BOLT BERANEK AND NEWMAN INC

CONSULTING · DEVELOPMENT · RESEARCH

AFCRL-67-0458

THE BBN 940 LISP SYSTEM

Daniel G. Bobrow
D. Lucille Darley
L. Peter Deutsch
Daniel L. Murphy
Warren Teitelman

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065

Project No. 8668

Scientific Report No. 9

This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 627, Amendment No. 2

15 July 1967

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

Contract Monitor: Stanley R. Petrick
Data Sciences Laboratory

Prepared for:

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS 01730

AFCRL-67-0458

THE BBN 940 LISP SYSTEM

Daniel G. Bobrow
D. Lucille Darley
L. Peter Deutsch
Daniel L. Murphy
Warren Teitelman

Bolt Beranek and Newman Inc
50 Moulton Street
Cambridge, Massachusetts 02138

Contract No. AF19(628)-5065

Project No. 8668

Scientific Report No. 9

This research was sponsored by the Advanced Research Projects Agency under ARPA Order No. 627, Amendment No. 2

15 July 1967

Contract Monitor: Stanley R. Petrick
Data Sciences Laboratory

Prepared for:

AIR FORCE CAMBRIDGE RESEARCH LABORATORIES
OFFICE OF AEROSPACE RESEARCH
UNITED STATES AIR FORCE
BEDFORD, MASSACHUSETTS 01730

ABSTRACT

This report describes the LISP system implemented at BBN on the SDS 94Ø Computer. This LISP is an upward compatible extension of LISP 1.5 for the IBM 7090, with a number of new features which make it work well as an on-line language. These new features include tracing, and conditional breakpoints in functions for debugging and a sophisticated LISP oriented editor. The BBN 94Ø LISP SYSTEM has a large memory store (approximately 50,000 free words) utilizing special paging techniques for a drum to provide reasonable computation times. The system includes both an interpreter, a fully compatible compiler, and an assembly language facility for inserting machine code subroutines.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

SECTION I

INTRODUCTION

LISP is a highly sophisticated list-processing language which is being used extensively in artificial intelligence research. This document describes the BBN 940 LISP system, which has a number of unique features which make it a very good on-line interactive system with a large drum memory. Ideally, a LISP system would have a very fast, random-access memory. However, magnetic core memory (the only large scale random-access memory available) is very expensive relative to serial memory devices such as magnetic drums or discs. Since average access time to a word on a drum or disc is many times slower than access to a word in a core memory, using a drum as a simple extension of core memory would reduce considerably the operating speed of such a system. We have developed special paging techniques which allow utilization of a drum for storage with a much smaller penalty in speed. These techniques are described in detail in Bobrow and Murphy's "Structure of LISP Using Two-Level Storage," (Comm. ACM, March 1967).

Although we have tried to be as clear and complete as possible, this document is not designed to be an introduction to LISP. Therefore, some parts may be clear only to people who have had some experience with other LISP systems. A good introduction to LISP is Clark Weisman, "LISP 1.5 Primer" (Dickenson Press 1967). Although not completely accurate with respect to the BBN 940 LISP system, the differences are small enough to be mastered by use of this manualand on-line interaction. Other important references, published by the MIT Press, are John McCarthy, LISP 1.5 Programmer's Manual and Berkeley and Bobrow (editors), The Programming Language LISP, Its Operation and Applications.

SECTION II

USING THE LISP SUBSYSTEM ON THE 94Ø

In order to use LISP, you must have in your files a sysout file of the basic system. This basic LISP system file, usually called LISP, contains a binary image of LISP after it has been initialized and loaded with the library. You do not need a copy of the library if you have this file.

Call LISP by typing LIS; the system will respond P; then type .; when LISP finally responds READY, and types ←, you are talking to the LISP supervisor, usually called evalquote. Then type the following:

```
     SYSIN (LISP)
```

After typing the above, the system will find and load the basic system binary file of this name from tape. When it has read it in successfully, it will respond with a T, and the LISP supervisor will again type ←, indicating that it is listening to you again.

When typing in to evalquote, typing a control-Q will clear the input line buffer erasing the entire line up to the last carriage return. Typing control-A erases the last character typed in, echoing a ↑ and the erased character; it will not go beyond the last carriage return. Pressing the RUBOUT button while in the middle of a typein to the LISP executive, evalquote, will clear the entire read buffer of everything back to the last ←, and LISP will again type ←.

The LISP read program counts parentheses, and echoes a carriage return when all left and right parentheses balance. Left and right brackets, "[" and "]", are super-parentheses. A right bracket will close all open left parentheses up to the last open left bracket; if there is no open left bracket, it will close the entire expression. For example:

```
    PRINT ((THIS IS A LISP SYSTEM FROM BBN (FOR THE 940]
```

will print the expression shown with enough right parentheses to close all lists; that is, the "]" is equivalent, in this case, to three right parentheses. Unpaired right parentheses are read as NIL.

To exit from LISP, type:

```
    LOGOUT ()
```

One can then execute any system commands, except those which start another subsystem, and continue LISP using the system CONTINUE command. This will revive the LISP system exactly as you left it, except that all open files will be closed, and you will be typing to evalquote, whether or not you executed the logout at the top level.

SECTION III

DATA TYPES AND THE ORGANIZATION OF VIRTUAL MEMORY

LISP operates in a 21-bit address space, though only that portion currently in use actually exists on the drum. A portion of the address space above that actually allocated for structures is used for representation of small integers, as described below. All data storage is contained within this virtual memory, including literal atoms, list structure, arrays and compiled code, large integers, floating point numbers, and pushdown list storage. This virtual memory is divided into pages of 256 words. References to the virtual storage are made via an in-core map which supplies the address of the required page if it is in core, or traps to a supervisory routine if the page is not in core. This drum supervisory routine selects an in-core page, writes it back on the drum if it has been changed, and reads the required page from the drum. Closed subroutine references to an in-core word through the map take approximately 40 microseconds. A reference to a word not in core, which must be obtained from the drum, takes up to 33 milliseconds, the drum maximum access time. It takes twice as long if a page must be written out on the drum before the referenced page can be read in.

Type Determination of Pointers

The virtual memory is divided into a number of areas as shown in Fig. 1. As can be seen from this map of storage, simple arithmetic on the address of a pointer will determine its type. We chose to allocate storage rather than provide in-core descriptors of storage areas, because they would take up valuable in-core space.

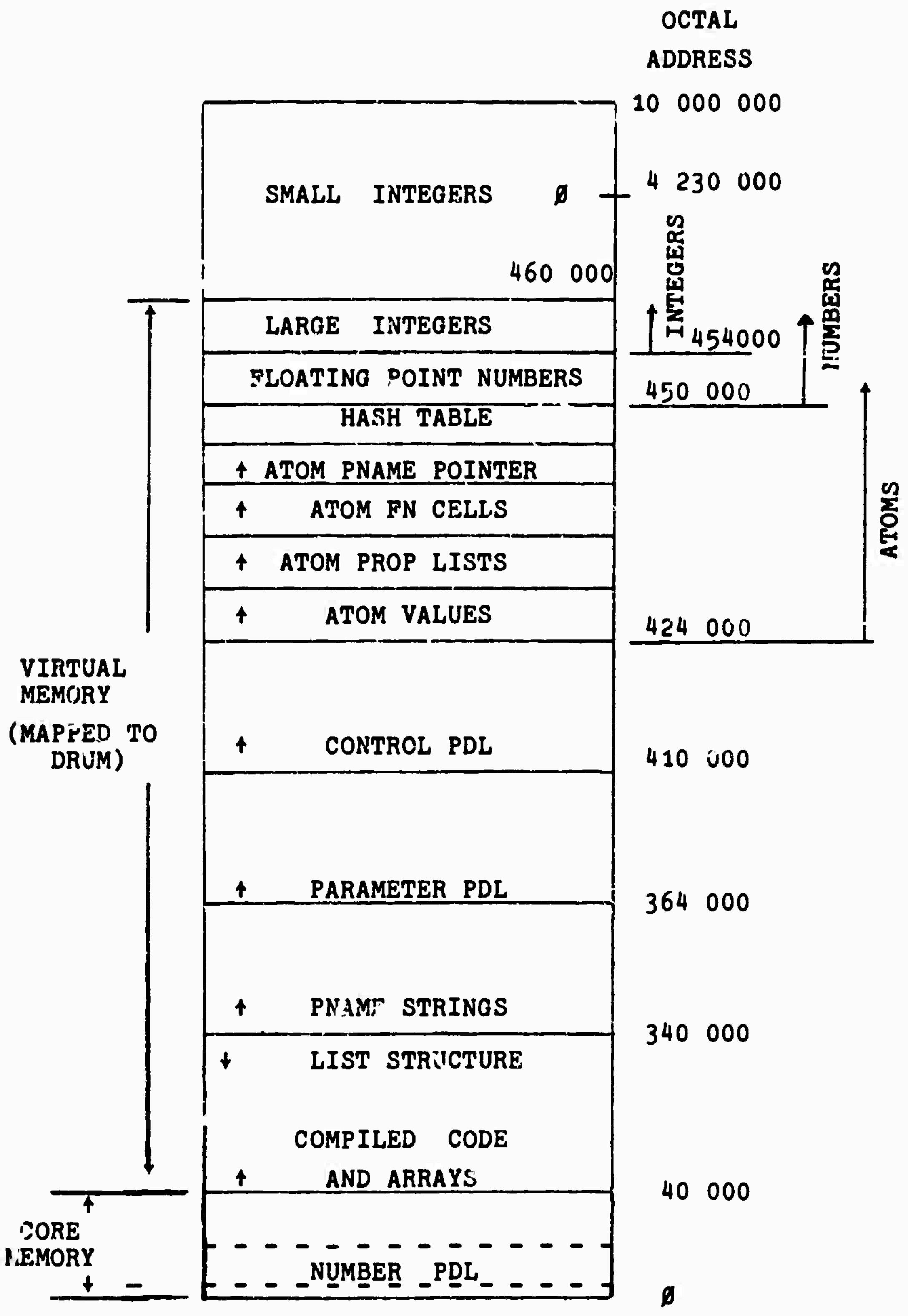


FIG. 1 MEMORY ALLOCATION IN LISP

Literal Atoms

A literal atom is constructed from any string of characters not interpretable as an integer or a floating point number. When a string of characters representing a literal atom is read in, a search is made to determine if an atom with the same print-name has been seen before. If so, a pointer to that atom is used for the current atom. If not, a new atom is created. Thus, as in all LISP systems, a literal atom has a unique representation.

Four cells (940 words) are associated with each literal atom. These cells contain pointers to the print-name of the atom, the function which it identifies, its top level or global value, and its property list. Since pointers to atoms occur in only one part of the address space, one can tell from a pointer (address) whether or not it is pointing to a literal atom.

Instead of having the four cells associated with each atom on the same page, each is put in a separate space in a position computable from the pointer to the atom.

Separating value cells and function cells, for example, is useful because most users will not use the same name for a global variable as they will for a function. Therefore, if the four cells were brought in whenever any one was asked for, it is likely that the other three cells would never be referenced. Yet, they use up room in core which could be used for other storage. Similarly, the print-name pointers associated with atoms are needed during input and output, but rarely during a computation. Therefore, during computation these cells are never in core.

Car of a literal atom usually contains the top level binding of the atom. If the atom has not yet been bound, the value cell contains the special atom NOBIND. *Cdr* of the atom is a pointer to the atom property list, initially NIL. The PNAME cell contains a pointer to a packed character table which contains the print-name of the atom. The function cell contains NIL until a function by that name is defined. One implication not immediately obvious is that car[NIL] = NIL, and cdr[NIL] = NIL. These latter two values are a significant convenience in programming.

Numerical Atoms

Integers

In LISP, most numerical atoms (numbers) do not have a unique representation; that is, a number of different pointers may reference numbers with the same value. This implies that for comparison of numbers, or for arithmetic operations, the values of the numbers must be obtained. The values of floating point numbers and large integers are stored in a "full word" space. Pointers to these values are used in list structure.

However, we utilize the fact that not all addresses in the virtual address space of the drum can legitimately appear as pointers in list structure. These "illegal" pointers are therefore used in the context of list structure to represent "small" integers directly, offset by a constant.

The input format for an integer is any string of digits, optionally preceded by a "+" or "-". Integers must have magnitude less than 2^{23}. "Small" integers are those of magnitude below approximately 2^{18} (an assembly parameter). A string of digits followed by a "Q" will be interpreted as an octal number.

Floating Point Numbers

Floating point numbers and operations are available in BBN LISP. They are stored in two contiguous 24 bit words in standard 940 format, in full word space. When creating an atom with read, ratom or pack, LISP will recognize as a floating point number a string of digits containing a decimal point. The letter "E" (exponent of 10; i.e. yyExx=yy * 10^{xx}) will also serve to designate a floating point number if preceded and followed by one or more digits. The following are legal floating point input strings.

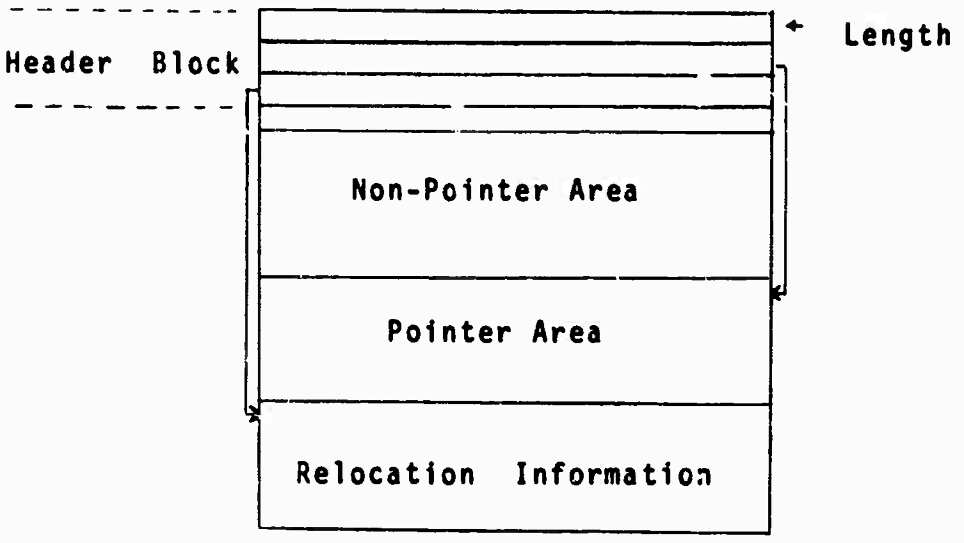
```
     5.     5.0     5E0     5E-3     5.2E+6     .3
```

The floating point/string conversion, and the floating point arithmetic are performed by the POP's and BRS's available in the 940 system. Additional information concerning conversion and precision is available from the system documentation of these routines.

The atom printing routine (used by prin1, prin2, prin3, unpack) will call the system conversion routine when it encounters a floating point datum. The output format is controlled by the function fltfmt[n] described later.

Arrays

Arrays in BBN LISP have the following format.



Typical Array

The HEADER BLOCK is four cells long and contains:

Cell:	Ø	Length of entire array.
	1	Relative address of first word of protected pointers.
	2	Relative address of first word of relocation information.
	3	Ø if an integer or symbolic array. 1 if a floating point array. Used as temporary storage in compliled code.

An array may contain both pointer and non-pointer data, separated as shown. Pointer data is assumed to be one of the standard LISP types, and the pointer data cells in all arrays are used as base cells for tracing during garbage collection. The non-pointer data, beginning in the fifth cell of the array, is of unrestricted type, and will not be used as trace pointers during garbage collection.

Relocation information contains the relative addresses of cells in the array which are to be relocated when the array is used as a compiled function, and is placed in core memory.

Examples:

1. Compiled code.
   a. Machine instructions and unboxed numeric literals are in the non-pointer area.
   b. Other literals and variable name pointers are in the pointer area.
   c. Relocation information area addresses all machine instructions whose address is within the same program, e.g., BRANCH instructions.

2. Array of lists.
   All data would be in the pointer area; the other areas would be of length Ø.

3. Array of unboxed numbers.
   All data would be in the non-pointer area; tne other areas would be of length Ø.

List Structure

List Structure is created in list space as shown in the memory map. Lists can contain pointers to all data types. As can be seen from the map, list space and array space grow toward each other. The total space available is an assembly parameter. Currently the space available is 96K (K=1024) SDS 940 24 bit words, which if used all for list storage would provide 48K words of free storage.

SECTION IV

FUNCTION TYPES

There are basically eight function types in the BBN LISP System. These eight types are characterized by three dichotomies. A function may independently have:

1. its arguments evaluated or unevaluated,
2. a fixed number of arguments or an indefinite number of arguments.
3. be defined by a LISP expression, or by machine code (which may be permanent system code, or compiled machine code).

Expressions used to define functions must start with either LAMBDA, or NLAMBDA; indicating that the arguments of this function are to be evaluated, or not evaluated, respectively. Following the LAMBDA or NLAMBDA may be any atom (except NIL) or a list of atoms (possibly empty). If there is a list of atoms, each atom in the list is the name of an argument for the function defined by the expression. Arguments for the function will be evaluated or unevaluated, as dictated by LAMBDA or NLAMBDA, and paired with these argument names. If an atom follows the LAMBDA or NLAMBDA, this function has an indefinite number of arguments. If it is an NLAMBDA expression, then the atom is paired to the list of arguments (unevaluated) of the function; that is, to <u>cdr</u> of the form in which this function name was <u>car</u>.

If a LAMBDA is followed by an atom, each of its arguments, <u>n</u>, will be evaluated in turn and placed on the parameter push down list. The atom following the LAMBDA is bound to the number of arguments which have been evaluated. A built-in function arg[m] returns the value of the mth argument of this function from the push down list. For m>n or m≤o, it is undefined.

Functions defined by expressions can be compiled by the LISP compiler, as described in the section on the compiler and lap. They may also be written directly in machine code and the LAP assembly language if the lap conventions are followed to allow linkage to LISP functions. Functions created both by the compiler and lap are referred to as compiled functions. Built-in system coded functions are called subroutines. To determine the type of any function fn, you can use the function fntyp[fn]. The value of fntyp is one of the following 12 types:

EXPR	CEXPR	SUBR
EXPR*	CEXPR*	SUBR*
FEXPR	CFEXPR	FSUBR
FEXPR*	CFEXPR*	FSUBR*

The types in the first column are all defined by expressions. The * suffix indicates an indefinite number of arguments (i.e. an atom following the LAMBDA or NLAMBDA). Functions of types in the first two rows evaluate their arguments. The types in the second column are compiled versions of the types in the first column, as indicated by the prefix C. In the third column are the parallel types for built-in subroutines. The prefix F again indicates no evaluation of arguments. Thus, for example, a CFEXPR* is a compiled form of an NLAMBDA expression with an atom following the NLAMBDA.

A standard feature of the BBN LISP system is that no error occurs if a function is called with too many or too few arguments. If a function is called with too many arguments, the extra arguments are evaluated but ignored. If a function is called with too few arguments, the unsupplied ones will be delivered as NIL. This applies to both built-in and defined functions.

There is a function progn of an arbitrary number of arguments which evaluates the arguments in order and returns the value of the last (i.e., it resembles and is an extension of prog2).

The conditional expression has been generalized so that instead of doublets it accepts n+1-tuplets which will be interpreted in the following manner:

```
(COND
   (P1 E11 E12 E13)
   (P2 E21 E22)
   (P3)
   (P4 E41))
```

will be taken as equivalent to (in LISP 1.5):

```
(COND
   (P1 (PROGN E11 E12 E13))
   (P2 (PROGN E21 E22))
   (P3 P3)
   (P4 E41)
   (T NIL))
```

This is not exactly true, but only because P3 is not evaluated a second time, if the value is needed in the third item in the second conditional expression. Thus, a list in a cond with only a predicate and no following expressions causes the value of the predicate itself to be returned. Note also that NIL is returned if all the predicates have value NIL. No error is invoked.

LAMBDA and NLAMBDA expressions also have implicit progn's; thus for example

```
        (LAMBDA (V1 V2) (F1 V1) (F2 V2) NIL)
```

is interpreted as

```
        (LAMBDA (V1 V2) (PROGN (F1 V1) (F2 V2) NIL))
```

The value of the last expression following LAMBDA (or NLAMBDA) is returned as the value of the expression. In this example, the function would always return NIL.

SECTION V

PRIMITIVE FUNCTIONS AND PREDICATES

Primitive Functions

car[x]	car gives the first element of a list x, or the left element of a dotted pair x. Nominally undefined for literal atoms, it usually gives the top level binding (value) of a literal atom x. For the usually undefined case of a number, its value is the number itself.
cdr[x]	cdr gives the tail of a list (all but the first element). This is also the right member of a dotted pair. If x is a literal atom, cdr[x] gives the property list of x. Property lists are usually NIL unless modified by the user. If x is a number, cdr returns NIL.

caar[x] = car[car[x]]
 	
cadr[x] = car[cdr[x]]
 	
cddddr[x] = [cdr[cdr[cdr[cdr[x]]]]
	All 30 combinations of nested cars and cdrs up to 4 deep are included in the system. Levels 1, 2 and 3 are subroutines; 4 is compiled. All are compiled open by the compiler.

cons[x;y]

<u>cons</u> constructs a dotted pair of <u>x</u> and <u>y</u>. If <u>y</u> is a list, <u>x</u> becomes the first element of that list. To minimize drum accesses the following algorithm is used for finding a page on which to put the constructed LISP word.

cons[x;y] is placed

1) on the page with <u>y</u> if <u>y</u> is a list and there is room; otherwise
2) on the page with <u>x</u> if <u>x</u> is a list and there is room; otherwise
3) on the same page as the last <u>cons</u> if there is room; otherwise
4) on a page in core if one is available with a specified minimum of storage; otherwise
5) on any page with a specified minimum of storage.
   The specified minimum is presently 20 LISP words in both cases.

The user may effect the operation of <u>cons</u> with the following function:

conspage[x]

causes the page on which <u>x</u> resides to be used for alternative 3 above instead of the result of the previous <u>cons</u>. If <u>x</u> is an atom, alternative 4 or 5 will be taken.

conscount[]	Returns the number of *conses* since LISP started up.
rplacd[x;y]	This very dangerous SUBR places in the decrement of the cell pointed to by *x* the pointer *y*. Thus it changes the internal list structure physically, as opposed to *cons* which creates a new list element. This is the only way to get a circular list inside of LISP; that is by placing a pointer to the beginning of a list in a spot at the end of the list. Using this function carelessly is one of the few ways to really clobber the system. The value of *rplacd* is *x*.
rplaca[x;y]	This SUBR is similar to *rplacd*, but it replaces the address pointer of *x* with *y*. The same caveats which applied to using *rplacd* apply to *rplaca*. The value of *rplaca* is *x*. *Rplaca* and *rplacd* of NIL are illegal.
quote[x]	This is a function that prevents its argument from being evaluated. Its value is *x* itself.

cond[x]

The argument for cond is a list. Each element of the list is itself a list containing n $\geq$ 1 items: the first is an expression whose value may be false or true (that is NIL, or anything which is not NIL); the rest may be any expressions. This is the conditional expression in the LISP system. The meaning of it is: if the first element of the first list is true (not NIL), then the following expressions are evaluated. The value of the conditional is the value of the last expression in this sublist. If there is only one element in the n-tuplet, then the value of the conditional is the value of this element if it is not NIL.

This value of a conditional agrees with that of LISP 1.5 for pairs of items, but allows additional flexibility. If the first element of the first list is false (=NIL), then the second sublist is considered, etc. Thus, the arguments are searched until a first element of a list is found which is not NIL. If none are found, the value of the conditional expression is NIL.

selectq[x;y_1;y_2;...;y_n;z]

This very useful function is used to select a sequence of instructions based on the value of its first argument $\underline{x}$. Each of the $\underline{y}_i$ is a list of the form

($\underline{s}_i$ $\underline{e}_{1i}$ $\underline{e}_{2i}$...$\underline{e}_{ki}$)

where $\underline{s}_i$ is the selection key.

If $\underline{s}_i$ is an atom the value of $\underline{x}$ is tested to see if it is eq to $\underline{s}_i$ (not evaluated). If so, the expressions $\underline{e}_{1i}$,...$\underline{e}_{ki}$ are evaluated in sequence, and the value of the selectq is the value of the last expression evaluated, i.e. $\underline{e}_{ki}$.

If $\underline{s}_i$ is a list, and if any element of $\underline{s}_i$ is eq to the value of $\underline{x}$, then $\underline{e}_{1i}$ to $\underline{e}_{ki}$ are evaluated in turn as above.

If $\underline{y}_i$ is not selected in one of the two ways described then $\underline{y}_{i+1}$ is tested, etc. until all the $\underline{y}$'s have been tested. If none is selected, the value of the selectq is the value of $\underline{z}$. $\underline{z}$ must be present.

An example of the form of a selectq is:

```
(SELECTQ (CAR X)
      (Q (PRINT FOO) (FIE X))
      ((A E I O U) (VOWEL X))
      (Y (TRY-AGAIN X))
      (COND((NULL X)NIL)
           (T (QUOTE STOP))))
```

which has 3 cases, Q,(A E I O U) and Y, and a default condition which is a cond.

selectq compiles open, and is therefore very fast; however it will not work for lists, large integers or floating point numbers since it uses a 24 bit open compare (an open eq).

progl[x_1;x_2;...;x_n]	This function evaluates its arguments in order, that is, x_1 then x_2 etc. It returns the value of its first argument x_1.
prog2[x;y]	Evaluates x, then y and returns y.
progn[x;y;...;z]	progn evaluates each of its arguments in sequence, and returns the value of its last argument as its value. It is an extension of prog2.

prog[args;e_1;e_2;...;e_n]

This feature allows the user to write an ALGOL-like program containing LISP statements to be executed and is identical to the prog in LISP 1.5. The first argument is a list of program variables. The rest is a sequence of (non-atomic) statements (expressions), and atomic symbols used as labels for transfer points. The value of a prog is determined by the function return. If no return is executed, the value of the prog is not guaranteed, but will not give an error.

go[x]

go is the function used to cause a transfer in prog. (GO A) will cause the program to continue at the label A.

A go can be used at any level in a prog. If a go is executed in an interpreted function which is not a prog, it will be executed in the last interpreted prog entered.

return[x]	A return is the normal end of a prog. Its argument is evaluated and is the value of the prog in which it appears. If a return is executed in an interpreted function which is not a prog, the return will be executed in the last interpreted prog entered.
set[x;y]	This function sets the atom which is the value of x, to the value of y, and returns the value of y.
setq[x;y]	This FSUBR is identical to set, except that the first argument is not evaluated. Example: If the value x is c, and the value of y is b, then set [x;y] would result in c having value b, and b returned. setq[x;y] would result in x having value b, and b returned. In both cases, the value of y is unaffected.
setqq[x;y]	Identical to setq except that neither argument is evaluated.

Predicates and Logical Connectives

atom[x]	atom[x]=T if x is an atom; NIL otherwise.
eq[x;y]	The value of eq is T if x and y are identical atoms, NIL otherwise. This includes numbers, if eq is called from an interpreted function. It is not guaranteed for floating point numbers and large integers when used in a compiled function, since it is compiled open as a 24 bit compare.
eqp[x;y]	Identical to eq, except that it is compiled closed, and hence will work for all numbers in compiled code.
neq[x:y]	The value of this function is T if x is not eqp to y, and NIL otherwise.
nill[]	Defined as NIL
null[x]	eq[x;NIL]
equal[x;y]	The value of this function is T if x and y are equal, that is, identical S-expressions, and NIL otherwise. Identical here means that they will print identically.

and[x_1;...x_n]	This function is an FSUBR and can take an indefinite number of arguments. Its value is the value of its last argument if none of its arguments has value NIL, and is NIL otherwise. Arguments past the first null argument are not evaluated.
or[x_1;...;x_n]	or is also an FSUBR and may have an indefinite number of arguments (including 0). or has value NIL if all of its arguments have value NIL, otherwise, it has the value of its first non-null argument. Arguments past this one are not evaluated.
not[x]	Same as null
memb[x;y]	This function determines if x is a member of list y, i.e. if there is an element of y eq to x. If so it returns the portion of the list starting with that element. If not it returns NIL.
member[x;y]	Identical to memb except that it uses equal instead of eq to check membership of x in y.

intersection[x;y]	This function returns with a list whose elements were members of both lists x and y.
union[x;y]	This function is entered with two lists. It returns with a list consisting of all elements included on either of the two original lists. If the same item is a member of both original lists, it is included only once on the new list.

SECTION VI

LIST MANIPULATION AND CONCATENATION

list[x_1;...;x_n]	The value of list is a list of the values of its arguments.
append[x;y]	This function copies the top level of list x and appends list y to this copy. The value is the combined list.
nconc[x;y]	This function is similar to append in effect, but it causes this effect by actually modifying the list structure x, and making the last element in the list x point to the list y. The value of nconc is a pointer to the first list x, but since this first list has now been modified, it is a pointer to the concatenated list.

tconc[x;p]	This function provides an effi- cient way for placing an item *x* at the end of a list. This list is the first item on *p*, that is, car[p]; cdr[p] is a pointer to the last element in this list; *x* is placed on the end of the list by modifying this structure, and *x* is placed on the list as an item. The effect of this function is equivalent to nconc[car[p];list[x]], with cdr[p] updated to point to the last ele- ment of the modified list.
lconc[x;p]	This function is similar to *tconc*, except that in this case *x* is a list. An entire list will be tacked on the end of car[p], and cdr[p] will be adjusted to be a pointer to the last element of this new combined list. Both *tconc* and *lconc* work correctly given null arguments.
attach[x;y]	This function attaches the element *x* on the front of the list *y* by doing an *rplaca* and an *rplacd*. This will not work correctly if *y* is an atom. Thus it is similar to *cons*, except that it modifies the contents of the first element of the non-null list *y*.

remove[x;l]	The function remove removes all occurrences of x from list l, giving a copy of x with all elements equal to x removed.
dremove[x;l]	This function is identical to remove, but actually modifies the list l when removing x, and returns x itself.
copy[x]	This function makes a copy of the list x. The value of copy is the (location of the) copied list. All levels of x are copied.
reverse[l]	This is a function to reverse the top level of a list. Thus, using reverse on (A B (C D)) gives ((C D) B A)
dreverse[l]	Identical to reverse but dreverse destroys the list l while reversing by modifying pointers, and thus does not use any additional storage.
subst[x;y;z]	This function gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z. It returns a copy of z with the changes made.

dsubst[x;y;z]	Identical to subst, but physically inserts a copy of x for y in z, thus changing the list structure z itself.
sublis[x;y]	Here x is a list of pairs: $((u_1.v_1)\ (u_2.v_2)\ \ldots\ (u_n.v_n))$ The value of sublis[x;y] is the result of substituting each v for the corresponding u in y. Copies the structure y with changes.
subpair[x;y;z]fl	Similar to sublis, except that elements on y are substituted for corresponding atoms on x in z. New structure is created only if needed, or if fl=T.
last[x]	This function has as its value a pointer to the last cell in the list x, and returns NIL if x is an atom. i.e. if x=(A B C) then last [x] = (C)

nth[x;n]	The arguments of nth are a list x and a positive integer n. Its value is a list whose first element is the nth element of list x. Thus if n = 1, it returns the list x itself. If n = 2, it returns cdr[x]. If n = 3, it returns cddr[x], etc. If n = 0 it returns cons[NIL,x].
length[x]	This function has as a value the length of the list x. If x is an atom, it returns Ø.

SECTION VII

PROPERTY LIST FUNCTIONS

put[x;y;z]	This function puts on the property list of x, the label y followed by the property z. The current value of z replaces any previous value of z with label y on this property list.
remprop[x;y]	This function removes all occurrences of the property with label y from the property list of x.
prop[x;y;u]	The function prop searches the list x for an item that is equal to y. If such an element is found, the value of prop is the rest of the list beginning immediately after that element. Otherwise, the value is u[], where u is a function of no arguments. Its effect is similar to memb and member, and they are more efficient when usable.

get[x;y]

This function gets from the list x the item after the atom y on list x. If y is not on the list x, this function returns NIL. For example, get[(A B C D);B] = C.

getp[x;y]

This function gets the property with label y from the property list of x.
NOTE: Both getp and get may be used on property lists. However, since getp searches a list two at a time, the latter allows one to have the same object as both a property and a value. e.g., if the property list of x is
(PROP1 A PROP2 B A C)

```
     then get[x;A]   = PROP2,
      but getp[x;A]  =   C.
```

deflist[x;p]

This function is used to put items on property lists. Its first argument x is a list of two element lists. The first of each is a name. The second element is the value to be stored after the property p on the property list of the name. The second argument p is the property that is to be used.

add[x;y;z]	This function adds the value z to the list appearing under the property y on the atom x. If x does not have a property y, the effect is the same as put[x;y;list[z]].
assoc[x;a]	If a is a list of dotted pairs, then assoc will produce the first pair whose first item is eq to x. If such an item is not found, assoc will return NIL.
sassoc[x;y;u]	The function sassoc searches y, which is a list of dotted pairs, for a pair whose first element is equal to x. If such a pair is found, the value of sassoc is this pair. Otherwise, the function u of no arguments is taken as the value of sassoc.

SECTION VIII

FUNCTION DEFINITION AND EVALUATION

getd[x]	This function gets the definition of the function whose name is the value of x. If x is not a defined function, the value of getd[x] is NIL; if x is a machine code function, the value is a number.
putd[x;y]	putd places the value of y into the function cell of the atom which is the value of x. This is the basic way of defining functions. putd is mnemonic for put definition on x. The value of putd is the definition (value of y).
putdq[x;y]	This function is similar to putd, but both arguments are considered quoted, and its value is x.

fntyp[fn]

This function returns NIL if the atom fn is not the name of a defined function. If fn is a function, then fntyp returns one of the following as defined in the section on function types:

EXPR	CEXPR	SUBR
EXPR*	CEXPR*	SUBR*
FEXPR	CFEXPR	FSUBR
FEXPR*	CFEXPR*	FSUBR*

The prefix F indicates unevaluated arguments; the prefix C indicates compiled code; and the suffix * indicates an indefinite number of arguments.

define[x]

The argument of define is a list. Each element of the list is itself a list containing two or more items. In a two-item list, the first item of each element of the list is the name of a function to be defined, and the second item is the defining LAMBDA or NLAMBDA expression. In longer lists, the first item is again the name of the function to be defined. The second is the LAMBDA list of variables and the remainder of the lists are forms for evaluation. As an example, consider the following two equivalent

	expressions for defining the function null. 1) (NULL (LAMBDA (X) (EQ X NIL))) 2) (NULL (X) (EQ X NIL)) define will not allow redefinition of a SUBR or FSUBR.
defineq[x;...;z]	This FEXPR is closely related to define. However, it can take an indefinite number of arguments, and it will treat them literally as if they were quoted. Each of the arguments must be a list, of the form described in define. Using defineq instead of define allows one to eliminate two pairs of parentheses in writing functions to be defined for loading with the function load.
eval[x]	eval evaluates the expression x and returns this value.
evala[x;a]	This is the regular eval from 7Ø94 LISP. Its first argument is a form which is evaluated by using the values obtained from a, a list of dotted pairs. That is, any variables appearing free in x, that also appear on a, will be given the value indicated on a.

evalr[x;a]	Same as evala except with list a reversed. Used by evala.
e[x]	This FEXPR is defined as eval; however, it is shorter and it removes the necessity for the extra pair of parentheses for the list of arguments for eval. Thus, when typing into evalquote one can simply type e followed by whatever one would type into eval and have it evaluated.
apply[fn;args]	apply applies the function fn to the arguments args. i.e. the arguments of fn, args, are not evaluated but given to fn directly.
nargs[fn]	Returns NIL if fn is not a function, and the number of arguments of fn if it is. It returns 1 for functions of type EXPR*, FEXPR*, CEXPR*, CFEXPR*, CSUBR* and CFSUBR*.
arglist[fn]	Returns with the list of arguments of the function fn. Causes an error if fn is a built-in function or undefined.

arg[n]	This function works with a function of type EXPR* or CEXPR*. It returns argument n of that function. It is undefined if $n \leq 0$ or $n \geq m$ where m is the number of arguments bound.
setarg[n;v]	Sets argument n of an EXPR* function to v.

SECTION IX

THE LISP EDITOR

The LISP editor allows rapid, convenient modification of list structures. Most often it is used to edit function definitions, often while the function itself is running. It is another important feature which allows good on-line interaction in the BBN-LISP system.

Editor Language Structure

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

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

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

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

where * indicates that this line was typed by the user.

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

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

Thus:

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

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

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

Note the use of several commands on a single line. In BBN LISP, a carriage return is printed automatically whenever a right parenthesis is typed which causes the parenthesis level to become a zero. Therefore, a non-atomic command is necessarily always the last command on its line.

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

Attention Commands

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

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

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

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

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

(NTH -n) may also be used, with the expected result:

```
*(NTH -3)
*P ↑
 ((X) Y (COND & &))
```

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

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

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

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

One more variation is provided by (F e n), which finds the nth occurrence of e anywhere in the structure. The search is done in printout order, so for example:

```
*↑ (F X 1)
*P
 (X)
*↑ (F X 2)
*P
 (X)
*Ø P
 (NUL X)
*↑ (F X 3)
*Ø P
 (CDR X Y)
```

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

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

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

Three facilities are available for saving information relating to the current state of the edit and later retrieving it. At any stage in the edit, a mark can be made and later returned to. The commands are MARK, which marks the current state for future reference; ←, which returns to the last mark without destroying it; and ←←, which returns to the last mark and forgets it. For example:

```
*↑ 4 2 P
((NUL X) Z)
*MARK ↑ (F CONS T)
*P
(CONS (CAR) (APPEND &))
*↑ P
(LAMBDA (X) Y (COND & &))
*←← P
((NUL X) Z)
*← P
?
```

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

A more drastic marking facility is available if it is desired to save the state of the edit in its entirety. Since changes are made as they are typed in, there is no simple way to undo part of an edit. However, the command COPY will make a copy of the entire state of the edit, which may be retrieved with RESTORE. This has the effect of undoing all changes made since COPY was given, since the copy is not affected by editing commands given after the copy was made. This facility is unlike MARK in that a second COPY obliterates the list saved by the first. Furthermore, since RESTORE retrieves the copied edit state and not a copy thereof, subsequent RESTOREs will definitely not have the desired effect.

Frequently it is desired to move or copy a sublist from one place in the structure being edited to another. No command for performing this particular operation is provided. However, it is possible to set a variable to the current list or a sublist thereof. The I command described below can then be used to treat this value exactly as though it had been typed in literally. In particular, the command (S v), where $\underline{v}$ is a variable name, sets $\underline{v}$ to the current list. (S v Ø) may also be used. Thus:

```
    *↑ (S EL2 2)
```

will result in setting the value of EL2 to the sublist (X).

<u>Modification commands</u>

Just as most general text editors contain INSERT, REPLACE, and APPEND commands, the LISP editor provides facilities for these three basic operations. To insert the S-expressions $\underline{e}_1 \ldots \underline{e}_m$ before sublist $\underline{n}$ of the current list, one simply gives the command (-n e_1 ... e_m), thus:

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

To replace the _n_th sublist with $\underline{e}_1 \ldots \underline{e}_m$, one gives the command $(n\ e_1 \ldots e_m)$, for example:

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

And to append at the end of the current list, one writes $(N\ e_1 \ldots e)$, thus:

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

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

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

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

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

Very often one wants to make a simple change in a list structure, without wanting to know exactly how to trace down the structure to the point where the emendation is to be made. The command (R $e_1 e_2$) replaces all occurrences of $\underline{e}_1$ in the current list and all its substructure by $\underline{e}_2$. This is done using a variant of subst called dsubst that runs faster, and physically replaces the old structure in the list by a copy of the new structure. For example:

```
*↑(R Z Y)
*4 2 P
((NUL X) Y)
```

The mechanism by which lists saved with the S command may be used, among other things, is (I c e_1, ... e_n), which is equivalent to ([atom[c]→c; T→eval[c]] eval[e_1] ... eval[e_n]). Thus for example, if EL2 has been set to (X) as per the sample above:

```
*↑ (I (CAR (QUOTE (F))) EL2 T)
*P
 (X)
```

because the I command is equivalent to (F (X) T).

Structure changing commands

The commands presented in the last section do not allow convenient alteration of the list structure itself, as opposed to components thereof. Consider, for example, the list (A B (C D E) F G). We can remove the parenthesis around (C D E), which is the third sublist, by (LO 3) (this stands for take Left paren Out). This produces the list (A B C D E). LO simply deletes all elements of the original list beyond the one specified. If we want to preserve them, we could say (BO 3), take Both parentheses Out, which produces (A B C D E F G). Conversely, if we want to take the partial list beginning at B and subordinate it one level, making (A (B (C D E) F G)), we can say (LI 2), i.e. put a Left parenthesis in before sublist 2 (and a matching right parenthesis at the end of the list). Again, if we want the matching right parenthesis inserted somewhere other than at the end of the list (after the F, for example), we can say (BI 2 4), put Both parentheses In around elements 2 through 4, which results in the list (A (B (C D E) F) G).

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

Printing commands

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

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

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

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

Another command which is available for examining the environment during editing is (E e), which simply prints the value of e without disturbing the state of the edit. This is done under errorset, so that one can actually try to run the function which one is editing. It should be mentioned that changes are made as soon as they are typed in, so that the state of the definition of a function (which is what is usually being edited) is always exactly what one expects. Also, the variable l contains the state of the edit, with the current list being car[l]. Thus, (E (CAR L)) will cause the current list to be printed by print.

Edit Macros

In editing a set of functions, to make a consistent change in a number of places, one must give the same sequence of commands a number of times. For example, to replace all occurrences of calls to (FOO &) by calls to (FIE & T), (where & stands for any expression), one would type

```
↑
(F FOO T)
(1 FIE)
(N T)
```

as many times as the replacement was necessary. To save this typing, one can define an edit Macro, called RF for example, by typing

```
(M RF ↑ (F FOO T) (1 FIE) (N T))
```

Then each time you type

```
RF
```

the sequence of commands, following the RF in the definition l'st, will be executed. If RF were made the last command in the list, the sequence would be repeated until FOO could not be found.

The simple ed   .cros described above cannot be given any arguments, and will always do exactly the same thing. One can also define macros which use parameters. For example, to define a macro to switch two items in a list, one would type

```
(M SW (A B) (S SW1 A) (S SW2 B) (I B SW1) (I A SW2))
```

where the list of argument names (A B) immediately follows the macro name, SW. To make this macro, SW switch items 2 and 7 in a list, one would type

```
(SW 2 7)
```

This command would substitute 2 for A, and 7 for B, in the macro definition following the argument list (A B); and then execute that sequence of commands with the substituted values. In this case, the sequence would be

```
(S SW1 2)
(S SW2 7)
(I 7 S' 1)
(I 2 SW2)
```

Note that a macro with no parameters is called by typing an atom (its name): a macro with parameters must be called by using its name as the first element of a list, followed by the values to be substituted for the parameters of the macro.

All the edit Macro definitions can be found on a free variable called EDITMACROS. This value can be edited by the editor, and will be the cumulative list of all macros defined since the current sysin was done. New definitions of macros supercede old ones. This feature lets you easily expand the repertoire of edit commands, and thus "program" the editor.

Using the editor

As presently interfaced to the outside world, the editor consists of a basic function for editing S-expressions, edite, and three special NLAMBDA functions for editing values, definitions, and property lists, respectively editv, editf, and editp. Thus,

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

would be used to begin the edit which has been used as the example. When editing is complete, STOP or OK will cause edite to exit with the edited list as value. The three interface functions all return as value the atom being edited, and place the new value in the appropriate place.

In fact, the work of the editor is done by two functions editcom and editdefault. Editcom assumes the existence of a free variable

L, initialized to list of the list being edited; a free variable Y, used to hold the copy made by COPY, if any; and a free variable M, to hold marks made by MARK. It accepts as its argument an editing command and performs the appropriate transformation on these three variables. Unrecognizable commands are passed to editdefault, which is currently defined as λ[[c];error[c]]; the edit is run by edite under an errorset.

A complete example, starting with the erroneous definition given at the beginning of Section IX and ending with the correct definition of append, is given below.

```
     *EDITF(APPEND)
      EDIT
     *(P Ø 1ØØ)
      (LAMBDA (X) Y (COND ((NUL X) Z) (T (CONS (CAR) (APPEND
          (CDR X Y))))))
     *(3)
     *(2 (X Y))
     *P
      (LAMBDA (X Y) (COND & &))
     *(R NUL NULL)
     *(R Z Y)
     *↑(F CAR T)
     *(N X)
     *↑(F CONS T)
     *3 (RI 2 2)
     *P
      (APPEND (CDR X) Y)
     *↑(P Ø 1ØØ)
      (LAMBDA (X Y) (COND ((NULL X) Y) (T (CONS (CAR X) (APPEND
          (CDR X) Y)))))
    *STOP
     APPEND
```

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

A Summary of the Editor Commands

Atoms

n>0	Makes nth element be current level list
n<0	Makes nth element from end be current level list
n=0	Makes previous level be current level list
COPY	Saves a copy of current work
RESTORE	Restores as current work earlier copy
P	Prints current level list to depth 3
↑	Makes current list be the top level list
MARK	Marks this point
←	Makes current level be last marked list
←←	Makes current level be last marked list and forgets mark
STOP or OK	Exit from editor

Other atoms give an error indication of ? if not defined as an Edit Macro. This can be changed by modifying the routine editdefault, currently defined as

```
(LAMBDA (C) (ERROR C))
```

Lists

(n e_1 e_2,..., e_k) $n>0$ $k\geq0$	Replace element _n_ by the _k_ elements _e_1_,..., _e_k_. Deletes the _n_th element if k=0
(n e_1 e_2,..., e_k) $n<0$ $k\geq1$	Inserts _e_1_,..., _e_k_ before _n_th element
(N e_1,..., e_k)	Adds _e_1_ to _e_k_ at end of current level list
(S name) and (S name Ø)	Sets _name_ to current level list
(S name n) $n\geq1$	Sets _name_ to _n_th element
(R old new)	Replaces all occurrences of the _old_ item by _new_ in current level list
(P n m) $n\geq0$	Prints element _n_ to depth _m_ (current list if n=0)
(F e)	Finds _e_ at current level; "&" matches any item, "--" matches any remaining list
(F e T)	Finds _e_ at any level
(F e n) $n\geq1$	Finds _n_th occurrence of _e_ any level

(NTH n)	$n \geq 1$	Makes nth element be first element of current list
	$n<0$	Makes nth element from the end be the first element on the list
(I command $e_1 \ldots e_k$)		Evaluates $e_1 \ldots e_k$ and then performs command as usual. Command can be a number, N, R, F, etc. If command is not atomic, it is evaluated.
(E e)		Evaluates and prints e
(LO n)		Removes left paren before element n (and removes a right paren at end of current list. If there are no more right parens at end of list, elements left hanging "drop off").
(LI n)		Inserts left paren before element n, (and a corresponding right paren at the end of the list).
(RO n)		Removes right paren after element n. It moves it to the end of the current list.
(RI n m)		Inserts right paren in element n after element m. In element n, it moves a right paren from the end of element n which must have more than m elements.

(BO n)	Removes both left and right parens around element <u>n</u>
(BI n m)	Inserts both left and right parens, making a sublist at position <u>n</u> containing elements <u>n</u> to <u>m</u> inclusive.
(M name c_1 c_2...c_n)	Defines <u>name</u> (an atom) as an Edit Macro equivalent to the sequence of commands $\underline{c}_1$, $\underline{c}_2$,..., $\underline{c}_n$.

All other lists cause errors, which print "?". See the statement on <u>editdefault</u> above.

Edit commands are all interpreted in one function <u>editcom</u> which accepts a single command as an argument. It and its subfunctions assume a free variable L initialized to <u>list</u> of the list to be edited; a free variable Y to hold a copy, if requested, and a free variable M to hold any marks created. With these restrictions <u>editcom</u> can be used as a subroutine (as it is in <u>breakin</u>, described in the section on debugging aids). <u>Edite</u> does the reading from the teletype, transmits the commands to <u>editcom</u>, and prints the "?" on errors. All <u>errors</u> and <u>rubouts</u> are caught by the <u>errorset</u> in <u>edite</u>.

SECTION X

ATOM, ARRAY, AND STORAGE MANIPULATION

pack[x]

The argument x of pack must be a list of atoms. The value of pack is a single atom whose print name is a packed version of the print names of all the atoms given in the list. Thus:

```
pack[(A BC DEF G)] = ABCDEFG
pack[(1 "." 3)] = 1.3 a floating
                      point number
```

unpack[x]

The argument of unpack should be an atom. The value of unpack is a list which contains, in order, the characters which make up the print name of that atom.

chcon[x;j]

Returns a list of numbers representing characters in print name of x which must be an atom.

```
  j = NIL  prin1 representation
    = T    prin2 representation
```

gensym[]	This function of no argument generates a unique symbol of the form Annnn, in which each of n's is replaced by a digit. Thus, the first one generated is A0001, etc. This is a way of generating new atoms for various uses within the system.
oblist[]	Creates a list of all atoms currently in the system.
reclaim[]	This function initiates a garbage collection and returns with the number of available LISP words in free storage. See minfs. Atoms with no property list, value or function definition, and not used in any list, are collected.
minfs[n]	Sets the minimum amount of free storage which will be maintained by the garbage collector. If, after automatic garbage collection, fewer than n free words are present, (as printed out by the garbage collector) sufficient storage will be added to raise the level to n.
gcgag[x]	If x=T garbage collector will print a message when entered. If x=NIL no message is printed. Previous setting is returned. Initially set to T.

logout[]	Deactivates users program and returns the user to the time-sharing system executive.
closer[a;x]	Stores x into location a. Both x and a must be numbers. $a<2^{14}$ actual core location $a\geq 2^{14}$ address in virtual address space.
ilp[x]	Unrestricted car of x.
openr[a]	Value is number in a as defined in closer.
loc[x]	Makes a number out of x, i.e. returns the virtual address of x.
vag[x]	The inverse of loc. Unboxes numbers. An unboxed number n which doesn't correspond to the address of a list structure or an atom is printed #n e.g. array pointers.
allocate[n]	Allocates an n word block in array (binary program) space. Returns a pointer to the address of the first word allocated. Returns NIL if no more array (binary program) space is available.

statistics[]	Prints out statistics on number of wraparounds of compiled code; number of mapped stores; total number of mapped references (*car*'s, *cdr*'s, *cons*'s, *rplaca*'s, *rplacd*'s, *getd*'s, etc.); total number of drum references.
storage[]	Prints out current status of storage including number of binary program (array) words in use; number of list words (two 94Ø words) in use; number of 94Ø words available; and number of words used up for print names.

Array Functions

Arrays and compiled code are both allocated out of a common array space. Arrays of pointers and unboxed integers may be manipulated by the following three functions:

array[n,p,v]	This function *allocates* a block of n+4 94Ø words, of which the first 4 are header information. The next $p \le n$ are cells which will contain unboxed integers, and are initialized to Ø. The last $n-p \ge 0$ will contain pointers initialized to *v*. If *p* is NIL it is assumed equal to Ø (i.e., a symbolic array). The value of this function is an unboxed number

	which is the location of the array in virtual memory, and is called an array pointer.
elt[a;m]	Has as value the $\underline{m}^{th}$ element of the array pointed to by $\underline{a}$. For out of bound calls, if m<1 or m>n, where $\underline{n}$ is the length of the array $\underline{a}$, *elt* gives element 1 if m<1, or element $\underline{n}$ if m>n.
seta[a;m;v]	Sets the value of the $\underline{m}^{th}$ element of $\underline{a}$ to $\underline{v}$. On out-of-bounds reference no store is made. The value of this function is always $\underline{v}$. It is the users responsibility to ensure that no pointers are placed in the non-pointer area. Any in that area will not be traced during garbage collection.
arraysize[a]	Returns the size of array $\underline{a}$ if $\underline{a}$ is an array pointer.

There will be three parallel functions, *arrayf*, *eltf*, and *setaf* which will manipulate arrays of unboxed floating point numbers. Until they are implemented, only *pointers* to floating point numbers can be stored in arrays. These will be useless until open floating point arithmetic is available

SECTION XI

FUNCTIONS WITH FUNCTIONAL ARGUMENTS

As in all LISP 1.5 Systems, arguments can be passed which can then be used as functions. Functions which use functional arguments should use variables with obscure names to avoid conflict of variable names with variables used free in a functional argument. There is no "FUNARG device" used in this system. All system functions standardly use variable names consisting of the function name concatenated with x or fn etc. A FUNARG device may be implemented in the future.

function[x]	Similar to quote except that x is the name or definition of a function used as an argument; must be used with all functional arguments.
map[x;fn1;fn2]	If fn2 is NIL (i.e. not provided) this function applies the function fn1 to successive tails of the list x. That is, first it computes fn1[x], and then fn1[cdr[x]], etc. until x is NIL; however, if fn2 is provided, fn2[x] is used instead of cdr[x] for the next call for fn1. Thus if fn2 were cddr, alternate elements of the list would be skipped. If fn2 is a conditional

	expression, then the next element to be looked at can be contingent on a computation.
mapc[x;fn1;fn2]	Identical to *map*, except that fn1[car[x]] is computed each time. If *fn2* is NIL, *fn1* is applied to each element of the list *x* in turn.
mapcar[x;fn1;fn2]	If *fn2* is NIL, this function applies the function *fn1* to each of the elements of the list *x*. It creates a new list which is a map of the old list in the sense that each element of the new list is the value of applying *fn1* to the corresponding element of the old list. If *fn2* is provided, fn2[x] is used instead of cdr[x] for each succeeding computation with *fn1*.
maplist[x;fn1;fn2]	This function computes successively the same values that *map* computes; it forms a new list consisting of successive values of applications of this function.
mapconc[x;fn1;fn2]	Identical to *mapcar* except that it does an *nconc* instead of a *cons*.
mapcon[x;fn1;fn2]	Identical to *maplist* except that it does an *nconc* instead of a *cons*.

This next set of functions is a slightly different extension of the mapping functions usually found in LISP 1.5. They are all defined by EXPR* type expressions, and make no recursive calls. The first argument to each is a function fn of n arguments. Following this first argument should be n lists. The mapping function iterates down the lists by successive cdrs until any one becomes empty and then returns the value specified. At each iteration, fn is applied to these lists, as specified. For example, the function

pair[x;y] could be defined as
maccar[function[cons];x;y]

mac[fn;x_1;x_2;...;x_n]	Similar to map. Applies fn to succes ive cdr's of x_1;x_2;...;x_n. Returns NIL.
macc[fn;x_1;x_2;...;x_n]	Similar to mapc. Applies fn to car's of successive cdr's of x_1;x_2;...;x_n. Returns NIL.
maclist[fn;x_1;x_2;...;x_n]	Similar to maplist. Applies fn to successive cdr's of x_1;x_2;...;x_n. Returns a consed list of these values.
maccar[fn;x_1;x_2;...;x_n]	Similar to mapcar. Applies fn to car's of successive cdr's of x_1;x_2;...;x_n. Returns a consed list of these values.

maccon[fn:x_1;x_2;...;x_n]	Similar to mapcon. Applies fn to successive cdr's of x_1;x_2;..;x_n. Returns an nconced list of these values.
macconc[fn;x_1;x_2;...;x_n]	Similar to mapconc Applies fn to car's of successive cdr's of x_1;x_2;...;x_n. Returns an nconced list of these values.

SECTION XII

VARIABLE BINDINGS AND PUSHDOWN LIST FUNCTIONS

A number of schemes have been used in different versions of LISP for storing the values of variables. These include:

1. Storing values on an association list paired with the variable names.

2. Storing values on the property list of the atom which is the name of the variable.

3. Storing values in a special value cell associated with the atom name, putting old values on the pushdown list, and restoring these values when exiting from a function.

4. Storing values on the pushdown list.

The first three schemes all have the property that values are scattered throughout list structure space, and, in general, in a paging environment would require references to many pages to determine the value of a variable. This would be very undesirable in our system. In order to avoid this scattering, and possible excessive drum references, we utilize a variation on the fourth standard scheme, usually only used for transmitting values of arguments to compiled functions; that is, we place these values on the pushdown list. But since we use an interpreter as well as a compiler, the variable names must be kept. The pushdown list thus contains pairs, each consisting of a variable name and its value. The interpreter need only search down the pushdown list for the binding (value) of a variable.

One advantage of this scheme is that the current top of the pushdown stack is usually in core, and thus, drum references are rarely required. Free variables work automatically in a way similar to the association list scheme.

An additional advantage of this scheme is that it is completely compatible with compiled functions which pick up their arguments on the pushdown list from known positions, instead of doing a search. To keep complete compatibility, our compiled functions put the names of their arguments on the pushdown list, although they do not use them to reference variables. Thus, free variables can be used between compiled and interpreted functions with no special declarations necessary. The names on the pushdown list are also very useful in debugging, for they provide a complete symbolic backtrace in case of error. Thus, this technique, for a small extra overhead, minimizes drum references, provides symbolic debugging information, and allows completely free mixing of compiled and interpreted routines.

There are three pushdown lists used in BBN 940 LISP: the first is called the parameter pushdown list, and contains pairs of variable names and values, and temporary storage of pointers; the second is a number stack for temporary storage of unboxed numbers; the third is called the control pushdown list, and contains function returns and other control information.

The following functions allow one to interrogate these pushdown lists from inside another function. The functions, nthfnback, evalv, setv, variables, and rename take an argument n which, if positive, is the number of function calls which have been made -

essentially the depth of nesting of functions from the top level. If n is negative, it references back from the current call level. The function nthfn returns as a value a positive number which is the number of call levels from the top (consistent with that needed by nthfnback, etc.). The argument n to nthfn (n>o) is interpreted as the nth preceding occurrence (i.e. counting back) of the function named.

nthfnback[n]	Returns the name of function called at call level (position) n
nthfn[fn;n]	Returns the position (number of call levels from top) of the nth occurrence back of function named fn.
evalv[var;n]	Returns the value of variable var evaluated starting at pushdown list position n
setv[var; n; val]	Sets the value of variable var starting at pushdown position n to value val
variables[n]	Returns list of variable names on pushdown list at pushdown position n
rename[old; n; new]	The variable named old at level n will be renamed new. The push-list cell containing the variable name is changed.

retfrom[n;v]	Returns from the function at position *n*, with value *v*. Thus an error[] under a *nlsetq* is equivalent to a retfrom[nthfn[nlsetq;1];NIL].
backtrace[n;m]	Prints out the untrace normally associated with errors, starting at position *n*, and going *back* to position *m* (i.e. n>m). If n=NIL; it is assumed equal to current position; if m=NIL; it is assumed equal to Ø.

SECTION XIII

ARITHMETIC FUNCTIONS

Integer Arithmetic

The following functions all work on integers. When given floating point numbers as arguments, these arguments are fixed (converted to integers) before any operation is performed. Most of these functions are compiled as open code.

plus[x_1;x_2;...;x_n]	Returns an integer $x_1+x_2+...+x_n$
minus[x]	- x
difference[x;y]	This function has for its value the numeric difference between its arguments.
add1[x]	x + 1
sub1[x]	x - 1
times[x_1;x_2;...;x_n]	Returnsan integer equal to the product of $\underline{x}_1,\underline{x}_2,...\underline{x}_n$
quotient[x;y]	Greatest integer in quotient x/y

remainder [x;y]	This function computes the number theoretic remainder for fixed-point numbers.
divide[x;y]	This function yields a dotted pair whose first member is quotient[x;y] and whose second member is remainder[x;y].
numberp[x]	T if _x_ is a number; NIL otherwise. This function works for floating point numbers as well as integers.
greaterp[x;y]	T if x>y; NIL otherwise
lessp[x;y]	T if x<y; NIL otherwise
zerop[x]	T if _x_ is zero; NIL otherwise
minusp[x]	T if _x_ is negative; NIL otherwise
logand[x;...;z]	This function takes the logical and of all of its argument, and return this value as an integer.
logor[x;...;z]	This function takes the logical or of all of its arguments, and return this value as an integer.
logxor[x_1;...;x_n]	Logical exclusive or of $\underline{x}_1,...,\underline{x}_n$

lsh[n;s]	Performs an arithmetic left shift of s$\geq$o on n. Equivalent to n * ^{s}.
rsh[n;s]	Performs an arithmetic shift of s$\geq$o on n. Equivalent to n * 2^{-s}.
abs[x]	Returns absolute value of x.

Floating Point Arithmetic

The floating point arithmetic functions available in BBN LISP are fplus, fminus, ftimes, fquotient, and fgtp. They will accept mixed arguments, i.e. integer or floating point. Just as the integer-type functions fix any floating arguments before performing their computation, the floating-type functions float any fixed arguments before performing a computation. Thus the result of a floating point function is guaranteed to be a floating point number.

The functions specifically related to floating point are:

fgtp[x;y]	Floating greaterp; compares by subtraction
fix[x]	Returns integer part of x
float[x]	Produces floating number
floatp[x]	Returns T if x is a floating point number, NIL otherwise
fminus[x]	Negative of x
fltfmt[x]	Output format control; x is defined as the time-sharing system formatting of floating point output
fplus[x_1;x_2;...;x_n]	Returns the sum of its arguments

fquotient[x;y]	Returns x/y
ftimes[x_1;x_2;...;x_n]	Product of its arguments

Equal and eqp will compare two floating point numbers for equality, and will float an integer to compare it to a floating point number. Eq when compiled is an open 24 bit compare which usually won't work for arithmetic comparisons. Equal uses eqp.

SECTION XIV

INPUT/CUTPUT FUNCTIONS

Opening and Closing Files

BBN 940 LISP 1.69 allows the user to have any number of files open at a given time. Restrictions in the time-sharing system currently limit this to a maximum of 2, however. A file is identified by its LISP File Name.

The three basic file manipulation operations are:

    infile[name;type]

used to open for input the file named name, which must be of type type (i.e., for binary, 2, or for symbolic, 3) if type is not NIL. Its value is the name of the file if it was opened successfully, or NIL otherwise. The standard input file is set to name. T is the name of the teletype as an input (or output) file.

    outfile[name;type]

opens for output the file name, which is set to type type if type is not NIL, and otherwise to type 3, symbolic. Its value is the name or NIL as for infile. It sets the standard output file to name.

closef[x]	Closes the named file. If x is NIL, it attempts to close the standard input file if other than teletype. Failing that, it attempts to close the standard output file if other than teletype. Failing either, it returns NIL. If it closes any file, it returns the name of that file. If it closes either of the standard files, it resets that standard file to teletype.
openp[x]	Returns NIL if x is not an open file, returns x if x is an open file.

At any given time one input and one output file are selected as primary (the exact meaning of this is given below). Normally these are both T for teletype input and output. The primary input file may be changed by

```
input[name]
```

which sets name to the primary input file. Its value is the name of the old primary input file. Similarly, the primary output file may be set with

```
output[name]
```

which has the obvious effect. To read the current setting of the primary input and output files

```
input[]
```

and

```
output[]
```

may be used.

Input/Output Transmission

Without exception, functions that actually read or write on files may be given an additional argument which is the name of the file on which the operation is to take place. If the additional argument is NIL, the primary file will be used.

The following functions perform output:

```
feed[n]
```

produces _n_ carriage returns and line feeds:

```
prin1[a]
```

prints its argument.

```
prin2[a]
```

prints the expression _a_ with double-quote marks inserted where required for it to read back in properly; both _prin1_ and _prin2_ print lists as well as atoms. Neither print a carriage return upon termination, both have value a.

prin3[a]	Prints a using double quotes for separation and break characters specified by setbrk and setsepr as described under ratom
print[x]	Prints the S-expression x; uses prin2; its value is x
spaces [n]	Produces n spaces; its value is NIL
terpri[]	Produces a carriage return and line feed; its value is NIL

If any print function is given an unboxed number n, it will print it as #n with n in octal.

The print functions print, prin1, prin2, and prin3 are all effected by a level parameter set by

    printlevel[n]

The variable n≥Ø controls the number of unpaired left parentheses which will be printed before any list will be printed as &.

Suppose x = (A (B C (D (E F) G) H) K)

Then if n = 2, print[x] would print

    (A (B C & H) K)

and if n = 3,

```
(A (B C (D & G) H) K)
```

and if n = 0, it prints as just

```
&
```

The value of printlevel[n] is the old parameter setting.

In order to change the level dynamically, while the system is printing at you, you can type control-P followed by a number, i.e. a string of digits, followed by a period. The print level will immediately be set to this number for this printout. If the print routine is currently deeper than the new level all unfinished lists above that level will be terminated by "--)". Thus, if a circular or long list of atoms, is being printed out, typing in

$P^c Ø.$

will cause the list to be terminated. After this printout, the level will be returned to its previous setting. Only printlevel (not P^c) changes the print level permanently.

character[n]	This function outputs on the teletype a single character with octal ascii representation (code) n. n must be a number.

Input Functions

read[]	Reads one S-expression from the current file
rdflx[x]	If *x* is NIL this function will try to read one S-expression with read[]; if no error occurred in reading, it will return with *list* of the S-expression that was read. If an error occurs in reading, it returns with NIL. If *x* is not NIL, it will attempt to read an S-expression and keep attempting to read it until it gets one without an error; each time it tries to read an S-expression and gets an error, it will print out *x*. In this case it returns with the S-expression itself (not *list* of the S-expression).
ratom[]	Reads in one atom from the standard file. Separation of atoms is defined by the functions *setsepr* and *setbrk*.
ratoms[a]	Calls *ratom* repeatedly until atom *a* is read. Returns a list of atoms read not including *a*.

setsepr[x]

Arguments should be octal numbers, e.g., 155q for carriage return.

setbrk[x]

Characters defined by *setbrk* will delimit atoms and be returned as separate atoms themselves. Characters defined by *setsepr* will not be returned and will serve only to separate atoms. For example, to make *ratom* read in ordinary format, space (0q), comma (14q), and carriage return (155q) are separation characters, and left paren (10q), right paren (11q), and period (16q) are break characters. Thus

```
setsepr[0q 14q 155q]
setbrk[10q 11q  16q]
```

would set up these characteristics. The value of *setsepr* and of *setbrk* is NIL. Use *chcon* to find numeric codes for characters. The tables are initially set to this standard LISP set of break and separator characters.

setseprc[x]

Same as *setsepr* except that *x* is a list of characters.

setbrkc[x]

Same as *setbrk* except that *x* is a list of characters.

ratest[x]	Performs two functions depending on setting of x.
	If x = T ratest returns indicator which is:
	T if a separator was encountered immediately prior to last atom read by ratom.
	NIL if there was no separator between last two atoms returned by ratom.
	If x = NIL it returns an indicator which is:
	T if last atom returned by ratom was a break character.
readc[]	Reads the next character. Not affected by setsepr and setbrk.

Input/Output Control Functions

These functions perform a variety of operations on the state of files. Those marked with * do not take the optional extra argument to indicate a file.

* clearbuf[]	Clears the input buffer of the file (not particularly useful for any file other than the teletype)

* radix[n;i]	Sets output radix to n and sign indicator to i. If i is T, negative numbers will print as sign and 23 bit value (normal). If i is NIL, all numbers print as 24 bit unsigned integers. Returns previous setting.
* control[j]	Sets modes for reading with ratom as follows:
j = T	Eliminates LISP'S normal line buffering (and also eliminates automatic detection of control-A and control-Q as line-editing characters on the TTY).
j = NIL	Restores line buffering (normal).
j = 0	Eliminates the echo of the character being deleted by control -A.
j = 1	Restores the echo (normal).
* linelength[n]	Sets the length of the print line for all files. The value is the former setting of the line length.
* position[]	Gives the character position on the print line. No guarantees are made about its meaningfulness if output is being done intermittently to more than one file.

* readp[]	Gives T if there is something in the input buffer (either the TSS input buffer or LISP'S line buffer) and NIL otherwise.

Special Functions

sysout[name]	Dumps the entire state of LISP on the file named. This name should not specify a drum file, since more than 38K of information (the maximum for a sequential drum file) will always be written. When the LISP system is reassembled, old *sysout* files are no longer readable.
sysin[name]	Restores the state of LISP from a *sysout* file. *Sysin* may only be done once after entering LISP. If it returns NIL, the file was not found, or was no longer a valid *sysout* file. *Sysin* will return T if it was successful.
rbin[x]	Reads one *word* from *x*, the specified file. This function returns the word as a number.
wbin[w;x]	Writes one word, *w*, on file specified by *x*. *W* must be a number.

Files opened for binary I-O should be closed by closef in the usual way.

Symbolic File Input

load[x;p] — load is a function which reads successive S-expressions from file x and evaluates each as it is read. If p = T, then load prints the value; otherwise it does not. load continues reading S-expressions and evaluating them, until it reads the single atom STOP followed by a carriage return, at which point it returns the value NIL. Using load is the standard way of getting functions in from files. It saves having to [illegible]

Symbolic File [illegible]

prettyprint[illegible] — [illegible]

prettydef[fns;file;vars]	This function is used for the creation of files containing systems of functions.

The arguments are interpreted as follows:

<u>fns</u> (first argument)	If a list, it is treated as a list of function names. If <u>fns</u> is an atom, it should have as a binding the list of functions for <u>prettydef</u>. The functions on the list are <u>prettyprinted</u> surrounded by a (DEFINEQ ...) so that they can be loaded with <u>load</u>. In addition, a SETQQ will be written which saves the list of functions on the named atom, and a PRINT will be written which informs the user of the named atom when the file is subsequently loaded.
<u>file</u> (second argument)	The name of the file on which the output is to be written. The following options exist: file=NIL The standard output file is used as determined by the last setting of <u>output</u>. file=atom The file <u>atom</u> is opened if not already open, and becomes the standard output file.

file=list

<u>Car</u> of the <u>list</u> is assumed to be the file name and is opened if not already open. The standard output file is not changed in this case.

<u>vars</u> (third argument)

This option is used where there are a number of atoms having top level bindings which the user wishes to save on the output file. The following options exist:

If <u>vars</u> is an atom, this atom is evaluated and should yield a list of atoms. For each atom in this list, a SETQQ will be written which will restore the top level binding to the atom when the file is loaded. In addition, a SETQQ and PRINT are written which save and print <u>vars</u> as described above for <u>fns</u>. If the list contains STOP as its last element, <u>endfile</u> will be called on the specified file, closing it as described above.

If <u>vars</u> is a list, the list is handled as above, except that the SETQQ and PRINT saving the list itself are not written.

As an additional option, if DATE is bound, "THIS FILE WRITTEN ON date" will be printed when the file is loaded.

Examples:

```
     PRETTYDEF((FOO1 FOO2) /FOO/)
```

The file /FOO/ is now open, regardless of whether it was open before. Furthermore, /FOO/ is the new output file.

```
     PRETTYDEF((FOO3 FOO4) (/FIE/))
```

The file /FIE/ is opened, if necessary, and FOO3 and FOO4 are written on it. /FIE/ is not closed. /FOO/ is still the output file.

```
     PRETTYDEF((FOO5 FOO6))
```

FOO5 and FOO6 are written on /FOO/

```
     PRETTYDEF((FOO7 FOO8) /FIE/ (STOP))
```

FOO7 and FOO8 are written on /FIE/ which is closed with a STOP at the end. The output file is now the teletype.

```
     SET(FOO(FOO1 FOO2 FOO3))
     PRETTYDEF(FOO /FOO/)
```

FOO1, FOO2, FOO3 are written on /FOO/. Also written on the file are (SETQQ FOO (FOO1 FOO2 FOO3))
and
(PRINT (QUOTE FOO)).

```
SET (FOOVAR (ZOT MUMBLE STOP))
SET (ZOT T)
SET(MUMBLE NIL))
PRETTYDEF(FOO /FUM/ FOOVAR)
```

The following are written on /FUM/: definitions of

```
FOO1, FOO2, and FOO3;
(SETQQ FOO (FOO1 FOO2 FOO3));
(SETQQ FOOVAR (ZOT MUMBLE STOP)); (SETQQ ZOT);
(SETQQ MUMBLE NIL); (PRINT (QUOTE FOO));
(PRINT (QUOTE FOOVAR)); and STOP.
```

The file is closed.

As you might surmise, the most convenient way to use _prettydef_ is as follows: set a variable to the list of the functions desired in a particular file, say FOO, and another variable to a list of variables to be set in that file, if any; _prettydef_ will do the rest. Then if you do

```
LOAD(/FUM/)
```

you will see

```
THIS FILE WAS CREATED ON 4-06 (if you had set DATE)
FOO
FOOVAR
STOP
```

and the file will be loaded.

clock[n]	for n=0	value of time of day clock, i.e., number of seconds since midnight
	for n=1	time of day user logged in
	for n=2	number of seconds of compute time since user logged in
	for n=3	time spent in garbage collections

time[x;n;g]

Time executes the computation _x_, _n_ number of times, and prints out the number of _conses_, total time/n if n≠1 and computation time per iteration. Garbage collection time is not included, i.e., it is subtracted out. If _n_ is NIL, it is set to 1. If _g_ is T, garbage collection time is also printed.

Example:

```
TIME ((CONS NIL NIL) 1000 T)
GARBAGE COLLECTION
2458 CELLS
1 CONSES
12/1000=0.12000E-01 SECONDS
GARBAGE COLLECTION TIME: 23 SECONDS
(NIL)
```

```
TIME ((PRETTYDEF (QUOTE (FOO))))
0 CONSES
9.0 SECONDS
(FOO)
```

SECTION XV

ERROR HANDLING AND DEBUGGING FUNCTIONS

Error Handling

Errors in BBN LISP are dichotomized into two classes: H errors for which the user can provide Help on the spot; and $\overline{H}$ errors, for which no help is possible. $\overline{H}$ errors in the LISP system normally cause a trap to a routine which prints an error message and unwinds the pushdown list. While unwinding the pushdown list, LISP prints the functions which have been entered, and their arguments. The most recently entered function is printed first, etc. until the top level <u>evalquote</u> if reached. This printout can be terminated by pressing RUBOUT; this will return you to the LISP executive. See <u>printlevel</u> for a discussion of modifying the printout without terminating it. The function

error[x]	induces an $\overline{H}$ error, printing a message <u>x</u>

An $\overline{H}$ error can be induced from the console by pressing the RUBOUT.

To prevent $\overline{H}$ errors from stopping all computation by unwinding to the top level, the following functions can be used:

errorset[form; flag]

This function calls eval with the value of form. If no error occurs in evaluation, it returns with a list containing one element, the value of eval[form]. If an error was encountered in the evaluation, it returns NIL. Note that NIL can only be returned if there was an error. A value NIL is returned as (NIL). The argument flag controls the printing of error messages. If flag=T, the error message is printed; if flag=NIL it is not.

On an error the pushdown list is unwound to the errorset, but no further. Printing the untrace of functions and arguments on unwinding to an errorset is controlled by esgag. If an error was induced by a RUBOUT, a second RUBOUT seen by LISP within 3 seconds will cause an immediate untrace past all errorsets.

esgag[g]

Sets the unwinding flag for errorset to g, and returns old value. If g=T an untrace will be printed on an unwind to an errorset. If g=NIL no untrace will be printed. Initially set to NIL.

ersetq[x]

An FEXPR equivalent to errorset, with the argument x quoted, and flag=T.

nlsetq[x]	An FEXPR equivalent to errorset, with x quoted, and flag=NIL.
quit[x]	Induces a "strong" error which will unwind through errorsets to the top level. It prints the error message x. An untrace is printed.
reset[]	Induces a "strong" error which will immediately return you to the top level with no untrace.

There are three types of errors which will allow the user to fix the mistake, and let the program continue. On these errors, the system will call breakl, described below, through either of two functions interrupt or faulteval. These Helpable errors are:

1) An unbound atom
   This usually occurs when an atom has been misspelled or not set at the top level, but may also occur because of an error in syntax. When this occurs the system will print the message

   ```
   UNBOUND ATOM name
   ```

   where name is the unbound atom, and breakl will print

   ```
   (name broken)
   ```

   Then all the options of breakl are available which will allow, for example, the user to set the atom, or return a value without setting it, editing the function with the error, etc.

2) Undefined car of form
An H error is induced, and the system types

UNDEFINED CAR OF FORM <u>atom</u>

where <u>atom</u> is the one for which the error occurred. This usually implies that the function has not yet been defined, or that its name was mistyped. The user can then define the function, or return a value etc. The entire form is bound in <u>breakl</u> to a variable called BRK1EXP.

3. Undefined function
If in compiled code, a function is called which is undefined, the system will print

UNDEFINED FUNCTION <u>function</u>

and <u>breakl</u> will then print

(<u>function</u> BROKEN)

where <u>function</u> is the function not defined.

The user may define this function as a LAMBDA expression with spread arguments only, if the function was also undefined at compiled time. The arguments (up to 12 of them) are bound in the interrupt routine to

ARG1, ARG2,..., ARG12

and can be examined in the usual way in breakl.

Inducing H errors

In addition to these errors detected by the system, the user may induce an H error by typing H^c (H with the control button pressed). At the next point a function is about to be entered, the system will type

INTERRUPTED BEFORE function

and breakl will type

(function BROKEN)

At this point the user can examine the status of his computation, by evaluating variables, or exploring the pushdown list with the appropriate functions (as of course can be done in any entry to breakl). The arguments are again bound to

ARG1, ARG2,..., ARG12

As usual, in breakl the function call will be continued if the user types OK or GO.

In all H errors, the function or atom in question will be bound to the variable FUNCTION. The form which will be evaluated on an EVAL, GO, or OK is bound to BRK1EXP. The number of interrupts which have been done before are bound to the variable INTERRUPT. If a new H error occurs within 6 call levels of an H breakl, the interrupt routine will <u>not</u> be entered again; an $\overline{H}$ error will be induced, and the user will be back in the earlier H interrupt. If a (GO <u>name</u>) or (RETURN <u>exp</u>) is evaluated, <u>break1</u> will be left immediately and quietly, and these functions executed in the last interpreted <u>prog</u> on the pushdown list. The user should avoid redefining the functions <u>faulteval</u> and <u>interrupt</u> which are called by the system on H-errors 1 and 2, and H-error 3 and H^c respectively. To suppress all calls to these functions, the user should set the free variable HELPFLAG to NIL.

<u>Debugging Functions</u>

There are three facilities in the system for easily modifying function definitions to allow a user to follow the flow of control and variable bindings in his programs. These three facilities together are called the break package. All three redefine functions in terms of a system function, <u>break1</u>, described below. <u>Trace</u> modifies a definition of a function <u>fn</u> so that whenever <u>fn</u> is called, its arguments (or some other values specified by the user) are printed. When the value of <u>fn</u> is found it is printed also.

<u>Break</u> modifies the definition of <u>fn</u> so that if a break condition (defined by the user) is satisfied, the process is halted temporarily on a call to <u>fn</u>. The user can then interrogate the state of the machine, perform any computations, and continue or return from the call.

<u>Breakin</u> allows the user to insert a breakpoint <u>inside</u> an expression defining a function. When the breakpoint is hit, and if a break condition (defined by the user) is satisfied, a temporary halt occurs and the user can again investigate the state of the computation.

Breakl

The basic function of the break package is breakl. It allows the user to interrogate the state of the world and to affect the course of the computation. Once a function is broken, the user may type in forms to eval and, under heavy errorset protection, see the value of the computations. In addition, he has the following options that are specifically recognized by breakl:

GO	Releases the break allowing the computation to proceed. When the function is evaluated, its value is printed.
OK	Similar to GO except value is not printed. When the function is evaluated, just the function name is printed.
ERROR	Causes an error return from breakl (all other errors will maintain the break). This is a useful way to get back to the preceding break.
↑	Unwinds to the top - i.e. it evaluates (RESET).
RETURN form	The value of form is returned as the value of the function broken.

EVAL	The computation proceeds but the break is maintained so that after the function is evaluated, a message to this effect is printed and the user can interrogate the value which is stored on the atom VALUE.

Whenever an error occurs inside of a break, either by RUBOUT, or otherwise, the break is maintained. Printing of the function value is done (with a function bpntØ) to a depth of 4; therefore circularities through the car are permissible.

Break

Break is an NLAMBDA which takes any number of functions to be broken. The functions may be of type EXPR, FEXPR, SUBR, etc., or even undefined. In the case of SUBRs, break will require the names of the arguments and will ask for them on the teletype. Break will usually establish unconditional breaks, i.e. the function will always be broken. To set up a conditional break, one can use a list instead of a function name in the call to break. The first element of the list should be the name of the function, the second the break condition, and the third - if present, a value to be printed. Thus

```
     BREAK(FOO1 (FOO2 (GREATERP N 5) (CAR X)))
     (FOO1 FOO2)
```

will cause a break in FOO1 whenever it is called, and a break in FOO2 whenever it is called with N greater than 5. In the latter case, (CAR X) will be printed, using bpntØ.

In general, if the break condition (the second element of the list) evaluates to T, the function will be broken, and the value of the third element will be printed. If the break condition evaluates to NIL, no break will occur. If the break condition evaluates to (NIL), then the value of the third element and the value of the function will be printed, but no break will occur. This is effectively what happens in <u>trace</u>.

<u>Trace</u>

<u>Trace</u> is also an NLAMBDA which takes any number of functions to be traced. In the normal mode of operation, the arguments of the function will be <u>bpntØed</u> as well as the value. To print out other values, list the function, followed by the values. Thus

```
TRACE(FOO1 (FOO2 Y) (FOO3 (CADR X)Z) (FOO4))
     (FOO1 FOO2 FOO3 FOO4)
```

will cause FOO1 to be traced, printing out all of its arguments, FOO2 to be traced printing out Y, FOO3 to be traced printing (CADR X) and Z, and FOO4 to be traced printing out nothing except the name FOO4. In every case, the value of the function is also printed. The features of <u>trace</u> are exemplified further by the following:

(1) The user can specify the values of interest to him in addition to or instead of the arguments of the function, by writing a list headed by the function followed by the values of interest, in place of just the function name.

Example:

```
TRACE(FOO (FOO1 Y (CAR Z)))
(FOO FOO1)

FOO(A B (C D))
FOO:
X=A              ... Arguments of FOO
Y=B
Z=(C D)

FOO1:
Y=A
(CAR Z)=NIL
```

etc.

(2) The user can specify the level to which the arguments, or values, are to be printed by writing (FN N X Y Z ...) in the call to trace. N is taken to be 4 if not specified by this device.

(3) If an error occurs, or RUBOUT is pressed, while a function is being traced, a normal break occurs and, the user can proceed from that point.

Example:

```
TRACE(FACTORIAL)
FACTORIAL

FACTORIAL(2)
FACTORIAL:
N=2

FACTORIAL
N=1

FACTORIAL:
RUBOUT       ... RUBOUT pressed here

(FACTORIAL BROKEN)  ... break occurs
N
0
EVAL

FACTORIAL EVALUATED
FACTORIAL
1
OK
FACTORIAL  ... exit from break

FACTORIAL = 1

FACTORIAL = 2
2
```

Breakin

The third way to use breakl is by means of breakin. Breakin inserts a call to breakl inside of a function definition. In other words, although it is impossible to break on eq, or quote, because so many functions use it, it is possible to break at the point eq or quote is called from some other function.

Breakin is a function of four arguments: FN, WHERE, WHEN, and WHAT. FN, WHEN, and WHAT play the same role as the three arguments shown when break is called with a list instead of an atom, i.e. they specify under what conditions a break should occur, and what is to be printed. The second argument, WHERE, specifies where the break is to be inserted.

WHERE can be either (BEFORE ...) (AFTER ...) (AROUND ...). "..." is used by the editor's find command to locate the correct point. Thus (BEFORE COND 3) will break before the third COND, and (AROUND (SETQ X --)) will break around the first place that X is set. With the exception of labels in a top level PROG, you cannot specify a BREAK AROUND, BEFORE, or AFTER an atom, because breakin automatically changes the atom to (atom --) when calling the editor. Thus, (BEFORE COND 3) is the same as (BEFORE (COND --) 3).

The first time that breakin is called, it copies the function definition. Subsequent times it merely searches for the appropriate location and smashes the function definition. If the location is not found, breakin prints "?". If the function is a machine code function or undefined, it is not possible to breakinside of it.

Unbreak

Unbreak restores functions modified by break, trace, or breakin to their original state. It is possible to do multiple breaks, traces or breakins in any combination without first performing unbreak. Unbreak is an NLAMBDA which takes an indefinite number of functions to be restored. The variable ALL is set to a list of all functions broken. Unbreak[ALL] will restore all functions to their original state. Since unbreak[FOO] does not remove FOO from the list ALL, a subsequent unbreak[ALL] will cause (FOO NOT BROKEN) to appear in the value of unbreak.

SECTION XVI

THE COMPILER AND LAP

The Compiler

The compiler is a separate sysout file on the system tape, usually called COMPILER. To use the compiler, copy your symbolic files onto the drum, enter LISP and do a SYSIN (COMPILER). You can now load your functions, thus defining them, and then use the function compile; or you can compile directly from drum using rcompile. The latter is recommended to avoid a clash of function names with the compiler. The compiler compiles to a LAP2 code which can be written out symbolically on the drum and loaded into any standard system, using loadc.

compile[x]	This will compile all the functions on the list x example: COMPILE((FOO FIE)) will compile FOO and FIE if they are defined
rcompile[]	This will compile from a drum file whose name will be requested after the compset questions have been answered

When either of these functions has been called, they call a function compset which asks a number of questions. Those that can be answered "yes" or "no" can be answered with YES, Y, or T for YES; and NO, N, or NIL for NO. The questions are:

(SETUP - TYPEOUT?)

This question should be answered YES only if you want to see the LAP and LAP2 code produced by the compiler printed on the teletype. If you answer 1 or 2 you will see the output of pass 1 or 2, respectively of the compiler. Usually one should answer NO to this question.

(STORE AND REDEFINE?)

This question should be usually answered NO, unless you want to work with your functions within the compiler system. If you answer YES, you will be asked the question

(SAVE EXPRS?)

If you answer this YES, the exprs will be saved on the property list of the function name. Otherwise they will be discarded.

(NO-SPREAD NLAMBDAS-)

If there are any NLAMBDA's with atomic argument lists called from your functions to be compiled which are not defined, answer the question with one of the following:

S	Means Same list as now on the free variable NLAMA

ADD (fn_1;...;fn_k)	Add <u>fn_1</u> to <u>fn_k</u> to list saved on NLAMA.
REMOVE (fn_1;...;fn_k)	Remove functions from NLAMA
EDIT	The editor will be called and you can edit the list of functions.
(fn_1;...;fn_k)	Set NLAMA to the list of functions
NIL, N, NO	Set NLAMA to NIL

Any other atom will cause a question mark to be printed and let you answer again. Then <u>compset</u> will ask:

    (SPREAD NLAMBDAS-)

Answer in the same way. The free variable used by the compiler is called NLAMS this time.

    (OUTPUT FILE)

This question is always asked. You should usually provide the name of a drum file on which you wish to save the LAP2 code generated. If you answer T, TTY or TELETYPE, the listing will be typed out on the teletype. If you answer N, NOTHING or NIL, output will <u>not</u> be done. If the file named is already open, it will continue to be used.

When the compiler is operating, it will normally print out the name of the function compiling, a list of its bound variables and a list of its free variables. When <u>compile</u> returns, it prints its list of the functions compiled. The value of rcompile is NIL.

When you have finished compiling all the functions you wish to dump on one drum file, print NIL on that file and close the file with closef[name].

The code dumped on the file can be loaded into any standard system by using

```
     loadc[file;fl[illegible]]
```

where if flag=T the names of the functions loaded will be printed.

Affecting the Compiled Code

There are three ways you can affect code compiled for you. You can make a function fn compile open (as an open LAMBDA expression) by putting the expression defining it (including the LAMBDA) on the property list of fn after the flag MACRO, and adding fn to the list which is the value of OPENFNS. Abs and memb are functions currently compiled open. The effect is the same as if you had written the LAMBDA expression in place of _fn_ wherever it appears in a function being compiled. This saves the time necessary to call a function (about a millisecond) at the price of more compiled code generated.

By putting on the property list of _fn_ under the flag MACRO an expression starting with an atom other than LAMBDA, one can obtain the usual MACRO feature of LISP. The atom which starts the list is bound to _cdr_ of the form in which _fn_ appears. The expression following the atom is evaluated, and the results of this evaluation are compiled. _List_, _mapc_ and _map_ are compiled using this technique. For example: _list_ has on its property list the expression (X (GLIST X)), where _glist_ is defined as

```
    (LAMBDA(L) (COND((NULL L)NIL) (T (LIST (QUOTE CONS) (CAR L)
    (GLIST (CDR L))))))
```

If the value of the result of this evaluation is the atom, INSTRUCTIONS, no code will be generated. It is then assumed the evaluation was done for effect and the necessary code has been added. This is a way of giving direct instructions to the _compiler_ if you understand it.

Finally, an expression following MACRO on the property list can start with a list of atoms which are dummy variables for a substitution MACRO. Each atom is paired with a corresponding element in the form containing _fn_. Then these elements are substituted for their paired atoms in the expression following the list of atoms, and this substituted expression is compiled. The functions

addl, _subl_, _neq_, _zerop_, _lessp_, _minusp_, _difference_, _ersetq_ and _nlsetq_ are all compiled open using these substitution macros. For example, on the property list of addl is the expression ((X)(PLUS X 1)).

LAP and LAP2

The compiler has two main passes. The first compiles into a fairly powerful macro language we call LAP; and this is expanded into a simple assembly language called LAP2. The user can write code directly in LAP to be compiled for LISP. It can be processed by the function

lap[fn;v;free;m;c]

Where fn is a function name; v is its list of bound variables; free is a list of variables used free; m is the maximum position used on the pushdown list; and c is the code to be compiled. LAP expects the flag LAPFLG to be set to NIL, T, 1, or 2 to determine printout on none, all, first or second pass code respectively. The variable STRF should be T or NIL, to store or not store the definition. The variable SVFLG should have value NIL (T only to save expr's) and LCFIL should be set to the name of an open file on which the output is to be placed. FTYP should have value EXPR.

The code is a list of instructions, which are lists, and atoms which are treated as labels. Instructions are lists with at least two elements. The first element, fn, can be an op code, a substitution macro, a lambda macro, or a function macro. These LAP macros are completely separate from the compiler macros. In the first three cases, fn has on its property list a property OPD with a value we will call mc. A function macro is the default case, and a list of code to be used is computed by applying fn to cdr of the instruction, and this new list is assembled. Useful function macros in the system will be described later.

If mc is a number, then fn is an opcode of the 940. The codes defined at the moment, with their values, are listed at the end of this section.

Instructions containing these codes as first elements are dumped in symbolic form and at load time are added to the second of the instruction. If the third element is I, the index bit is set in the instruction.

The substitution macros are those where mc is a list which starts with a list of dummy variables for the macro. Corresponding elements of the instruction are substituted for the variables in the macro which is cdr[mc], and this new list of instructions is compiled, before the next instruction on the original code is compiled. When mc starts with LAMBDA, (a lambda macro) similarly to the default case, a list of instructions is computed by applying mc to cdr of the instruction. The substitution and lambda macros in the system are listed at the end of this section.

The important ones of this group are: (where A indicates accumulator)

(LDV X)	Load variable X into A
(STV X)	Stores A into X
(LDT N)	Load A from stack position N (a stack position is a pair of 94Ø words n≥Ø)
(STT N)	Store A in stack position N and store Ø in variable slot. This is important to prevent garbage collector foul ups.
(NSTT N)	Store A in stack position N, do not store Ø.
(LQT X)	Load X quoted as a constant
(LDN N)	Load an unboxed integer from position N
(STN N)	Store an unboxed integer in position N
(CLL L K U)	Call function L with K/2 args, and U/2 positions used up on the pushlist through last argument of function called
(CLLA L K U)	Calls a functional argument L as above
(RET)	Standard return, only one per function is usual
(BE N B)	Branch to B if A=N
(BNE N B)	Branch to B if A≠N
(BN B)	Branch to B if A=NIL
(BNN B)	Branch to B if A≠NIL
(BA B)	Branch to B if A an atom
(BNA B)	Branch to B if A not atom
(BI B)	Branch to B if A a number
(BNI B)	Branch to B if A not number
(BIS B L)	Branch to B if A=L a quoted constant
(BNS B L)	Branch to B if A≠L a quoted constant
(JUMP B)	Branch to B

(CONSCLL N)	Calling sequence for *cons* of element in stack position *N* with contents of A.
(CCALL OP)	Used with either op=CARCLL or op=CDRCLL to call *car* or *cdr* A.

The important function macros are:

(PRGREF OP B)	This must be used for all instructions whose address *B* is a number relative to the beginning of the code and therefore must be relocated on loading. In computing *B*, remember that LAP inserts 4 instructions at the beginning of your code for initialization.
(BRANCH OP B)	Must be used for all instructions which reference a label (branch point) in the code.
(RELREF OP N)	Used when *N* is relative to current location.
(LITREF OP EXP)	Stores *EXP* in a list of literals and computes the address of the literal for use in the compiled code. Used to get any literal quantity into the code.
(VREF OP X)	Computes the position on the stack of the variable named *X*.
(STKREF OP N)	Computes the actual address on stack of position *N*.

For further information, consult the document "An Annotated LISP Compiler" by Bobrow, Murphy and Deutsch (forthcoming).

MACROS for the compiler

The following expression when loaded with the compiler defines all the MACROS used by the compiler.

```
 (DEFLIST(QUOTE(
   (LIST (X (GLIST X)))
   (ADD1 ((X)
       (PLUS X 1)))
   (SUB1 ((X)
       (PLUS X -1)))
   (NEQ ((X Y)
       (NOT (EQ X Y))))
   (ZEROP ((X)
       (EQ X 0)))
   (LESSP ((X Y)
       (GREATERP Y X)))
   (MINUSP ((X)
       (GREATERP 0 X)))
   (DIFFERENCE ((X Y)
       (PLUS X (MINUS Y))))
   (ABS (LAMBDA (X)
       (COND
         ((GREATERP 0 X)
           (MINUS X))
         (T X))))
   (ERSETQ ((X)
       (ERRORSET (QUOTE X)
         T)))
   (MAP (X (LIST (SUBPAIR (QUOTE (MAPF MAPF2))
           (LIST (CFNP (CADR X))
             (COND
               ((CDDR X)
                 (CFNP (CADDR X)))
               (T (QUOTE CDR))))
           (QUOTE (LAMBDA (MAPX)
               (PROG NIL
                 LP  (COND
                        ((NULL MAPX)
                          (RETURN)))
                      (MAPF MAPX)
                      (SETQ MAPX (MAPF2 MAPX))
                      (GO LP)
                 ))))
         (CAR X))))
 (MAPC (X (LIST (SUBPAIR (QUOTE (MAPCF MAPCF2))
           (LIST (CFNP (CADR X))
             (COND
               ((CDDR X)
                 (CFNP (CADDR X)))
               (T (QUOTE CDR))))
          (QUOTE (LAMBDA (MAPCX)
               (PROG NIL
                 LP  (COND
                       ((NULL MAPCX)
                         (RETURN)))
                     (MAPCF (CAR MAPCX))
                     (SETQ MAPCX (MAPCF2 MAPCX))
                     (GO LP)
                 ))))
        (CAR X))))
 (MEMB (LAMBDA (X Y)
     (PROG NIL
       LP  (RETURN (COND
                 ((ATOM Y)
                   NIL)
                 ((EQ X (CAR Y))
                   Y)
                 (T (SETQ Y (CDR Y))
                   (GO LP))))
       )))
 (NLSETQ ((X)
     (ERRORSET (QUOTE X)
       NIL)))
 (VAG (X (CEXPR (CAR X))
     (COND
       ((EQ (CAADR CODE)
            (QUOTE ENBOX))
          (RPLACA (CDR CODE)))
       (T (STORE (QUOTE (UNBOX)))))
     (QUOTE INSTRUCTIONS)))
  (LOC (X (CEXPR (CAR X))
      (COND
        ((EQ (CAADR CODE)
             (QUOTE UNBOX))
           (RPLACA (CDR CODE)))
        (T (BOX SP)))
      (QUOTE INSTRUCTIONS)))
  (ARG (X (CEXPR (LIST (QUOTE VAG)
           (CAR X)))
      (STORE (LIST (QUOTE VREF)
           (QUOTE SUB)
           (COND
             (ARGARG)
             (T (ERROR (QUOTE (FUNCTION 'ARG' NOT LEGAL))))))
)
      (STORE (LIST (QUOTE ARGN)
           ARGARG))
      (QUOTE INSTRUCTIONS)))
))(QUOTE MACRO))
```

LAP MACROS

The following expression when loaded with the compiler defines the substitution and lambda macros for Lap.

```
  (DEFLIST(QUOTE(
    (CSP1 ((LV LF LT)
        (LITREF LDA LV)
        (LITREF LDX LF)
        (LITREF LDB LT)
        (PRGREF VAL (PLUS ENTER PLITORG 1))))
    (VST1 ((PP LV V)
        (LITREF LDA PP)
        (LITREF LDB LV)
        (PRGREF VAL (PLUS IPV PLITORG V))))
    (BE ((N B)
        (STKREF SKE N)
        (RELREF BRU 2)
        (JUMP B)))
    (BNE ((N B)
        (STKREF SKE N)
        (JUMP B)))
    (LDV (LAMBDA (S)
        (VREF (QUOTE LDA)
          S)))
    (STV (LAMBDA (S)
        (VREF (QUOTE STA)
          S)))
    (LDT (LAMBDA (S)
        (STKREF (QUOTE LDA)
          S)))
    (STT (LAMBDA (S)
        (STKREF (QUOTE STA)
          S)))
    (NSTT (LAMBDA (S)
        (STKREF (QUOTE NSTA)
          S)))
    (LQT (LAMBDA (X)
        (LITREF (QUOTE LDA)
          X)))
    (LDN (LAMBDA (S)
        (NREF (QUOTE LDA)
          S)))
```

```
(STN (LAMBDA (N)
    (NREF (QUOTE STA)
      N)))
(CLL ((L K U)
    (LITREF LDA U)
    (LITREF LDB K)
    (LITREF CLLX L)))
(CLLA ((L K U)
    (LITREF LDA U)
    (LITREF LDB K)
    (VREF CLLXA L)))
(ARGN ((A)
    (LSH 1)
    (ARGSUB A)
    (ADD PPPTR)
    (CAXB 0)
    (LDA 0 1)
    (CBX 0)))
(ARGSUB (LAMBDA NIL
    (LITREF (QUOTE ADD)
      (PLUS -2 (VREF1 A)))))
(RET (NIL (VAL RETURN)))
(BN ((B)
    (SKE SYSNIL)
    (RELREF BRU 2)
    (JUMP B)))
(BNN ((B)
    (SKE SYSNIL)
    (JUMP B)))
(BA ((B)
    (SKG SYSTAT)
    (RELREF BRU 2)
    (JUMP B)))
(BNA ((B)
    (SKG SYSTAT)
    (JUMP B)))
    (UNBOX (NIL (VAL UNBOX)))
    (ENBOX ((N)
        (VAL (PLUS ENBOX N))))
    (NEG (NIL (CNA 0)))
    (DVD ((N X)
        (RSH 23)
        (DIV N X)))
    (DIVIDE ((S)
        (STTN S)
        (SWAP 0)
        (ENBOX S)
        (STKREF SXMA S)
        (ENBOX S)
        (STKREF XMA S)
        (CONSCLL S)))
    (BI ((B)
        (SKG SYSNUM)
        (RELREF BRU 2)
        (JUMP B)))
    (BNI ((B)
        (SKG SYSNUM)
        (JUMP B)))
    (BIS ((B L)
        (LITREF1 SKE L)
        (RELREF BRU 2)
        (JUMP B)))
    (BNS ((B L)
        (LITREF1 SKE L)
        (JUMP B)))
    (CONSCLL ((N)
        (CAB 0)
        (STKREF LDA N)
        (VAL (PLUS CONSCLL (TIMES N 2)))))
    (CCALL ((OP)
        (VAL OP)
        (LDX PPPTR)))
    (CLLX ((N)
        (VAL (PLUS XCLL N))))
    (CLLXA ((N X)
        (VAL (PLUS XCLL N X))))
    (SWAP (NIL (XAB 0)))
    (JUMP ((B)
        (PRGREF BRU (GBS B))))
    (MPY ((N X)
        (MUL N X)
        (LSH 23)))
))(QUOTE OPD))
```

OPCODES currently defined for LAP

The following expression loaded with the compiler defines the Opcodes for Lap.

```
(DEFLIST(QUOTE(          (CAXB 4600440Q)
  (LDA 7600000Q)         (CBX 4600020Q)
  (STA 3500000Q)         (CNA 4601000Q)
  (NSTA 3500000Q)        (BRU 100000Q)
  (LDB 7500000Q)         (BRX 4100000Q)
  (STB 3600000Q)         (BRM 4300000Q)
  (LDX 7100000Q)         (BRR 5100000Q)
  (STX 3700000Q)         (SKE 5000000Q)
  (EAX 7700000Q)         (SKG 7300000Q)
  (XMA 6200000Q)         (SKM 7000000Q)
  (SXMA 6200000Q)        (SKA 7200000Q)
  (ADD 5500000Q)         (SKB 5200000Q)
  (ADM 6300000Q)         (SKN 5300000Q)
  (MIN 6100000Q)         (SKR 6000000Q)
  (SUB 5400000Q)         (RSH 6600000Q)
  (MUL 6400000Q)         (LRSH 6624000Q)
  (DIV 6500000Q)         (RCY 6620000Q)
  (ETR 1400000Q)         (LSH 6700000Q)
  (MRG 1600000Q)         (LCY 6720000Q)
  (EOR 1700000Q)         (NOP 2000000Q)
  (CLA 4600001Q)         (EXU 2300000Q)
  (CLB 4600002Q)         (VAL 0Q)
  (CAB 4600004Q)         (BIO 5760000Q)
  (CBA 4600010Q)         (BRS 5730000Q)
  (XAB 4600014Q)         (CTRL 5720000Q)
                       ))(QUOTE OPD))
```

INDEX TO FUNCTIONS

name of function	description page
ersetq	95
esgag	95
eval	37
evala	37
evalr	38
evalv	70
faulteval	99
feed	79
fgvp	75
fix	75
float	75
floatp	75
fltfmt	75
fminus	75
fntyp	36
fplus	75
fquotient	76
ftimes	76
function	64
gcgag	60
gensym	60
get	33
getd	35
getp	33
go	22
greaterp	73
ilp	61
infile	77
input	78
interrupt	99

name of function	description page
prin3	80
prog	22
progn	21
prog1	21
prog2	21
prop	32
put	32
putd	35
putdq	35
quit	96
quote	18
quotient	72
radix	85
ratest	84
ratom	82
ratoms	82
rbin	86
rcompile	107
rdflx	82
read	82
readc	84
readp	86
reclaim	60
remainder	73
remove	29
remprop	32
rename	70
reset	96
retfrom	71
return	23

Following is a list of all atoms with initialized top level values in the basic system and those values.

blank	space
space	space
tab	tab
comma	,
eqsign	=
xeqs	=
f	nil
nil	nil
period	.
pluss	+
lpar	(
rpar	)
slash	/
t	t
t	t
qmark	?
xdol	$
xsqu	'
xdqu	"
xlbr	[
xrbr	]
xarr	←
uparr	↑
colon	:
xgreater	>
xlesser	<
xnum	#
xper	%
xamp	&
xat	@

xsem	;
xexclaim	!
xcr	carriage return
bkslash	\

Unclassified
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, Massachusetts 02138	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE

THE BBN 940 LISP SYSTEM

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Interim Scientific Report

5. AUTHOR(S) *(First name, middle initial, last name)*
Daniel G. Bobrow, D. Lucille Darley, L. Peter Deutsch, Daniel L. Murphy, Warren Teitelman

6. REPORT DATE July 1967	7a. TOTAL NO. OF PAGES 131	7b. NO. OF REFS 0

8a. CONTRACT OR GRANT NO. AF 19(628)-5065 - ARPA Order No.627	9a. ORIGINATOR'S REPORT NUMBER(S) BBN Report No. 1539 Scientific Report No. 9
b. PROJECT NO. 8668 Amendment No. 2	
c. DoD Element 6154501R	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* AFCRL-67-0458
d. DoD Subelement n/a	

10. DISTRIBUTION STATEMENT
Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

11. SUPPLEMENTARY NOTES This research was sponsored by the Advanced Projects Agency	12. SPONSORING MILITARY ACTIVITY Air Force Cambridge Research Laboratories (CRB) L. G. Hanscom Field Bedford, Massachusetts 01730

13. ABSTRACT

This report describes the LISP system implemented at BBN on the SDS 940 Computer. This LISP is an upward compatible extension of LISP 1.5 for the IBM 7090, with a number of new features which make it work well as an on-line language. These new features include tracing, and conditional breakpoints in functions for debugging and a sophisticated LISP oriented editor. The BBN 940 LISP SYSTEM has a large memory store (approximately 50,000 free words) utilizing special paging techniques for a drum to provide reasonable computation times. The system includes both an interpreter, a fully compatible compiler, and an assembly language facility for inserting machine code subroutines.

DD FORM 1 NOV 65 1473 (PAGE 1)
S/N 0101-807-6811

Unclassified
Security Classification

A-31408

Unclassified
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
LISP List Processing Language Paging Systems Drum Systems for List Structure List Structures Symbol Manipulation Language						

DD FORM 1 NOV 65 1473 (BACK)
S/N 0101-807-6821

Unclassified
Security Classification

A-31409