine argument is the value of the corresponding subroutine upon evaluation.

(3) Simple argument values. The value of a simple argument is the contents of the word pointed to by the argument; i.e., the value of simple argument x is C(C(x)); i.e., contents of the contents of x.

Available primitives. Primitives may also be distinguished by the number of arguments each requires. The categories and contents are:

(1) Ø arguments: LISTIN, PAUSE, EXIT

(2) 1 argument: HD, TL, LSTOUT, NULL, ATOM, DFUNC, RETURN, MACH, MINUS, NUMBER, QUOTE, ENTRY

(3) 2 arguments: EQ, SET, SETHD, SETTL, GREATP, IF, CONS

(4) indefinite number of arguments: AND, PROG, COND, BEGIN, GO, OR, PLUS, TIMES. The arguments are ended by a word which points to a word which contains -1. In what follows, this will be referred to as END.

Primitive descriptions:

AND

#Ø
args: indefinite (all logical)
definition: returns the logical AND of the values of the arguments.
special cases/error exits: Ø arguments: returns T; 1 argument: returns the argument value; no error exits.

HD

#1
args: 1 (pointer to list)
definition: returns the down pointer of the list cell pointed to by the argument.
special cases/error exits: null argument: returns Ø; atomic argument: returns Ø; no error exits.

LISTIN

#3
args: Ø
definition: inputs a list or an atom from the standard input device (see OPERATING PROCEDURES) and returns a pointer to the list or atom.

Input format: (1) lists: begin with an open parenthesis, end with a matching closing parenthesis. Spaces are assumed to precede open and close parentheses. Thus (A) and (Aƀ) are equivalent, where ƀ represents a space. (2) atoms: begin with other than open parenthesis and end with a space. Since open and closing parentehses are assumed preceded by a space, the following is a valid list: (A(B)). Spaces other than those which terminate atoms are ignored. The atom Øƀ is input as a null atom.

special cases/error exits: other than system malfunction, the only error exits occur in the event the first non-blank character is a close parenthesis, or if not enough list storage is available to store the list/atom.

LSTOUT

#4
args: 1 (pointer to list/atom)

definition: outputs a list or an atom onto the standard output device (see OPERATING PROCEDURES) and returns the argument. No character is put out following the list/atom output. Thus ABCØ on input is just ABC on output. A null list or atom is output as Ø.

special cases/error exits: none

NULL

#5
args: 1 (pointer to list/atom)

definition: returns T(1) if the argument is null; returns F (Ø) otherwise. Note: NULL;X is roughly equivalent to EQ;NIL;X where NIL is the address of a word containing zero.

special cases/error exits: none.

ATOM

#6
#args: 1 (pointer to list/atom)

definition: returns T(1) if the argument is an atom or is null; returns F (Ø) otherwise.

special cases/error exits: none.

PROG

#7
args: indefinite (described below)

definition: permits the in-line coding of LISP-8 instructions (arguments). The first set of arguments are local variables (simple arguments) to be pushed down on entry, popped on exit. These are terminated by an END (an address of any location containing a - 1), and are all set to null atoms initially. Following the END are the instructions. These are executed sequentially and are terminated by another END. The value returned is that of the last argument. See also GO and RETURN below.

special cases/error exits: none.

EQ
#10
#args: 2 (both pointers to atoms)
definition: compares the atoms pointed to by the args. Returns T (1) if they are equal; returns F (0) otherwise.
special cases/error exits: error exit occurs if either argument is not atomic.

COND
#11
#args: indefinite (described below)
definition: the number of arguments must be even. Odd numbered arguments must return logical values. Even numbered ones may be of any type. COND begins by evaluating the first argument. If the value returned by this argument is T (1) then the second argument is evaluated, the interpreter drops through to the terminating END, and the value of the second argument is the value of the COND. If the value of the first argument is not T, then the procedure is repeated with the third and fourth arguments, etc., until either an odd numbered argument has value T or until the END is encountered. In the latter case, the value is undefined.
special cases/error exits: none.

SET
#13
#args: 2 (simple; any type)
definition: the first argument is set to the value of the second argument.
special cases/error exits: none.

BEGIN
#14
#args: indefinite

definition: the same as PROG but without the pushdown/initialization of local variables. The first set of arguments (up to and including the first END) is not included.
special cases/error exits: none.

RETURN
#15
#args: 1 (any type)
definition: causes a PROG or BEGIN to drop through to the terminating END with the value of the RETURN argument. For nested PROG's or BEGIN's, RETURN drops through the PROG or BEGIN for which the RETURN is an argument (instruction). May be used only as the argument of a PROG or BEGIN or as the argument of an IF which is the argument of a PROG or BEGIN or which itself is the argument of an IF which is the argument of a PROG or BEGIN or which itself is the argument of an IF, etc.
special cases/error exits: none.

SETHD
#16
#args: 2 (pointer to non-null list; pointer to list/atom)
definition: the down pointer of the list cell pointed to by the argument is set to the value of the second argument.
special cases/error exits: error exit occurs if the value of the first argument is null or atomic.

SETTL
#17
#args: 2 (pointer to non-null list; pointer to list)
definition: the right pointer of the list cell pointed to by the argument is set to the value of the second argument.
special cases/error exits: error exit occurs if the value of the first argument is null or atomic, or if the second argument value is not a list pointer.

CONS
#20
#args: 2 (pointer to list/atom; pointer to list/null atom)

definition: returns the value of the new cell whose down pointer is the value of the first argument and whose right pointer is the value of the second argument.

special cases/error exits: error exit occurs if second argument is not a list or a null atom, or if list storage has been exhausted.

OR

#21
args: indefinite (all logical)

definition: returns the logical OR of the values of the arguments.

special cases/error exits: Ø arguments: returns F (Ø); 1 argument: returns the argument value; no error exits.

GO

#22
args: arbitrary (primitive or subroutine (with arguments) ... simple argument (location of instruction within PROG or BEGIN))

definition: causes the interpreter to continue evaluation at the location specified by the simple argument. An arbitrary number of non-simple arguments (with their arguments) may precede the simple argument. In that case, the non-simple arguments are evaluated as encountered, and control is transferred when the simple argument is encountered, and the value of the final non-simple argument becomes the value of the PROG or BEGIN at that point. Example: GO;LOC causes control to be transferred to LOC, assuming LOC is the location of an instruction within a PROG or BEGIN. GO; SET; X;HD;Y;LOC causes X to be set to the HD of Y, and control is transferred to LOC, the value of the PROG or BEGIN at that point being the HD of Y.

special cases/error exits: none.

MACH

#23
args: 1 (location of a machine language subroutine)

definition: executes a JMS to the indicated subroutine. An argument may be gotten in the subroutine by coding JMS I EV. The AC will contain the value of the argument upon return. A value may be entered on the push-down list by executing JMS I PUSH;LOC where the contents of LOC will be pushed down. The AC is clear on return. To pop a value, execute JMS I POP;LOC. The contents of LOC will upon exit be set to the top of the push-down list before popping. The AC is undisturbed. Use of the push and pop routines should be with great care since the system will blow up if the push-down list is not the same on exit from the subroutine as it was on

entry. As arguments are gotten via EV, they should be stored in locations in the collection shelf unless the user knows that no call on EV and no code in the subroutine is liable to cause a garbage collection (LISTIN, CONS, PLUS, MINUS, and TIMES are all liable to cause garbage collections) or unless the argument values are not lists or atoms. Furthermore, these locations in the collection shelf should be pushed down on entry and popped on exit. After all arguments have been gotten, the user should code JMS I TEST;JMP RET where RET is the location of code which will pop all pushes and exit from the subroutine. By way of example, consider the following subroutine: (assume LIST1 and LIST2 are defined on the collection shelf)

```
SUBR, Ø
  JMS I PUSH          /PUSH CONTENTS OF LIST1
    LIST1
  JMS I PUSH          /PUSH CONTENTS OF LIST2
    LIST2
  JMS I EV            /GET 1ST ARGUMENT
  DCA LIST1           /AND STORE IT
  JMS I EV            /GET 2ND ARGUMENT
  DCA LIST2           /AND STORE IT
  JMS I TEST          /TEST
  JMP SUBRET          /EXECUTE THIS IF NO OPERATION TO BE DONE
  .......             /EXECUTE FOLLOWING CODE OTHERWISE
  (subroutine body)
  .......
  TAD LIST1           /RETURN LIST1 AS VALUE
SUBRET, JMS I POP     /POP CONTENTS OF LIST2.  NOTE:  DONE IN REVERSE
    LIST2             /  ORDER AS PUSHES.
  JMS I POP           /POP CONTENTS OF LIST1.
    LIST1
  JMP I SUBR          /RETURN WITH CONTENTS OF LIST1 BEFORE
                      /  POP AS VALUE OF THE SUBROUTINE
```

If the subroutine is liable to be invoked recursively, then the return address should be saved. For example, the above should be modified to appear as follows:

```
SUBR, Ø
  JMS I PUSH          /PUSH RETURN ADDRESS
    SUBR
    .
    . (as above)
    .
  JMS I POP           /POP RETURN ADDRESS
    SUBR
  JMP I SUBR
```

The interpreter may be invoked via EVAL within a machine language subroutine as follows:

```
JMS I PUSH
  LISP+200
EVAL
   .
   . (EVAL argument)
   .
JMS I POP
  LISP+200
```

Other routines available: CVDEC will convert a binary word in the AC to a numeric atom the address of whose head is returned in the AC. Invocation of this routine might cause a garbare collection. CVBIN will

convert a numeric atom whose address is in the AC to a binary word, the value of which is returned in the AC (error exit is taken if the AC does not point to an atom).

PLUS

#24

#args: indefinite (all numeric atoms)

definition: each atom is converted to binary, the sum is taken and converted to a numeric atom with leading zeros stripped. Returns a pointer to this atom. Does not check to make sure argument values are numeric.

special cases/error exits: Ø args: returns Ø; 1 arg; returns arg value. If sum is zero, returns a null atom. Error exit occurs if any argument value is non-atomic.

MINUS

#25

#args: 1 (numeric atom)

definition: converts the arg to binary, negates it, converts it back to a numeric atom, and returns a pointer to that atom. Does not check to make sure the arg value is numeric.

special cases/error exits: If result is zero, returns a null atom. Error exit occurs if the argument value is non-atomic.

TIMES

#26

#args: indefinite (all numeric atoms)

definition: each atom is converted to binary, the product is taken, and converted to a numeric atom with leading zeros stripped. Returns a pointer to this atom. Does not check to make sure args are numeric.

Special cases/error exits: Ø args: returns 1; 1 arg; returns arg value. If product is zero, returns a null atom. Error exit occurs if any argument value is non-atomic.

NUMBER

#27

#args: 1 (atomic)

definition: returns T (1) if arg value is numeric; returns F (Ø) otherwise

special cases/error exits: Error exit occurs if arg value is non-atomic.

GREATP

#30
args: 2 (both numeric atoms)

definition: returns T(1) if value of first arg is numerically greater than the value of the second arg. Returns F(Ø) otherwise.

special cases/error exits: error exit occurs if either argument value is non-atomic.

IF

#31
args: 2 (logical; any type)

definition: if first argument is value T, then executes second argument; else does not execute second arg.

special cases/error exits: none.

PAUSE

#32
args: Ø

definition: causes HLT (74Ø2) to be executed. To continue, CONTINUE on the console should be depressed. The location of the PAUSE is displayed in the AC; the MQ is zero.

special cases/error exits: none.

NOT

#33
args: 1 (logical)

definition: returns T if argument value is F; else returns F.

special cases/error exits: none.

EXIT

#34
args: Ø

definition: clears various machine flags and executes a JMP to 76ØØ.

special cases/error exits: none.

QUOTE

#35
args: 1 (primitive without args or sub without args)

definition: returns the number of the primitive or subroutine which is its argument.

special cases/error exits: none.

ENTRY

#36
args: 2 (described below)

definition: used in conjunction with QUOTE. Causes execution of the argument whose number is pointed to by the 1st argument of ENTRY. The arguments for the indicated argument follow the 2nd ENTRY argument, which is a word containing Ø.
special cases/error exits: none.

Entry to the interpreter: Entry is accomplished by means of EVAL (see OPERATING PROCEDURES). EVAL accepts one argument and returns the value of the argument in the AC. The argument immediately follows the call on EVAL.

Initialization: The interpreter must be initialized as described in OPERATING PROCEDURES.

Interpreter flow. Upon entry to the interpreter via EVAL, a pointer is set to the word containing the call to the interpreter. Control is then given to a routine called EVAL2. This routine (which is recursive) advances the pointer and inspects the word pointed to. If the contents of this word are less than or equal 378 then a primitive invocation is identified, and control is passed to the specified primitive. Otherwise, the interpreter checks its subroutine table to determine if a subroutine invocation is specified. If so, the subroutine linkages are established and the subroutine is executed. Otherwise, the AC is set to the contents of the word pointed to, and return from EVAL2 is effected.

If the invocation of a primitive is specified, then control is passed to the machine language routine in the interpreter which processes the call. If the primitive requires an argument, EVAL2 is invoked, which returns in the AC value of the argument. Since EVAL2 is recursive, it is apparent that the argument of a primitive may itself be a primitive or a subroutine or a simple argument.

To illustrate the above, suppose EQUAL has been defined as a subroutine with two arguments (see the section describing subroutine definition), and that X and Y are simple.

Then:

```
EVAL                          entry to the interpreter
HD                            invocation to the primitive HD
X                             argument of HD
(AC is now set to a pointer   to the HD of X)

EVAL                          entry to the interpreter
HD                            invocation to the primitive HD
HD                            arg of HD: another invocation of HD
X                             arg of 2nd HD
(AC is now set to a pointer   to the HD of the HD of X)

EVAL                          entry to the interpreter
EQUAL                         invocation of the subroutine EQUAL
X                             1st argument of EQUAL
Y                             2nd argument of EQUAL
(AC is now set to the value   returned by EQUAL)

EVAL                          entry to the interpreter
EQUAL                         invocation of the subroutine EQUAL
HD                            1st arg or EQUAL: invocation of HD
X                             arg of HD
HD                            2nd arg of EQUAL: invocation of HD
HD                            arg of HD: another invocation of HD
Y                             arg of previous HD
(AC is now set to the value   returned by EQUAL)
```

Purpose. A LISP-8 subroutine is used for much the same purpose as a subroutine is used in a recursive algorithmic language, such as ALGOL or PL/I. A list of formal variables is specified which are set by the interpreter to the values of the actual arguments. These formal variables may then be acted on by the subroutine definition.

Header. This consists of the name of the subroutine, a comma, and a carriage return. Example: the header for a subroutine called EQUAL would appear as

```
EQUAL,
```

Formal variables. Following the Header is a set of Ø or more formal variables followed by a word which points to a word which contains -1. When the subroutine linkages are established, the formal variables are pushed down and set to the values of the arguments in the calling code. Upon exit from the subroutine, these formal variables are popped. Example: suppose END is the label of a word containing -1. Suppose further that X and Y are in the collection shelf (see OPERATING PROCEDURES). Then the following would specify a subroutine with 2 arguments and with formal variables X and Y:

```
X
Y
END
```

Note that if a particular formal variable is to be operated on as a pointer to a list or an atom, then the formal variable definition must appear in the collection shelf. Otherwise it need not.

The definition. Following the set of formal variables is a single LISP-8 argument (with its arguments). Any simple arguments which appear in the definition are assumed global unless they appear in the set of formal variables or in the set of local variables in a PROG statement which encompasses the occurrence of the argument.

Recursion. A subroutine may be invoked recursively.

Relationship of calling code to formal variables. If an argument in the calling code is simple, then upon exit from the subroutine the argument will be set to the value of the corresponding formal variable thus reflecting any changes which might have occurred in the value of the formal variable.

Tapes provided. PØLOC (S), LISP-8 (B:46ØØ-74ØØ, S:12 tapes), ORIGIN (S).

Storage required. 12 pages plus 15 (1Ø) locations on page zero, plus additional push down store required.

Reassembling LISP-8. First read in ORIGIN, then the 12 LISP-8 symbolic tapes. Reassembly at another origin may be accomplished by re-setting the value of the symbol LISP in ORIGIN.

Assembling a LISP-8 problem program. First read in ORIGIN then PØLOC then the problem program tape(s). The problem program tape(s) should begin with the user's collection shelf (see below).

Initializing the interpreter. Prior to the first execution of EVAL, the interpreter must be initialized giving certain parameters:

1. The first location available for list storage (if odd, the next even location is assumed).
2. The negation of the total number of locations available (if the absolute value is odd, then the absolute value is assumed to be one less than specified).
3. The location of an ouput routine (see below). This value is placed into the location called EVOUT in PØLOC
4. The location of an input routine (see below). This value is placed into the location called EVIN in PØLOC.
5. The negation of the number of words desired for push-down storage. This storage is allocated directly before the interpreter. If there is an overlap between this storage and the list storage, then the list storage space is reduced accordingly.
6. The locations of subroutines in the problem program.
7. A word containing zero.

The symbol INIT is defined in PØLOC as a JMS to the initializing routine in the interpreter. Thus an initialization might be:

```
INIT        /JMS TO INITIALIZING ROUTINE
START       /ADDRESS OF 1ST AVAILABLE LOCATION
START-LISP  /-(TOTAL # OF LOCATIONS AVAILABLE)
OUTPUT      /LOCATION OF AN OUTPUT ROUTINE
INPUT       /LOCATION OF AN INPUT ROUTINE
-2ØØ        /2ØØ LOCATIONS TO BE USED IN PUSH-DOWN STACK
SUBA        /LOCATION OF SUBROUTINE CALLED SUBA
COPY        /LOCATION OF SUBROUTINE CALLED COPY
FIND        /LOCATION OF SUBROUTINE CALLED FIND
Ø           /END OF SUBROUTINE LIST
```

Note: the symbol LISP has been defined in PØLOC as 5ØØØ. Thus it may be used as shown above to define available storage.

Output routine. This user-provided machine language routine must process 6-bit ASCII characters, packed 2 to a word, ignoring @ characters. The LSTOUT primitive does a JMS to the routine with the 2 characters to be output in the AC. The device accessed is called the standard output device.

Input routine. This must provide the interpreter with 6-bit ASCII characters right packed in the AC upon exit from the input routine. If any of bits Ø-5 are set, the 6-bit value will be used, but open parenthesis, close parenthesis, quote, and space will not be recognized by LISTIN which invokes this routine. The device accessed is called the standard input device.

Error exits. When a routine takes an error exit (such as the case in which either argument value for a call on EQ is not atomic), it goes to a routine which halts, displaying in the AC the address of the last argument evaluated, and displaying in the MQ the location plus one from which the exit was taken. The user may inspect the listings to determine the error. Restart is not possible.

Garbage collection. When list storage is exhausted and another cell is required (by LISTIN, CONS, PLUS, TIMES, or MINUS), a garbage collection phase is entered. First, bit 11 of the right pointer of each cell is set to 1. Then the collection shelf and the push-down list are inspected. Those cells comprising lists and atoms pointed to by locations in the collection shelf and the push-down list are noted by setting the right pointer of each of these cells to Ø. After inspection, those cells whose right pointer has bit 11 equal to 1 are collected into a free string from which requests for cells are made. Those cells whose right pointer has bit 11 equal to Ø are left unaffected. The garbage collector makes use of the push-down list, so an error exit on account of insufficient push-down storage is possible. In addition, if it is impossible to collect any cells, an error exit is taken. If any location in the collection shelf or in the stack does not point to a cell, that location is ignored.

Collection shelf. This consists of a set of locations directly following those locations in PØLOC. These locations, along with the push-down list are inspected whenever garbage collection takes place. Any list or atom which is stored in list storage will be lost (i.e., collected) if a pointer to it does not appear in the collection shelf or in the push-down list. The collection shelf is terminated by a word containing -1.

SYNTACTIC SUGARING

Consider one of the examples given under THE INTERPRETER, interpreter flow. It was:

```
EVAL
EQUAL
HD
X
HD
HD
Y
```

This is perfectly acceptable, but it is quite confusing as to the relationships between the arguments. A more intelligible way of writing the above would be:

```
EVAL
 EQUAL
  HD
   X
  HD
   HD
    Y
```

This notation makes it much clearer what is an argument of what. Realizing from experience that the above notation, while better than the first type cited, was still difficult to use and read, the decision was made to modify the assembler in such a way that an open parenthesis is interpreted as a semicolon (though printed during pass 3 as an open parenthesis), and a close parenthesis is interpreted as a space. Thus we could write the above as

```
EVAL(EQUAL(HD(X);HD(HD(Y))))
```

which is interpreted as EVAL;EQUAL;HD;X;HD;HD;Y which is clearly equivalent to

```
EVAL
EQUAL
HD
X
HD
HD
Y
```

in this way a LISP-8 program could be written using either notation, or a combination, at the user's discretion.

The assembler in use by the author contains the above modifications, but it is an in-house version and not presently available through DECUS. Users may readily make the modifications themselves (simply check for open parenthesis, and if found, jump to the routine handling semicolons; similarly for close parenthesis).

Users of MACRO-8 will probably wish to use other than open and close parentheses for the sugaring since these are used for other purposes; angle brackets are a possible substitution.

The symbolic LISP-8 tapes provided make use of the PSUEDO OP's EJECT (which ejects a page; the assembler in use automatically paginates pass 3 output into 8½ x 11 inch sheets) and PAGE which is the equivalent of *.+177&7600. It was included inasmuch as the assembler in use does not have the & operator. If the user's assembler does not have the EJECT or the PAGE options, then EJECT may be removed via the editor. For the first occurrence of PAGE (on the second LISP-8 tape) substitute via the editor *LISP+200; for the second occurrence substitute *LISP+400, etc.