BBN report #1346

AFCRL-66-180

# THE BBN-LISP SYSTEM

Daniel G. Bobrow
D. Lucille Darley
Daniel L. Murphy
Cynthia Solomon
Warren Teitelman

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

Contract No. AF19(628)-5065
Project No. 8668
Scientific Report No. 1

February, 1966

(The work reported was supported by the Advanced Research Projects Agency, P.R. No. CRL-56176, ARPA Order No. 627, dated 9 March 1965.)

Prepared for:

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

Distribution of this document is unlimited.

AFCRL-66-180

# THE BBN-LISP SYSTEM

Daniel G. Bobrow
D. Lucille Darley
Daniel L. Murphy
Cynthia Solomon
Warren Teitelman

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

Contract No. AF19(628)-5065
Project No. 8668
Scientific Report No. 1

February, 1966

(The work reported was supported by the Advanced Research Projects Agency, P.R. No. CRI-56176, ARPA Order No. 627, dated 9 March 1965.)

Prepared for:

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

TABLE OF CONTENTS

## FOREWORD

The work reported here was performed at Bolt Beranek and Newman Inc in Cambridge, Massachusetts for the Advanced Research Projects Agency under Contract No. AF 19(628)-5065.

THE BBN-LISP SYSTEM

## ABSTRACT

This report describes in detail the BBN-LISP system. This LISP system has a number of unique features; most notably, it has a small core memory, and utilizes a drum for storage of list structure. The paging techniques described here allow utilization of this large, but slow, drum memory with a surprisingly small time penalty. These techniques are applicable to the design of efficient list processing systems embedded in time-sharing systems using paging for memory allocation.

# SECTION I.

## INTRODUCTION

LISP is a highly sophisticated list-processing language which is being used extensively in the artificial intelligence research program at Bolt Beranek and Newman. This report describes our LISP system, which has a number of unique features. Ideally, a LISP system would have a very large, 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 approximately 1000 times slower than access to a word in a core memory, using a drum as a simple extension of core memory would reduce the operating speed of a system by a factor of 1000.

We have developed a special paging technique which allows utilization of a drum for storage with a much smaller time penalty. This technique allows us to make effective use of a LISP system on our PDP-1 which has only 8392 18-bit words of 5 microsecond core memory and 92,312 words on a drum with an average access time of 16.5 milliseconds. In addition, the techniques reported here would improve the speed of operation of LISP systems embedded in time-sharing systems using paging for memory allocation. In these time-sharing systems the user is allocated only a small portion of core memory at any time, although his program can address a large virtual memory. The portion of his data structure and/or program not in core is kept in a slower secondary storage medium such as a drum or disc. Thus, to the user it is very similar to the situation on our PDP-1, except that a hardware mechanism makes the program transparent to the medium of storage of any page of his program.

Section II of this report describes the internal structure of the BBN-LISP system, and the mechanisms used to facilitate fast use of drum storage. Section III describes the LISP functions which are built into the basic system. Section IV contains listings of those functions which are defined in LISP.

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.

## SECTION II.

## THE INTERNAL STRUCTURE OF THE BBN-LISP SYSTEM

The BBN-LISP System uses only a small core memory, but achieves a large memory capacity by utilizing a drum. This drum is used in three ways. First, the working program is divided into three overlays, the read and print (input-output) program, the garbage collector, and the interpreter of S-expressions. Only one of these overlays is in core at any time, although a number of sub-programs common to all three remain in core at all times.

Secondly, the drum contains a large push-down list for use in running recursive programs. This push-down list is double-buffered; that is, the section of the push-down list used most recently is in core and the section used immediately preceding this section is also there, so that traveling between buffers does not necessitate a drum reference.

The third way of utilizing this large secondary store, the drum, is for storage of list structure. If the entire remaining drum storage was used simply as an extension of core memory, an access to the drum would be needed each time a new list element was referenced; and LISP would be reduced to operating at drum rotation speed. Instead, the drum storage of list structure is divided into pages. Each page is currently 258 words (decimal); and each page contains its own free storage list. The <u>cons</u> algorithm, for constructing a new list element, works as follows.

To construct z = cons [x;y]:

1) If *y* is not an atom, and there is room on the page with *y*, then place *z* on this page

2) Otherwise, if *x* is not an atom, and there is room on the page with *x*, put *z* on that page

3) Otherwise, place *z* on the page in core with maximum free storage.

This algorithm tends to minimize cross linkages between pages and to limit any single structure to a very few pages. Thus when working with this structure, it is unlikely that one will make references to more than a few pages for a relatively long period of time. Since these pages can reside in core, no drum references are needed. For example, in entering the definition of a function, the entire definition tends to appear on a single page. Thus, during the interpretation of a function, multiple drum references are usually unnecessary.

Although we have not yet run this LISP system on a large problem where we can make a reasonable timing comparison, we can give the following anecdotal evidence for the increase in speed due to this paging system. The run light on the PDP-1 goes off when a drum swap is taking place. In an older version of PDP-1 LISP the drum was treated as an extension of core memory. When any problem was run, the run light seemed to go off completely, indicating that the machine was spending almost all of its time doing drum transfers. In this system, however, the run light seems to burn as brightly as the rest, indicating that relatively few drum transfer operations occur except when going between the three overlay packages, that is, when going from input and output back to the interpreter or going into a garbage collection.

On the research computer, because of the drum storage, we currently have in use an effective free storage list of approximately 25,000 LISP words, i.e., double word pairs (pointers). Each LISP word is, of course, two 18-bit PDP-1 words. In the extended version of LISP that will be used on the hospital system we will have 256,000 LISP words for free storage.

There are a number of differences between this system and 7094 LISP aside from the storage conventions. For example, the value of a variable is stored in a special value cell for that variable, that is, as car of the atom name. An atom is distinguished by its address, which is located in a fixed region of virtual memory space. Thus one need not reference a structure, but only look at its address, in order to tell whether or not it is an atom. If x is an atom, then cdr[x] is the property list of the atom, as in 7094 LISP. However, the print name of the atom is not to be found on this property list. The user can only get at the print name with the instructions pack and unpack. Similarly, the definition of an atom as a function is hidden away from the user in a special cell associated with the atom name. Two functions, getd[x] and putd[x;def] are used to get the definition of a function, and place the definition in the function cell of an atom, respectively. The value of getd[x] on an atom defined as a machine language subroutine is a numerical constant which bears some relationship to the instruction that must be executed to obtain access to the subroutine.

When a new function is entered, the old values of its variables are pushed down on the push-down list, and the current values are stored in the value cells. Since the current value of any variable is always to be found in its value cell, free variables are no problem. However, there is the usual anomalous case of conflicting free variables in functional arguments. This can be circumvented by using sufficiently unique variable names.

Because of the way variable values are stored, the main interpreter, <u>eval</u>, obviously does not use an A-list, and is therefore a function of only one argument. The function <u>evala</u> defined in the BBN-LISP System will simulate the effect of the usual eval[x;a], <u>a</u> being an A-list.

Different LISP systems employ different methods to achieve the following two effects in functions labelled FEXPR's in 7094 LISP. These two effects are (1) giving a function the ability to have an indefinite number of arguments, and (2) giving a function the ability to receive its arguments unevaluated.

On the 7094 an FEXPR is defined by putting the function definition on the property list after the flag, FEXPR, and treating it as a special case in the interpreter. In BBN-LISP we call functions which have abilities (1) and (2) FEXPR's, but we define them differently. The way an FEXPR is defined in BBN-LISP is as follows: instead of the usual lambda followed by a list of variables, the defining form is preceded by nlamda followed by a list containing a single variable. When a function with an nlamda is entered, everything following the function name in the form to be evaluated is placed on a single list and becomes the value of the single argument of this FEXPR. This is passed to the function unevaluated. In order to evaluate any portion of this list, an explicit call to eval must be made. See "defineq" in the listings for an example of the use of this device. A third reason FEXPR's and FSUBR's are used on 7094 LISP is to make the A-list available to a program. However, since there is no A-list in BBN-LISP this will not concern us here.

Another major difference between BBN-LISP and 7094 LISP is due to the fact that the 7094 has floating point hardware, and the PDP-1 does not. Any floating point machinery would have to be interpreted on the research computer. This would be expensive in both time and space, and, therefore, in this version of LISP there is only integer arithmetic. A compiler is being planned for the PDP-1 and will be described in a later document.

# SECTION III.

## DESCRIPTION OF FUNCTIONS IN BBN-LISP

cons[x;y]
SUBR

cons constructs a dotted pair of x and y. If y is a list, x becomes the first element of that list.

car[x]
SUBR

car gives the first element of a list x, or the left element of a dotted pair x. Nominally undefined for atoms, it gives the binding (value) of an atom x.

cdr[x]
SUBR

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 an atom, cdr[x] gives the property list of x.

caar[x] = car[car[x]]
SUBR

cadr[x] = car[cdr[x]]
SUBR

cddddr[x] = cdr[cdr[cdr[cdr[x]]]]
SUBR

All 30 combinations of nested cars and cdrs up to 4 deep are included in the system.

eq[x;y]
SUBR

The value of eq is T if x and y are identical atoms, including numbers; otherwise the value is NIL. (Will give T for lists if their internal representations are identical, NIL otherwise.)

| | |
|---|---|
| null[x] SUBR | eq[x;NIL] |
| atom[x] SUBR | Its value is T if x is an atom; NIL otherwise. |
| oblist[] SUBR | Gives a list of all atoms in the system. |
| not[x] EXPR | Its value is true if its argument is false, and false otherwise. |
| quote[x] FSUBR | This is a function that prevents its argument from being evaluated. Its value is x itself. |
| cond[x] FSUBR | 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 expression, then the value of |

the conditional is the value of this expression. This value coincides with the value in 7090 LISP 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.

prog[l]
FSUBR

This feature allows the user to write an ALGOL-like program containing LISP statements to be executed. The argument is a list, the first element of which is a list of program variables. The rest of the list is a sequence of statements, and atomic symbols used as labels for transfer points.

go[x]
FSUBR

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

list[x1;...;xn]
FSUBR

The value of list is a list of the values of its arguments.

| | |
|---|---|
| return[x] SUBR | <u>return</u> is the normal end of a <u>prog</u>. Its argument is evaluated and is the value of the <u>prog</u> in which it appears. |
| print[x] SUBR | Prints <u>x</u>, followed by carriage return, on specified devices (see <u>punchon</u>, <u>typeout</u>). Value is <u>x</u>. |
| prin1[x] SUBR | Prints one atom, <u>x</u>, with no space or carriage return following. Value is <u>x</u>. |
| terpri[] SUBR | Prints a carriage return. Value is NIL. |
| punchon[x] SUBR | Turns punch on for <u>print</u> if x = T; turns punch off if x = NIL. Value is former setting of <u>punchon</u>. |
| typeout[x] SUBR | If x = T, turns typewriter on for printing. If x = NIL, turns typewriter off. Value is former setting of <u>typeout</u>. |
| read[] SUBR | Reads on S-expression from specified device (see <u>typein</u>). |
| punch[x] EXPR | This function sets <u>punchon</u> to t, sets <u>typeout</u> to nil, punches <u>x</u>, and then restores <u>punchon</u> and <u>typeout</u> to their original values. |

**typein[x]**
SUBR

If x = T read-in device is set to typewriter. If x = NIL read-in device is set to reader. Value is former setting of typein.

**ratom[]**
SUBR

Reads in one atom from read-in device. Separation of atoms is as defined by the functions setsepr and setbrk.

**setsepr[x]**
FSUBR

**setbrk[x]**
FSUBR

These are both FSUBRS and may have up to 18 arguments each. Arguments should be octal numbers, e.g., 77q for carriage return. 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 (33q), and carriage return (77q) are separation characters, and left paren (57q), right paren (55q), and period (73q) are break characters. Thus

```
Thus  setsepr[0q  33q  77q]
      setbrk[57q  55q  73q]
```

would set up these characteristics. The value of setsepr and of setbrk is NIL.

clearbuf[]
SUBR

This SUBR clears the input and output buffers of the sequence break package, including the sequence break reader, ratom, read, and typein line buffers, and sets the case to lower case. This means that if you have just done a read and the S-expression did not complete a line, whatever else is on that line will be lost. However, it is very useful if you want to initialize the system, or an error has been made, and you want to clear out what has been read in on a line.

readin[x]
SUBR

If x = T, readin sets the teletype input to the paper tape reader. Specifically, it eliminates the line-feed echo after a carriage return, and the delete characters, rubout and colon, are not recognized. Setting x to NIL restores the status to normal. This function returns its previous value.

feed[n]
SUBR

The value of n must be a number. This function outputs on the teletype n carriage return-line feeds or n carriage returns depending on the setting of readin.

character[n]
SUBR

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

prog1[x;y]
SUBR

This function evaluates both its arguments in order, that is, x first and then y, and then returns the value of x.

prog2[x;y]
SUBR

The purpose of this function is to allow the evaluation of x, before returning y.

progn[x;y;...;z]
FSUBR

progn is an FSUBR which evaluates each of its arguments in sequence, and returns the value of its last argument as its value. It is an extension of prog2.

set[x;y]
SUBR

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]
FSUBR

This FSUBR is identical to set, except that the first argument is quoted.

Example: If the value of 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. The value of y is unaffected.

setn[x;y]
SUBR

<u>setn</u> requires and checks for an atom as the value of the first argument, and a number as the second. If the first argument is <u>not</u> already defined as a number, the value of the second will be moved to a new cell in FWS (Full Word Space), the location of which will be stored in the value cell of the first argument. Otherwise, <u>setn</u> replaces the FWS cell containing the previous numeric value of the first argument by the numeric value of the second. If the second argument was the most recent number added to FWS, the cell containing its value is returned to the free list.

<u>Example</u>:

(SETN (QUOTE P) (PLUS P 1))

creates a new cell in FWS containing the old value of P plus 1. This value gets moved to the FWS cell containing the old value.

The following will lose:

(PROG .. (SET (QUOTE A) B)

(SETN (QUOTE A) (PLUS A 1)) ...)

because the cell containing the value of A is the same as that for B. To avoid the problem, the first SET should have been a SETN so that a unique numeric value cell would have been assigned for A.

| | |
|---|---|
| setqq[x;] | Identical to *setq* except that neither argument is evaluated. |
| setnq[x;y]<br>FEXPR | This FEXPR is identical to *setn* except that the first argument is quoted. |
| putd[x;y]<br>SUBR | *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* *d*efinition on *x*. Value of *putd* is the definition (value of *y*). |
| putdq[x;y]<br>FEXPR | This function is similar to *putd*, but both arguments are considered quoted, and its value is *x*. |
| getd[x]<br>SUBR | This function *gets* the *d*efinition 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 SUBR or FSUBR, the value is a number. |
| fntyp[x]<br>SUBR | This function gives EXPR, FEXPR, SUBR, FSUBR or NIL depending on whether *x* is an EXPR, FEXPR, SUBR, FSUBR or not defined, respectively. |
| eval[x]<br>SUBR | *eval* evaluates the expression *x* and returns this value. |

errorset[form;arg]
SUBR

This function calls eval with the value of form, and returns with a list of this value if no error is encountered. If an error is encountered on the call to eval, errorset returns with the value NIL. If arg is T, the message from error is printed; the message is not printed if arg = NIL.

ersetq[x]
FEXPR

This FEXPR is defined as (ERRORSET (CAR X) T); that is, it is the same as errorset with the argument quoted and the error flag set to T.

nlsetq[x]
FEXPR

This FEXPR is identical to ersetq except that the error flag is set to NIL and the error comment from error will not be printed out.

error[x]
SUBR

error induces an error with message x.

quit[]
SUBR

quit induces a "strong" error, i.e., will unwind a program through errorsets to the top level.

equal[x;y]
SUBR

The value of this function is T if x and y are equal, that is, identical S-expressions, and NIL otherwise. It is as fast as eq for atoms.

and[x]
FSUBR

This function is an FSUBR and can take an indefinite number of arguments. Its value is T if none of its arguments has value NIL, and is NIL otherwise.

or[x]
FSUBR

*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 value T.

rdflx[x]
EXPR

If *x* is NIL this function will try to read one S-expression from the typewriter 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).

append[x;y]
EXPR

This function copies list *x* and appends list *y* to this copy. The value is the combined list.

nconc[x;y]
SUBR

This function is similar to append, in effect, but it actually causes this effect by 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.

nnconc[x;y]
SUBR

This function is the same as nconc. nnconc is used by the trace programs so that nconc itself can be traced.

attach[x;y]
EXPR

This function attaches x to the front of the list y by doing an rplaca and an rplacd.

tconc[x;p]
EXPR

This function provides an efficient way for placing an item x at the end of a list p. 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 element of the modified list.

lconc[x;p]
EXPR

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.

last[x]
EXPR

This function has as its value a pointer to the last cell in the list *x*, and returns NIL if *x* is an atom.

length[x]
EXPR

This function has as a value the length of the list *x*. If *x* is an atom, it returns 0.

prettyprint[x]
EXPR

The argument of *prettyprint* is a list of names of functions; it prints and/or punches (depending on the settings) the definitions of the named functions in a pretty format. It utilizes the functions *printdef*, *endline*, and *superprint*. This latter function does all the work.

prettydef[x]
EXPR

This function of one argument (a list of function names) prints and/or punches "(DEFINEQ", followed by the *prettyprint* listing of each of these functions, followed by a right paren. A tape punched by <u>prettydef</u> can be loaded by the function <u>load</u> if a STOP is punched on the end of the tape. The value of <u>prettydef</u> is <u>x</u>.

define[x]
EXPR

The argument of <u>define</u> is a list. Each element of the list is itself a list containing either two or three 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 nlamda expression. In a three-item list the first item is again the name of the function to be defined. The second is the lambda list of variables and the third is the form for the expression. 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))
```

defineq[x;...;z]
FEXPR

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 as an element of the list which is the argument for 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.

load[x]
EXPR

load is a function which reads successive S-expressions from the paper tape reader, and evaluates each as it is read. If x = 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 the paper tape reader; it saves having to write sequences of E(EVAL (READ)).

unpack[x]
SUBR

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.

pack[x]
SUBR

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.

remob[x]
SUBR

The argument of _remob_ must be an atom. The effect of applying _remob_ to this atom is to remove all trace of this atom from the system. This is a good way of reclaiming space from atoms which are no longer needed but it is very dangerous, and _remob_ should be used with care. A future mention of the same atom name will have no connection with the old one that was formerly there. In addition, any lists which point to this old atom will now be incorrect.

member[x;y]
SUBR

This SUBR checks to see if _x_ is a member of the list _y_. If so, it returns the value T; if not, it returns the value NIL.

rplacd[x;y]
SUBR

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.

rplaca[x;y]
SUBR

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.

gensym[]
SUBR

This function of no argument generates a unique symbol of the form Annnn, in which each of the n's is replaced by a digit. Thus the first one generated is A 0001, etc. This is a way of generating new atoms for various uses within the system.

disp[x;y]
SUBR

This function displays one point on the cathode ray tube at the point whose coordinates are (x;y) and returns T if the light pen saw the displayed point, and NIL otherwise.

displis[l]
SUBR

The argument of this function is a list of successive x and y coordinates to be displayed.
For example:
displis[(1 2 1 3 1 4)]
will successively display the points at coordinates (1,2), (1,3) and (1,4).
This is faster than displaying each of these three points individually by using disp.

logand[x;...;z]
FSUBR

This FSUBR will take the logical AND of all of its argument as octal numbers and return this value.

logor[x;...,z]
FSUBR

This function, an FSUBR, will take the logical OR, bit-wise, of all of its arguments, and return this number

e[x]
FEXPR

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.

get[x;y]
EXPR

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.

trace[x]
EXPR

This function has as an argument a list of names of functions. It changes the definition of these functions so that when each function is entered, the values of the arguments of this function are printed; when the value of this function is computed this value is printed. Thus, trace[(plus ratom)] would cause plus and ratom to be redefined so that this tracing takes place. The value of trace is the value of its argument x. The work of trace is done by the function trac1.

tracp[x;y]
EXPR

This function tells whether the function named x with definition y has been traced. Its value is T if the function is being traced, and NIL otherwise.

untrace[x]
EXPR

This function undoes what trace does, and restores the original definition of the function.

A word of warning: do not trace the following functions or you will get in an infinite loop because these functions are used within traci itself: print; cons; set; fntyp; eval; return; evalprint; car; cdr; null; go.

mapc[x;fn]
EXPR

This function applies the function fn to each of the elements of the list x. It returns the value NIL.

mapcar[x;fn]
EXPR

This function applies the function fn 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 fn to the corresponding element of the old list.

mapconc[x;fn]

Identical to mapcar except that it does an nconc instead of a cons.

mapcon[x;fn]

Identical to maplist except that it does an nconc instead of a cons.

map[x;fn]
EXPR

This function applies the function fn to successive tails of the list x. That is, first it computes fn[x], and then fn[cdr[x]], etc. until x is NIL. This function returns NIL.

maplist[x;fn]
EXPR

This function computes successively the same values that map computes; it forms a new list consisting of successive values of applications of this function.

assoc[x;a]
EXPR

If a is a list of dotted pairs, then assoc will produce the first pair whose first item is x. If such an item is not found, assoc will return NIL.

sassoc[x;y;u]
EXPR

The function sassoc searches y, which is a list of dotted pairs, for a pair whose first element is 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.

copy[x]
EXPR

This function makes a copy of the list x. The value of copy is the location of the copie list.

**intersection[x;y]**
EXPR

This function returns with a list whose elements were members of both lists x and y.

**union[x;y]**
EXPR

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.

**prop[x;y;u]**
EXPR

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~~.

**reverse[l]**
EXPR

This is a function to ~~reverse the~~ top level of a list~~.~~ ~~Thus, using~~ reverse on

(A B (C D)) = ((C D) B A)

**subst[x;y;z]**
EXPR

This function gives the result of substituting the S-expression x for all occurrences of the atomic symbol y in the S-expression z.

| | |
|---|---|
| sublis[x;y]<br>EXPR | Here _x_ is a list of pairs:<br>$((u_1 . v_1)\ (u_2 . v_2)\ \ldots\ (u_n . v_n))$<br><br>The value of sublis[x;y] is the results of substituting each _v_ for the corresponding _u_ in _y_. |
| evala[x;a]<br>SUBR | This is the regular _eval_ in the 7094 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 in _x_ that also appear on _a_ will be given the value indicated on _a_. |
| apply[fn;args;a]<br>SUBR | _apply_ applies the function _fn_ to the arguments _args_ with the values obtained from _a_, i.e. the arguments of _fn_ on _args_ are _not_ evaluated but given to _fn_ directly. _a_ is used to evaluate free variables in _fn_ as described above. |
| remove[x;l]<br>EXPR | The function _remove_ removes all occurrences of _x_ from list _l_. |
| remprop[x;y]<br>EXPR | This function _removes_ all occurrences of the _property_ with label _y_ from the property list of _x_. |
| put[x;y;z]<br>EXPR | 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. |

add[x;y;z]
EXPR

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 p, the effect is the same as put[x;y;list[z]].

getp[x;y]
EXPR

This function gets the property with label y from the property list of x.

NOTE: Both prop and get may also 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;ind]
EXPR

This function is used to put any indicator on a property list. The first argument is a list of pairs (where the first of the pair is a name and the second party of the pair is the property to be stored with the indicator on the property list of the name) and the second argument is the indicator that is to be used.

select[x;$y_1$;$y_2$ ...;$y_n$;z]
FSUBR

An example of arguments for this function is:
[q; ($q_1$ $e_1$); ($q_2$ $e_2$); ...($q_n$ $e_n$); e]

The $q_i$'s are evaluated in sequence until one is found such that $q_i$ = q, and the value of select is the value of the corresponding $e_i$. If no such $q_i$ is found the value of select is that of e.

selectq[x;y;z]
FSUBR

selectq is identical to select except that the $q_i$'s are not evaluated--only q.

time[x n]
EXPR

This function performs computation x n times and indicates average time in tenths of seconds.

gcgag[x]
SUBR

If x=T garbage collector will print message when entered. If x=NIL, no message is printed.

reclaim[]
SUBR

This function initiates a garbage collection and returns with the number of available LISP words in free storage.

field[n]
SUBR

This function calls field n from the drum. (See description of system program linking.)

nth[x;n]
EXPR

This EXPR has as inputs 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.

editf[x]
EXPR

This EXPR gets the expression which is the definition of the function named x and gives it to edite.

editv[x]
EXPR

This EXPR gets the value of the atom x and gives it to edite for editing.

editp[x]
EXPR

This EXPR gets the property list of the atom x, etc.

edite[x]
EXPR

This function is the executive for an editing facility for LISP expressions. The argument of edite must be a list to be edited. When edite has been called, it prints out EDIT, and then waits for input from the on-line teletype (or the reader if typein is set to NIL).

The input that may be typed in may be a positive integer, a negative integer, or zero, or one of these as the first element of a two-element list, or NIL, or one of several special lists described below. Typing in NIL terminates editing.

This editing program allows you to edit any subexpression within the current level expression, that is, you can replace or delete any subexpression of this expression, or insert anything before any subexpression of this expression. An input (n exp) where n is a positive integer will replace the nth expression in the current level expression by exp; if n is a negative integer it will put exp just before the nth subexpression in the current level expression. (n) where n is a positive integer (with no expression following this integer) will delete the nth expression.

Warning: Typing "(1)", where current expression is a singleton, will not have desired effect.

An input of 0 will take you up to the next higher level expression. If the input to edit is a positive integer, the nth-subexpression of the current expression will become the expression that can be edited.

An important thing to note is that all editing is final in the sense that any changes that are requested are put in with rplacas and rplacds. It is the original expression which has been modified to give the edited version; to return to the original expression you must re-edit. However, by using the COPY and RESTORE feature, the user can protect himself against errors in editing. The function edite calls edit1f, edit2f, edit2af, and edit3f to do all the work.

Other special commands are:

| | |
|---|---|
| COPY | copies and saves entire expression being edited as it currently exists. |
| RESTORE | Restores expression as of last copy: the current level expression will be the current level expression at last copy. RESTORING without copying will have no effect. |
| p | Same as (p o). |
| (p n) | Prints the nth subexpression of the current expression to a level of 2, using LEVELN described below. If n is zero, prints current expression to level 2. |
| (p n m) | Prints nth subexpression to a level m. |
| | All printing may be interupted. |

(N $e_1$ $e_2$ ...)

which will tack the expressions $e_1$ $e_2$, ... to the end of the current expression.

(E exp) will print the value of eval [exp]. (I n exp) will compute v = eval[exp] and then act as if edit were given (n v). This allows you to insert the value of a computation in the current expression, at subexpression n. (n must be a number).

(LI n) will insert a left parenthesis immediately before subexpression n in the current expression and a matching right paren at the end of this current expression. For example, if
e = (A B C)
(LI 2) yields (A (B C)).

(LO n) will remove a left paren from the nth subexpression, and take a corresponding right paren from the end of the current expression, e.g.,
for e = (A (B C) D)
(LO 2) yields (A B C)

(RO n) will remove a right paren from the nth subexpression of the current expression, and insert one in at the end of the current top level expression, e.g.,
for e = (A (B C) DE)
(RO 2) yields (A (B C DE))

(RI m n) will insert a right paren in the nth subexpression of the mth subexpression of the current expression, removing one from the end of the mth subexpression, e.g.,
for e = (A B (C D E) F)
(RI 3 1) yields
(A B (C) D E F)

leveln[x n]

Abbreviates list x at level n, using the symbol ampersand, "&," to indicate greater depth. For example, leveln [(A (B C) (D (E F) G)) 2] is (A (B C) (D & G)).

The following 9 functions form a Break Package which allows the user to make a break conditional upon the result of some computation and thus arrest the operation of a function. He may interrogate the broken function as to the current values of its arguments or other variables, or perform arbitrary LISP computations; then he may return with a specified value for it without actually entering it. Alternatively, the user may just "crack" a function, i.e., print out the result of some computation before executing the function and print out the final value of this function.

break[fn;when;what]
EXPR

break is a function of three arguments: the function to be broken, under what condition to break, and what to print out when a break occurs If when = T, the function always breaks. If when = (NIL) a crack is made in fn. If what = NIL, no information is printed out. break redefines fn using break1 so that at the time the function would have been entered, break1 is entered instead with the definition of the function and information regarding the conditions for breaking.

unbreak[fn]
EXPR

unbreak redefines fn as it was before the break and returns the value fn. If fn is not broken when unbreak is called, the value of unbreak is (FN NOT BROKEN).

breaklist[l]
FEXPR

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

unbreaklist[l]
FEXPR

This function performs (UNBREAK FN) for each function on the list l.

breakat[fn;where;when;what]
EXPR

This function is similar to break except that instead of inserting a break at the beginning of fn, it allows the user to insert a break at any top-level place in fn. The argument where indicates the label or statement at which the break is to occur. The other arguments are used as in break.

unbreakat[fn;where]
EXPR

This function removes a break inserted by breakat.

breakprog[fn;l]
EXPR

breakprog is entered with the name of a function and a list of places in that function where a break is desired. breakprog performs (BREAKAT FN WHERE T NIL) for each place on the list l.

unbreakprog[fn]
EXPR

This function performs (UNBREAKAT FN WHERE) for each place where a break exists in fn.

break1[form;when;fn;what]
FEXPR

Although this function is not entered directly by the user, it is the heart of all the break functions and is entered when a break occurs. After the specified information is printed out, break1 listens to the typewriter or teletype for inputs. If STOP is input, a normal exit is achieved. If RETURN FOO is input break1 returns (EVAL FOO). If QUIT is input, it performs (ERROR FN). If EVAL is input, it evaluates fn. If a normal exit is subsequently achieved via the STOP command, break1 does not reevaluate fn, but uses the value obtained by the EVAL command. The EVAL feature is useful for evaluating a broken function without "letting go" of the break, e.g., to examine the effect of executing a broken function. If OK is input, a normal return is made without printing the value of the function. Any other input to break1 is evaluated, and its value printed. This function uses bp1 to do part of its work.

Arithmetic Functions (all arguments must be numbers)

| | |
|---|---|
| greaterp[x;y]<br>SUBR | T if x > y;<br>NIL otherwise |
| lessp[x;y]<br>EXPR | T if x < y;<br>NIL otherwise |
| zerop[x]<br>EXPR | T if x is zero;<br>NIL otherwise |
| minusp[x]<br>EXPR | T if x is negative;<br>NIL otherwise |
| numberp[x]<br>SUBR | T if x is a number;<br>NIL otherwise |
| add1[x]<br>EXPR | x + 1 |
| sub1[x]<br>EXPR | x - 1 |
| plus[x;y]<br>FSUBR | x + y (This FSUBR may have any number of arguments.) |
| minus[x]<br>SUBR | - x |
| times[x;y]<br>FSUBR | product of x and y (This FSUBR may have any number of arguments.) |

| | |
|---|---|
| quotient[x;y] SUBR | greatest integer in quotient x/y |
| difference[x;y] EXPR | This function has for its value the algebraic difference between its arguments. |
| remainder[x;y] EXPR | This function computes the number theoretic remainder for fixed-point numbers. |
| divide[x;y] SUBR | This function yields a dotted pair whose first member is quotient[x;y] and whose second member is remainder [x;y]. Remainder is defined in terms of divide. |

Following is a list of all atoms with APVAL's (permanent values) in the basic system, and their values.

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

## SECTION IV.

## LISTINGS OF S-EXPRESSIONS OF EXPR'S AND FEXPR'S

```
(prog nil
   (cond
      ((null (fntyp (quote putdq))) (putd (print (quote putdq))
(quote (nlamda (x) (prog2
         (putd (car x) (cadr x))
         (car x))))))))
(return (putdq load (lambda (x) (prog (xx yy zz)
   (clearbuf)
   (setq zz (typein nil))
l1 (cond
      ((equal (setq xx (read)) (quote stop)) (return (prog2
         (clearbuf)
         (typein zz)))))
   (setq xx (eval xx))
   (cond
      (x (print xx)))
   (go l1))))))
```

```
(putdq define
   (lambda (x) (cond
      ((null x) nil)
      (t (cons ((lambda (y) (prog2
            (putd (car y) (cond
               ((null (cddr y)) (cadr y))
               (t (cons (quote lambda) (cdr y)))))
            (car y)))
         (car x)) (define (cdr x)))))))

(putdq defineq
   (nlamda (x) (define x)))




(add
   (lambda (x y z) (prog nil
      loop  (cond
               ((null (cdr x)) (rplacd x (list
                  y
                  (list
                     z))))
               ((equal (cadr x) y) (rplaca (cddr x) (append
(caddr x) (list
                  z))))
               ((setq x (cddr x)) (go loop)))
            (return y))))

(add1
   (lambda (x) (plus
      x
      1)))

(append
   (lambda (x y) (cond
      ((null x) y)
      (t (cons (car x) (append (cdr x) y))))))
```

```
(assoc
   (lambda (xsas ysas) (cond
      ((null ysas) nil)
      ((equal (caar ysas) xsas) (car ysas))
      (t (assoc xsas (cdr ysas))))))

(attach
   (lambda (x y) (rplaca (rplacd y (cons (car y) (cdr y)))
x)))

(copy
   (lambda (x) (cond
      ((null x) nil)
      ((atom x) x)
      (t (cons (copy (car x)) (copy (cdr x)))))))

(deflist
   (lambda (l ind) (prog nil
      loop  (cond
               ((null l) (return nil)))
            (put (caar l) ind (cadar l))
            (setq l (cdr l))
            (go loop))))

(difference
   (lambda (x y) (plus
      x
      (minus y))))

(e
   (nlamda (xeeee) (eval xeeee)))

(ersetq
   (nlamda (ersetx) (errorset (car ersetx) t)))

(get
   (lambda (x y) (cond
      ((null x) nil)
      ((equal (car x) y) (cadr x))
      (t (get (cdr x) y)))))
```

```
(getp
   (lambda (x y) (prog (z)
               (setq z (cdr x))
      loop     (cond
                  ((null z) (return nil))
                  ((eq (car z) y) (return (cadr z))))
               (setq z (cddr z))
               (go loop))))

(intersection
   (lambda (x y) (cond
      ((null x) nil)
      ((member (car x) y) (cons (car x) (intersection
(cdr x) y)))
      (t (intersection (cdr x) y)))))

(last
   (lambda (x) (prog (xx)
      l        (cond
                  ((atom x) (return xx)))
               (setq xx x)
               (setq x (cdr x))
               (go l))))

(lconc
   (lambda (x p) (prog (xx)
               (return (cond
                  ((null x) p)
                  ((cdr (setq xx (last x))) (error (list
                     (quote lconc)
                     x)))
                  ((null p) (cons x xx))
                  ((null (car p)) (rplaca (rplacd p xx) x))
                  (t (prog2
                     (rplacd (cdr p) x)
                     (rplacd p xx))))))))

(length
   (lambda (x) (prog (n)
               (setq n 0)
      l        (cond
                  ((atom x) (return n)))
               (setq x (cdr x))
               (setq n (add1 n))
               (go l))))

(lessp
   (lambda (x y) (cond
      ((equal x y) nil)
      ((greaterp x y) nil)
      (t t))))
```

```
(map
   (lambda (mapx mapf) (cond
      ((null mapx) nil)
      (t (prog2
         (mapf mapx)
         (map (cdr mapx) mapf))))))

(mapc
   (lambda (mapcx mapcf) (cond
      ((null mapcx) nil)
      (t (prog2
         (mapcf (car mapcx))
         (mapc (cdr mapcx) mapcf))))))

(mapcar
   (lambda (mpcrx mpcrf) (cond
      ((null mpcrx) nil)
      (t (cons (mpcrf (car mpcrx)) (mapcar (cdr mpcrx) mpcrf
))))))

(mapcon
   (lambda (mpcnx mpcnf) (cond
      ((null mpcnx) nil)
      (t (nconc (mpcnf mpcnx) (mapcon (cdr mpcnx) mpcnf
))))))

(mapconc
   (lambda (mpcncx mpcncf) (cond
      ((null mpcncx) nil)
      (t (nconc (mpcncf (car mpcncx)) (mapconc (cdr mpcncx) mpcncf
))))))

(maplist
   (lambda (mplstx mplstf) (cond
      ((null mplstx) nil)
      (t (cons (mplstf mplstx) (maplist (cdr mplstx) mplstf
))))))

(minusp
   (lambda (x) (greaterp 0 x)))

(nill
   (nlamda (xnil) nil))

(nlsetq
   (nlamda (nlsetx) (errorset (car nlsetx) nil)))
```

```
(not
   (lambda (x) (cond
      ((null x) t)
      (t nil))))

(prop
   (lambda (x y u) (cond
      ((null x) (u))
      ((equal (car x) y) (cdr x))
      (t (prop (cdr x) y u)))))

(punch
   (lambda (x) (prog (y z)
            (setq y (punchon t))
            (setq z (typeout nil))
            (print x)
            (punchon y)
            (typeout z)
            (return x))))

(put
   (lambda (x y z) (prog nil
      loop  (cond
               ((null (cdr x)) (rplacd x (list
                  y
                  z)))
               ((equal (cadr x) y) (rplaca (cddr x) z))
               ((setq x (cddr x)) (go loop)))
            (return y))))

(rdflx
   (lambda (x) (prog (xx yy)
            (setq yy (typein t))
            (cond
               (x (go r1)))
            (setq xx (ersetq (read)))
            (go r2)
      r1    (cond
               ((setq xx (nlsetq (read))) (setq xx (car xx
)))
               ((print x) (go r1)))
      r2    (typein yy)
            (return xx))))
```

```
(remainder
   (lambda (x y) (cdr (divide x y))))

(remove
   (lambda (a x) (cond
      ((null x) nil)
      ((equal a (car x)) (remove a (cdr x)))
      (t (cons (car x) (remove a (cdr x)))))))

(remprop
   (lambda (x y) (prog nil
      loop  (cond
               ((null (cdr x)) (return y))
               ((equal (cadr x) y) (rplacd x (cdddr x)))
               (t (setq x (cdr x))))
            (go loop))))

(reverse
   (lambda (x) (prog (u)
      loop  (cond
               ((null x) (return u)))
            (setq u (cons (car x) u))
            (setq x (cdr x))
            (go loop))))

(sassoc
   (lambda (xsas ysas usas) (cond
      ((null ysas) (usas))
      ((equal (caar ysas) xsas) (car ysas))
      (t (sassoc xsas (cdr ysas) usas)))))

(setnq
   (nlamda (xsetnq) (setn (car xsetnq) (eval (cadr xsetnq)
))))

(setqq
   (nlamda (x) (set (car x) (cadr x))))

(soundexin
   (nlamda (x) (mapcar x (quote (lambda (ysdx) (put (soundex
 ysdx) (quote name) ysdx))))))

(soundexout
   (lambda (x) (getp x (quote name))))

(sub1
   (lambda (x) (plus
      x
      -1)))
```

```
(sub2
   (lambda (a z) (cond
      ((null a) z)
      ((equal (caar a) z) (cdar a))
      (t (sub2 (cdr a) z)))))

(sublis
   (lambda (a y) (cond
      ((atom y) (sub2 a y))
      (t (cons (sublis a (car y)) (sublis a (cdr y)))))))

(subst
   (lambda (x y z) (cond
      ((equal y z) x)
      ((atom z) z)
      (t (cons (subst x y (car z)) (subst x y (cdr z)))))))

(tconc
   (lambda (x p) (prog (xx)
            (return (cond
               ((null p) (cons (setq xx (cons x nil)) xx))
               ((null (car p)) (prog2
                  (rplaca p (cons x nil))
                  (rplacd p (car p))))
               (t (rplacd p (cdr (rplacd (cdr p)
(rplacd (cons x (cdr p)) nil))))))))))

(time
   (lambda (x n) (prog (y m c c1)
            (setq m n)
            (setq c (clock))
      t1    (cond
               ((zerop m) (setq c1 (clock)))
               (t (progn
                  (setq y (eval x))
                  (setq m (sub1 m))
                  (go t1))))
            (setq m (divide (plus
               c1
               (minus c)) n))
            (prin1 (car m))
            (prin1 period)
            (prin1 (quotient (times
               (cdr m)
               10) n))
```

```
                (prin1 blank)
                (print (quote seconds))
                (return y))))

(union
   (lambda (x y) (cond
      ((null x) y)
      ((member (car x) y) (union (cdr x) y))
      (t (cons (car x) (union (cdr x) y))))))

(zerop
   (lambda (x) (equal x 0)))
```

```
(break
   (lambda (fn when what) (prog (xx yy zz)
              (cond
                 ((null (setq xx (getd fn))) (return (prog2
                    (putd fn (list
                       (quote nlamda)
                       (quote (1))
                       (list
                          (quote break1)
                          nil
                          when
                          (setq xx (list
                             fn
                             (quote (undefined))))
                          what)))
                    xx)))
                 ((eq (setq yy (fntyp fn)) (quote fsubr)) (return
 (cons fn (quote (is an fsubr)))))
                 ((null (eq yy (quote subr))) (go b2)))
              (setq yy (rdflx (print (cons fn (quote (is a subr

 need args)))))))
              (putd (setq zz (gensym)) xx)
              (setq xx (putd fn (list
                 (quote lambda)
                 yy
                 (cons zz yy))))
        b2    (cond
                 ((eq (caaddr xx) (quote break1)) (setq xx (
list
                    (car xx)
                    (cadr xx)
                    (cadr (caddr xx))))))
              (putd fn (list
                 (car xx)
                 (cadr xx)
                 (list
                    (quote break1)
                    (caddr xx)
                    when
                    (list
                       fn)
                    what)))
              (return fn))))
```

```
(unbreak
   (lambda (fn) (prog (xx yy)
             (return (cond
                ((null (setq xx (getd fn))) (cons fn (quote
 (not a function))))
                ((and
                   (or
                      (eq (setq yy (fntyp fn)) (quote expr)
)
                      (eq yy (quote fexpr)))
                   (eq (caaddr xx) (quote break1))) (prog2
                   (putd fn (list
                      (car xx)
                      (cadr xx)
                      (cadr (caddr xx))))
                   fn))
                (t (cons fn (quote (not broken)))))))))

(breaklist
   (nlamda (x) (maplist x (quote (lambda (x) (break (car x
) t nil))))))

(unbreaklist
   (nlamda (x) (maplist x (quote (lambda (x) (unbreak (car
 x)))))))

(breakprog
   (lambda (bpx bpy) (maplist bpy (quote (lambda (z) (breakat
 bpx (car z) t nil))))))

(unbreakprog
   (lambda (x) (prog (xx)
             (setq xx (bp1 x))
       u1    (cond
                ((eq (caadr xx) (quote break1)) (rplacd xx
(cddr xx)))
                ((setq xx (cdr xx)) (go u1))
                (t (return nil)))
             (go u1))))
```

```
(breakat
   (lambda (fn where when what) (prog (a)
            (setq a (bp1 fn))
      b1    (cond
               ((equal (car a) where) (return (prog2
                  (rplacd a (cons (list
                     (quote break1)
                     nil
                     when
                     (list
                        fn
                        (quote at)
                        where)
                     what) (cdr a)))
                  where)))
               ((setq a (cdr a)) (go b1)))
            (return (cons where (quote (not found)))))))

(unbreakat
   (lambda (fn where) (prog (a)
            (setq a (bp1 fn))
      u1    (cond
               ((equal (car a) where) (return (cond
                  ((eq (caadr a) (quote break1)) (prog2
                     (rplacd a (cddr a))
                     where))
                  (t (cons fn (append (quote (not broken at
)) (list
                     where)))))))
               ((setq a (cdr a)) (go u1)))
            (return (cons where (quote (not found)))))))
```

```
(break1
   (nlamda (brk1x) (prog (brk1xx brk1yy brk1zz)
               (cond
                  ((null (setq brk1xx (eval (cadr brk1x)))) (
return (eval (car brk1x))))
                  ((null (equal brk1xx (quote (nil)))) (go b0
)))
               (print (append (quote (crack in)) (caddr brk1x
)))
               (cond
                  ((cadddr brk1x) (print (eval (cadddr brk1x)
))))
               (go b3)
         b0    (setq brk1yy (print (append (quote (break in))
 (caddr brk1x))))
               (cond
                  ((cadddr brk1x) (print (eval (cadddr brk1x)
))))
         b1    (cond
                  ((eq (setq brk1xx (rdflx brk1yy)) (quote quit
)) (error (caddr brk1x)))
                  ((eq brk1xx (quote stop)) (go b3))
                  ((eq brk1xx (quote return)) (go b2))
                  ((eq brk1xx (quote eval)) nil)
                  ((eq brk1xx (quote ok)) (go b3))
                  ((and
                     (ersetq (setq brk1xx (eval brk1xx)))
                     (nlsetq (print brk1xx))) (go b1))
                  ((print brk1yy) (go b1)))
               (cond
                  ((null (setq brk1zz (ersetq (eval (car brk1x
))))) (print brk1yy))
                  ((print (append (caddr brk1x) (quote (evaluated
)))) (set (caaddr brk1x) (car brk1zz))))
               (go b1)
         b2    (cond
                  ((and
                     (setq brk1zz (rdflx nil))
                     (setq brk1zz (ersetq (eval (car brk1zz))
))) (go b4))
                  ((print brk1yy) (go b1)))
         b3    (cond
                  ((or
                     brk1zz
                     (setq brk1zz (ersetq (eval (car brk1x)))
)) nil)
                  ((print brk1yy) (go b1)))
         b4    (cond
                  ((eq brk1xx (quote ok)) (print (caddr brk1x
)))
                  ((prog2
                     (print (append (quote (value of)) (caddr
 brk1x)))
                     (null (nlsetq (print (car brk1zz))))) (print
 (quote ok))))
               (return (car brk1zz)))))
```

```
(bp1
   (lambda (x) (prog (xx)
                 (return (cond
                    ((and
                       (or
                          (eq (setq xx (fntyp x)) (quote expr))
                          (eq xx (quote fexpr)))
                       (eq (caaddr (setq xx (getd x))) (quote prog
))) (caddr xx))
                    (t (error (cons x (quote (not a program))))
))))))
```

```
(prettydef
   (lambda (x) (prog (a)
                  (setq a (punchon t))
                  (prin1 (quote "("))
                  (print (quote defineq))
                  (prettyprint x)
                  (print (quote ")"))
                  (punchon a)
                  (return x))))

(prettyprint
   (lambda (l) (map l (quote (lambda (j) (prog (t1)
                  (terpri)
                  (prin1 lpar)
                  (print (car j))
                  (printdef (cond
                     ((getd (car j)))
                     (t (quote undefined))))
                  (prin1 rpar)
                  (terpri)))))))

(printdef
   (lambda (e) (prog (i iunit iunitl)
                  (setnq i 1)
                  (setq iunit (quote "   "))
                  (setq iunitl 3)
                  (prin1 iunit)
                  (superprint e)
                  (return nil))))

(superprint
   (lambda (e) (cond
      ((atom e) (cond
         ((member e (quote (""" " " "(" ")"
"
" "   " "." ","))) (prin1 (pack (list
                  (quote """)
                  e
                  (quote """))))))
         (t (prin1 e))))
      (t (prog (ep m)
               (setq ep e)
               (prin1 lpar)
```

```
a       (cond
           ((member (car ep) (quote (and
              or
              select
              selectq
              list
              plus
              times
              cond
              prog2
              progn))) (go pl))
           ((eq (car ep) (quote prog)) (go pp))
           ((atom (car ep)) nil)
           ((or
              (eq (caar ep) (quote lambda))
              (eq (caar ep) (quote nlamda))) (go pl
)))
        (superprint (car ep))
        (setq ep (cdr ep))
        (cond
           ((null ep) (return (prin1 rpar)))
           ((atom ep) (go pd)))
        (prin1 blank)
        (go a)
pk      (setnq i (sub1 i))
pd      (prin1 blank)
        (prin1 period)
        (prin1 blank)
        (prin1 ep)
        (return (prin1 rpar))
pl      (setnq i (add1 i))
        (superprint (car ep))
pm      (setq ep (cdr ep))
        (cond
           ((null ep) (go pj))
           ((atom ep) (go pk)))
        (endline)
        (superprint (car ep))
        (go pm)
pj      (setnq i (sub1 i))
        (return (prin1 rpar))
pp      (prin1 (car ep))
        (setq ep (cdr ep))
        (setnq i (add1 i))
        (cond
           ((null ep) (go pj))
           ((atom ep) (go pk)))
        (prin1 blank)
        (superprint (car ep))
py      (setq ep (cdr ep))
        (cond
```

```
                ((null ep) (go pj))
                ((atom ep) (go pk)))
            (endline)
            (cond
                ((atom (car ep)) (go pz)))
            (prin1 iunit)
            (prin1 iunit)
      px    (setnq i (plus
                i
                2))
            (superprint (car ep))
            (setnq i (plus
                i
                -2))
            (go py)
      pz    (prin1 (car ep))
            (setnq m (plus
                iunitl
                iunitl
                (minus (length (unpack (car ep))))))
      aa    (setnq m (sub1 m))
            (prin1 blank)
            (cond
                ((null (or
                    (zerop m)
                    (minusp m))) (go aa)))
            (setq ep (cdr ep))
            (cond
                ((null ep) (go pj))
                ((atom ep) (go pk))
                ((atom (car ep)) (go pz)))
            (go px))))))

(endline
   (lambda nil (prog (j)
           (setnq j i)
           (terpri)
     a     (cond
              ((zerop j) (return nil))
              ((minusp j) (error i)))
           (prin1 iunit)
           (setnq j (sub1 j))
           (go a))))
```

```
(trace
   (lambda (x) (prog (a b c g)
               (setq a x)
      loop     (cond
                  ((null x) (return a)))
               (setq b (getd (setq c (car x))))
               (setq x (cdr x))
               (cond
                  ((null b) (progn
                     (print (cons c (quote (undefined))))
                     (go loop)))
                  ((tracp c b) (progn
                     (print (cons c (quote (was traced))))
                     (go loop))))
               (putd (setq g (gensym)) b)
               (putd c (list
                  (quote nlamda)
                  (quote (q1qq))
                  (list
                     (quote trac1)
                     (list
                        (quote quote)
                        c)
                     (list
                        (quote quote)
                        g)
                     (quote q1qq))))
               (go loop))))

(untrace
   (lambda (x) (prog (a b c g)
               (set (quote a) x)
      loop     (cond
                  ((null x) (return a)))
               (set (quote g) (car x))
               (set (quote x) (cdr x))
               (cond
                  ((tracp g (set (quote b) (getd g))) (progn
                     (set (quote b) (cdaddr b))
                     (putd (cadar b) (getd (set (quote c) (cadadr
 b))))
                     (remob c)))
                  (t (print (cons g (quote (not traced))))))
               (go loop))))
```

```
(tracp
   (lambda (x y) (and
      (eq (fntyp x) (quote fexpr))
      (eq (caaddr y) (quote trac1)))))

(trac1
   (lambda (ctrac gtrac xtrac) (prog (atrac)
            (print (cons ctrac (quote (entered with))))
            (set (quote xtrac) (cond
               ((eq (fntyp gtrac) (quote fsubr)) (print xtrac
))
               ((eq (fntyp gtrac) (quote fexpr)) (print xtrac
))
               (t (evalprint xtrac))))
            (set (quote atrac) (eval (cons gtrac xtrac)))
            (print (cons ctrac (quote (has value))))
            (return (print atrac)))))

(evalprint
   (lambda (xvalp) (prog (avalp)
      loop  (cond
               ((null xvalp) (return avalp)))
            (set (quote avalp) (nnconc avalp (list
               (list
                  (quote quote)
                  (print (eval (car xvalp)))))))
            (set (quote xvalp) (cdr xvalp))
            (go loop))))
```

```
(editf
   (lambda (x) (prog2
      (putd x (edite (getd x)))
      x)))

(editv
   (lambda (x) (prog2
      (set x (edite (eval x)))
      x)))

(editp
   (lambda (x) (prog2
      (rplacd x (edite (cdr x)))
      x)))

(edite
   (lambda (x) (prog (l y c)
            (typein t)
            (setq l (list
               x))
            (print (quote edit))
      a     (cond
               ((null (ersetq (setq c (read)))) (go a))
               ((null c) (return (car (lastr l))))
               ((numberp c) (edit1f c))
               ((eq c (quote copy)) (setq y (copy l)))
               ((eq c (quote restore)) (setq l (cond
                  (y y)
                  (t l))))
               ((eq c (quote p)) (edit3f (quote (p 0))))
               ((atom c) (print qmark))
               ((numberp (car c)) (edit2f c))
               (t (edit3f c)))
            (go a))))

(edit1f
   (lambda (c) (cond
      ((eq c 0) (cond
         ((null (cdr l)) (print qmark))
         (t (setq l (cdr l)))))
      ((greaterp c 0) (cond
         ((greaterp c (length (car l))) (print qmark))
         (t (setq l (cons (car (nth (car l) c)) l)))))
      (t (print qmark)))))
```

```
(edit2f
   (lambda (c) (cond
      ((greaterp (car c) 0) (cond
         ((greaterp (car c) (length (car l))) (print qmark
))
         (t (rplaca l (edit2af (sub1 (car c)) (car l) (cdr
 c) nil)))))
      ((or
         (eq (car c) 0)
         (null (cdr c))
         (greaterp (minus (car c)) (length (car l)))) (print
 qmark))
      (t (rplaca l (edit2af (sub1 (minus (car c))) (car l)
 (cdr c) t))))))

(edit2af
   (lambda (n x r d) (prog2
      (cond
         ((null (eq n 0)) (rplacd (nth x n) (nconc r (cond
            (d (cdr (nth x n)))
            (t (cddr (nth x n)))))))
         (d (attach (car r) x))
         (r (rplaca x (car r)))
         ((rplaca x (cadr x)) (rplacd x (cddr x))))
      x)))

(edit3f
   (lambda (x) (cond
      ((eq (car x) (quote i)) (edit2f (list
         (cadr x)
         (eval (caddr x)))))
      ((eq (car x) (quote e)) (ersetq (print (eval (cadr x
)))))
      ((eq (car x) (quote n)) (nconc (car l) (cdr x)))
      ((eq (car x) (quote p)) (bpnt (cdr x)))
      ((member (car x) (quote (ri ro li lo))) (errorset (nconc
 x (quote ((car l)))) t))
      (t (print qmark)))))
```

```
(bpnt
   (lambda (x) (prog (y n)
            (cond
               ((zerop (car x)) (setq y (car l)))
               ((greaterp (car x) (length (car l))) (go b1
))
               ((minusp (car x)) (go b1))
               (t (setq y (car (nth (car l) (car x))))))
            (cond
               ((null (cdr x)) (setq n 2))
               ((null (numberp (cadr x))) (go b1))
               ((minusp (cadr x)) (go b1))
               (t (setq n (cadr x))))
            (return (cond
               ((nlsetq (print (leveln y n))) nil)
               (t (print (quote edit)))))
      b1    (return (print qmark)))))

(leveln
   (lambda (x n) (cond
      ((atom x) x)
      ((zerop n) (quote ∧))
      (t (mapcar x (quote (lambda (x) (leveln x (sub1 n)))
))))))

(nth
   (lambda (x n) (cond
      ((atom x) nil)
      ((greaterp n 1) (nth (cdr x) (sub1 n)))
      (t x))))


(lastr
   (lambda (x) (cond
      ((null x) (error (quote (null list))))
      ((null (cdr x)) x)
      (t (lastr (cdr x))))))
```

```
(ri
   (lambda (m n x) (prog (a b)
            (setq a (nth x m))
            (setq b (nth (car a) n))
            (cond
               ((or
                  (null a)
                  (null b)) (return (print qmark))))
            (rplacd a (nconc (cdr b) (cdr a)))
            (rplacd b nil))))

(ro
   (lambda (n x) (prog (a)
            (setq a (nth x n))
            (cond
               ((or
                  (null a)
                  (atom (car a))) (return (print qmark))))
            (rplacd (lastr (car a)) (cdr a))
            (rplacd a nil))))

(li
   (lambda (n x) (prog (a)
            (setq a (nth x n))
            (cond
               ((null a) (return (print qmark))))
            (rplaca a (cons (car a) (cdr a)))
            (rplacd a nil))))

(lo
   (lambda (n x) (prog (a)
            (setq a (nth x n))
            (cond
               ((or
                  (null a)
                  (atom (car a))) (return (print qmark))))
            (rplacd a (cdar a))
            (rplaca a (caar a)))))
```

# APPENDIX A

# OPERATING THE BBN-LISP SYSTEM

APPENDIX A.1

LISP LOADER

The LISP loader allows one to load several drum fields from either paper tape or magnetic tape. In addition, there is provision for transferring a system from drum to mag tape. A complete system is treated as a file on tape (each core load is one block of the file) and all tape commands are in terms of files rather than blocks. Teletype should be connected to channel 0 of the 630 scanner.

Instructions for Loading System Programs onto the Drum

The LISP loader can be used for setting up the drum fields of the system programs, including itself. To do this:

1. Read into core 1 the system program to be placed on a drum field.

2. Read into core 1 the program at location 0 for that drum field.

3. Read into core 0 the LISP loader.

4. Type: nd
where n is the octal number of the drum field onto which to dump core 1.

Instructions for Loading LISP with the Loader

1. Load mag tape of system on tape drive and set it to automatic on unit 1.

2. Read into core 0 the paper tape of the LISP loader. The mag tape will be rewound and the LISP loader will be waiting for typein. (The LISP loader starts at 300.)

3. Type: nr
where n is the octal number of the file to be read in. 26 drum fields will be read off of the mag tape onto the drum and the typewriter will type out n < m where n is the first block number read (starting with 0) and m is the last +1 block number read.

4. Type: 1
This will take the user to LISP.

Instructions for Writing LISP onto Mag Tape with the Loader

1. From LISP call the drum field with the LISP loader, FIELD (25Q), or read into core 0 the paper tape of the LISP loader.

2. Type: nw
where n is the octal number of the file that you wish to write.

List of Commands Available in the LISP Loader (n is an octal number)

| | |
|---|---|
| l | calls LISP |
| e | calls the editor on field 26 |
| nr | reads onto the drum from mag tape file n |
| nw | writes current drum system on mag tape file n |
| nd | dumps core 1 onto relative drum field n |
| nc | reads relative drum field n into core 1 |
| np | preserves core 0 on relative drum field n |
| ng | gets registers 0-177 on relative drum field n and transfers to 0 |
| nu | selects the mag tape unit to be used. Starting the program at 300 automatically selects unit 1. |
| nb | sets the base field on the drum to n, i.e., drum loading will begin on field n from either core or mag tape. The base is initially set to 1. The first relative field n is 1, not 0. Relative field n is absolute field "n - 1 + base". |
| nf | sets the number of fields in a file. This value is initially set to 26 octal. |
| o | rewind (origin) |
| ns | space tape n files forward (or backward if n is negative). If n is zero the ~~tape~~ will be moved to the beginning of the current file. Spacing backwards has been known ~~to cause~~ trouble. |

Error Printouts

| | |
|---|---|
| nOf | tried to reference file 0 or drum field 0 (either absolute or relative) |
| fle | file error -- while searching for a designated file, a file longer than 64 blocks was encountered. |
| una | tape unit not available. If this is the first thing that happens it is because the program has attempted to rewind unit 1 and cannot for some reason. |
| pmc n | bad parity or missed character on reading or checking tape block n |
| nch | saw no characters for 6 inches |
| ept | saw tape end point |
| wcf n | write check failure mag tape block n |
| drf n | drum read fail, absolute field n |
| nem | no end mark has been entered |
| dwe | drum write error |

# APPENDIX A.2

## USING LISP FROM THE COMPUTER ROOM TELETYPE

To use LISP from the computer room teletype: Connect the teletype to channel 0 of the scanner and then load the LISP system as described in Appendix A.1, LISP LOADER. The teletype will carriage-return and be waiting for input into evalquote.

Manual restart should never be used as there are no known ways to cause the system to halt or crash (if either does occur, record all particulars and deliver to D. Murphy). The following, however, do exist:

| | |
|---|---|
| start 202 | reinitializes all sequence break routines and restarts |
| start 203 | reinitializes entire system, i.e., kills everything and redefines only initial SUBR's and FSUBR's. |

## APPENDIX A.3

## USING LISP FROM A REMOTE DATASET

To use LISP from a remote dataset: The LISP system should be loaded and running as described in Appendix A.1, LISP LOADER. Then:

> Set the channel 0 dataset phone to "auto" (the channel 0 phone is the one on which the number 491-5120 appears).
>
> From the remote dataset, push the "tel" button, and when the dial tone is heard in the attached receiver, dial 491-5120. The phone in the computer room will be answered automatically, and a tone will be transmitted. When this tone is heard, the "ORIG" button should be pressed, establishing the connection.

Special Codes for Control (see standard chart of teletype codes for complete set)

| Octal Code | Character | Function |
|---|---|---|
| | rubout | deletes the line being typed in types out and deletes the last character typed in |
| | break key | causes an interrupt followed by an untrace. A second depression of this key halts the untrace. |

| Octal Code | Character | Function |
|---|---|---|
| 204 | control D | HANGUP, when transmitted by either computer or user, causes immediate hangup on both ends |
| 207 | control G | Bell |
| 211 | control I | Horizontal tab, on output only, causes carriage to be moved to next predefined tab stop |
| 221 | control Q | reader on: starts paper tape reader if tape is loaded |
| 223 | control S | reader off: when appearing on paper tape only, causes reader to stop after reading next character |

# APPENDIX B

## INDEX TO FUNCTIONS

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) | 2a. REPORT SECURITY CLASSIFICATION |
|---|---|
| Bolt Beranek and Newman Inc.<br>Cambridge, Massachusetts | Unclassified |
| | 2b. GROUP |

3\. REPORT TITLE

The BBN-LISP System

4\. DESCRIPTIVE NOTES (Type of report and inclusive dates)

Scientific Report No. 1

5\. AUTHOR(S) (Last name, first name, initial)

Daniel G. Bobrow, D. Lucille Darley, Daniel L. Murphy, Cynthia Solomon, Warren Teitelman

| 6. REPORT DATE | 7a. TOTAL NO. OF PAGES | 7b. NO. OF REFS |
|---|---|---|
| February 1966 | 82 | 0 |

| | |
|---|---|
| 8a. CONTRACT OR GRANT NO.<br>AF 19(628)-5065 - ARPA Order No. 627 | 9a. ORIGINATOR'S REPORT NUMBER(S)<br>BBN Report No. 1346 |
| b. PROJECT NO.<br>8668 | |
| c. | 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)<br>AFCRL-66-180 |
| d. | |

10\. AVAILABILITY/LIMITATION NOTICES

Distribution of this document is unlimited

| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY |
|---|---|
| ARPA Order No. 627, dated 9 March 1965. | Hq. AFCRL, OAR (CRB)<br>United States Air Force<br>L.G. Hanscom Field, Bedford, Mass. |

13\. ABSTRACT

This report describes in detail the BBN-LISP system. This LISP system has a number of unique features; most notably, it has a small core memory, and utilizes a drum for storage of list structure. The paging techniques described here allow utilization of this large, but slow, drum memory with a surprisingly small time penalty. These techniques are applicable to the design of efficient list processing systems embedded in time-sharing systems using paging for memory allocation.

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| LISP<br>List Processing Language<br>Paging Systems<br>Drum Systems for List Structure<br>List Structures<br>Symbol Manipulation Language | | | | | | |

## INSTRUCTIONS

1. ORIGINATING ACTIVITY: Enter the name and address of the contractor, subcontractor, grantee, Department of Defense activity or other organization *(corporate author)* issuing the report.

2*a*. REPORT SECURITY CLASSIFICATION: Enter the overall security classification of the report. Indicate whether "Restricted Data" is included. Marking is to be in accordance with appropriate security regulations.

2*b*. GROUP: Automatic downgrading is specified in DoD Directive 5200.10 and Armed Forces Industrial Manual. Enter the group number. Also, when applicable, show that optional markings have been used for Group 3 and Group 4 as authorized.

3. REPORT TITLE: Enter the complete report title in all capital letters. Titles in all cases should be unclassified. If a meaningful title cannot be selected without classification, show title classification in all capitals in parenthesis immediately following the title.

4. DESCRIPTIVE NOTES: If appropriate, enter the type of report, e.g., interim, progress, summary, annual, or final. Give the inclusive dates when a specific reporting period is covered.

5. AUTHOR(S): Enter the name(s) of author(s) as shown on or in the report. Enter last name, first name, middle initial. If military, show rank and branch of service. The name of the principal author is an absolute minimum requirement.

6. REPORT DATE: Enter the date of the report as day, month, year; or month, year. If more than one date appears on the report, use date of publication.

7*a*. TOTAL NUMBER OF PAGES: The total page count should follow normal pagination procedures, i.e., enter the number of pages containing information.

7*b*. NUMBER OF REFERENCES: Enter the total number of references cited in the report.

8*a*. CONTRACT OR GRANT NUMBER: If appropriate, enter the applicable number of the contract or grant under which the report was written.

8*b*, 8*c*, & 8*d*. PROJECT NUMBER: Enter the appropriate military department identification, such as project number, subproject number, system numbers, task number, etc.

9*a*. ORIGINATOR'S REPORT NUMBER(S): Enter the official report number by which the document will be identified and controlled by the originating activity. This number must be unique to this report.

9*b*. OTHER REPORT NUMBER(S): If the report has been assigned any other report numbers *(either by the originator or by the sponsor)*, also enter this number(s).

10. AVAILABILITY/LIMITATION NOTICES: Enter any limitations on further dissemination of the report, other than those imposed by security classification, using standard statements such as:

(1) "Qualified requesters may obtain copies of this report from DDC."

(2) "Foreign announcement and dissemination of this report by DDC is not authorized."

(3) "U. S. Government agencies may obtain copies of this report directly from DDC. Other qualified DDC users shall request through ______________."

(4) "U. S. military agencies may obtain copies of this report directly from DDC. Other qualified users shall request through ______________."

(5) "All distribution of this report is controlled. Qualified DDC users shall request through ______________."

If the report has been furnished to the Office of Technical Services, Department of Commerce, for sale to the public, indicate this fact and enter the price, if known.

11. SUPPLEMENTARY NOTES: Use for additional explanatory notes.

12. SPONSORING MILITARY ACTIVITY: Enter the name of the departmental project office or laboratory sponsoring *(paying for)* the research and development. Include address.

13. ABSTRACT: Enter an abstract giving a brief and factual summary of the document indicative of the report, even though it may also appear elsewhere in the body of the technical report. If additional space is required, a continuation sheet shall be attached.

It is highly desirable that the abstract of classified reports be unclassified. Each paragraph of the abstract shall end with an indication of the military security classification of the information in the paragraph, represented as *(TS), (S), (C), or (U)*.

There is no limitation on the length of the abstract. However, the suggested length is from 150 to 225 words.

14. KEY WORDS: Key words are technically meaningful terms or short phrases that characterize a report and may be used as index entries for cataloging the report. Key words must be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location, may be used as key words but will be followed by an indication of technical context. The assignment of links, roles, and weights is optional.