THE BBN-LISP SYSTEM

REFERENCE MANUAL
APRIL 1969

(D.G. Bobrow, D.L. Murphy, W. Teitelman)

Bolt Beranek and Newman Inc

THE BBN - LISP SYSTEM

REFERENCE MANUAL
APRIL 1969

(D.G. Bobrow, D.L. Murphy, W. Teitelman)

Bolt Beranek and Newman Inc

ACKNOWLEDGEMENTS

The design, construction and documentation of this system was sponsored by the Information Processing Techniques section of the Advanced Research Projects Agency. The basic design and implementation of this paged LISP was done by D.G. Bobrow and D.L. Murphy. L.P. Deutsch also participated in the programming of the system and wrote the first versions of our LISP compiler and editor. Many of the extended interactive features are the work of Warren Teitelman and the existence of this expanded revised edition of the reference manual is due primarily to his perseverence and effort. The authors are grateful to Rita Doherty and Karen O'Sullivan for their hard work and patience through endless revisions of the text.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

SECTION I

INTRODUCTION

This document describes the BBN-LISP system currently implemented on the SDS 940. It is a dialect of LISP 1.5 and the differences between IBM 7090 version and this system are described in Appendix 1 and 2. Principally, this system has been expanded from the LISP 1.5 on the 7090 in a number of different ways. BBN-LISP is designed to utilize a drum for storage and to provide the user a large virtual memory, with a relatively small penalty in speed (using special paging techniques described in Bobrow and Murphy 1967). Secondly, this system has been designed to be a good on-line interactive system. Some of the features provided include sophisticated debugging facilities with tracing and conditional breakpoints, a sophisticated LISP oriented editor within the system, and compatible compiler and interpreter. Utilization of a uniform error processing through a user accessible function has allowed the implementation of a do-what-I-mean feature which can correct errors without losing the context of the computation. The philosophy of the DWIM feature is described in Teitelman 1969. In addition to the sub-systems described in this manual, a complete format directed list processing sub-system (FLIP, Teitelman, 1967) is available within BBN-LISP. There is also an assembler for inserting machine code sub-routines within BBN-LISP, and facilities for using the CRT display and CALCOMP plotter.

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 only be clear to people who have had some experience with other LISP systems. A good introduction to

LISP has been written by Clark Weissman (1967). Although not completely accurate with respect to BBN-LISP, the differences are small enough to be mastered by use of this manual and on-line interaction. Another useful introduction is given by Berkeley (1964) in the collection of Berkeley and Bobrow (1966).

Changes to this manual will be issued by replacing sections or pages which are faculty and reissuing the index and table of contents at periodic intervals.

Bibliography

Berkeley, E.C. (1964)"LISP, A Simple Introduction" in Berkeley, E.C. and Bobrow, D.G. (1966).

Berkeley, E.C. and Bobrow, D.G. (editors), (1966), The Programming Language LISP, Its Operation and Applications, MIT Press, 1966.

Bobrow, D.G. and Murphy, D.L. (1967) "The Structure of a LISP System Using Two Level Storage", Communications of the ACM, V15 3, March 1967.

McCarthy, J. et al, LISP 1.5 Programmers Manual, MIT Press, 1966.

Teitelman, W. "Toward a Programming Laboratory" in Walker, D. (ed) International Joint Artificial Intelligence Conference. May, 1969.

Teitelman, W. FLIP, A Format Directed List Processor in LISP, BBN Report 1967.

Weissman, C. (1967) LISP 1.5 Primer, Dickenson Press (1967).

SECTION II

USING THE LISP SUBSYSTEM ON THE 940

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*. The system so obtained contains all of the basic functions and programming and debugging aids described in the manual, including the LISP compiler and FLIP. Typing SYSGET(T) to *evalquote* will return you to this initial system at any time. Typing control-C will take you instantly back to the LISP executive at any time except during garbage collection. To get the effect of typing to a Lisp executive *eval*, type E and a space followed by the expression to be evaluated. This effect is achieved by the function *e* described in section 8.

When typing in to the LISP *read* function (used by *evalquote* and most other programs), 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 control-R while in the middle of a typein to the LISP executive, *evalquote*, will clear the entire read buffer of everything to the last ←, and LISP will again type ← . Several other control characters are interpreted by the LISP input fork, and their functions are summarized in Appendix 3.

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 the descriptors take up valuable in-core space.

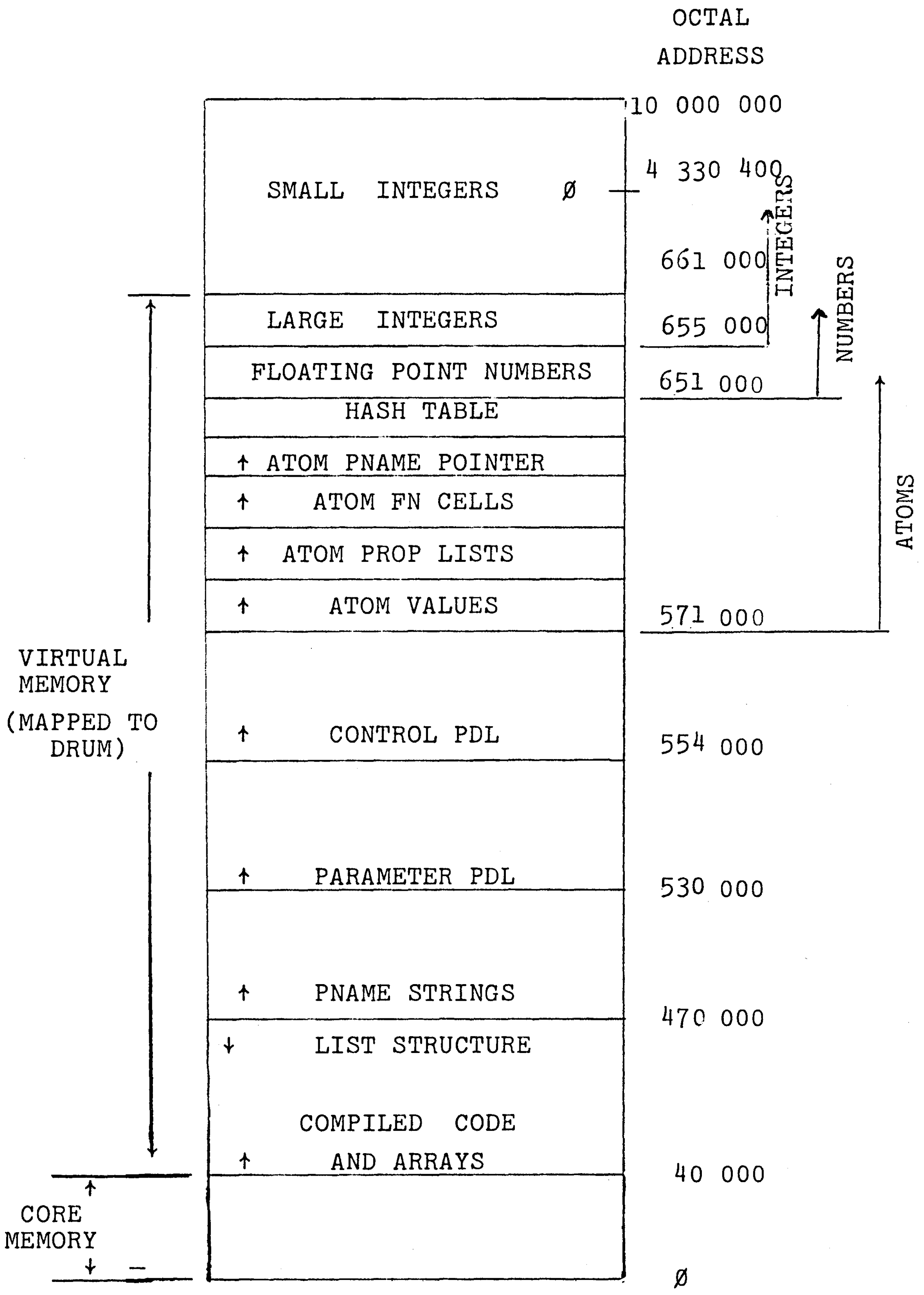


FIG. 1 MEMORY ALLOCATION IN LISP

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 determined by its print name. Special syntactic characters can be included in print names through the use of the quote mark, " (see the description of the function read).

Four cells (94∅ 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 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.

---

* and initialized with value NOBIND, property list NIL, function definition NIL.

Car of a literal atom usually contains the top level binding of the atom. If the atom has not yet been set 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. It has been defined that car[NIL] and cdr[NIL] are NIL, and cannot be changed. 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 21 bit virtual address space 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, as indicated in Fig. 1.

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 94Ø 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.

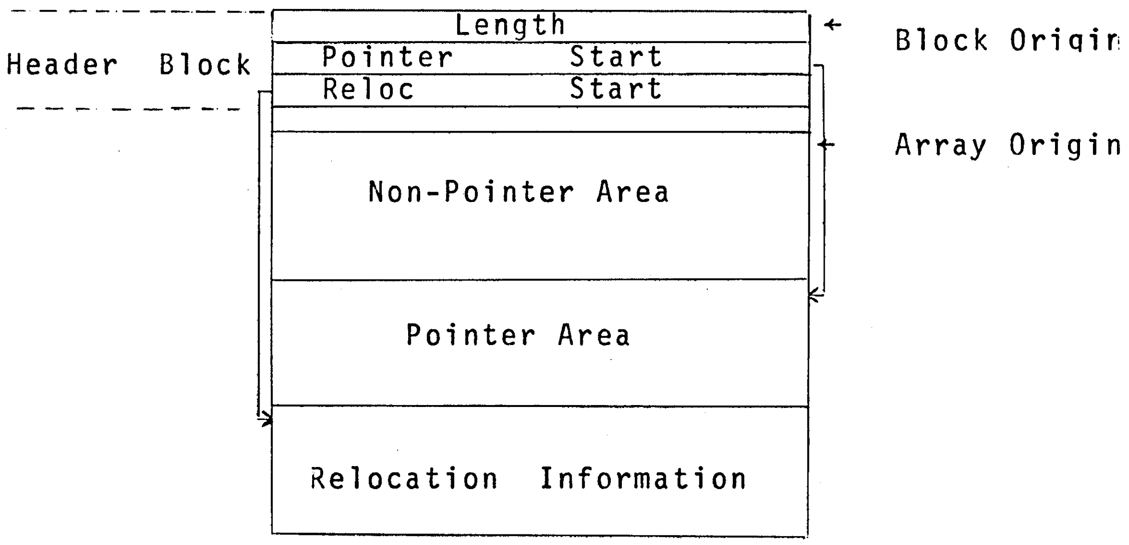
```
5.    5.0    5E0    5E-3    5.2E+6    .3
```

The floating point/string conversion, and the floating point arithmetic are performed by the POP's and BRS's available in the 94Ø 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 block=arraysize + 4.

1 Address of first word of protected pointers, relative to Array Origin.

2 Address of first word of relocation information, relative to Block Origin.

3 Used for temporary storage during garbage collection.

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; the 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. The space available in the 4-1-68 LISP system is 144K (K=1024) SDS 940 24 bit words, which if used all for list storage would provide 72K words of free storage.

Shared LISP

The LISP System as presently implemented contains nearly 90,000 words of compiled code constituting the miscellaneous functions, Editor, Compiler, Break and other service packages. A sharing mechanism enables one copy of this code residing on the drum to be used by all active users of LISP. This practice results in a considerable saving of drum space over that required if each user had a separate, private copy of these functions. When a user starts a LISP on his console, the virtual memory is set to contain all the shared pages which constitute the basic system. In addition, roughly 1,000 words of private list storage are also provided. As the user adds his own private functions and data to the system, private pages are assigned to contain them. Thus a running system will typically contain some number of private pages and the shared pages of the basic LISP system.

Fundamental to the proper operation of the sharing mechanism is the requirement that no individual user be permitted to change the contents of a shared page. Therefore, the shared pages in the virtual memory are initially set to be <u>read-only</u>. This means that the user can do <u>car</u> of the list structure on a shared page but not <u>rplaca</u>. However, circumstances do arise when it is necessary for the user to change his virtual memory in a place where a shared page has been mapped. For example, the user may set the top level value of an atom contained in the original shared system, i.e. change the contents of the value cell. To properly handle this situation, the LISP page turning routine takes special note of any attempt to store data into a shared page and makes a _private_ copy of the page, assigning it to a new place on the drum. This procedure is invoked automatically and is invisible to the user.

Garbage Collector

The garbage collector is a routine which serves to locate cells no longer in use by the running program and make them again available for storage. The various data spaces in LISP which may need to be garbage collected in this way include lists, arrays (and compiled-code), large integers, floating point numbers, atoms, and print-names.

An automatic garbage collection is usually initiated whenever a cell is needed in a space which has become exhausted. This happens most frequently when the allocated free list words have become exhausted by repeated _cons_es. A garbage collection will also be initiated whenever print-name space is exhausted. The garbage collection initiated for either of these reasons will reclaim lists, numbers, atoms, and print-names. A garbage collection initiated when array space is exhausted will collect these spaces, and in addition, will compact array space. This means that unused arrays will be eliminated, and still-in-use arrays will be moved so as to be contiguous.

When either large integers or floating-point numbers are exhausted, a special type garbage collection called _number collection_ is initiated. This operation identifies still-in-use numbers by performing a linear sweep over all spaces. This may result in the retention of some numbers which are no longer in use. Therefore, if a number collection is unsuccessful in obtaining free number cells, a regular garbage collection is initiated.

The user can initiate a regular garbage collection at any time via the function reclaim described in Section 10. Note that the depletion of atom space will not cause an automatic garbage collection. Instead, the error ATOM SPACE FULL is generated. However, in this case, an explicit reclaim may be successful in recovering atoms.

Allocation of List Space

Normally, a user will have in use for list structure only a small portion of the total space available for this purpose. In order to prevent scattering lists over many pages(which increases access time), LISP allocates and places on the free list only a portion of the total list structure space. A garbage collection will be initiated whenever this allocated portion becomes exhausted, whether or not additional space is available. After a garbage collection, additional pages will be allocated to list space if necessary to raise the total number of available free words to the minimum, a parameter set by minfs (described in Section 10). The two number spaces, atom space, and print name space have fixed boundaries, and an error will be generated if additional space is needed and none is available. Note that list space and array space are allocated from a common area. Array space recovered by a garbage collection can be subsequently used by list space because array space is compacted. However list space is not and cannot reasonably be compacted, so acquiring all of LISP's memory for list structure will prevent any further allocation of arrays for compiled functions.

Shared Areas

The garbage collector takes special note of the shared areas of virtual memory. Specifically, compiled functions and arrays residing on shared pages are not traced for the purpose of identifying list structures and numbers to be retained. Instead, a separate list, created at the time that the shared system was loaded, serves this purpose. This results in a considerable saving of time over what would be required if the garbage collector had to trace through all 90,000 words of compiled code in the shared system.

Initially, LISP memory is over half allocated to the shared portion of the systems. If a particular user requires more than the remaining space for his program, it is possible to remove portions of the shared system using the function flushcode (described in Section 22). The portion flushed is automatically available for allocation to array space. Atoms in this portion that are now no longer used, e.g. function and argument names, can be reclaimed via use of the function atomgc described in Section 10.

SECTION IV

FUNCTION TYPES

There are basically twelve function types in the BBN LISP System. These twelve types reflect three characteristics. 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, by 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 a list of atoms (possibly empty) or any literal atom (except NIL). 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. This is called "spreading" the arguments, and the function is called a spread-LAMBDA or spread-NLAMBDA. If an <u>atom</u> 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>. Such a function is called a "nospread" function.

If a LAMBDA is followed by an atom, each of its <u>n</u> arguments 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, arg[m] 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 using the ASSEMBLE directive of the compiler. Functions created by the compiler, whether from S-expressions or ASSEMBLE directives, 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). 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. Functions of types in the first two rows evaluate their arguments. The prefix F in the third and fourth rows 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]]	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.
cadr[x] = car[cdr[x]]	
cddddr[x] = [cdr[cdr[cdr[cdr[x]]]]	

cons[x;y]	cons constructs a dotted pair of x and y. If y is a list, x 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 y if y is a list and there is room; otherwise
2) on the page with x if x is a list and there is room; otherwise
3) on the same page as the last cons 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 cons with the following function:

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

conscount[]	Returns the number of *conses* since LISP started up.
rplacd[x;y]	This 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.

$cond[c_1;c_2;\ldots;c_k]$

The conditional function of LISP, cond, takes an indefinite number of arguments, $\underline{c}_1,\underline{c}_2,\ldots\underline{c}_k$, called clauses. Each clause $\underline{c}_i$ is a list $(\underline{e}_{1i}\ldots\underline{e}_{ni})$ of $n\geq 1$ items. The clauses are considered in sequence as follows: the first expression $\underline{e}_{1i}$ of the clause $\underline{c}_i$ is evaluated and its value is classified as false (equal to NIL) or true (not equal to NIL). If the value of $\underline{e}_{1i}$ is true, the expressions $\underline{e}_{2i}\ldots\underline{e}_{ni}$ that follow in clause $\underline{c}_i$ are evaluated in sequence, and the value of the conditional is the value of $\underline{e}_{ni}$, the last expression in the clause. In particular, if n=1, i.e., if there is only one expression in the clause $\underline{c}_i$, the value of the conditional is the value of $\underline{e}_{1i}$.

If $\underline{e}_{1i}$ is false, then the remainder of clause $\underline{c}_i$ is ignored, and the next clause $\underline{c}_i+1$ is considered. If no $\underline{e}_{1i}$ is true for any clause, the value of the conditional expression is NIL.

This conditional expression form gives the same value as LISP 1.5 for clauses of exactly two items but allows additional flexibility.

selectq$[x;y_1;y_2;\ldots;y_n;z]$

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

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

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 $\underline{e}_{1i},\ldots\underline{e}_{ki}$ 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_).

prog1[x_1;x_2;...;x_n]

This function evaluates its arguments in order, that is, $\underline{x}_1$ then $\underline{x}_2$ etc. It returns the value of its first argument $\underline{x}_1$.

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_.

rpt[n;form]	Evaluates the expression form n times. Returns the value of the last evaluation.
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, if flow of control "falls off the end".
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.
arrayp[x]	is T if x is an array; NIL otherwise.
listp[x]	is T if x is a nonatomic list-structure, i.e., created by one or more CONSes NIL otherwise. Since arrays are not atoms, and will fail an atom test, listp should be used to distinguish bona fide list structure from atoms, numbers, arrays, et al.
nlistp[x]	not[listp[x]]
eq[x;y]	The value of eq is T if x and y are pointers to the same structure in memory, and NIL otherwise. eq is compiled open by the compiler as a 24 bit compare of pointers. Its value is not guaranteed T for equal numbers which are not small integers. See eqp.

eqp[x;y]	The value of *eqp* is T if *x* and *y* are pointers to the same structure in memory, or if *x* and *y* are numbers and have the same value. Its value is NIL otherwise.
neq[x;y]	The value of this function is T if *x* is not *eq* to *y*, and NIL otherwise.
nill[]	Defined as (LAMBDA NIL NIL)
null[x]	eq[x;NIL]
equal[x;y]	The value of this function is T if *x* and *y* are isomorphic, that is, *x* and *y* print identically; the value of *equal* is NIL otherwise.
and[x_1;...x_n,]	This function is an FSUBR and can take an indefinite number of arguments (including 0). Its value is the value of its last argument if all of its arguments have non-null value, otherwise NIL. and[]=T. Arguments past the first null argument are not evaluated.

or[x_1;...x_n,]	This function is also an FSUBR and can take an indefinite number of arguments (including 0). Its value is that of the first argument whose value is non-null, otherwise NIL. or[]=NIL. Arguments past the first non-null arguments are not evaluated.
not[x]	Same as null; that is, eq[x;NIL].
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 are 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. It is more efficient to make x be the shorter 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.
nlist[x_1;...,x_n,]	Returns a list of the value of all arguments (same as LIST), but deletes all NIL's at the end of this list. Example: (NLIST T T NIL T NIL NIL) = (T T NIL T)
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. If x is NIL, it returns y.
nconc[u;v]	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. If x is NIL, it returns y itself.
nconcl[lst;x]	performs nconc[lst;list[x]]. The cons will be on the same page as lst.

tconc[x;p]	This function provides an efficient 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 element 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 *l* 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 thus does not use any additional storage.
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 S-expression *y* in the S-expression *z*. Substitution occurs whenever *y* is *equal* to *car* of some sub-expression of *z* or when *y* is

subst[x;y;z] (cont.)	both atomic and *eq* to *cdr* of some subexpression of *z*. For example: subst[A;B;(C B (X . B))] gives (C A (X . A)) subst[A;(B C);((B C) D B C)] gives (A D B C), not (A D . A) The value of *subst* is a copy of *z* with the appropriate changes.
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: $((u_1.v_1)\ (u_2.v_2)\ \ldots\ (u_n.v_n))$ with each u_i atomic. 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.

lsublis[x;y]	x is a list of pairs as for <u>sublis</u>, except that the v_i are substituted as <u>segments</u> of a list, not as items. For example, sublis[((A B C)); (X A Y)] = (X (B C) Y) but lsublis[((A B C));(X A Y)] = (X B C Y). Note also that lsublis[((A)); (X A Y)] = (X Y) <u>lsublis</u> is destructive: it physically changes the list structure of y itself.

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.
count[x]	Returns the number of LISP words in the list structure *x*. Returns Ø if *x* is an atom.

ldiff[x;y;z]	y is a tail of x, i.e., the result of applying some number of CDRs to x ldiff[x;y] gives a list of all elements in x but not in y, i.e., the list difference of x and y. Thus (LDIFF X (NTH X (ADD1 M))) gives the first M elements of X, (LDIFF X (MEMBER (QUOTE FOO) X)) gives all elements in X up to the first FOO. If z is not NIL the value of ldiff is effectively nconc[z;ldiff[x;y]], i.e. the list difference is added at the end of z.
editnth[x;n]	similar to the function nth except n may be positive or negative. If n is positive, car of value is nth element of x. If n is negative, car of value is nth element of x counting from the end. i.e., editnth[x;-1] = last[x]. If n is too large (or too small), editnth generates an error. Note that nth does not.
li[n;x]	equivalent to executing the edit command (LI N) when x is the current level list.
ri[m;n;x] bi[m;n;x] lo[n;x] ro[n;x] bo[n;x]	equivalent to corresponding edit command

makelist[n;m]

makes a list of length m consisting of the contents of cells n, ... n+m-1. For example, if FOO is an array pointer,

```
(MAKELIST (PLUS (LOC FOO) 4) 3)
```

is a list consisting of the first three elements in the array FOO.

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. Its value is *z*.
remprop[x;y]	This function *rem*oves all occurrences of the *prop*erty 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.
changeprop[x;prop1;prop2]	Changes name of property *prop1* to *prop2* on atom x, (does not affect the value of the property). Value is x. If *prop1* not found, value is NIL.

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.
getl[x;y]	y is a list of properties. getl searches the property list of x, two at a time, and returns the property list as of the first property on y that it finds, e.g., with above property list, getl[x;(PROP2 PROP3)]=(PROP2 B 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 no arguments, if given, is taken as the value of *sassoc*. Otherwise, its value is NIL.

Note: Many atoms in the system already have property lists, usually for use by the compiler. Be careful not to clobber their property lists by using *rplacd*.

SECTION VIII

FUNCTION DEFINITION AND EVALUATION

getd[x]	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 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* *d*efinition 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*.
movd[from;to;copyflg]	Moves definition of *from* to *to* i.e., redefines *to*. If *copyflg*=T, a *copy* of the definition of *from* is used.

fntyp[fn]	This function returns NIL if <u>fn</u> if not the name of a defined function, or a function definition. Otherwise <u>fntyp</u> 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 <u>F</u> indicates unevaluated arguments; the prefix <u>C</u> indicates compiled code; and the suffix * indicates an indefinite number of arguments.

define[x]	The argument of <u>define</u> 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 generate an error on encountering an atom where a defining list is expected.

If _dfnflg_=T, its normal setting, an attempt to _redefine_ a function _fn_ will cause _define_ to print the message (fn REDEFINED) and to save the old definition of _fn_ using _savedef_ before redefining it.

Note: _define_ will operate correctly if the function is already defined and _broken_, _advised_, or _broken-in_.

savedef[fn]

Saves the definition of _fn_ on its property list under property EXPR, CODE, or SUBR depending on its type. If _fn_ is a list, _savedef_ operates on each function in the list.

unsavedef[fn;prop]

Restores the definition of _fn_ from _prop_. If _prop_ is not given, _unsavedef_ looks under EXPR, CODE, and SUBR, in that order, before giving an error. If _dfnflg_=T, the current definition of _fn_ is saved using _savedef_. Thus one can use _unsavedef_ to switch back and forth between two definitions of the same function, keeping one on its property list and the other in the function cell.

	If *fn* is a list, *unsavedef* operates on each function in the list.
defineq[x;...z]	This FEXPR is closely related to *define*. However, it takes an indefinite number of arguments which are not evaluated. 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*. Since *defineq* calls *define*, *dfnflg* affects its operation as well as that of *define*.
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]	fn is either the name of a function or its definition. Value of arglist is the list of names of the arguments, or in the case of a non-spread function, the single atom that is the name of the argument. By convention, arguments to all functions of type SUBR are u, v, and w, in that order. For functions of type SUBR*, FSUBR, FSUBR*, or undefined functions, arglist causes a helpable error.

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) via the function editf, e.g. EDITF(APPEND). However, it can be used to edit variables, via editv, property lists, via editp, or arbitrary expressions, via edite. (Editf, editv and editp all use edite, see p. 9.18). 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 in 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 the last command on its line. No commands on a line are performed until the user or the system types a carriage return.

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 _n_th sublist, and Ø, to revert to the superlist. If _n_ is a positive integer, then -_n_ examines the _n_th 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 _n_th element 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)
```

Both the (F e T) and (F e n) commands will automatically ascend to higher level expressions if the structure e is not found in the current list. The entire search is done in printout order, starting with the current list, and, if e is not found, proceeding to those portions of higher level lists that would be printed subsequently to the current list. Thus:

```
* 4 2 P
((NUL X) Z)
*(F X T)
*P
(X)
*(F CONS T)
*P
(CONS (CAR) (APPEND &))
```

```
*Ø P
(T (CONS & &))
*(F NUL T)
(F NUL T) ?
```

The question mark typed after the command in error is the editor's all-purpose comment: it simply means something is wrong with the indicated command. In this case, it is because the search failed to find a NUL following the current position, although of course a NUL does appear earlier in the structure.

Another useful variation of the find command is provided by (F e N), to be distinguished from (F e n), n a number. Here N means Next, and the search skips over the first element in the current list, and then proceeds exactly the same as (F e T). Thus the command (F e 2) will produce the same results as the command (F e T) followed by (F e N). The find-next command is useful for cycling through a large structure and examining and/or changing several instances of the same expression. It is also extremely useful in conjunction with edit macros, which are explained later.

For all of the four variants of the F command described, the argument e need not be a literal S-expression. The symbol & can be used to match any single element of a list; the symbol -- to match with the rest of any list. Thus in our example, (F (NUL &) T) will find (NUL X) and (F (CDR --) T) will find (CDR X Y), as will (F (CDR & &) T), but not (F (CDR &) T).

These two special symbols can be useful in finding a particular expression which is similar to many others. For example, if there are many places in a program where the variable X is set, (F SETQ T) may not find the expression you are looking for, nor perhaps will (F (SETQ X &) T). It may be necessary to type (F (SETQ X (LIST --)) T) to find the correct expression. However, the usual technique in this case is to pick out a unique expression or atom which occurs just prior to the desired expression and then do two F commands. This "homing in" process seems to be more convenient than ultra-precise specification of _e_.

For all find commands, if _e_ is atomic, it will be the first element of the current list after the find command has (successfully) operated. If _e_ is nonatomic, the corresponding structure will _be_ the current list. To insert before or after this expression, or to delete or replace it, the command UP, described on p. 9.29, can be used to make the current list list be the first element in the next higher list.

The find commands can be used on a list structure that is circular through a _car_ chain by appropriately setting the free variable _maxlevel_. This variable determines how "deep" the editor will search before abandoning a given structure, where the depth of a structure is the number of unpaired left parentheses preceding it in a printout. _Maxlevel_ is initially set to 100 .

An abbreviated form exists for doing the usual _find_ command (F e N). Typing F e (with no parentheses) achieves the same effect. After the F is typed in the editor expects a next expression to be typed in as the search goal. See 9.26 for a more complete explanation.

Three facilities are available for saving information relating to the current state of the edit and later retrieving it. At any stage i 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.

Frequently one wants 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, with a command (S v), or to the _nth_ sublist of the current list with (S v n), _n_ positive or negative. The _I_ command described below can then be used to treat this value exactly as though it had been typed in literally. Thus:

```
     *↑ (S EL2 2)
```

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

Modification commands

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 _n_ of the current list, one simply gives the command (-n $e_1 \ldots e_m$), thus:

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

To replace the <u>n</u>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)
```

To append the elements $\underline{e}_1 \ldots \underline{e}_m$ to the end of the current list, one gives the command (N $\underline{e}_1 \ldots \underline{e}_m$).

```
*(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 $\underline{e}_1$ by $\underline{e}_2$ in the current list and all its substructure. 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)
```

A mechanism by which lists saved with the S command may be used, is (I c e_1, ... e_n), which is equivalent to ([atom[c]→c; T→eval[c]] eval[e_1] ... eval[e_n]).

If EL2 has been set to (X) as shown above:

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

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

<u>Structure changing commands</u>

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 element 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 one wants 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 <u>n</u>th sublist (or, if n=0, the current list) to <u>m</u> 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 evaluates e and prints its value 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. Typing E e (without parentheses) also causes e to be evaluated.

The command (E e T) causes the expression e to be evaluated without being printed. It is primarily useful for defining macros.

The command PP causes the current list to be printed in a pretty form using printdef, a subordinate function of prettyprint (see 14.14). PP is equivalent to the two commands (S FOO) (E (PRINTDEF FOO) T).

This completes the discussion of the commands built into the editor. The following section on Edit Macros describes ways of augmenting this set. It should be emphasized that all user inputs are interpreted as edit commands, and have no bearing on any external functions. The command (F X T) will not be affected by the existence of a user function named F. The work done by the commands LI,LO,RI,RO,BI, and BO happen to be carried out by edit functions of the same name; but as far as the user is concerned, he is not calling these functions when he types (LI 2), but merely giving the editor a command which in some mysterious way it carries out. The only way the user can call a function explicitly is to use either the E or I command, (or to use a macro which uses an E or an I command)

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 list, will be executed. If RF were made the last command in the list, the sequence would be repeated until FOO could not be found, that is if RF were defined by

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

The simple edit macro 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, which is listed to indicate that SW will always be used with arguments. 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 SW1)
    (I 2 SW2)
```

An example of a macro which calls a function is

```
    (M (FOO) (N FN) (NTH N) (S FIE 1) (I 1 (FN FIE)))
```

Thus typing (FOO -1 CADR) would cause the last element in the current list to be replaced by CADR of that element by executing the following sequence of edit commands:

```
    (NTH -1)
    (S FIE 1)
    (I 1 (CADR FIE))
```

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 its "arguments" which are substituted for the parameters of the macro in its definition. A macro with arguments may have a fixed or indefinite number of arguments parallel to the FEXPR and FEXPR* function types. A macro with a fixed number of arguments, such as SW, has, following its name in the macro definition, a parameter list containing the argument names. The arguments in the call to the macro are substituted in the macro definition before executing that sequence of editor commands. A macro with an indefinite number of arguments is indicated by having an atomic parameter list following its name in its definition. In this case, the entire list of arguments is substituted for this atom in the macro definition, and then the sequence of editor commands is executed.

All edit macro definitions are kept on a free variable called EDITMACROS. New definitions supercede old ones, or the value of EDITMACROS itself can be edited to delete, replace, or change macro definitions. Purely local macros, i.e., those that will not be used after the current editing job, can be defined using D, for define, in place of M. These will not be saved on EDITMACROS, but their definitions will, temporarily, supercede any macros of the same name that appear on EDITMACROS.

The macro feature allows the user to easily expand the repertoire of edit commands, and thus "program" the editor. Note that entirely new editing operations can be implemented by defining an appropriate function, and then introducing it to the editor's vocabulary via a macro which calls the function. For example, if no find feature were provided in the editor, the user could define a function FIND, and a macro

```
(M F (X Y)(E (FIND X Y) T))
```

for doing the job.

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, 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 edited expression in the appropriate place.

The editor can be used as a subroutine by giving edite a list of commands to be executed as its second argument. Each command will be executed and, if no errors occur, the edited list returned after their completion. Otherwise, edite goes into normal on-line mode and waits for user commands. Editf, editv, and editp all accept an indefinite number of commands to be interpreted in this fashion: they are each non-spread NLAMBDA type functions, where CAR of their argument is the function/property list/value to be edited, and CDR the (optional) commands to be supplied to EDITE.

Edite uses editl, which takes the edit push-down list, L, as its argument, executes commands until an OK is reached, and then returns the new L as its value.

The user can also write his own editing programs which directly call editcom, the function that does the work in the editor. The workings of this function are explained below under Internal Organization of Editor.

Since all input and output commands in the editor specify the file as teletype, it is possible to edit a function when input and/or output standard files are other than the teletype.

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)))))
*OK
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.

Internal Organization of the Editor

The work of the editor is done by the function editcom, which interprets and executes a single edit command and editcoms which takes a list of commands. editl does the reading from the teletype, and transmits commands to editcom under errorset protection, until an OK command is given. All errors and control-R's are caught by this errorset, and cause editl to print a "?".

Editcom accepts a single command as an argument and uses as free variables L, M, and EM which are normally bound in editl. If the user wishes to define a function which calls editcom or editcoms directly, these variables should be bound in that function. Their interpretation is:

L	is the edit push-down list. It is initialized to list of the expression being edited. (CAR L) is always the current list being examined; (CADR L) is the list you would be examining if you gave a 0 command, etc. Each operation which descends to a lower structure, such as the F command or a number, causes the higher level structure(s) to be attached at the front of L. Operations which ascend, such as 0, or ↑, take things off the front of L.
M	is a list of marks made by the MARK command. It need not be bound if marks are not used. The command MARK simply performs (SETQ M (CONS L M)), the command ← does (SETQ L (CAR M)).

EM contains the list of editmacros being used. It is initialized to the value of EDITMACROS. M commands change EM and EDITMACROS, D commands just change EM. EM is searched whenever an unfamiliar command is encountered.

When _editcom_ is given a command that it does not recognize, it searches EM using _assoc_ on the command, if atomic, or _car_ of the command if a list. If the command has been defined as a macro, _editcom_ performs as described earlier. Otherwise, _editcom_ calls _editdefault_, a function of one argument, which is currently defined as (LAMBDA (C) (ERROR C)). This causes an error which is caught by the _errorset_ in _editl_ However, _editdefault_ can be redefined by the user. In fact, edit default has been redefined to implement some of the operations of the expanded editor described below. For example to treat all unrecognizable forms as expressions to be evaluated ala _break_, one would define _editdefault_ as (LAMBDA (C) (PRINT (EVAL C))). If any error occurred in the evaluation, it would still be caught by the _errorset_ in _editl_. Another possibility might be to have _editdefault_ search the property list of the indicated command to look for a macro definition, or treat the command as a function call with L as its argument, etc.

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
P	Prints current level list to depth 3
PP	Prettyprints current level list
↑	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
F	equivalent to (F X N) where X is the next expression read, e.g. *F COND
E	equivalent to (E X) where X is next expression read, e.g. *E (EDITV FOO)
OK	Exit from editor

Other atoms are given to editdefault.

Lists

(n e_1 e_2,..., e_k) $n>0$ $k\geq 0$	Replace element n by the k elements e_1,..., e_k. Deletes the nth element if k=0
(-n e_1 e_2,...,e_k) $n>0$ $k\geq 1$	Inserts e_1,..., e_k before nth element
(N e_1,..., e_k)	Adds e_1... e_k at end of current level list
(S name) and (S name ∅)	Sets name to current level list
(S name n)	Sets name to nth element, if n>0, Sets name to nth element from end of list, n<0. Gives error if no such element.
(R old new)	Replaces all occurrences of the old item by new in current level list
(P n m) $n\geq 0$	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)	Finds next occurrence of e excluding first element in current list
(F e n) $n\geq 1$	Finds nth occurrence of e any level

(NTH n) $n \geqslant 1$	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 $\underline{e}_1 \ldots \underline{e}_k$ 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
(E e T)	Evaluates e but does not print.
(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 mth subelement of element n. 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 _n_
(BI n m)	Inserts both left and right parens, making a sublist at position _n_ containing elements _n_ to _m_ inclusive.

All of the above six commands, LO, LI, RO, RI, BO, and BI, accept positive or negative numbers as arguments. Negative numbers are positions relative to the end of the list.

(M name c_1 c_2 ... c_n)	Defines _name_, as an Edit Macro equivalent to the sequence of commands $\underline{c}_1$, $\underline{c}_2$, ... $\underline{c}_n$, if _name_ is atomic, and $\underline{c}_2$, ... $\underline{c}_n$ with substitution of arguments for parameters when car[name] appears as _car_ of a non-atomic command, and _name_ is listed in definition.
(D name c_1 c_2 ... c_n)	Same as _M_ but effect is temporary - confined to this call to _edite_.

All other lists are given to _editdefault_.

The Expanded Editor

Supereditflg

All of the commands described below are available to the user whether or not he sets SUPEREDITFLG to T. With SUPEREDITFLG = T, any unrecognized commands will be interpreted as (F command N), (*) e.g., (CAR X) is equivalent to (F (CAR X) N). With SUPEREDITFLG = NIL, these will, of course, cause an error and the editor will print a ?. SUPEREDITFLG can be set to T by using the edit macro bell (control G), which flips the state of SUPEREDITFLG and prints its new value, or by setting it yourself.

New F Command

If the user wishes to operate with SUPEREDITFLG = NIL, or for those cases where he wants to locate an expression which would normally be recognized as a command, and therefore not be searched for, e.g., P, PP, or a number, the following abbreviated form of the F command is available:

F expression

will cause expression to be found a la SUPEREDITFLG. Note that in this form, the user is giving two "commands" to the editor to express a single find operation. The effect is the same as though SUPEREDITFLG were set to T only for the next input after the F, except that no attempt will be made to treat this input as an edit command so that F P, or F 6 will work.

(*) (F command) is done first (using MEMB), so that if LOOP is a PROG label, and that PROG is the current level list, LOOP will find the label before looking for any nested GO'S.

In the discussion that follows, the examples in the text assumes SUPEREDITFLG is set to T. The identical operation can be performed with SUPEREDITFLG = NIL if the user substitutes F expression for expression wherever a find command is intended.

The Location Routine

All of the commands in the expanded editor use a single routine for locating the place at which the operation is to begin. In this discussion, the symbol @ will be used to mean a location specification. @ has no meaning to the editor, it is purely a notational device. The following options may appear at @.

1.	@ NIL	Effectively a NOP.
2.	@ atomic	The single command @ is executed, e.g., @ = 3 means operation is to begin at 3rd element of current list. @ = COND means operation is to begin at the next COND, i.e., the command COND is executed, and the next occurrence of COND is found.
3.	@ a list	Each of the commands in @ is executed and the operation begins after the last one is successfully complete, e.g., (COND 3) locates the second clause in the next COND (the COND itself counts as 1). Note, (CAR X) will first find CAR and then find X. It is equivalent to ((F CAR N) (F X N)), or to (F CAR F X). To find (CAR X) itself, use

((CAR X)) which is equivalent to
((F (CAR X) N)), or (F (CAR X)).

If the execution of the commands in @ is not successful, i.e., an error occurs, the location tries again from the point that the error occurs until it is successful or until no progress is being made. Thus, if the first COND beyond the current point contained only one clause, @ = (COND 3) would then look for the next COND after that, etc. At the point that there were no more CONDs remaining in the list being edited, the locator routine would give up. If this occurs, the status of the edit reverts to its state when the locater was entered.

The locater routine can be called by the user directly via the macro LC. To locate @, type (LC . @), e.g., to locate (COND 3) type (LC COND 3). To locate COND, type (LC COND), i.e., @ = (COND) which is equivalent to @ = COND, since both consist of the single command COND.

UP

Another command used by all of the commands described below is UP. The effect of UP is the following:

1. If the result of typing P is an element in your list structure, then after UP, that element will be the first element in your current list.

2. If the result of typing P is a tail in your list structure, then UP has no effect.

Examples:

Your current level list structure is (COND ((NULL X) (RETURN Y))).

```
1.  * 1 P UP P
    COND
    (COND ((NULL X) (RETURN Y)))

2.  *-1 P UP P
    ((NULL X) (RETURN Y))
    (((NULL X) (RETURN Y)))

3.  *NULL P UP P
    (NULL X)
    ((NULL X) (RETURN Y))

4.  * X P UP P
    (X)
    (X)
```

```
5.  *(NTH 2) P UP P
    (((NULL X) (RETURN Y)))
    (((NULL X) (RETURN Y)))
```

This explanation covers the HOW and WHAT of UP, the WHY will become clear in the explanation of the commands given below.

Insertion, Replacement, and Deletion Commands

The basic editor provides commands for inserting elements before a certain position in the current level list, and for replacing or deleting specified positions in the current level list. However, since the operation is tied up with the location, it is impossible for the user to give single commands for deleting the last element in the list, inserting a certain structure before the second element from the end of the current list, etc. The following three commands are more general than the basic editor's commands, and do provide such a capability. In the description, @ indicates a location operation, and expr a sequence of expressions (possibly null).

1.	(B @ expr)	locates @,does an UP, and inserts expression Before current point, i.e., effectively does (-1 expr).
2.	(A @ expr)	locates @, does an UP, and inserts expression after this point, i.e., does either a (-2 expr) or (N expr) whichever is appropriate.
3,	(: @ expr)	locates @, does an UP, and replaces first element with expr. If expr is null, it deletes the first element.

All three commands leave edit position as of locating @ but do not change marks.

```
Examples:  Current list is (COND ((NULL X) (RETURN Y)))

     1.  *(A NULL (PRINT Y))
         P
         ((NULL X) (PRINT Y) (RETURN Y))

     2.  *(: X (CDR X))
         P
         (NULL (CDR X))

     Current list is (COND ((NULL X) (RETURN Y))
                     ((NULL (SETQ Z (CDR Z)))  (GO LP))
                     (T (ERROR))

     3.  *(: T) Ø P
         (COND ((NULL X) (RETURN Y)) ((NULL &) (GO LP)))

     4.  * (B GO (PRINT Y) (PRINT X)) P
         ((PRINT Y) (PRINT X) (GO LP))

     5.  *(: (NULL NULL) (EQ X Z)) P
         ((EQ X Z) (GO LP))

     6.  *(: (3 1) (EQ X Z)) P
         ((EQ X Z) (GO LP))

     7.  *(B -1 ((EQ X Z) (GO LP1)) P
         (((EQ X Z) (GO LP1)) (T (ERROR)))
```

An exception to the above procedure occurs when the expression is to be replaced by a function. In this case, UP is not performed. For example:

(: CAR X) will replace (CAR &) by X; (: CAR CDR) will replace just CAR by CDR; (: CAR (CDR X)) will replace (CAR &) by (CAR X).

Note that (: NIL) or just (:) deletes the current level list. (: NIL expr) replaces it with <u>expr</u>. Similarly (A NIL expr) and (B NIL expr) insert <u>expr</u>, respectively, after and before current level list.

Switching and Moving Expressions

Note: The SW and MV commands described below require two location specifications. In both cases, the location of the second position is begun at the same point that the first location started; i.e., the commands save the state of the edit upon entering, and return to that position for the second location. However, a MARK is performed after the first location so that by making <— be the first command for the second location, the user can begin that locating process where the first one left off.

The Switch Command

(SW @1 . @2)	@1 is located, an UP performed, SW1 set to the first element, and the current position saved. Then @2 is located, an UP performed, SW2 set to the first element, which is then replaced by a copy of SW1. A copy of SW2 then replaces the original SW1. SW leaves marks and position unchanged.

Examples:

(SW -1 2)	The last element in the list is switched with the second element.
(SW 2 3 1)	switches second element with first element of third element in current level list.
(SW RETURN GO)	The first (RETURN --) is switched with the first (GO --). Note that they may be on entirely different levels. However, one should not be <u>inside</u> of the other.

<u>The Move Command</u>

(MV @1 C . @2)	@1 located, an UP performed, the variable MV1 set to the first element. @2 is then located and the operation indicated by C is performed. Then the former occurrence of MV1 is deleted. MV leaves marks and position unchanged.

Examples:

(MV RETURN B -1)	will find the first RETURN expression and insert it <u>before</u> the last element in the current level list.
(MV (COND SETQ) : GO)	finds the first SETQ after the first COND, and <u>replaces</u> the first GO by that SETQ.

(MV -3 N ↑ 3) takes the third element from the end of the current list and attaches it at the end of the third expression from the top.

The Extract Command

This command is designed to replace a certain expression by one of its subexpressions.

(XTR @1 . @2) locates @1, does an UP, saves the position, locates @2 (beginning from the point @1 left off), does an UP, sets the variable XTR1 to the first expression, returns to saved position, and replaced first expression by XTR1. XTR leaves marks and position unchanged.

Examples:

(XTR COND SETQ)	replaces first COND by first SETQ,
(XTR COND 2 2)	replaces first COND by second expression in its first clause.

Note: While the XTR command is designed to replace an expression by a subexpression, there is no check made to see that the result of locating @2 is in fact inside of @1.

The Embed Command

The embed command is designed to replace a particular expression by a larger expression containing it.

(MBD @ X)	@ is located, an UP performed, and the first expression replaced by the result of substituting it in X for the variable *. MBD leaves current position as the new super-expression, with an extra MARK which is set to the original position when MBD was entered.
(MBD @ X1 ... Xn)	(X1 must be atomic) Same as (MBD @ (X1 ... Xn *))

Examples:

(MBD NIL (COND ((NULL X) *)))

replaces current-expression by (COND ((NULL X) current-expression)))

(MBD 2 QUOTE)	quotes second expression
(MBD -1 SETQ X)	replaces last expression by (SETQ X last-expression).

<u>Miscellaneous Commands</u>

(* N)	moves editor to current expression plus or minus N, i.e., (* 1) equivalent to the command UP followed by 2. If the current expression is the 4th, (* -1) is equivalent to 0 followed by 3.
BK	for <u>b</u>ac<u>k</u>, same as (* -1).
NX	for <u>n</u>e<u>x</u>t, same as (* 1)
DELETE	same as (:)
?	(P 0 100)

TTY	calls editl and sets L to new value, i.e. accepts commands from user. Useful for functions that call editor as subroutine, e.g. breakin[FOO (AFTER COND SETQ TTY)] allows the user to interact before the break is inserted.
STOP	used in connection with TTY command, same as OK command given to next higher call to editl, i.e. aborts the editing operation of the subroutine that was calling the editor.
(SECOND . X)	locates SECOND X, no change if not found.
(THIRD . X)	as above
(ORR X1 X2 ... Xn)	The sequence of commands X1 is executed. If successful, ORR returns. If not, the state of the edit is restored to its original state and X2 executed, etc. This is a way of executing commands conditionally, e.g., the command SECOND is defined as (ORR ((LC . X)(LC . X)))
(## . commands)	sequence of commands is executed for value, not effect, i.e., (## 2 -1) has as its value the last element in the second element of the current list. ## does not change the state of the edit. ## is also an nlambda-nospread function.
(LCL . commands)	Commands are executed locally, i.e., find commands will not be allowed to search beyond current list.

(IF form)	If the value of form is NIL, an error is generated. Designed for use with ORR and locating routine. For example, (ADD1 (IF (NUMBERP (## 2))) as a location specification will find the first ADD1 followed by a number. IF does not change state of the edit.
(LP . commands)	sequence of commands is executed repeatedly until an error occurs, e.g., (LP PRINT (N T)) will attach a T at the end of all PRINTs. LP will print number of successful iterations.
(LPQ . commands)	same as LP but does not print number of iterations.

Note: the routines that handle A, B, :, and MBD commands make special checks (of a flag) so that the user can do commands like (B PRINT (PRIN1 ZOT))) or (LP (MBD X LIST)) without getting in an infinite loop.

(← X)	does repeated 0 commands until finds a position for which first element is X, e.g., (← COND) takes you up to the COND containing the expression that is the current level list. If not found, no change is made in the state of the edit. Note: it is unnecessary to have SUPEREDITFLG set to T to execute (← COND). Also (← F COND) is the same as (← F).

Miscellaneous Features

(X CONTAINING . @)	Locates X, then locates @ locally, i.e., find commands will not go outside of the expression headed by X, and then backs up to X using the ← command, so that (X CONTAINING Y) will find the inner X in (X ... (X ...Y) ..). Note: X will be located regardless of the setting of SUPEREDITFLG: (X CONTAINING . @) is identical to (LC F X (LCL . @) (← X)). Example: (SECOND (COND CONTAINING (SETQ CONTAINING CDR))
(EVERY . @)	is built into the locate routine. Whenever (CAR @) = EVERY, the locate routine looks back up the push down list and finds the command containing this specification, and instead performs the corresponding LP command, e.g., (MBD (EVERY X) LIST) is equivalent to (LP (MBD X LIST)).
(ALL . @)	same as EVERY.

Sentence Format

In addition to the command followed by arguments format, the user can employ a more flexible, sentence-type format when communicating with the editor. The chief advantages of this format are that the names of the commands and the order of the arguments are somewhat more intuitive, and that there are considerably fewer parentheses required.

The following is a list of sentence-types permissible: (*)

```
(INSERT ... BEFORE ...)
(INSERT ... AFTER ...)
(INSERT ... FOR ...)
(PUT ... BEFORE ...)
(PUT ... AFTER ...)
(REPLACE ... BY ...)
(REPLACE ... WITH ...)
(CHANGE ... TO ...)
(DELETE ...)
(EMBED ... IN ...)
(EMBED ... WITH ...)
(MOVE ... TO AFTER ...)
(MOVE ... TO BEFORE ...)
(SURROUND ... WITH ...)
(SURROUND ... IN ...)
(EXTRACT ... FROM ...)
(SWITCH ... AND ...)
```

Examples:

```
(INSERT (PRINT Y) AFTER -1 NULL)
(REPLACE CDR WITH CAR)
(REPLACE CDR WITH (CAR X))
(EMBED EVERY PRINT IN (COND (FLG *)))
(DELETE (COND CONTAINING RETURN))
(EXTRACT (SECOND SETQ) FROM (COND CONTAINING GO))
```

(*) "..." indicates a <u>segment</u> of a list, i.e., no parentheses are used around "..." even if it consists of several elements, see examples.

Summary of New Commands

Following is a complete list of the commands in the expanded editor. The ordering is that of their position on EDITMACROS.

UP

NX

BK

:

A

B

MBD

SW

XTR

MV

LP

*

LC

ORR

LPQ

?

DELETE

TTY

STOP

← (when used in a list)

SECOND

THIRD

LCL

EDITA

The increasing number of applications of LISP that involve arrays have motivated the implementation of EDITA, an editor for arrays. EDITA can be used on any LISP array, including those containing list structure or unboxed numbers, or both, or on compiled function definitions.

To the user, EDITA looks very much like DDT with some LISP extensions. It is a function of one argument:* the array or function to be edited. This can be specified directly or indirectly, i.e., you can type EDITA(A), or perform (EDITA A) inside of some other form. EDITA performs an EVAL on its first argument if it is not already an array or a function.

Once inside of EDITA, individual "registers" or cells in the array may be examined by typing their address followed by a slash, e.g.,

```
4/          6
```

i.e., (ELT A 4)=6.** An address consists of a number or a LISP form whose value is a number, or a series of numbers or forms which yield numbers. In the latter case, the address is computed as the sum of the forms, e.g.,

```
4 X (MINUS (CAR Y)) /         . . .
(CAR (CHCON (QUOTE A))) /     . . .
```

* An optional second argument can be a list of commands to edita. These are executed exactly as though they had come from the teletype.

** If the register is in the unboxed area of the array, the boxed contents are printed.

The variable "." has the value of the address of the current (last) register examined, and the variable $ has the value of the last register in the array, i.e., (ARRAYSIZE A), e.g.,

```
$ (MINUS .) /          . . .
. ./                   . . .
```

are acceptable. Since EDITA uses its own read program, it is not necessary to surround the period in double quotes. Also, since carriage return has a special meaning, the balancing of parentheses in any LISP expression is indicated by a space, instead of a carriage return as with the LISP reader. This is the explanation of the extra space before the slash in some of the examples above.

A slash is really a command to EDITA to "open" the indicated register. Only one register at a time can be open, and only open registers can be modified. To change the contents of a register, the user first opens it, types a form* and then closes the register with a carriage return, e.g.,

```
4/    6    (FACTORIAL .) ↩
./   24
```

Note: Computations can be executed while a register is opened without changing its contents. The contents of a register are changed only when it is explicitly closed by a carriage return, line feed, or ↑. If the register is in the unboxed region of the array, an unbox will be automatically performed before storing the new value into the array.

* In the boxed region of the array, only non-atomic expressions are evaluated; in the unboxed, all expressions are evaluated.

If a form is typed followed by a carriage return when no register is open, the form is simply evaluated and its value typed, e.g.,

```
4/      4096 ↲
(RADIX 8)↲
(10 T) ↲
./      10000Q
```

Used in this way, EDITA behaves the same as BREAK.

EDITA also recognizes the following commands:

OK	which causes a return from EDITA with the value the array being edited.
linefeed	which closes any open register and opens the <u>next</u> register, i.e., it is the same as carriage return followed by (ADD1 .)
↑	same as carriage return followed by (SUB1 .)

=	(when not preceded by a space) causes EDITA to type value of last expression, e.g.,
	`.=4/   6   (PRIME 6)=11 (PRIME 7)↲`
	`./     13`
	If a register is open, the = command also operates to negate the effect of any previous user input so that if the register is closed following an = command, it will not be changed.

;Q	has value of last expression typed by EDITA, e.g.,
	`4/   6  (FACTORIAL .)=24 (ADD1 ;Q)↲`
	`./   25`

/

(when register is already open), if preceded by user input, EDITA prints the contents of the indicated register, otherwise EDITA prints the contents of the register whose address is the contents of the currently open* register, e.g.,

```
4/    6    /    10   ↵
6/   10    4/    6
```

This command does not affect the currently open register.

Tab (control I)

similar to / except it closes the currently open register (if any) and opens the indicated register, e.g.,

```
4/    6    tab
6/   10    9tab
9/   11
```

If the contents of the currently open register is not a number, but is another array , or the name of a function - tab will call EDITA on it.

* In all cases only low order 14 bits used; Underlined characters were typed by user.

?	negates all user input not yet processed, leaves state of registers unchanged.
AD1,AD2/	where AD1 and AD2 are addresses, causes the contents of registers AD1 through AD2 inclusive to be typed, "." is set to AD2 after completion.
;W expr	searches array and prints all registers whose contents "equal" <u>expr</u>, in the sense of the match used by the editor, e.g., ;W (&) will find and print all registers containing a list consisting of a single element. After search, "." is left set to the last such register.
expr ;W	same as above except that since <u>expr</u> will have been evaluated before the ;W command was read, its value will be used in the search, e.g., (CADR X) ;W

If the search command is prefaced by an address and a comma as in FOO 2, ;W NIL or 25,X ;W the search will begin at the indicated register, otherwise it begins at register ".", the last opened register. If the search is to begin in the unboxed region of the array, the value to be searched for must be a number and is compared with the result of boxing each element in the unboxed region. The variable MASK can be set prior to the search for comparison with just selected bits in the word. The search terminates at the end of the unboxed region of the array.

If the value to be searched for is not numeric, no attempt will be made to search the unboxed region of the array, regardless of the value of "." or the address specified, i.e., the search automatically begins at BOXED if "." is less than BOXED.

:name	defines name as either (1) the contents of register ".", or if no register open (2) the address typed just before the :, or if none was typed, (3) the value of ".". * For example,

```
4/    6    :FOO
:FIE
. 1:FUM
```

defines FOO as 6, FIE as 4 and FUM as 5.

EDITA keeps its "symbol tables" on two free variables, USERSYMS and SYMLST. USERSYMS is a list of elements of the form (name . value) and is used for encoding input, i.e., all variables on USERSYMS are bound to their corresponding values during evaluation of any expression inside EDITA. SYMLST is a list of elements of the form (value . name) that is used for decoding addresses,

* Only low order 14 bits are used.

and in the case of editing compiled functions, decoding instructions. USERSYMS is initially set to NIL, while SYMLST contains certain system parameters such as PPPTR and SPCELL.* Since the : command adds the appropriate information to these two lists, new definitions will remain in effect even if the user exits from EDITA and then reenters it later.

Note that the user can effectively define symbols without using the : command by appropriately binding USERSYMS and/or SYMLST before calling EDITA. Also, he can thus use different symbol tables for different applications.

Some general comments

Although EDITA uses its own read program,** which is not line buffered, it does respond to control-A and control-Q in the same way as the LISP read program does in almost all situations. In those cases where it cannot delete previous characters because they have already been processed, EDITA will ring the teletype bell to signal its frustration. Similarly double quotes can be used to input expressions containing break characters for EDITA such as /, (, ?, etc.

EDITA is buffered against errors and rubouts. Whenever an error occurs or a control-R is typed, EDITA responds with a ? and closes any open registers (without modifying them) and clears any flags that may have been set during the user's last request. It is quite safe to hit control-R at any time during EDITA's operation.

EDITA will not allow the user to reference any registers outside the bounds of his array.

* It is not necessary to place a symbol on USERSYMS that already has a binding, such as PPPTR or SPCELL, since the correct value will be obtained when the form in which it appears is evaluated.

** Actually, EDITA uses the FLIP read program so do not flush FLIPREAD if you plan to use EDITA.

Using EDITA on compiled functions

Since a compiled function is actually an array with the instructions of the function corresponding to the unboxed region of the array and the literals to the boxed region, EDITA could be used on a function definition exactly the same as though it were an array generated via ARRAY. However, certain extensions and modifications to EDITA have been made to facilitate its use with compiled functions with the result that EDITA operates somewhat differently when working with a function than with an array, although the basic idea and philosophy of its use remains the same.

The first difference to be noted is that by convention the first element of an array has address 1, while the first instruction of a function definition has address 0. In other words, the instruction LDA 25 loads the accumulator with what would be register 26 in an array. EDITA takes care of this problem automatically both on input and output by following one address convention for functions and the other for arrays.

The greatest difference - and the most useful feature of EDITA in conjunction with functions - is the decoding of instructions. EDITA decodes the contents of all cells in the unboxed portion of the function definition (in the boxed region, EDITA operates exactly as it does with an array except that the address convention is that of function definitions) and types their mnemonic, address portion, and indirect bit or index register if any. The address portion is further decoded using SYMLST. Here are some examples: LDX PPPTR; STA 25,2; LDA* FOO; CONSCLL. The decoding of the address portion also notes references to literals in the function definition and prints the literal preceded by an "=", instead of its address, e.g., LDA =(-ARGUMENTS ?); XCLL =PRINT. References to small integers are also detected, e.g., SUB =1; SKG =5.

Symbolic input is available, although it is not quite as sophisticated as the output decoding. Op codes are recognized, as are small integers (this is why the = command must be preceded by a space: to distinguish it from a small integer). However, the indirect bit must be indicated by means of the variable J, and the index register by I. For example, the following is permissible input: LDX PPPTR; STA 25 I (which would be typed back as STA 25,2) LDA FOO J, SUB =1. LDA =(-ARGUMENTS ?) is <u>not</u> permissible. To input this the user would have to know the address of the register containing the literal (-ARGUMENTS ?).

When an op code is seen, all subsequent arithmetic is done in the low 14 bits of the word only, so that LDA -1 is equivalent to LDA 37777Q, not to SKD* 37777Q.

There are two other small differences when editing a function. First, read-only out of bounds references <u>are</u> permitted, e.g., 3/ LDX PPPTR / 27. Secondly, the search option will automatically mask out the address or instruction portion of the word if the value to be searched for is just an instruction or an address. For example, LDA =5 ;W will find all occurrences of that instruction, LDA ;W will find all LDA instructions, =5 ;W will find all references to the small integer five. The user can of course still set the variable MASK.

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.

nthchar[atom;n]

Returns the nth character of atom as a single-character atom. Equivalent to (CAR (NTH (UNPACK atom) n)), but is faster and does no CONS'es. See note after loc, p. 10.4.

nchars[atom]

Returns number of characters in atom. Thus the last character in an atom is given by

```
     nthchar[atom;nchars[atom]]
```

chcon[x;j]	Returns a list of numbers representing characters in print name of x which must be an atom. j = NIL prinl representation = T prin2 representation
character[n]	n is a character code. Value is atom having single character as its P-name, e.g., character[8]="(".
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[flg]	Initiates a garbage collection. If flg is T, all spaces are collected: list words, atoms, large numbers, floating point numbers, arrays and binary programs. If flg is NIL, array space (identical to binary program space) is not collected, but all others are. Value of reclaim is number of list words available and will be $\geq$ the setting of minfs unless the total list space has been exhausted. See p. 3.8-3.12 for more detailed discussion of garbage collection.

atomgc[flg]

initiates a reclaim which also collects any unused atoms that were previously in the shared system but have been released by flushing some portion of it. See p. 3.8-3.12 for discussion of garbage collection and p. 22.8 for flushcode. Argument flg is the same as in reclaim.

gctrp[n]

garbage collection trap. Causes a (simulated) control-H interrupt when number of free words left equals n, i.e. when a garbage collection would occur in n more conses.

At this point, the user can turn off the display, list a file, logout of LISP, etc. arm described on p. 22.2 shows how the user can automate this procedure.

Value of gctrp is last setting. If n=NIL, value of gctrp is number of words left, i.e. (GCTRP (PLUS (GCTRP) -10)) will cause a trap after 10 more conses.

minfs[n]	Sets the minimum amount of free storage which will be maintained by the garbage collector. If, after any garbage collection, fewer than n free words are present, sufficient storage will be added in 128 word chunks to raise the level to n. The user may also change the setting of minfs at any time, even during a garbage collection, by typing control-F followed by a number (which becomes the new setting) followed by a period.
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. Closes all open files.
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.
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.

Note: for alphabetizing purposes, it is useful to note that the atoms consisting of a single character are stored in ASCII code order, i.e. loc[A] < loc[B] < loc[C] etc.

vag[x] — The inverse of loc. x must be a number; value is the unbox of x. An unboxed number n which doesn't correspond to the address of a list structure or an atom is printed #n, n is given in octal, e.g. array pointers are printed this way.

Note: unboxed numbers should not be passed around as ordinary values because they can cause trouble for the garbage collector. Everything in LISP is essentially an unboxed number, i.e. an address. However, certain unboxed numbers are recognized as being of certain data types, e.g. integers, atoms, list structure, etc. If you creat an unboxed number that happens to correspond to an address in list structure, the garbage collector will not be able to distinguish this from a bona fide list structure. For example, suppose the value of x were 150000, and you created (VAG X), and this just happened to point into the free storage list! The next garbage collection would be disastrous.

allocate[n] — Allocates an n word block in array (binary program) space. Returns a pointer to the address of the first word allocated. If sufficient space is not available, a garbage collection of array space (RECLAIM T) is initiated. If this is unsuccessful in obtaining sufficient space, an error is generated.

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.); number of drum reads;

number of drum writes; number of drum reads for binary function loading; number of function calls from binary code. Names and locations of cells printed are bound at top level to STATCELLS.

clearstat[]	Sets to Ø all statistics cells in the list bound to STATCELLS.
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

Space for 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 $p \leq n$ are cells which will contain unboxed integers, and are initialized to Ø. The last $n-p \geq 0$ 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 the location of the array in virtual memory, and is called an array pointer. Array-pointers print as #n, where n is the octal representation of the pointer. Note than, #n will be read as an atom, and not an array pointer.

elt[a;m]

Has as value the m^{th} element of the array pointed to by a. For out of bound calls, if m<1 or m>n, where n is the length of the array a, elt gives element 1 if m<1, or element n if m>n.

seta[a;m;v]

Sets the value of the m^{th} element of a to v. On out-of-bounds reference no store is made. The value of this function is always 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 a if a is an array pointer.

arrayp[x]

Returns T if x is a pointer into the active array area, otherwise NIL. No check is made to ensure that x actually addresses the header of a legitimate array.

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]	Identical to quote for interpreted code. When compiled, function[x] will cause x to also compile, quote[x] will not.
map[mapx;mapfn1;mapfn2]	If mapfn2 is NIL (i.e. not provided) this function applies the function mapfn1 to successive tails of the list mapx. That is, first it computes mapfn1[mapx], and then mapfn1[cdr[mapx]], etc. until mapx is NIL (mapx is reset at each iteration so that its value is always the current tail); however, if mapfn2 is provided, mapfn2[mapx] is used instead of cdr[mapx] for the next call for mapfn1. Thus if mapfn2 were cddr, alternate elements of the list would be skipped. If mapfn2 is a

	conditional expression, then the next element to be looked at can be contingent on a computation.
mapc[mapx;mapfn1;mapfn2]	Identical to *map*, except that mapfn1[car[mapx]] is computed each time. If *mapfn2* is NIL, *mapfn1* is applied to each element of the list *x* in turn.
mapcar[mapx;mapfn1;mapfn2]	If *mapfn2* is NIL, this function applies the function *mapfn1* to each of the elements of the list *mapx*. 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 *mapfn1* to the corresponding element of the old list. If *mapfn2* is provided, mapfn2[mapx] is used instead of cdr[x] for each succeeding computation with *mapfn1*.
maplist[mapx;mapfn1;mapfn2]	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[mapx;mapfn1;mapfn2]	Identical to *mapcar* except that it does an *nconc* instead of a *cons*. This makes it useful for constructing a new list from an old one where a variable number of elements is to be inserted at each iteration.

mapcon[mapx;mapfn1;mapfn2]	Identical to *maplist* except that it does an *nconc* instead of a *cons*.
map2c[mapx;mapy;mapfn1;mapfn2]	Identical to *mapc* except *mapfn1* is a function of two arguments, and mapfn1[car[mapx];car[mapy]] is computed each time. Terminates if *either* *mapx* or *mapy* become NIL.
map2car[mapx;mapy;mapfn1;mapfn2]	Identical to *mapcar* except *mapfn1* is a function of two arguments and mapfn1[car[mapx];car[mapy]] is used to assemble the new list. Terminates if either *mapx* or *mapy* become NIL.
mapa[mapary;mapfn1;mapfn2;mapn]	Cycles through *mapary*, an array, applying at each iteration the function *mapfn1*, a function of two arguments, to *mapary* and *n* the index of iteration. *n* is initially set to *1*, and reset to mapfn2[n], if mapfn2 is given, otherwise add1[n]. Process continues until *n* exceeds *mapn*, if given, or else arraysize[mapary]. The value of *mapa* is *mapary*. Example: the following function will copy an array:

```
(LAMBDA (A) (MAPA (ARRAY (ARRAYSIZE A))
        (FUNCTION (LAMBDA (A1 N)
         (SETA A1 N (ELT A N))))))
```

maprint[lst;fl;l;r;s;pfn;c]	is a general printing function. It cycles through lst applying pfn (or prinl if pfn not given) to each element of the lst. Between each application it performs prinl of s, or " " if not given. If l is given, it is printed (prinl) initially; if r is given, it is printed (prinl) at the end. fl is the file used for all printing, c a special argument used by prettyflip. For example, maprint[x;NIL;"(",")"] is equivalent to print. To print a list on the tty with commas between each element and a final "." one could use maprint[x;T;NIL;".";","].
every[everyx,everyf]	is true if everyf applied to each element in everyx is not NIL, e.g., every[(X Y Z); ATOM]=T.
some[somex;somef]	is NIL if somef applied to every element in somex is NIL, otherwise it is the list beginning with the first element that satisfies somef, e.g., somef[x,(LAMBDA (X) (EQUAL X Y))] is equivalent to member[x,y].
mapdl	see p. 12.9
searchpdl	see p. 12.10

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 two pushdown lists used in BBN 94Ø 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 called the control pushdown list, and contains function returns and other control information.

However, it is more convenient for the user to consider the push-down list as a single "list" containing the names of functions that have been entered but not yet exited, and the names and values of the corresponding variables. The multiplicity of push-down lists in the actual implementation is for efficienty of operation only.

The Push-Down List and the Interpreter

In addition to the names and values of arguments for functions, information regarding partially-evaluated expressions is kept on the push-down list. For example, consider the function FACT:

```
←PRETTYPRINT((FACT]

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (TIMES N (FACT (SUB1 N)))))))
NIL
←
```

As soon as FACT is entered, the interpreter begins evaluating the implicit PROGN following the LAMBDA (see p. 4.3-4.4). The first function entered in this process is COND. COND begins to process its list of clauses. After calling ZEROP and getting a NIL value, COND proceeds to the next clause and evaluates T. Since T is not NIL, the evaluation of the implicit PROGN that is the consequent of the T clause is begun (see p. 4.3). This requires calling the function TIMES. However before TIMES can be called, its arguments must be evaluated. The first argument is immediately evaluated, but the second involves a recursive call to FACT, and another implicit PROGN, etc.

Note that at each stage of this process, some portion of an expression or argument list has been evaluated, and another is awaiting evaluation. This information is recorded on the pushdown list as follows:

1. Whenever a FSUBR* function is entered, i.e. COND, PROG, OR, AND, SETQ, PROGN, or implicit PROGN (*), the variable NLAMBDA is bound on the push-down list to the rest of the expression following the FSUBR*. (Since implicit PROGNs do not appear on the push-down list as specific function calls, this binding will appear following the variables of the function previously called.) As the FSUBR* processes its "argument list," the binding of NLAMBDA is updated so that car of its value is always the expression currently being worked on. This is more than just a diagnostic device: the slot on the push-down list corresponding to the binding of NLAMBDA is actually where the FSUBR* keeps the expression it is processing. If a function subsequently entered modifies this binding, the FSUBR* would continue with the evaluation of the modified expression when control returned to it.

2. Whenever a form is encountered that is headed by a function of type EXPR, EXPR*, CEXPR, CEXPR*, SUBR, or SUBR*, i.e. those requiring evaluation of arguments, the function name is entered on the push-down list and the variable LAMBDA is bound to its argument list. The arguments are then evaluated in turn from left to right. As each argument is evaluated, it is bound to a variable name selected from the atoms a,b,c, ... z (**),i.e. the first is bound to a, the second to b, and so forth, and LAMBDA is bound to <u>cdr</u> of its previous value. Thus <u>car</u> of the value of LAMBDA is always the

---

(*) QUOTE, GO, and FUNCTION are also FSUBR* type functions, but since no evaluation of forms takes place inside of them, they do not modify the push-down list.

(**) Note that these are lower-case characters not present on model teletype model 33 keyboards. They print out as %A, %B etc. as described on page 14.4. To evaluate %A, perform (EVAL (CHARACTER 65)), %B, (EVAL (CHARACTER 66)), etc.

argument currently being evaluated.

When all of the arguments to the function are evaluated, the function is called, and the values of the arguments bound to the names of the arguments of the function. The bindings for LAMBDA, and a,b, etc., disappear. Thus a function has actually been entered if and only if it does not have a binding for the variable LAMBDA.

The following untrace illustrates the above discussion:

```
←PRETTYPRINT((FACT))

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N)
        L)
      (T (TIMES N (FACT (SUB1 N))))))))
NIL
←FACT(2)

ERROR
(L IS UNBOUND ATOM)
UNTRACE:
   NLAMBDA (L)
COND
   NLAMBDA ((COND ((ZEROP N) L) (T (TIMES N (FACT (SUB1 N))))))
   N 0
FACT
   LAMBDA ((FACT (SUB1 N)))
   %A 1
TIMES
   NLAMBDA ((TIMES N (FACT (SUB1 N))))
COND
   NLAMBDA ((COND ((ZEROP N) L) (T (TIMES N (FACT (SUB1 N))))))
   N 1
FACT
   LAMBDA ((FACT (SUB1 N)))
  %A 2
TIMES
   NLAMBDA ((TIMES N (FACT (SUB1 N))))
COND
   NLAMBDA ((COND ((ZEROP N) L) (T (TIMES N (FACT (SUB1 N))))))
   N 2
FACT

←
```

The Push-Down List and Compiled Functions

In addition to the function names and values of arguments to compiled functions, the push-down list contains the names and values for all _free_ variables used in compiled functions, as well as any variables that are _locally_ bound by PROGs or open-LAMBDA expressions.* The free variables follow the arguments to the function on the push-down list. Locally bound variables are stored on the push-down list following the _next_ function call. See p. 16.26-27 for more detail.

(*) In interpreted functions, the PROG or open-LAMBDA would be called as a regular function and the bindings of their variables would automatically appear as arguments on the push-down list.

Pushdown List Functions

The following functions allow one to interrogate the pushdown list(s) from inside another function. The convention used by these functions regarding push-down list positions is that the position number, n, 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. For example, on the previous page, the position of the last call to fact is either -2 or 6.

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, e.g. nthfnback[nthfn[fn,1]] = 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, including LAMBDA, NLAMBDA, and %A-type bindings if any, as well as free and locally bound variables for compiled functions as described earlier.
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, i.e. jumps back up the pushdown list through all intervening function calls.
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.
baktrace[n]	Like backtrace except it skips over calls to breakl, faultl, faulteval, interrupt, error, etc. Used by the BT macro in break.
rtfrm[rtfn;rtform;rtn]	is an NLAMBDA that provides a convenient way of calling retfrom, e.g. (RTFRM FOO X 2) is equivalent to (RETFROM (NTHFN (QUOTE FOO) 2) X). It does a retfrom from nthfn[rtfn;rtn], (if rtn is not given, 1 is used) with the value of rtform.

mapdl[mpdlfn;mpdln]

cycles back up the push-down list, starting at position mpdln, (if mpdln=NIL, it is set to nthfn[mapdl;1]) applying mpdlfn to the function entered at that push-down position, i.e., to nthfnback[mpdln] and then decrementing mpdln by 1 until it reaches Ø. For example:

```
mapdl[(LAMBDA (X) (COND
  ((EQ (FNTYP X) (QUOTE EXPR))
      (PRINT X)))))
```

will print all EXPRs on the push down list.

Note: Negative value for mpdln may be used.

Value of mapdl is NIL.

searchpdl[srchfn;srchm]

searches the push-down list until it finds a position for which srchfn, applied to the function called at that position, is not NIL. For example,

```
(SEARCHPDL (FUNCTION (LAMBDA (X)
    (NOT (ATOM (GETD X))))))
```

will find the last EXPR called.

If srchm is not given, the search begins with the function called just before searchpdl. If srchm is supplied and is not a number, the search begins as of (NTHFN SRCHM 1), otherwise search begins with srchm. Note that srchm is bound to the push down list position at all times, so that srchfn can use it for calling evalv, setv, or retfrom. The value of searchpdl is (function . position).

Push-Down Handles

This section describes how to write functions which directly manipulate the push-down lists, e.g. an nthfn that starts at a specified point, or one that searches forward instead of backward, a form of variables that checks to see if a particular variable is bound without creating a list of all variables, etc.

There are four free variables which provide a direct handle on the two push lists:

CP	Control PDL Pointer
ICP	Initial CP
PP	Parameter PDL Pointer
IPP	Initial PP

The value of each of these variables (also stored under the property COREVAL on their property list) is a number which is the location of a cell in core which contains a virtual memory address. These addresses define the bounds of their respective push-lists: IPP and ICP are the initial values, PP and CP the current values (actually they point to the first cell not used). Thus if both stacks are empty, i.e. at the top level

```
          (OPENR CP)  =  (OPENR ICP)
    and   (OPENR PP)  =  (OPENR IPP)
```

For each function called, the contents of CP are increased by 4, corresponding to the four cells required on the control push-down list for information about this function call. The first two of these cells contain the functions return and its virtual address, so should not be of interest to the user. However, the third cell, i.e. (OPENR ICP)+2,+6,+10... contains the functions name. Thus, the following definition of nthfn is equivalent to the machine coded one currently in our system.

```
(NTHFN
  (LAMBDA (FN N)
    (PROG (X Y)
          (SETQ X (PLUS (OPENR CP)
              -4))
          (SETQ Y (OPENR ICP))
      LP  (COND
            ((EQ (CAR (VAG (PLUS X 2)))
                FN)
              (COND
                ((ZEROP (SETQ N (SUB1 N)))
                  (RETURN (QUOTIENT (DIFFERENCE X Y)
                      4))))))
          (COND
            ((NEQ (SETQ X (PLUS X -4))
                Y)
              (GO LP)))
          (RETURN NIL)
      )))
```

The fourth cell on the control push-down list is a pointer to the first cell on the parameter push-down list used by this function. For each variable bound locally by a function, the contents of PP are increased by two, i.e. each variable uses two cells on the parameter push-down list. The first cell, i.e. (OPENR IPP)+0,+2,+4..., contains the value of the variable; the second contains its name. Thus, variables can be recognized on the parameter push-down list by the appearance of an atom, the variable's name, in an odd cell*. Thus, the following definition can be used for variables:

* The parameter push-down list is used for temporary storage by the interpreter, and also contains information about bindings of free variables in compiled functions. Thus, from the user's standpoint, it may contain some "garbage".

```
(VARIABLES
  (LAMBDA (N)
    (PROG (LST FROM TO Z)
          (SETQ FROM (OPENR (SETQ N (PLUS (OPENR ICP)
                  (TIMES N 4)
                  3))))
          (SETQ TO (OPENR (PLUS N 4)))
       LP (COND
            ((EQ FROM TO)
              (RETURN LST))
            ((ATOM (SETQ Z (CAR (VAG (ADD1 FROM)))))
              (SETQ LST (NCONC LST (LIST Z)))))
          (SETQ FROM (PLUS FROM 2))
          (GO LP)
       )))
```

The following function is presented as an example of a "new" push-down list function. Its value is T if the variable VAR is bound by the function at position N on the push-down list, otherwise NIL. In other words, it is equivalent to (MEMBER VAR (VARIABLES N)).

```
(VARIABLE?
  (LAMBDA (N VAR)
    (PROG (FROM TO)
          (SETQ FROM (OPENR (SETQ N (PLUS (OPENR ICP)
                  (TIMES N 4)
                  3))))
          (SETQ TO (OPENR (PLUS N 4)))
       LP (COND
            ((EQ FROM TO)
              (RETURN NIL))
            ((EQ (CAR (VAG (ADD1 FROM)))
                VAR)
              (RETURN T)))
          (SETQ FROM (PLUS FROM 2))
          (GO LP)
       )))
```

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]	Returns an 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]	x 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 x_1,...,x_n

lsh[n;s]	Performs an arithmetic left shift of $s \geq o$ on n. Equivalent to $n * 2^{s}$.
rsh[n;s]	Performs an arithmetic shift of $s \geq o$ on n. Equivalent to $n * 2^{-s}$.
lrsh[x;n]	Performs a logical right shift of x by n>Ø places.
abs[x]	Returns absolute value of x.
rand[m,n]	Returns a random integer r, $m \geq r \geq n$. Uniformly distributed in the range $m \leq r \leq n$.

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
fixp[x]	Returns T if x is an integer, NIL otherwise.
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 LISP normally operates with fltfmt[0]. Another useful format

is 3DDWW000Q, where DD is the number of digits following the decimal point, and WW the total field width.

Thus with fltfmt[30205000Q],
.62400000E+2 will be typed as 62.40
.38000000E-1 will be typed as 0.04

Numbers outside this range will be typed with E notation.

See time-sharing manual for complete description of floating formats.

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
expt[m,n]	Returns as floating point number the value of m to nth power. m and n may be positive, negative, fixed or floating point numbers except that if m is negative and n fractional an error occurs.
log[x]	value is natural logarithm of x as a floating point number. x can be integer or floating point.
antilog[x]	value is floating point number whose logarithm is x. x can be integer or floating point.

sine[theta]	Truncates theta to nearest 5 degrees and returns sine of theta as floating point number. Uses table look-up.
cosine[theta]	sine[theta+90]

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/OUTPUT FUNCTIONS

Opening and Closing Files

All input (output) functions in BBN LISP can specify their source (destination) file with an optional extra argument which is the name of the file. This file must be opened as specified below. If the extra argument is not given (has value NIL), the file specified as "primary" for input (output) is used. Normally these are both T for teletype input and output. However, the primary input (output) file may be changed by

input[name]	Sets name to the primary input file. Its value is the name of the old primary input file. If name=NIL, value is current primary input file which is not changed.
output[name]	Same as input except operates on primary output file.

Any file which is made primary must have been previously opened for input (output).

The user may have a maximum of 3 files open simultaneously, in addition to the teletype input and output files, and the output file NOTHING.

The three basic file manipulation operations are:

infile[name]	Opens for input the file named name and sets it as the primary input file. The value of infile is the name of the previous primary input 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 previous primary output file. It sets the standard (primary) 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.

Input/Output Transmission

Most of the functions described below have an (optional) argument file which specifies the name of the file on which the operation is to take place. If that argument is NIL, the primary file will be used.*

Note: in all 940 files, end-of-line is indicated by the characters carriage-return and line-feed in that order. Unless otherwise stated, carriage-return appearing in the description of an output function means carriage-return and line feed.

On input from files, LISP will skip all line-feeds which immediately follow carriage-returns. On input from teletype, LISP will echo a line-feed whenever a carriage-return is typed.

The following functions perform output:

prinl[x,file]	prints x on file
prin2[x,file]	prints x with double quote marks inserted where required for it to read back in properly

Both prinl and prin2 print lists as well as atoms; neither print a carriage return upon termination; both have value x. prinl is usually used only for explicitly printing formatting characters, e.g. (PRIN1 (QUOTE ".")) might be used to print a period at the end of a sentence. prin2 is used for printing S-expressions which can then be read back into LISP with read i.e. atoms containing the regular LISP formatting characters in their print names will be printed with surrounding double-quote marks. If radix=8, prin2 puts a Q after numbers but prinl does not.

* file is used for tutorial purposes only. The arguments to all subrs, which includes prinl, prin2 etc., are u, v, and w, as described in arglist, p. 8.5.

prin3 [x,file]	Prints x using double quotes for separation and break characters specified by setbrk and setsepr; p. 14.7.
print[x,file]	Prints the S-expression x using prin2; followed by a carriage-return linefeed. Its value is x.

If any print function is given an atom containing a lower case character, c, and the output file is the teletype character will print as %C.* Similarly, control characters print as &C on the teletype. For all files, unboxed numbers print as #N, where N is the octal representation. (See p. 10.5).

spaces [n, file]	Produces n spaces; its value is NIL
terpri[file]	Produces a carriage return; its value is NIL
xcr[]	Produces a carriage return without a line feed; for teletype only; its value is NIL. Note: this carriage return is not detected by position.

The print functions print, prinl, prin2, and prin3 are all affected by a level parameter set by

printlevel[n]	Sets print level to n, value is old setting. Initial value is 100000.

* The line printer will print lower-case characters as lower-case characters. If the file is printed on the teletype, e.g. by the copy command, lower-case are printed as upper-case.

The variable n 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

```
     &
```

If n is negative, action is similar except that a carriage return is inserted between all occurrences of right paren followed by left paren. 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 or exclamation point. 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. If a period was used to terminate the number, level will be returned to its previous setting after this printout. If an exclamation point was used, the printlevel is changed permanently. This setting effects both print and printx which is the name of the printing function called by evalquote.

Note: printlevel only affects teletype output. Output to all other files acts as though level is infinite.

Input Functions

read[file;flg]	Reads one S-expression from *file*. Atoms are delimited by parentheses, brackets, spaces, carriage returns. To input an atom which contains one of these syntactic delimiters enclose the atom in double quotes; e.g., "A,B,( ] C." A double quote immediately following the first double quote will be considered part of the print name, not as a terminator. To have a double quote internal to a quoted print name, use three double quotes; e.g. "("""")A" will *prinl* as (")A, and "A"""" as A". If *flg* = T, then *read* will not count parens.
readx[file;flg]	Read program used by evalquote; same as *read*.
rdflx[x]	If *x* is NIL this function will try to read one S-expression with read[T]; i.e. from teletype. 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 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[file;flg]	Reads in one atom from file. Separation of atoms is defined by tables set by *setsepr* and *setbrk*, if *flg* = NIL. If *flg* = T, *ratom* uses the LISP tables.
ratoms[a,file]	Calls *ratom* repeatedly until atom *a* is read. Returns a list of atoms read not including *a*.
setsepr[x]	*Sets* *sepa*r*ator characters
setbrk[x]	*Sets* *br*ea*k* characters

Both *setsepr* and *setbrk* are of type EXPR*. Arguments are octal numbers which are ASCII codes for teletype characters, e.g., 155q for carriage return.

Characters specified by *setbrk* will delimit atoms, and be returned as separate atoms themselves by *ratom*. Characters specified by *setsepr* will be ignored and serve only to separate atoms. *Read* does not use *ratom*, but if it did, space (0q), and carriage return (155q) would be separator characters; and left paren (10q), right paren (11q), left bracket (73q), right bracket (75q), double quote (2q), and period (16q) would be break characters.
Thus

```
setsepr[0q 155q]
setbrk[10q 11q 73q 75q 2q 16q]
```

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

setseprc[x]	Same as setsepr except that the arguments should be single character atoms.
setbrkc[x]	Same as setbrk except that the arguments should be single character atoms. Use setlsepr, setlbrk.
setlsepr[u] setlbrk[u]	If u = NIL causes all separator (break) characters to be cleared. If u is a single character atom or a numeric code for a character, this character is added to the set of separator (break) characters. Returns T if this character was previously a separator (break) character, NIL otherwise. Error 10, "illegal argument" occurs if number is out of range, or atom is not character atom.
ratest[x]	Performs three 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.

If x = 1 then it returns:

T if last atom read contained double quotes (on READ or (RATOM x T)) NIL otherwise.

readc[file;flg]	Reads the next character. Allows paren counting and line buffering if flg = T.
unreadc[n,file]	If n is the code for a single character or a single character atom, it will be placed at the beginning of the input buffer and thus taken as the next character read. May not be done two or more times without intervening read.

Input/Output Control Functions

These functions perform a variety of operations on the state of files.

clearbuf []	Clears the input buffer of TTY.
radix[u;v]	Sets output radix to u and sign indicator to v. If u is T, negative numbers will print as sign and 23 bit value (normal). If u is NIL, all numbers print as 24 bit unsigned integers. Returns previous setting.
control[u;v]	If v is not NIL, the system echo table is set to v, which is the 2nd argument to BRS 12(A REGISTER). The value of u sets modes for reading with ratom as follows:
u = T	Eliminates LISP'S normal line buffering, automatic detection of control-A and control-Q as line-editing characters on the TTY and paren counting.
u = NIL	Restores line buffering (normal).
u = -1	Restores line buffering and causes characters in current line not yet read to be reprocessed by paren counter and line buffer handler.
u = 0	Eliminates the echo of the character being deleted by control-A.
u = 1	Restores the echo (normal). Value is old setting.

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 output buffer (either the TSS input buffer or LISP'S line buffer) and NIL otherwise.

Special Functions

sysout[name]	Saves the user's private memory on the file *name*. The value of *sysout* is the number of words required. Note: whenever the LISP system is reassembled, old *sysout* files are no longer readable.
sysin[name]	Restores the state of LISP from a *sysout* file. Value of *sysin* is the number of private pages (256 words) read. If *sysin* returns NIL or INCOMPATIBLE, it was unable to read the file *name*.
sysget[name]	Initializes LISP and then does sysin[name]. If name = T, just initializes LISP.

rbin[file]	Reads one 940 24 bit <u>word</u> from <u>file</u>, the specified file. This function returns the word as a number.
wbin(w;file]	Writes one word, <u>w</u>, on file specified by <u>file</u>. <u>W</u> must be a number.
ginfn[name] goutfn[name]	Obtains <u>system</u>'s file number for previously opened input (output) file. Useful for performing direct I/O from hand-coded function. <u>Cannot</u> be used as file argument for I/O functions. See <u>ginfx</u>, <u>goutfx</u>.
ginfx[name] goutfx[name]	These functions obtain the <u>LISP</u> file index for the previously opened input (output) file. Can be used as argument to I/O SUBR in place of file name. It is somewhat faster especially in case of repeated calls e.g. READC, RATOM, PRIN1, etc.
filetype[name]	Obtains number indicating type for previously opened input or output file.
copyfile[from;to]	copies file <u>from</u> to file <u>to</u>, e.g., COPYFILE[/FOO/; "LINE PRINTER"]. Value is <u>to</u>.
delfile[file]	delete <u>file</u>. Value is <u>file</u> if found, otherwise NIL.
renamefile[old,new]	renames <u>old</u> to <u>new</u>. Value is OLD-FILE if <u>new</u> is an old file, NEW-FILE if <u>new</u> is a new file, or NIL if unsuccessful.

Symbolic File Input

load[x;flg;p]

load is a function which reads successive S-expressions from file x and evaluates each as it is read, until it reads either NIL, or the single atom STOP, followed by a carriage return, at which point it returns the value NIL.

If p=T, load prints the value of each S-expression; otherwise it does not. flg affects the treatment of expressions beginning with defineq: if flg=NIL, dfnflg is bound to T and the expression evaluated. If flg=T, dfnflg is bound to NIL and the expression evaluated. (This reversal is used so that with the normal usage of load, with only the first argument specified, defineq will operate as though dfnflg were T, its normal setting.) If flg=PROP, defineq is not called. Instead, the function definitions are stored on the property lists.

Thus if the function definitions for a particular file were all

	compiled, and one wished to edit a function definition, and make a new copy of the file containing it, without disturbing the compiled definitions, one would perform load[file;PROP], followed by editf[fn], since *editf* automatically goes to the property list if the function definition cannot be edited, followed by prettydef[fns;file], since *prettydef* automatically goes to the property list if the function definition cannot be prettyprinted.
readfile[v;x]	reads successive S-expressions from file *x* until the single atom STOP is read, makes a list of these S-expressions, and sets *v* to this list. Value is *x*.

Symbolic File Output

writefile[v;x]	writes successive S-expressions from *v* onto file *x*. If *v* is atomic, its value is used. If *x* is not open, it is opened and the date printed out. When *v* is finished, a STOP is printed on *x* and it is closed. Value is *x*.

prettyprint[x]	If x is an atom it will be evaluated to yield a list of functions. The definitions of the functions will be printed in a pretty format. If x is a list, it is used directly as the list of functions.

Example:

```
(FACTORIAL
  (LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (TIMES N (FACTORIAL (SUB1 N)))))))
```

Note: prettyprint will operate correctly on functions that are broken, broken-in, advised, or have been compiled with their definitions saved on their property lists - it prints the original, pristine definition.

A facility for documenting LISP functions is provided in prettyprint. Any S-expression beginning with * is interpreted as a comment and printed in the right margin.* Example:

```
(FACTORIAL
  (LAMBDA (N)                                              (* COMPUTES N!)
    (COND
      ((ZEROP N)                                           (* Ø!=1)
        1)
      (T                                                   (* RECURSIVE DEFINITION:
                                                            N!=N*N-1!)

        (TIMES N (FACTORIAL (SUB1 N)))))))
```

These comments actually form a part of the function definition. Accordingly, * is defined as an NLAMBDA NONSPREAD function that returns its argument, i.e. it is equivalent to QUOTE. When running an interpreted function, * is entered the same as any other

* Comments begin in column firstcol, initially set to 50, and end in column lastcol, initially set to 74.

LISP function. Therefore, comments should only be placed where they will not harm the computation. For example, writing (TIMES N(FACTORIAL (SUB1 N)) (* RECURSIVE DEFINITION)) in the above function would cause an error when TIMES attempted to multiply N, N-1!, and RECURSIVE.

For compilation purposes, * is defined as a macro which compiles into no instructions. Thus, if you compile a function with comments, and load the compiled definition into another system, the extra atom and list structures storage required by the comments will be eliminated. This is the way the comment feature is intended to be used.

endfile[x]	Prints STOP on and closes the file specified by x.
printdef[e]	**prints the expression e on the primary output file in a pretty format, i.e., prettyprint does printdef[getd[fn]] after appropriately setting output files, printlevel, etc.**
prettydef[prtyx;prtyy;prtyl]	This function is used for the creation of files containing systems of functions.

The arguments are interpreted as follows:

prtyx (first argument)	If a list, it is treated as a list of function names. **If prtyx is an** atom, it should have as a binding the list of functions for prettydef. The functions on the list are prettyprinted surrounded by a (DEFINEQ ...) so that they can be loaded with load. In addition, a

rpaqq 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.

prtyy
(second argument)

is 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 *output*.

file=atom
: The file *atom* is opened if not already open, and becomes the standard output file.

file=list
: *Car* of the *list* is assumed to be the file name and is opened if not already open. The standard output file is not changed in this case.

prtyl
(third argument)

is a list of commands, or if atomic, its value is used as a list of commands and an *rpaqq* is written which saves the list of commands on the named atom, and a *print* which informs the user of the named atom when the file is subsequently loaded, exactly as with the first argument.

These commands are used to save on the output file top level bindings of variables, property lists of atoms, miscellaneous LISP forms to be evaluated upon loading, arrays, and advised functions. It also provides for evaluation of forms at output time.

The interpretation of each command in the command list is as follows:

1. if STOP, the file is closed.

2. if atomic, an RPAQQ is written which will restore the top level binding of this atom then the file is loaded.

3. if of the form (PROP property atom1 ... atomn) for each atom following <u>property</u>, an appropriate DEFLIST will be written which will restore the <u>property</u> for each corresponding atom1 ... atomn when file is loaded. If <u>property</u>=ALL, the entire property list will be written with an RPLACD. If <u>property</u> is a list, DEFLISTs will be written for each property.

4. if the form (ARRAY ...), each atom following ARRAY should have an array as its value, and an appropriate expression will be written which will set the atom

to an array of exactly the same size, type, and contents upon loading.

5. If of the form (P ....), each S-expression following P will be printed on the output file, and consequently evaluated upon loading.

6. If of the form (E ....), each form following E will be evaluated immediately, i.e., while prettyprint is operating.

7. If of the form (ADVISE fn1 ... fnm) an appropriate expression will be written for each of the _m_ functions which will upon loading allow the user to reinstate the advice using the function READVISE.

8. If of the form (FNS fn1 ... fnm), a _defineq_ is written with the definitions of fn1 ... fnm, exactly as though (fn1 ... fnm) where the first argument to _prettydef_, e.g. suppose the user wanted to set some variables or perform some computations in a file _before_ defining functions, for example, do a _minfs_, he would then write the definitions using this option instead of using the first argument to _prettydef_.

9. If of the form (COMMANDS com1 ... comn), each of the commands com1 ... comn will be interpreted as one of the above eight command types.

In each of the nine commands described above, if the atom "*" follows the command type, the form following the *, i.e., caddr, is evaluated and its value used in executing the option, e.g., (FNS * FOOFNS), (PROP * FOOPROPS). Note that in the latter case, (CAR FOOPROPS) will be the property(ies) to be saved, and (CDR FOOPROPS) the list of atoms for which this property is to be saved.

Note that (COMMANDS * form) provides a way of computing what should be done by prettydef.

Example:

```
SET(FOOFNS (FOO1 FOO2 FOO3))
SET(FOOVARS(FIE (PROP MACRO FOO1 FOO2) (P (MOVD (QUOTE FOO1)
  (QUOTE FIE1))) STOP)
PRETTYDEF(FOOFNS /FOO/ FOOVARS)
```

would create a file /FOO/ containing

1. A message which prints the time and date the file was made (done automatically)
2. DEFINEQ followed by the definitions of FOO1, FOO2, and FOO3
3. (PRINT (QUOTE FOOFNS))
4. (RPAQQ FOOFNS (FOO1 FOO2 FOO3))
5. (PRINT (QUOTE FOOVARS))
6. (RPAQQ FOOVARS (FIE ... STOP)
7. (RPAQQ FIE value of fie)
8. (DEFLIST (QUOTE((FOO1 PROPERTY) (FOO2 PROPERTY))) (QUOTE MACRO))
9. (MOVD (QUOTE FOO1) (QUOTE FIE1))
10. STOP

printfns[x]	x is a list of functions. printfns prints DEFINEQ and prettyprints the functions. Used by prettydef, i.e. command (FNS * FOO) is equivalent to command (E (PRINTFNS FOO)).
printdate[]	prints the expression at beginning of prettydefed files that types date upon loading.
tab[pos,minspaces,file]	performs appropriate number of spaces to move to position pos. If position + minspaces is greater than pos, does terpri and then spaces [pos]. If minspaces not given, 1 is used.
makefile[file;flg]	does prettydef[fileFNS;/nfile/; fileVARS] where /nfile/ is first unused file, e.g., if user's file directory contains /1FOO/ and /2FOO/ makefile[FOO] does prettydef[FOOFNS;/3FOO/ FOOVARS]. If fileVARS is unbound atom, makefile uses (STOP). If flg = T, file is also listed using listfile.
listfile[files]	Lists each file in files on line printer using utility. Value is files.

SECTION XV

DEBUGGING AND ERROR HANDLING

Debugging Facilities

Debugging a collection of LISP functions involves isolating problems within particular functions and/or determining when and where incorrect data is being generated and transmitted. In the BBN-LISP system, there are three facilities which allow the user to (temporarily) modify selected function definitions so that he can follow the flow of control in his programs, and obtain this debugging information. These three facilities together are called the break package, All three redefine functions in terms of a system function, breakl, described below.

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

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

Breakin allows the user to insert a breakpoint inside an expression defining a function. When the breakpoint is reached 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.

The following two examples illustrate these facilities. In the first example, the user traces the function FACTORIAL. TRACE redefines FACTORIAL so that it calls BREAK1 in such a way that it prints some information, in this case the arguments and value of

```
PRETTYPRINT((FACTORIAL))

(FACTORIAL
   (LAMBDA (N)
      (COND
         ((ZEROP N)
           L)
         (T (TIMES N (FACTORIAL (SUB1 N)))))))
NIL
←TRACE(FACTORIAL)
(FACTORIAL)
←FACTORIAL(5)

FACTORIAL:
N = 5


    FACTORIAL:
    N = 4


        FACTORIAL:
        N = 3


            FACTORIAL:
            N = 2


                FACTORIAL:
                N = 1

FACTORIAL:
N = 0


ERROR
(L IS UNBOUND ATOM)
(FACTORIAL BROKEN)
:N
0
:RETURN 1
FACTORIAL = 1

                FACTORIAL = 1
            FACTORIAL = 2
        FACTORIAL = 6
    FACTORIAL = 24
FACTORIAL = 120
120
←
```

```
 PRETTYPRINT((FACTORIAL))

 (FACTORIAL
   (LAMBDA (N)
      (PROG (M)
            (SETQ M 1)
         LOOP(COND
               ((ZEROP N)
                  (RETURN M)))
            (SETQ M (TIMES M N))
            (SETQ N (SUB1 N))
            (GO LOOP)
         )))
NIL

←BREAKIN(FACTORIAL (AFTER LOOP) (LESSP N 2))
SEARCHING...
FACTORIAL
←FACTORIAL(5)

((FACTORIAL) BROKEN)
:NN

ERROR
(NN IS UNBOUND ATOM)
(FACTORIAL BROKEN AFTER LOOP)
:N
1
:M
120
:OK
(FACTORIAL)

((FACTORIAL) BROKEN)
:N
0
:OK
(FACTORIAL)
120
←
```

FACTORIAL, and then goes on with the computation. When an error occurs on the fifth recursion, BREAK1 reverts to interactive mode, and a full break occurs. The situation is then the same as though the user had originally performed BREAK(FACTORIAL) instead of TRACE(FACTORIAL), and the user can evaluate various LISP forms and direct the course of the computation. In this case, the user examines the variable N, and instructs BREAK1 to return 1 as the value of this call to FACTORIAL. The rest of the tracing proceeds without incident. The user would then presumably edit FACTORIAL to change L to 1. In the second example, the user has constructed a non-recursive definition of FACTORIAL. He uses BREAKIN to insert a call to BREAK1 just after the PROG label LOOP. This break is to occur only on the last two iterations, i.e., when N is less than 2. When the break occurs, the user looks at the value of N, mistakenly typing NN. However, the break is maintained and no damage is done. After examining N and M, the user allows the computation to continue by typing OK. A second break occurs after the next iteration, this time with N=0. When this break is released, the function FACTORIAL returns its value of 120.

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. breakl uses the ready character ":" to indicate it is ready to accept forms for evaluation (by eval). 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 and allows the computation to proceed. BREAK1 evaluates BRKEXP, its first argument, prints the value, and returns it as the value of the break. BRKEXP is set up by the function that created the call to BREAK1. For BREAK or TRACE, BRKEXP is (dummy ARG1 ARG2 ... ARGN), where dummy has the original definition of the function being broken and ARG1, ARG2, ... ARGN are its arguments. BRKEXP is NIL for BREAKIN using BEFORE or AFTER, and the indicated expression for BREAKIN AROUND.
OK	Same as GO except the value of BRKEXP not printed.
EVAL	Same as GO or OK except that the break is maintained after the evaluation. The user can then interrogate the value which is bound on the variable VALUE, and continue with the break. Typing GO or OK following EVAL will not cause reevaluation, but another EVAL will. This is a useful option when the user is not sure whether or not the break will produce the correct value and wishes to be able to do something about it if it is wrong.

RETURN <u>form</u>	The value of <u>form</u> is returned as the value of the break. For example, one might use the EVAL command and follow this with RETURN (REVERSE VALUE).
↑	Calls <u>error</u> and aborts the break. i.e. makes it "go away" without returning a value. This is a useful way to unwind to a higher level break. All other errors, including those encountered while executing the GO, OK, EVAL, and RETURN option, maintain the break.

Once a break occurs, control of the computation reverts to the user. The computation does not proceed without a specific instruction from the user. In most cases the user will simply check the values of certain key variables, and allow the computation to proceed. If he is not interested in the value of the break, he probably will use the OK command, and BREAK1 will "quietly go away." If he is interested in seeing the value, but fairly certain that it will be correct, he may use the GO command. If he wants to see the value of the break, but still have the option of further interaction with BREAK1, he will use the EVAL command, after which he can examine the variable VALUE. By means of the RETURN command, he can force BREAK1 to return an appropriate value, even when his function is still "buggy." This can be useful in localizing a problem in a large program. If substituting the omniscient user for a faulty function corrects the bug, then the problem has been isolated.

BREAK1 puts all of the power of LISP at the users command. For example, he can insert new breaks on subordinate functions simply by typing:

(BREAK <u>function1</u> <u>function2</u>)

or he can remove old breaks and traces if too much information is being supplied:

(UNBREAK function3 function4)

He can edit functions, including the one currently broken:

(EDITF function)

For example, the user might type EVAL, see that the value was incorrect, call the editor, repair the bug, and type EVAL again, all without leaving the break.

Similarly, the user can prettyprint functions, define new functions or redefine old ones, etc., load a file, compile functions, time a computation, etc. In short, anything that he can do at the top level EVALQUOTE can be done while inside of the break. In addition, the user can examine the pushdown list, via the functions described in section 12, and even force a return back to some higher function via the function retfrom.

Brkcoms

The fourth argument to breakl is brkcoms, a list of break commands that breakl interprets and executes exactly as though they were teletype input. One can think of brkcoms as another input file which always has priority over the teletype. Whenever brkcoms=NIL, breakl reads its next command from the teletype. Whenever brkcoms is not NIL, breakl takes as its next command car[brkcoms] and sets brkcoms to cdr[brkcoms]. For example, suppose the user wished to see the value of the variable x after a function was evaluated.

He would set up a break with brkcoms=(EVAL (PRINT X) OK), which would have the desired effect. The function trace uses brkcoms: it sets up a break with two commands; the first one prints the arguments of the function, or whatever the user specifies, and the second is the command GO, which causes the function to be evaluated and its value printed.

Note: if brkcoms is not NIL, the value of a break command is not printed. If you desire to see a value, you must print it yourself, as in the above example with the command (PRINT X).

Note: Whenever an error occurs, brkcoms is set to NIL, and a full interactive break occurs.

Breakmacros

Breakl specifically recognizes the five atomic commands ↑, GO, RETURN, EVAL, and OK. Whenever an atomic command is given breakl that it does *not* recognize, either via brkcoms or the teletype, it searches the list breakmacros for the command. The form of breakmacros is (... (macro command1 command2 ... commandn) ...). If the command is defined as a macro, breakl simply appends its definition, which is a sequence of commands, to the front of brkcoms, and goes on. If the command is not contained in breakmacros, it is evaluated as before.

The following breakmacros are currently defined:

?=	if from _brkcoms_, looks at next command on _brkcoms_ and prints the _value_ of each member of that list. For example, if _brkcoms_ were (EVAL ?= (X Y) OK), the user would see (_function_ BROKEN) _function_ EVALUATED X= value of x Y= value of y _function_ If the next command on _brkcoms_ is T, ?= operates on all of the arguments of the broken function, i.e. if the arguments of the broken function are X and Y, ?= T is equivalent to ?= (X Y). ?= is used by _trace_. ?= typed in by user is equivalent to the commands ?= and T on _brkcoms_, i.e. the names and values of the arguments to the broken function are printed.
!EVAL	function is first unbroken, then evaluated, and then rebroken. Very useful for dealing with recursive functions.
!OK	Function is first unbroken, evaluated, rebroken, and then exited, i.e. !OK is defined as !EVAL followed by OK.
!GO	Function is first unbroken, evaluated, rebroken, and exited with value typed, i.e. !EVAL followed by GO.

ARGS	the names of the arguments of the broken function are printed.
%A	evaluates and prints the value of the atom a, (lower-case A), see p. 12.4.
Bell (Control-G)	reverses the setting of brkevqflg, and prints new value, i.e. does (PRINT (SETQ BRKEVQFLG (NULL BRKEVQFLG)))

If brkevqflg=NIL, its normal setting, breakl operates in eval mode, uses : as its ready character and expects one input which it evaluates using eval (except if input is one of the five commands ↑, GO, EVAL, OK or RETURN or a breakmacro).

If brkevqflg=T, breakl operates in evalquote mode, its ready character is ←, and it then expects two inputs, a function and its arguments, which it gives to evalquote. However, the five regular commands as well as any breakmacros are not affected by the setting of brkevqflg and will always be recognized. evalquote mode is useful if you have a lot of expressions to evaluate that would otherwise have to be quoted, e.g.

```
:(PUT (QUOTE FOO) (QUOTE FIE) (QUOTE ...))
```

vs

```
←PUT(FOO FIE ...)
```

in evalquote mode.

BT	Prints a backtrace of function names only. If BT is followed by a function name, on the same line of input, the backtrace starts from the last occurrence of that function, e.g. BT FOO prints FOO and all functions entered above it in reverse chronological order. A number (position on the push list) can be used instead of a function name. Otherwise backtrace starts from the current position except that calls to error, faulteval, breakl, etc., are initially skipped.

BTV

Prints a backtrace of function names *with* variables. If followed by a number on the same line of input, *printlevel* is set to that number during the backtrace only, e.g.

```
BTV 2
```

will print a backtrace with the values of all variables being printed to a depth of 2. If this number is not supplied, *printlevel* is set to *0* for backtrace. In both cases *printlevel* is restored after backtrace, even if interrupted by a control-R. Note however, if the backtrace is aborted by a control-C, the *printlevel* will not have been restored.

Miscellaneous

The fifth argument to breakl is brkfile, and determines the output file for breakl. If it is NIL, the teletype is used. However, brkfile can be used to dump diagnostics onto a disc file, or even onto the file NOTHING. If brkfile is not open, it is opened. If an error occurs, brkfile is set to NIL and all output goes to the teletype.

brkx is a prog variable in breakl which is bound in breakl but not used. It is available for the user, to provide a local binding for computations associated with breaks.

Break Functions

breakl[brkexp;brkwhen;brkfn;brkcoms;brkfile]

is an nlambda. brkwhen determines whether a break is to occur. If its value is NIL, brkexp is evaluated and returned as the value of breakl. Otherwise a break occurs and an identifying message is printed using brkfn. Commands are then taken from brkcoms or the teletype and interpreted. The commands GO, OK, RETURN, and ↑, are the only ways to leave breakl. The command EVAL causes brkexp to be evaluated, and saves the value on the prog variable value. Other commands can be defined for breakl via breakmacros.

break∅[fn;when;coms;file] sets up a break on the function _fn_ by redefining _fn_ as a call to _break1_ with the original definition of _fn_ as _brkexp_,and _when_, _fn_, _coms_, and _file_ as _brkwhen_, _brkfn_, _brkcoms_ and _brkfile_. Puts property BROKEN on property list of _fn_ with value a _gensym_ defined with the original definition. Adds _fn_ to the list _brokenfns_. Value is _fn_.

If _fn_ is non-atomic and of the form (fn1 IN fn2), _break0_ first calls a function which changes the name of _fn1_ wherever it appears inside of _fn2_ to that of a new function, fn1-IN-fn2, which it initially defines as _fn1_. Then _break0_ proceeds to break on fn1-IN-fn2 exactly as described above. This procedure is useful for breaking on a function that is called from many places, but where one is only interested in the call from a specific function, e.g. (RPLACA IN FOO), (PRINT IN FIE), etc. It is similar to _breakin_ described below, but can be performed _even when FN2 is compiled_ whereas _breakin_ only works for interpreted functions.

If _fn1_ is not found in _fn2_, break0 returns the value (fn1 NOT FOUND IN fn2).

If fn1 is found in fn2, in addition to breaking fn1-IN-fn2 and adding fn1-IN-fn2 to the list brokenfns, break0 adds fn1 to the property value for the property NAMESCHANGED on the property list of fn2 and adds the property ALIAS with value (fn2 . fn1) to the property list of fn1-IN-fn2. This will enable unbreak to recognize what changes have been made and restore the function fn2 to its original state.

If fn is nonatomic and not of the above form, break0 is called for each member of fn using the same values for when, coms, and file specified in this call to break0. This associativity permits the user to specify complicated break conditions on several functions without retyping, e.g.

```
breakO[(FOO1 (PRINT IN FOO2)), (NEQ X T)
    (EVAL ?= (Y Z) OK)]
```

Associativity is also available for breaking in e.g. breakØ[((PRINI PRINT) IN (FOO FIE)),T]

Value is list of individual values.

break[x] is a nonspread nlambda. For each atomic argument, it performs breakØ[atom;T]. For each list, it performs apply [BREAKØ;list]. For example, BREAK(FOO1 (FOO2 (GREATERP N 5)))

will establish breaks on FOO1 and FOO2, the latter a conditional break.

trace[x] is a nonspread _nlambda._ For each atomic argument, it performs breakØ[atom;T; (TRACE ?= T GO)]* (see discussion of _brkcoms_ and _break-macros_ in text). For each list, it performs

breakØ[car[list];T;list[TRACE,?=, cdr[list],GO]]*

For example, TRACE(FOO1 (FOO2 Y)) will cause both FOO1 and FOO2 to be traced. All the arguments of FOO1 will be printed; only the value of Y will be printed when FOO2 is entered.

Note: the user can always call _breakØ_ himself to obtain combination of options of _break1_ not directly available with _break_ and _trace_. These two functions merely provide convenient ways of calling _breakØ_, and will serve for most uses.

(*) The flag TRACE is checked for in _break1_ and causes the message "function": to be printed instead of (function BROKEN).

breakin[fn,where,when,coms,file]

<u>breakin</u> is an <u>nlambda</u>. Its last three arguments correspond to the last three arguments of <u>break0</u>, except if <u>when</u> is NIL, T is used. <u>where</u> specifies where in the definition of <u>fn</u> the call to <u>break1</u> is to be inserted. There are three possibilities: (BEFORE ...), (AFTER ...), or (AROUND ...). "..." is used by the expanded editor's locate routine to find the correct point for the break. For example, (BEFORE COND) will insert a break before the first occurence of <u>cond</u>, and (AROUND (SETQ X --)) will break around the first place X is set. Note that specifications such as (AROUND ALL (SETQ X --)) for breaking around every place x is set, and (AROUND (SECOND (COND CONTAINING RETURN))) are perfectly acceptable. See p. 9.26-9.40 for description of expanded editor.

If <u>fn</u> is a compiled function, <u>breakin</u> returns UNBREAKABLE as its value.

If <u>fn</u> is interpreted, <u>breakin</u> types SEARCHING... while it calls the editor. If the location specified by <u>where</u> is not found, <u>breakin</u> types

(NOT FOUND) and exits. If it is found, *breakin* adds the property BROKEN-IN with value to T to the property list of *fn*, and adds *fn* to the list *brokenfns*.

Because of the operation of the UP command in the expanded editor, (BEFORE COND) and (BEFORE X) will both have the desired effect. The first will insert a break before the entire expression beginning with COND, the second before X itself. A special check is made to avoid inserting a break in the wrong place when a variable may appear as car of an expression, as in the case of a *cond* or *selectq*, i.e. the X in (COND ... (X ..) ...) will not satisfy the locating routine.

Another special check is made to avoid inserting a break *anywhere* inside of an expression headed by any member of the list *nobreaks*, initialized to (GO QUOTE *), since this break would never be activated.

It is possible to insert multiple break points, with a single call to breakin by using a *list* of the form ((BEFORE ...) .. (AROUND ...)) for *where*. It is also possible to call *break* or *trace* on a function which has been modified by *breakin* , and conversely to *breakin* a function which has been redefined by a call to *break* or *trace*.

unbreak[x]

unbreak is a non-spread *nlambda*. It takes an indefinite number of functions modified by *break*, *trace*, or *breakin* and restores them to their original state by calling *unbreak0*. unbreak[] will unbreak all functions on *brokenfns*, a list that is updated by *break0* and *breakin*. Value is list of values of *unbreak0*.

unbreak0[fn]

restores *fn* to its original state. If *fn* was not broken, value is (NOT BROKEN) and no changes are made. If *fn* was modified by *breakin*, *unbreakin* is called to edit it back to its original state. If *fn* was created from (fn1 IN fn2), i.e. if it has a property ALIAS, the function in which *fn* appears is restored to its original state. All dummy functions that were created by the break are eliminated.

Note: unbreak0[(fn1 IN fn2] is allowed: *unbreak0* will operate on fn1-IN-fn2 instead.

unbreakin[fn]	performs the appropriate editing operations to eliminate all changes made by *breakin*. *fn* may be either the name or definition of a function. Does not check to see if any changes were made. Value is *fn*.
changename[fn,from,to]	changes all occurrences of *from* to *to* in *fn*. *fn* may be compiled. Value is *fn* if *from* was found, otherwise NIL. Does not perform any modifications of property lists.
virginfn[fn,flg]	is the function that knows how to restore functions to their original state regardless of any amount of breaks, breakins, advising, compiling and saving exprs, etc. It is used by *prettyprint*, *define*, and the *compiler*. If flg=NIL, as for *prettyprint*, it does not modify the definition of *fn* in the process of producing a "clean" version of the definition. If flg=T as for the *compiler* and *define*, it physically restores the function to its original state, and prints the changes it is making, e.g. FOO UNBROKEN, FOO UNADVISED, etc. Value is the virgin function definition.

valv[x,fn,n]

a useful form of evalv for inside of a break. valv is an nlambda. x is the name of a variable to be evaluated, using evalv, as of the nth occurrence of the function fn, i.e. nthfn[fn,n]. If n is NIL, 1 is used. If fn is NIL, the last position used by valv is used (it is saved on the free variable vvnlast). For example,

```
:(VALV X MATCH)
value of X as of last call to MATCH
:(VALV Y)
value of Y as of last call to MATCH
```

Error Handling in LISP

There are currently twenty four different error types in the BBN LISP system. These are discussed in greater detail below. However, by far the most common "error condition" in LISP programs: unbound atoms and undefined functions, is not treated as an error at all, but handled in a special way by the BBN interpreter. The basic difference between a bona fide error, and an unbound atom or undefined function, is that errors are detected *after* they occur, e.g. PDL OVERFLOW, NONXMEM, or else they are detected inside a low-level function (a SUBR, SUBR* or FSUBR*), like *plus* or *setq* e.g. NUN-NUMERIC ARG, NON-ATOMIC ARG, ILLEGAL REGISTER COMMAND, whereas unbound atoms and undefined functions are detected by the interpreter when it attempts to evaluate a LISP form. Consequently, the system is in a better position to allow the user to correct unbound atom and undefined function error conditions than the more basic errors, although these too are "helpable" in the BBN LISP system.

Unbound atoms and undefined functions

Whenever the interpreter encounters an atomic form with no binding on the push-down list, and whose value is the atom NOBIND,(*) the interpreter calls the function faulteval. Similarly, faulteval is called when a non-atomic form is encountered, car of which is not a function.(**). The value returned by faulteval is used by the interpreter as the value of the form. faulteval is defined to print either UNBOUND ATOM or UNDEFINED CAR OF FORM, followed by the name of the atom or car of the form, and then to call breakl giving it as brkexp the offending form. Once inside the break, the user can set the atom, define the function, return a specified value for the form using the RETURN command, etc, or abort the break using the ↑ command. If the break is exited with a value, the computation will proceed exactly as though no error had occurred.

This call to breakl can be inhibited by setting to NIL the variable helpflag, which is initially set to T. In this case, faulteval instead calls the function error (p. 15.33). It is at this point that the unbound atom or undefined car of form actually becomes a LISP error. Similarly, error is called instead of breakl if the "error"

(*) All atoms are initialized (when they are created by the read program) with their value cells (car of the atom) NOBIND, their function cells NIL, and their property lists (cdr of the atom) NIL.

(**) See Appendix 2 for complete description of BBN LISP interpreter.

occurs within helpdepth, initially set to 4, function calls from the top level evalquote or the last errorset (p. 15.33). In the example below, a break occurs when FOOO is evaluated inside the mapcar, but not when typed in to e. Of course, the user can set helpdepth to any value he wishes. For helpdepth=0, breaks will always occur.

```
←SET(FOO (1 2 3))
(1 2 3)
←E (MAPCAR FOOO (FUNCTION ADD1))
UNBOUND ATOM FOOO

(FOOO BROKEN)
:RETURN FOO
FOOO = (1 2 3)
(2 3 4)
←E FOOO

ERROR
(FOOO IS UNBOUND ATOM)
UNTRACE:
   XEEEE FOOO
E
```

Undefined function calls from compiled code

Frequently, a function will be compiled when some of the functions it calls are not defined. The compiler merely generates instructions for calling the function exactly as though it were defined as a LAMBDA expression (p. 16.7). However, if the function is undefined at run time, the system routine that performs the actual calling of functions will instead call the function interrupt, analagous to faulteval in the interpreter. Interrupt is defined to type UNDEFINED FUNCTION followed by the function name and then call breakl.

If the function was undefined at compile time, its arguments will have been evaluated (see p. 16.7). In this case, they may be examined using the function arg, p. 8.6. e.g. arg(1), arg(2), or the function breaknargs can be used to make a list consisting of all of the arguments to the function. For example, the expression given to breakl as brkexp is (APPLY FUNCTION (BREAKNARGS INTERRUPTARGS)) where interruptargs is the argument to interrupt, and is bound to the number of arguments for the undefined function, i.e. the number of arguments with which is was called, and the variable function is bound by interrupt to the undefined function name. If the user defines function as a LAMBDA expression, and executes the OK, GO, or EVAL command in breakl, the correct value will be computed.

As with faulteval, the value returned by interrupt is used as the value for the function call, and the computation proceeds exactly as though no error had occurred.

Similarly, the call to breakl can be inhibited by setting the variable helpflag to NIL, and no break occurs if the error was within helpdepth function calls from the top or the last errorset. In these cases, error is called instead.

Inducing an interrupt

The user can induce an interrupt and subsequent call to breakl at any point in a computation by typing control-H. At the next point a function is about to be entered, interrupt is called instead. Interrupt types INTERRUPTED BEFORE followed by the function name, and then calls breakl exactly as though the function were undefined. The arguments to the function can be examined using arg or breaknargs. If the user types OK, GO, or EVAL the function call will be continued. Control-H interrupts are not affected by the setting of helpflag or helpdepth. However, they only occur when a function is about to be entered. If the program is computing in a function which after compilation, does not call any other functions, (pp. 16.11-16.14) computation will not be interrupted until that function is exited.

"Real" errors

The conventional treatment of errors in a LISP system is to cause a trap to a routine which prints an error message and unwinds the pushdown list. While unwinding the pushdown list, the system prints the names and arguments of all functions that have been entered but not yet exited, with the most recently entered function printed first.(*) If the function errorset has been entered, unwinding proceeds only as far as the most recent call to it, and errorset then returns NIL as its value, to indicate an error occurred. Otherwise, unwinding proceeds until the top level evalquote.

In the BBN LISP system, this unwinding process takes place only as a last resort, i.e. if the variable helpflag is NIL, or if the error type (p. 15.29) is specified as non-helpable, or if the error occurred within helpdepth function calls from the top or the last errorset. Otherwise, the error diagnostic is printed and breakl is called to allow the user to examine the state of the computation and proscribe the next action.

Unfortunately, the user may not always be able to make a correction and proceed as if no error had occurred, as he can with calls to faulteval and interrupt. When an error occurs in a low-level routine, the state of the computation must be backed up to the last function call before breakl can be called. For example, if the compiled function hypotenuse were defined as:

(*) In the BBN system, this printout can be terminated by pressing control-R, or portions of it can be skipped by judicious use of rubout, or the printlevel can be modified by using control-P, or the user can always bomb back to the top level via control-C. See appendix 2, p. 23.7.

```
(HYPOTENUSE
   (LAMBDA (X Y)
      (EXPT (FPLUS (FTIMES X X)
            (FTIMES Y Y))
         0.50000000)))
```

and the user performed:

```
←E (FTIMES (SINE 30) (HYPOTENUSE 3))

NON-NUMERIC ARG
NIL
IN HYPOTENUSE

(HYPOTENUSE BROKEN)
:BTV
UNTRACE:
    Y NIL
    X 3
HYPOTENUSE
    LAMBDA &
    %A 0.50000000
FTIMES
    XEEEE &
E

:(SETQ Y 4)
4
:EVAL
HYPOTENUSE EVALUATED
:VALUE
5
:OK
HYPOTENUSE
2.50000000
←
```

the computation would be preserved as of the time hypotenuse was called, since ftimes and fplus compile open (p. 16.10). Thus the partial results of the computation would be lost when the error occurred. In this particular case, the user could proceed as shown.

However, if the computation made some changes in the program's environment before the error occurred, the programmer may not be able to simply repeat the computation. For example, if hypotenuse were defined as:

```
←PRETTYPRINT((HYPOTENUSE))

(HYPOTENUSE
   (LAMBDA (X Y)
     (EXPT (FPLUS (SETQ X (FTIMES X X))
                (FTIMES Y Y))
       0.50000000)))
NIL

←E (FTIMES (SINE 30) (HYPOTENUSE 3))

NON-NUMERIC ARG
NIL
IN HYPOTENUSE

(HYPOTENUSE BROKEN)
:X
9
:RETURN (HYPOTENUSE 3 4)
HYPOTENUSE = 5
2.50000000
←
```

The user must evaluate each situation individually to decide whether or not he can continue, or should force a returned value, or perform a retfrom back to some higher level.

Error types

There are currently twenty four error types in the BBN LISP system:

0	NONXMEM
1	BREAK
2	CAR OF NUMBER
3	PDL OVERFLOW
4	UNDEFINED FUNCTION
5	FUNCTION 'ARG' NOT LEGAL
6	ATOM STORAGE FULL
7	PNAME STORAGE FULL
8	UNREASONABLE LINE LINE LENGTH
9	ILLEGAL RADIX SETTING
10	ILLEGAL INPUT FORMAT
11	ILLEGAL REGISTER COMMAND
12	ILLEGAL FILE NAME
13	NOT USED (UNLUCKY)
14	NON-NUMERIC ARG
15	NON-ATOMIC ARG
16	ATTEMPT TO CLOBBER NIL
17	NUMBER STORAGE EXCEEDED
18	ERROR
19	ILLEGAL GO
20	ILLEGAL RETURN
21	QUIT
22	INCOMPATIBLE
23	NOT FOUND
24	TOO MANY CHARACTERS IN ATOM

Explanation of error types

NONXMEM	reference to *non*-e*x*istent *mem*ory. Can occur if array-pointer or other unboxed number is treated as list structure, i.e. program tries to take car of it, but more frequently an indication that system is sick.
BREAK	User types control-R
CAR OF NUMBER	occurs when interpreter tries to take car of number, e.g. (COND 387 (T NIL))
PDL OVERFLOW	occurs from infinite recursion, where infinite means more than 1500 nested function calls.
UNDEFINED FUNCTION	very rare - means calling routine is very confused - normally it calls *interrupt* on undefined function as described above
FUNCTION 'ARG' NOT LEGAL	*arg* used inside a function that was *not* a no-spread, evaluate type function
ATOM STORAGE FULL	too many atoms, (current system can hold 3100 *new* atoms,) if a *reclaim* does not collect any of the user's atoms, he can continue by flushing part of the system, p. 22.8, and then performing an *atomgc*, p. 10.3.
PNAME STORAGE FULL	can occur if you have many atoms with long names
UNREASONABLE LINE SETTING	linelength[n], n > 999.
ILLEGAL RADIX SETTING	radix[n], n > 255.
ILLEGAL INPUT FORMAT	*read* is confused, e.g. it saw an expression like (A .)
ILLEGAL REGISTER COMMAND	*openr* or *closer* given an illegal address.

ILLEGAL FILE NAME	attempt to read from, write on, or close a file that is not open
NON-NUMERIC ARG	from numeric functions like plus, times, etc.
ERROR	call to function error
ILLEGAL GO	go to nonexistent label
ILLEGAL RETURN	call to return from outside a prog
QUIT	call to quit
INCOMPATIBLE	from sysin, see p. 14.11
NOT FOUND	from sysin see p. 14.11
TOO MANY CHARACTERS IN ATOM	> 86

The list of non-helpable errors, nherrors, is initially set to (0 1 18). The rationale behind this is that NONXMEM are usually system malfunctions; control-R means abort-(control-H should be used to request for interaction), and error type 18, a call to error should not be helpable: the function help, p.15.33, is available for that purpose. However, the user can set nherrors to any list of error type numbers for which he does not wish the system to go into a break.

Error Messages

errorn[]	returns information about the last error in the form (n m) where n is the error type number and m is the argument to errorm which would normally be printed out after the error message. Thus if an unbound atom FOO had been encountered, errorn[] would yield (18 (FOO IS UNBOUND ATOM)). In the example with hypotenuse on page 15.28 errorn would yield (14 NIL).
errorm[n;m]	prints message corresponding to an errorn that yield (n m). For example, errorm[18,(FOO IS UNBOUND ATOM)] would print ERROR (FOO IS UNBOUND ATOM) errorm[14;NIL] would print NON-NUMERIC ARG NIL and errorm[24] would print out just TOO MANY CHARACTERS IN ATOM

Error Functions

error[x]	causes an error, type 18, with message X.
help[helpx;helpy]	Generates an error with message *helpx,* that is helpable i.e. *breakl* will be called, if either *helpflag* or *helpy* is T, regardless of the depth. *help* is a convenient way to program a default condition, or to terminate some portion of a program which theoretically the computation is never expected to reach.
errorset[ersetx;ersetflg]	performs eval[ersetx]. Note that *errorset* is a lambda-type of function, and that its arguments are evaluated *before* it is entered, i.e. errorset[x] means eval is called with the *value* of *x*. If no error occurs in the evaluation, the value of *errorset* is a list containing one element, the value of eval[x]. If an error did occur, the value of *errorset* is NIL. Note that NIL can be returned only if there was an error. If the value eval[x] is NIL, the value of *errorset* is (NIL). The argument *ersetflg* controls the printing of error messages. If ersetflg=T, the error message is

printed; if *ersetflg*=NIL it is not. If *ersetflg* = IGNORE, the errorset is ignored. Thus you can make an errorset "go away" while still inside of it.

Note: errorset is defined as just (LIST (EVAL ERSETX)). All of the errorset-ing effect is performed in *errorx*, described below, after an error occurs.

ersetq[ersetx]	*nlambda* performs errorset[ersetx;t], i.e. (ERSETQ (FOO)) is equivalent to (ERRORSET (QUOTE (FOO)) T)
nlsetq[nlsetx]	*nlambda*, performs errorset[nlsetx;NIL].
esgag[x]	sets *esgag* to x, returns old value. If *esgag* is T, an untrace will be printed during unwinding to an errorset. If it is NIL, no untrace will be printed. Initially set to NIL.
quit[x]	Induces a "strong" error which will unwind through *errorsets* to the top level. It prints the error message x and an untrace.
reset[]	Induces a "strong" error which will immediately return you to the top level with no untrace. *reset* is esentially a programmable control-C.

Errorx

For completeness, and a summary of the error handling facilities, this section describes howerrorx, the basic error handling routine of the system is written using nthfn, nthfnback, errorm, errorn, backtrace, reset, evalv, and retfrom.

Errorx is called for all 23 error types. It first performs an errorn to determine the error number. If this number is not a member of the list nherrors and helpflag is T, and the difference between nthfn[ERRORSET;1] the position of the last call to errorset, (0 is used if nthfn returns NIL i.e. no calls to errorset) and nthfn[ERRORX;1] is not greater than helpdepth, errorx calls errorm to print the error message, and then calls breakl.

If there were no calls to errorset, or the value of ersetflg was IGNORE for all of the calls to errorset, errorx calls errorm to print the message, calls backtrace to print the untrace, and calls reset to get back to the top level.

Otherwise, errorx looks at the value of ersetflg with evalv to determine whether to print a message. If this value is T, errorx prints the error message using errorm. If esgag is T, errorx calls backtrace to print the untrace. Finally, errorx does a retfrom[nthfn[errorset;1]NIL] to return NIL from the last errorset.

SECTION XVI

THE COMPILER AND ASSEMBLER

The Compiler

The compiler is available in the regular LISP system. It may be used to compile individual functions as requested or all function definitions in a standard format LOAD file. The resulting code may be loaded as it is compiled, so as to be available for immediate use, or it may be written onto a file for subsequent loading. The compiler also provides a means of specifying sequences of machine instructions for special purposes.

The most common way to use the compiler is to compile from a symbolic file, producing a corresponding file which contains a logical set of functions in compiled form which can be quickly loaded. An alternate way of using the compiler is to compile from functions already defined in the user's LISP system. In this case, the user has the option of specifying whether the code is to be saved on a file for subsequent loading, or the functions redefined, or both. In either case, the compiler will ask the user certain questions concerning the compilation. The first question is

(LISTING?)

The answer to this question controls the generation of a listing and is explained in full below. However, for most applications, the user will want to answer this question with either <u>ST</u> or <u>F</u>, which will also specify an answer to the rest of the questions which would otherwise be asked. ST means the user wants the compiler to Store the new definitions; F means the user is only interested in compiling to a File, and no storing of definitions is performed. In both cases, the compiler will then ask the user one more question:

(OUTPUT FILE?)

to which the user can answer

NIL	no output file.
file-name	file is opened if not already opened, and compiled code is written on the file.*

Example:

```
COMPILE((FACT FACT1 FACT2))
(LISTING?)
ST
(OUTPUT FILE?)
/CFACT/
(FACT COMPILING)
     .
     .
(FACT REDEFINED)
     .
     .
     .
(FACT2 REDEFINED)
(FACT FACT1 FACT2)
```

This process caused the functions FACT, FACT1, and FACT2 to be compiled, redefined, and the compiled definitions also written on file /CFACT/ for subsequent loading.

---

* Note some compiler functions will leave the output file open, others do not. Consult the description of each particular function.

Compiler Functions

compile[x] — This will compile all the functions on the list _x_. Returns a list of the functions compiled. Leaves output file open.

Note: Certain compiler functions leave the output file, if any, open so the user can perform several compilations to the same file. When finished, compiled files should be closed by performing endfile[file-name].

recompile[prettyfile;compiledfile;fns]

The purpose of _recompile_ is to allow the user to update a compiled file without necessitating a complete recompilation. _recompile_ does this by using the results of a previous compilation, and is considerably faster than compiling an entire file from scratch.

compiledfile is a disc file containing functions in compiled form. _prettyfile_ is a disc file made by _prettydef_. _recompile_ makes a new file that is equivalent to performing a tcompl((prettyfile)). (If the output file from _tcompl_ would have the same name as _compiledfile_, the user is asked to name the output file.) Every function defined in _prettyfile_ that appears on the list _fns_ is compiled from its definition in _prettyfile_. For all other functions in _prettyfile_, _recompile_ reads from _compiledfile_ until it finds the compiled version and then simply copies it onto the output file. Note that the user can thus modify an old compiled file so as to add new functions by prettydefing them and then including them on the list *fns*. Similarly, he can delete functions by not putting them in *prettyfile*. *Warning*: this procedure assumes that the *order* of the functions in *compiledfile* follows that of *prettyfile*.

Note: when a function is compiled from an in core definition, i.e., via *compile* as opposed to *recompile* or *tcompl*, which use definitions from a file, and it has been modified by *break*, *breakin* or *advised*, the function is restored to its original state before compilation. If the user wishes to compile a function *with* its advice, he should use the function *cadvice* described on page 19.10.

rcompile[]	Compiles from a file whose name will be requested after the *compset* questions have been answered. This should be a disc file because it will be open during the entire compilation. The value of this function is NIL. Closes output file.
tcompl[x]	*x* is a list of file names. Performs a *rcompile* for each of the files in the list. The user is not asked to specify an output file for each file. Instead, the output from the compilation of each file will be written on a file of the same name prefixed with a *c*. The value of *tcompl* is a list

of the names of the output files. All output files will be properly terminated and closed. Note: due to present restrictions of the 940 file system, only disc files (names beginning with a slash) should be used.
Example:

```
TCOMPL ((/SYM1/  /SYM2/  /SYM3/))
```

creates files

```
/CSYM1/, /CSYM2/, /CSYM3/
```

compile2[name;def]   Compiles <u>def</u>, redefines <u>name</u> if <u>strf</u>=T, (described below). This is the function to call if you wish to use the compiler as a subroutine, i.e., from another function as opposed to direct input from teletype:

Compiler Questions

The compiler uses the free (top level) variables LAPFLG, STRF, SVFLG, NLAMA, NLAML, LCFIL and LSTFIL which determine various modes of operation. These variables are set by the answers to the "compset" questions. When any of the top level compiling functions have been called, the function compset is called 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:

(LISTING?)

The answer to this question controls the generation of a listing. Possible answers are:

1	Prints output of pass 1, the LAP macro code.
2	Prints output of pass 2, the LAP2 machine code.
YES	Prints output of both passes.
NO	Prints no listings.

The variable LAPFLG is set to your answer.

The LAP and LAP2 code is usually not of interest to the user. There are three other possible answers to this question, each of which specifies a complete mode for compiling. They are:

S	Same as last setting
F	Compile to File (no definition of function)
ST	Store new definitions

Implicit in these three are the answers to the questions on disposition of compiled code, expr's and NLAMBDA's, so these questions will be skipped. These questions are discussed below.

(STORE AND REDEFINE?)

YES Causes each function to be redefined as it is compiled. The compiled code is stored and the function definition changed. The variable STRF is set to T.

NO Causes function definitions to remain unchanged. The variable STRF is set to NIL.

The answer ST for the first question implies YES for this question, F implies NO, and S makes no change.

(SAVE EXPRS?)

If you answer this YES, SVFLG will be set to T, and the exprs will be saved on the property list of the function name. Otherwise they will be discarded. The answer ST for the first question implies YES for this question, F implies NO, and S makes no change.

When compiling the call to a function, the compiler must prepare the arguments in one of three ways:

1. Evaluated (SUBR, SUBR*, EXPR, EXPR*, CEXPR, CEXPR*)
2. Unevaluated, spread (FSUBR, FEXPR, CFEXPR)
3. Unevaluated, not spread (FSUBR*, FEXPR*, CFEXPR*)

In attempting to determine which of these three is appropriate, the compiler will examine the definition of the called function if there is one, otherwise it will check all the functions in the file being compiled, and failing this, <u>will assume type 1 above</u>. Therefore, if there are type 2 or 3 functions called from the functions being compiled, and they are only defined in a separate file, the following two questions must be answered.

(NO-SPREAD NLAMBDAS-)

The answer to this question sets the variable NLAMA. If there are any NLAMBDA's with atomic argument lists called from your functions to be compiled, and they 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 $\underline{fn}_1$ to $\underline{fn}_k$ 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 compset will ask:

(SPREAD NLAMBDAS-)

Answer in the same way. The free variable used by the compiler is NLAML this time. The answers ST, F, or S to the first question leave the settings of NLAMA and NLAML unchanged.

(OUTPUT FILE)

This question is always asked except under TCOMPL. You should usually provide the name of a disc file on which you wish to save the 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. The free variable LCFIL is set to the name of the file.

When the compiler is operating, it will normally print on the teletype the name of the function compiling, a list of its bound variables and a list of its free variables.

When you have finished compiling all the functions you wish to dump on one disc file, close the file <u>endfile</u>.

The code dumped on the file can be loaded into any standard system with <u>load</u>.

Compiler Structure

The compiler has two principal passes. The first compiles its input into a prefix macro assembly language called LAP. The second pass expands (and optimizes) the LAP code and produces a simple assembly language (one instruction per line) called LAP2. This output is either dumped onto a file and/or loaded into array (binary program) space and the function redefined.

The input for the compiler can be either a standard LISP function definition (the normal usage), or an _assemble_ form, which allows direct machine language coding within LISP in a convenient form.

The compiled code generated always links between functions by using a special call-enter pair of routines. This is necessary because a function may not be in core when called, and this is checked in a binary function hash table. A function must be brought into the in-core binary program buffer to run.

The linkage routines also set up the parameter and control push lists as necessary for variable bindings and return information. In some cases discussed below, the linkage routine can be avoided (saving about a millisecond a call) by compiling short functions "open." Some often used functions, such as _car_ and _cdr_, are always called open by the compiler (a complete list is given later).

Open Functions

It is useful to know what LISP forms do not result in function calls after they are compiled since function calls take a significant time. Thus, it is more economical to perform

```
(AND (NULL (EQ (CAR X) 4)) (OR Y (NULL (ATOM Z))))
```

in a compiled function than to call another function. In addition to functions such as addl, subl, memb, etc. which compile open via macros, the compiler specifically checks for certain functions like plus, times, car, cdr, etc. and handles them in an efficient way. Below is a list of those functions which when compiled do not result in external function calls. Note: that mapc and map will require a call if their functional argument requires one.

```
    ABS               LRSH
    ADD1              LSH
    AND               MAP
    ARRAYP            MAPC
    ASSEMBLE          MEMB
    ATOM              MINUS
    CAR               MINUSP
    CAAR              NEQ
    CAAAR             NLISTP
     etc.             NOT
    CDDDAR            NULL
    CDDDDR            NUMBERP
    COND              OR
    DIFFERENCE        PLUS
    DIVIDE            PROG
    EQ                PROG1
    FIXP              PROG2
    FLOATP            PROGN
    FMINUS            QUOTE
    FPLUS             QUOTIENT
    FQUOTIENT         REMAINDER
    FTIMES            RETURN
    GO                RSH
    GREATERP          SELECTQ
    LESSP             SETQ
    LISTP             SUB1
    LOC               TIMES
    LOGAND            VAG
    LOGOR             ZEROP
    LOGXOR
```

Affecting the Compiled Code

There are three ways to affect code compiled for you. You can make a function fn compile open (as an open LAMBDA or NLAMBDA expression) by putting the expression defining it (including the LAMBDA or NLAMBDA) 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 this 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 actually compute the LISP expression to be compiled in place of the call to fn. 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 result of this evaluation is 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)))))
```

this causes (LIST X Y Z) to be compiled as

```
(CONS X (CONS Y (CONS Z NIL))).
```

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

* The evaluation is done by the function expandcomp, which takes two arguments, the property value for MACRO, and cdr of the form in which fn appears, and returns the expression to be compiled. This is the function to break on if you want to see if your macro is expanding correctly.

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 then used as 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)). Thus, (ADD1 (CAR X)) is compiled as (PLUS (CAR X) 1). Note that a function like times2 defined as (LAMBDA (X) (PLUS X X)) could be compiled open or could be made a substitution macro. The macro, however, would cause (TIMES2 (FOO X)) to compile as (PLUS (FOO X) (FOO X)) and consequently (FOO X) would be evaluated twice. In this case it is better to use an open macro - i.e., put (LAMBDA (X) (PLUS X X)) on the property list of TIMES2, so that its argument would only be evaluated once.

Note:
Expressions that begin with FUNCTION will always be compiled as separate functions named by attaching a gensym to the end of the name of the function in which they appear, e.g. FOOA0003. This latter function will be called at run time. Thus if FOO is defined as (LAMBDA (X) ... (FOO1 X (FUNCTION ...)) ...) and compiled, then when FOO is run, FOO1 will be called with two arguments, X, and FOOA000n, and then FOO1 will call FOOA000n each time it must use its functional argument. A considerable savings in time can be achieved by defining FOO1 as a macro of type two:

```
        (MACRO X (LIST (SUBST (CADADR X) (QUOTE FN) *)
                     (CAR X)))
```

where * is the definition of FOO1 as a function of just its first argument and FN is the name used for its functional argument. This expression will be evaluated at compile time and produce an expression to be compiled that contains the actual definition of the function that would have been the second argument to FOO1 had FOO1 not been compiled open. Thus you save the function call to FOO1 and each of the function calls to its functional argument. For example, if FOO1 operates on a list of length ten, eleven function calls will be saved. Of course, this savings in time costs space, and the user must decide which is more important.

Free Variables and EVQ

As discussed in section 12, free variables used by a function are detected at compiled time so that when the compiled function is entered, its free variables can be bound locally. This procedure saves searching the entire push-down list each time a free variable is used in the compiled function. However, if the user knows that the particular portion of the function that references the free variable will only be reached infrequently, he may opt to search the push-down list only when the value of the free variable is needed. This can be done by using the form (EVQ variable) instead of variable. (For interpreted purposes, EVQ is defined as (LAMBDA (X) X).) Note that if a free variable will be used more than once in a function, it is more efficient to search for its binding once, when the function is entered, than each time the variable is used.

Changing the Binary Program Buffer

While running binary code, a program ring buffer of 4K is used to contain active program. The size of this buffer can be affected by the following functions:

contractl[]	Contracts the in-core binary program buffer by one LISP (256 word) page, thereby also adding one virtual page buffer. Returns value of new higher boundary. Will not contract beyond a predetermined minimum amount, an assembly parameter (2K in 4-1-68 LISP).
expandl[]	Expands the in-core binary program buffer by one LISP (256 word) page, thereby also removing one virtual page buffer. Returns value of new lower boundary. Will not expand beyond predetermined maximum amount. (8K in 4-1-68 LISP)

Note: Expanding the BP buffer will usually not be very effective in speeding up programs unless the code used is just larger than the current buffer size. Then expanding the buffer will allow an all in core operation, rather than continuous shuffling of code back and forth from the drum. Contracting the buffer is advantageous only when a relatively small compiled program is to be used for a considerable period, with a data base that requires more than 20 buffers.

<u>Assemble</u>

Using the LISP compiler, it is possible to define functions partially or completely in machine language. Machine language is specified by the pseudo-function <u>assemble</u>. <u>assemble</u> is, in fact, a compiler directive, and has no independent definition. Thus, it is not possible to interpretively run functions defined using <u>assemble</u>.

The format of ASSEMBLE is similar to that of PROG.

    (ASSEMBLE V S_1 S_2. . . S_n)

Each of the statements S_n are interpreted sequentially during compilation according to the rules given below. V is a list of variables to be bound during compilation, <u>not</u>, it must be noted, during the running of the object code. Interpretation of each S will usually result in the generation of one or more instructions of object code. Some S, however, may result in no object code being generated. Note than an ASSEMBLE statement can appear anywhere in a LISP function, e.g., you can write

```
(SETQ Z (PLUS X (LOC (ASSEMBLE NIL (BRS 42))) Y))
```

The value of the pseudo-function <u>assemble</u> is determined by what is left in the A register after the execution of the sequence of <u>assemble</u> instructions.

Assemble Statements

If S is an atom, it is taken as a label identifying the next cell to be assembled. Otherwise, it is one of the following types of statements.

```
     (CQ C1 C2 ...)
```

C1, C2, ... are literal S expressions which are compiled in order in the usual way.

```
     (C E1 E2 ...)
```

Same as CQ except the En are first evaluated and then compiled. The above two statements provide the ability to mix regular compilation with assembly. The value of the A register may be obtained within a compile statement by use of the function AC. It must, however, appear as the first argument to be evaluated in the expression.

Example:

```
          (CQ (PLUS FOO 1))
          (C (CONS (QUOTE FN1)(CDR FOO)))
          (CQ (PLUS (LOC (AC)) FIE -1))
```

```
     (E E1 E2 ...)
```

The expressions E1, E2, ... are evaluated in order for effect, i.e., no code is produced.

Example:

```
          (E (SETQ SP (PLUS SP -3)))
```

(RETURN)

Assembles an instruction which causes a return from the function being compiled (not from the ASSEMBLE expression), with the contents of the A register as the value. A return from the ASSEMBLE expression is done by "falling through" or branching to the instruction following the last statement. The value is the contents of the A register at that time.

(CALL NAME N)

Assembles a call to the function NAME giving N arguments. The N arguments should be in stack positions
SP-N+1, SP-N+2, ..., SP-1, SP. See Section "Compiler Conventions."
Note: A, B, X registers are destroyed.

(SETQ VAR)

Assembles an instruction which stores the A register in the variable VAR.

Note: The contents of the X register are an index to the parameter stack and are used by compiled code whenever a variable or temp storage cell is referenced. If the code specified by an assemble directive changes the X register, it should (<u>must</u>) be restored with (LDX PPPTR) before executing any regular compiled code.

Lap Macros

If CAR of the statement is an atom which has a LAP property-list macro definition, e.g., LDV, STV, etc., the arguments are evaluated and the results assembled. If CAR of the statement is a defined function, the function is called, without evaluating the arguments, and the result is treated as code. This feature would normally not be of use to the programmer; it is used by LAP in interpreting first pass code generated by the compiler.

Assemble Macros

If CAR of the statement has a property list value following the flag AMAC, it is assumed to be an assembler macro call. There are two types of assembler macros, substitution and lambda. A substitution macro is defined by an S expression, CAR of which is a list of dummy symbols. The arguments of the call will be substituted for corresponding appearances of the dummy symbols in CDR of the defining form and the resulting list of statements will be assembled.

If CAR of the defining form is the atom LAMBDA, the entire defining form will be applied to the arguments of the call. Note that either of these may be indefinitely recursive.

Example:

```
      DEFLIST ((

      (UBOX ((VAR LOC)
             (CQ (VAG VAR))
             (STA LOC)))

      (UBOXN (LAMBDA XX
                (PROG (YY)

                LP (COND
                     ((NULL XX)
                         (RETURN (CAR YY)))
                     (T (SETQ YY (TCONC (LIST (QUOTE UBOX)
                                           (CAR XX)
                                           (CADR XX))
                                        YY))
                        (SETQ XX (CDDR XX))
                        (GO LP)))

      )AMAC)
```

The above defines two macros, one of each type. The first takes two arguments and expands into instructions which place the unboxed value of a numeric variable in a local cell. The second does the same thing for an indefinite number of pairs of arguments. For each pair, it constructs a call to the first macro.

the call:

```
     (UBOX SUM XSUM)
```

expands into:

```
    ((CQ (VAG SUM))
     (STA XSUM))
```

the call (UBOXN S1 L1 S2 L2 ...)

first expands into

```
    ((UBOX S1 L1)
     (UBOX S2 L2)
          .
          .
          .
               )
```

Machine Instructions

If CAR of the statement is an atom defined as a machine instruction, e.g., by having a property OPD with numeric value (see compiler conventions),

```
     (LDA A1 A2)
```

the remainder of the statement may contain Ø, 1 or 2 expressions.

If either A1 or A2 or both are not present, Ø is assumed as their value.

A2 may be used to specify indexing and indirect addressing when required. I specifies indexing (equivalent to a value of $2ØØØØØØØ_8$), and J specifies indirect addressing (equivalent to a value of $4ØØØØ_8$). Otherwise, A2 may be any expression which evaluates to a number and will be added into the assembled word.

If A1 is a number, it is added unchanged into the assembled word.

If A1 is non-atomic, it is evaluated and the result added into the assembled word which is assumed to refer to a <u>stack position</u>, and handled accordingly.

If A1 is atomic, it is one of the following:

=	Specifies that A2 is a literal. The instruction will be assembled to address a cell which contains A2, e.g., (LDA = NIL). If A2 is a number, it will be unboxed. Works only for S expressions, does not work for local program symbols (tags).
*	Has the value of the current location, e.g., (BRU * 1) is a jump to next location.

A member of V, the list of variables given to ASSEMBLE, or the variable SP or CODE: is evaluated and assembled as a <u>stack position</u> (as for A1 non-atomic).

One of the system defined atoms such as SYSNIL, SPCELL, etc.: the top-level value of the atom is added into the assembled word.

Any other atom is assumed to refer to a tag in the program. Note: the detection of an undefined label does not occur until pass 2 of the compiler by which time all labels have been translated into generated symbols. Thus, the error comment "UNDEFINED LABEL" will inform the user of the problem, but will not specify which label is missing.

If CAR of a statement is a number, it is treated as if it were preceded by an opcode of value Ø.

Examples:

```
      (LDA BUF I)                     label reference, indexing
      (ADD = 47)                      numeric literal
      (SKG = Ø)
      (BRU * 2)                       relative address
      (BRU LOC1)
      (E (BOX SP))                    evaluate - compiles an ENBOX
      (CAB)
      (LDA = NIL)                     non-numeric literal
      (PCONS)                         fast CONS
      (CQ (RETURN (TCONC (AC) TCL)))  return, use of A register
                                          in compile
LOC1  (STA (PSTEP))                   stack reference
      .
      .
      .
      (LDA SP)                        stack reference
      (STA SPCELL 1)                  global symbol reference
      (STA SPCELL (PLUS J 10))        indirection and address arithmetic
      .
      .
      .
TMP1  (Ø)                             temp storage
TMP2  (Ø)
BUF   (BSS 1ØØ)                       block definition
```

When using locations within the function for temporaries, remember that the core copy of a compiled function <u>may</u> be overwritten any time another compiled function is called or a return effected. In this case, all internal changes will be lost.

Use

ASSEMBLE should appear in a function defined with the usual defining forms. To relieve the user of the burden of unnecessary detail, as much of the function as possible should be compiled. For example, to obtain a variable, it is best to write

```
      (CQ VAR)
```

to load the value of a variable into the A register.

Thus the function

```
      (LAMBDA (X) X)
```

could be written

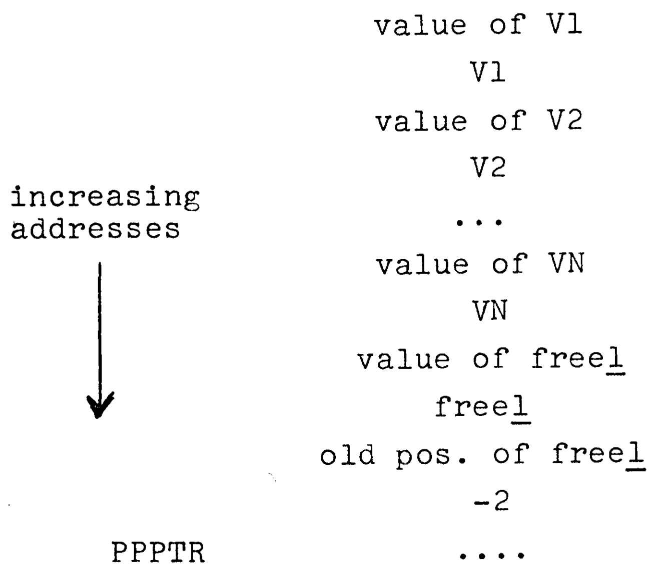
```
      (LAMBDA (X) (ASSEMBLE NIL
                    (CQ X)))
```

and would compile identically.

Compiler Conventions

The user of assemble should understand the following basic things about how compiled code is run. As explained in Section XII, all variable bindings and temporary storage of values are kept on the parameter pushdown list. When a compiled function is entered, the parameter pushdown list contains, in ascending order of addresses:

1. Pairs of words containing the names and values of arguments passed to the function.
2. Blocks of four words containing the value, name, and old locations of free variables used in the function and a flag so indicating.
3. Room for temporary storage, for arguments to lower level functions, and for PROG and LAMBDA bindings appearing in the body of the function.



The index register (and a cell called PPPTR) contain a pointer to the first cell of the temporary block (just after the free variable bindings). This portion of the stack is guaranteed to be in core, and the compiler keeps a variable MSP which contains the maximum stack position used. SP is the variable which usually contains the last stack position used. The function pstep adds one to SP, updates MSP if necessary and returns the incremented SP. Each increment by 1 of SP changes the PDL position by two cells. If any functions are called, care must be taken that the garbage collector and free variable searcher are not confused by random things on the push list. Use of the LAP macros STT (for store temporary) and STN (for store number unboxed) will avoid such problems. STT compiles into code which stores the value in the value word and Ø in the name word of the stack position, thus erasing any old name left from earlier calls. This also indicates to the G.C. that the value word of the pair contains a pointer to be traced. STN stores an unboxed integer in the value word and a -1 in the name word as a flag to the G.C. not to mark from this value word.

There are a number of values which are stored on atoms which may vary for different system assemblies. These are dumped in symbolic form on LAP files to make these files compatible across assemblies.

The following are programmed operator instructions used by compiled code:

CARCLL	car of A register
CDRCLL	cdr " " " "
CONSCLL	cons of A and B registers
UNBOX	unbox number in A register (VAG)
ENBOX	enbox quantity in A register (LOC)
XCLL	function call
RETURN	function return
IPV	initialize prog variables
ENTER	enter function and setup args.

The following top-level bindings are the location of cells containing quantities of interest or used for communication.

SYSNIL	contains NIL
SYST	contains T
SYSTAT	contains lower boundary of atoms
SYSNUM	contains lower boundary of numbers
SYSINT	contains lower boundary of integers
TOPBPS	contains upper boundary of array space in use
FREELW	contains lower boundary of list space in use
CTEMP	communication with garbage collector
INTZRO	intzro+n for $-30 \leq n \leq 30$ contains n
SPCELL	first of a block of 100 cells for general use
PPPTR	index to push list in core

The following is a list of all machine operation, and programmed operator codes defined in the computer system.

ADC	5700000
ADD	5500000
ADM	6300000
BAC	4600012
BIO	57600000
BRM	4300000
BRR	5100000
BRS	57300000
BRU	100000
BRX	4100000
BSS1	0
BXC	4600022
CAB	4600004
CAX	4600400
CAXB	4600440
CBA	4600010
CBX	4600020
CIO	56100000
CLA	4600001
CLAB	4600003
CLB	4600002
CLX	24600000
CNA	4601000
CTRL	57200000
CXA	4600200
DIV	6500000
EAX	7700000
EOR	1700000
ETR	1400000
EXU	2300000
FAD	55600000
FDV	55300000
FMP	55400000
FSB	55500000
GCI	56500000
LCY	6720000
LDA	7600000
LDB	7500000
LDE	4600140
LDP	56600000
LDX	7100000
LRSH	6624000
LSH	6700000
MIN	6100000
MRG	1600000
MUL	6400000
NOD	6710000
NOP	2000000
NSTA	3500000
OVT	2200101
PFFV	13400000
PLAI	11200000
PMFN	13500000
PSAI	11300000
PSTR	14100000
RCY	6620000
ROV	2200001
RSH	6600000
SKA	7200000
SKB	5200000
SKD	7400000
SKE	5000000
SKG	7300000
SKM	7000000
SKN	5300000
SKR	6000000
STA	3500000
STB	3600000
STE	4600122
STP	56700000
STX	3700000
SUB	5400000
SUC	5600000
SXMA	6200000
TCI	57400000
TCO	57500000
VAL	0
WCH	56400000
WCI	55700000
WIO	56000000
XAB	4600014
XMA	6200000
XXB	4600060

Appendix

This section contains listings of those compiler and lap macros which are normally included with the compiler system. There are no assemble macros pre-defined.

Compiler Macros

```
(DEFLIST(QUOTE(
  (LIST (X (GLIST X)))
  (ADD1 ((X)
      (PLUS X 1)))
  (SUB1 ((X)
      (PLUS X -1)))
  (NEQ ((X Y)
      (NOT (EQ X Y))))
  (NLISTP ((X)
      (NOT (LISTP X))))
  (ZEROP ((X)
      (EQ X Ø)))
  (MINUSP ((X)
      (GREATERP Ø X)))
  (DIFFERENCE ((X Y)
      (PLUS X (MINUS Y))))
  (ABS (LAMBDA (X)
      (COND
        ((GREATERP Ø X)
          (MINUS X))
        (T X))))
  (ERSETQ ((X)
      (ERRORSET (QUOTE X)
        T)))
  (EVQ (X (COND
        ((ATOM (CAR X))
          (STORECOMP (LIST (QUOTE LFV)
              (CAR X))))
        (T (CEXPR (CAR X))))
      (QUOTE INSTRUCTIONS)))
  (MAP (X (LIST (SUBPAIR (QUOTE (MAPF MAPF2))
          (LIST (CFNP (CADR X))
            (COND
              ((CDDR X)
                (CFNP (CADDR X)))
              (T (QUOTE CDR))))
          (QUOTE (LAMBDA (MACROX)
              (PROG NIL
                LP  (COND
                      ((NULL MACROX)
                        (RETURN)))
                    (MAPF MACROX)
                    (SETQ MACROX (MAPF2 MACROX))
                    (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 (MACROX)
            (PROG NIL
              LP  (COND
                    ((NULL MACROX)
                      (RETURN)))
                  (MAPCF (CAR MACROX))
                  (SETQ MACROX (MAPCF2 MACROX))
                  (GO LP)
              ))))
      (CAR X))))
(MEMB (LAMBDA (MACROX MACROY)
    (PROG NIL
      LP  (RETURN (COND
              ((NULL MACROY)
                NIL)
              ((EQ MACROX (CAR MACROY))
                (IFPRED T MACROY))
              (T (SETQ MACROY (CDR MACROY))
                (GO LP))))
      )))
(NLSETQ ((X)
    (ERRORSET (QUOTE X)
      NIL)))
(VAG (X (CEXPR (CAR X))
    (COND
      ((EQ (CAADR CODE)
          (QUOTE ENBOX))
        (RPLACA (CDR CODE)))
      (T (STORECOMP (QUOTE (UNBOX)))))
    (QUOTE INSTRUCTIONS)))
(LOC (X (CEXPR (CAR X))
    (COND
      ((EQ (CAADR CODE)
          (QUOTE UNBOX))
        (RPLACA (CDR CODE)))
      (T (BOX SP)))
    (QUOTE INSTRUCTIONS)))
```

```
  (FRPLAC (X (CEXPR (CAR X))
      (STS)
      (CEXPR (CADR X))
      (STORECOMP (LIST (QUOTE MSAI)
          SP))
      (SETQ SP (SUB1 SP))
      (QUOTE INSTRUCTIONS)))
  (ASSEMBLE (ASEMX (ASEM1 ASEMX))
    )
  (AC (X (QUOTE INSTRUCTIONS)))
  (IFPRED (AA (COND
        (EBRF (CAR AA))
        (T (CADR AA)))))
  (ARG (X (CEXPR (LIST (QUOTE VAG)
          (CAR X)))
      (STORECOMP (LIST (QUOTE ARGN)
          (COND
            (ARGARG)
            (T (ERROR (QUOTE (FUNCTION 'ARG' NOT LEGAL)))))))
      (QUOTE INSTRUCTIONS)))
  (SETARG (X (CEXPR (LIST (QUOTE VAG)
          (CAR X)))
      (STORECOMP (LIST (QUOTE STN)
          (PSTEP)))
      (CEXPR (CADR X))
      (STS)
      (LACOMP (SUB1 SP))
      (STORECOMP (LIST (QUOTE SARGN)
          (COND
            (ARGARG)
            (T (ERROR (QUOTE (FUNCTION 'SETARG' NOT LEGAL)))))
          SP))
      (SETQ SP (PLUS SP -2))
      (QUOTE INSTRUCTIONS)))
  (LSH (X (SHIFTCOMP (CAR X)
        (CADR X)
        (QUOTE LSH))))
  (RSH (X (SHIFTCOMP (CAR X)
        (CADR X)
        (QUOTE RSH))))
  (LRSH (X (SHIFTCOMP (CAR X)
        (CADR X)
        (QUOTE LRSH))))
))(QUOTE MACRO))
```

Lap Macros

```
(CSP1 ((LV LF LT)
    (LITREF LDA LV)
    (LITREF LDX LF)
    (LITREF LDB LT)
    (PRGREF PENT (PLUS PLITORG 1))))
(SETIX ((N P)
    (LDV N)
    (UNBOX)
    (LSH 1)
    (CNA Ø)
    (ARGSUB N)
    (ADD PPPTR)
    (STN P)))
(VST1 ((PP LV V)
    (LITREF LDA PP)
    (LITREF LDB LV)
    (PRGREF PIPV (PLUS PLITORG V))))
(BE ((B N)
    (STKREF SKE N)
    (RELREF BRU 2)
    (JUMP B)))
(BNE ((B N)
    (STKREF SKE N)
    (JUMP B)))
(LDV (LAMBDA (S)
    (VREF (QUOTE LDA)
      S)))
(STV (LAMBDA (S)
    (VREF (QUOTE STA)
      S)))
(LFV (LAMBDA (S)
    (LITREF (QUOTE PATV)
      S)))
(LDT (LAMBDA (S)
    (STKREF (QUOTE LDA)
      S)))
(STT (LAMBDA (S)
    (STKREF (QUOTE STA)
      S)))
(NSTT (LAMBDA (S)
    (STKREF (QUOTE NSTA)
      S)))
```

```
(MSAI (LAMBDA (S)
    (STKREF (QUOTE PSAI)
      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)
    (STKREF CLLXA L)))
(ARGN ((A)
    (CLB Ø)
    (LSH 1)
    (STKREF ADD A)
    (CAXB Ø)
    (LDA Ø I)
    (CBX Ø)))
(SARGN ((A B)
    (CLB Ø)
    (LSH 1)
    (STKREF ADD A)
    (CAB Ø)
    (LDT B)
    (XXB Ø)
    (STA Ø I)
    (CBX Ø)))
(ARGSUB (LAMBDA (A)
    (LITREF (QUOTE ADD)
      (PLUS -2 (VREF1 A)))))
(RET (NIL (PRETN Ø)))
(BN ((B)
    (SKE SYSNIL)
    (RELREF BRU 2)
    (JUMP B)))
(BNN ((B)
    (SKE SYSNIL)
    (JUMP B)))
```

```
(BAP ((B)
     (SKG TOPBPS)
     (LITREF SKG 16383)
     (RELREF BRU 2)
     (JUMP B)))
(BNAP ((B)
     (SKG TOPBPS)
     (LITREF SKG 16383)
     (JUMP B)))
(BA ((B)
     (SKG SYSTAT)
     (RELREF BRU 2)
     (JUMP B)))
(BNA ((B)
     (SKG SYSTAT)
     (JUMP B)))
(BLST ((B)
     (SKG SYSTAT)
     (SKG TOPBPS)
     (RELREF BRU 2)
     (JUMP B)))
(BNLST ((B)
     (SKG SYSTAT)
     (SKG TOPBPS)
     (JUMP B)))


(UNBOX (NIL (PFVE Ø)))
(ENBOX ((N)
     (PMKN N)))
(FENBOX ((N)
     (PMFN N)))
(FUNBOX (NIL (PFFV Ø)))
(NEG (NIL (CNA Ø)))
(DVD ((N X)
     (RSH 23)
     (DIV N X)))
(DIVIDE ((S)
     (STTN S)
     (SWAP Ø)
     (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)))
(BIF ((B)
    (SKG SYSINT)
    (SKG SYSNUM)
    (RELREF BRU 2)
    (JUMP B)))
(BUF ((B)
    (SKG SYSINT)
    (SKG SYSNUM)
    (JUMP B)))
(BII ((B)
    (SKG SYSINT)
    (RELREF BRU 2)
    (JUMP B)))
(BUI ((B)
    (SKG SYSINT)
    (JUMP B)))
(BIS ((B L)
    (LITREF1 SKE L)
    (RELREF BRU 2)
    (JUMP B)))
(BNS ((B L)
    (LITREF1 SKE L)
    (JUMP B)))
(BR1 ((B)
    (PRGREF BRU (GBS B))))
(BR2 ((B)
    (RELREF BRU 2)
    (PRGREF BRU (GBS B))))
(CONSCLL ((N)
    (CAB Ø)
    (STKREF LDA N)
    (PCONS (TIMES N 2))))
(CLLX ((N)
    (PCLL N)))
(CLLXA ((N X)
    (PCLL N X)))
(SWAP (NIL (XAB Ø)))
(JUMP ((B)
    (PRGREF BRU (GBS B))))
(MPY ((N X)
    (MUL N X)
    (LSH 23)))
(BSS (LAMBDA (N)
    (SETQ LOC (PLUS LOC N -1))
    (LIST (LIST (QUOTE BSS1)
        N))))
```

SECTION XVII

USING FORKS AND THE HYBRID PROCESSOR IN LISP

The FORK logic provided by the 940 time-sharing system is available for use by LISP programmers. Use of this very powerful feature has made possible the efficient running of the hybrid processor for display output and speech input. Other operations are also available, such as running independent subsystems under control of LISP.

A fairly complete grasp of the machine-language environment provided by the time-sharing system and the 940 hardware is necessary for a complete understanding of the basic operation of forks under LISP. However, a large class of jobs may be performed using the existing system functions which require only a minimal knowledge of fork operations.

We will first discuss the storage organization of the LISP system and the conventions which have been established for the use of forks. The basic nature of forks will be discussed in sufficient detail to give anyone with a moderate knowledge of 940 machine language a good understanding of their mechanics. Those not interested in programming at the machine language level should, nonetheless, be able to get a general picture of the nature of forks sufficient to understand the functions described below.

Forks

In a time-sharing system such as that running on the 94Ø, there may be several users running apparently simultaneously, each with his own "machine" (which may appear very much different from the actual machine), memory, files, etc. Obviously, the monitor program must have the ability to keep track of several programs at once. The 94Ø system makes it possible for the individual user to make use of this ability if he chooses. That is, he may cause the monitor to handle not one, but several "programs" for him at one time. These are called <u>forks</u> in the 94Ø system.

It is important to understand the concept of a fork. A fork is a complete process, capable of executing instructions and, in general, of performing all the operations of machine language programs. A fork is specified by several items:

1. Central registers (PC, A, B, X)
2. Memory (two relabeling registers)
3. Status (running, waiting for I/O, etc.)

The first two of these are needed to define a fork. That is, if the contents of the A, B and X registers are specified, and memory is provided (presumably containing instructions), a computation may be performed by simply telling the "computer" where to start (the function of the PC). The status is then determined by the nature of the instructions and how far the "computer" has gone.

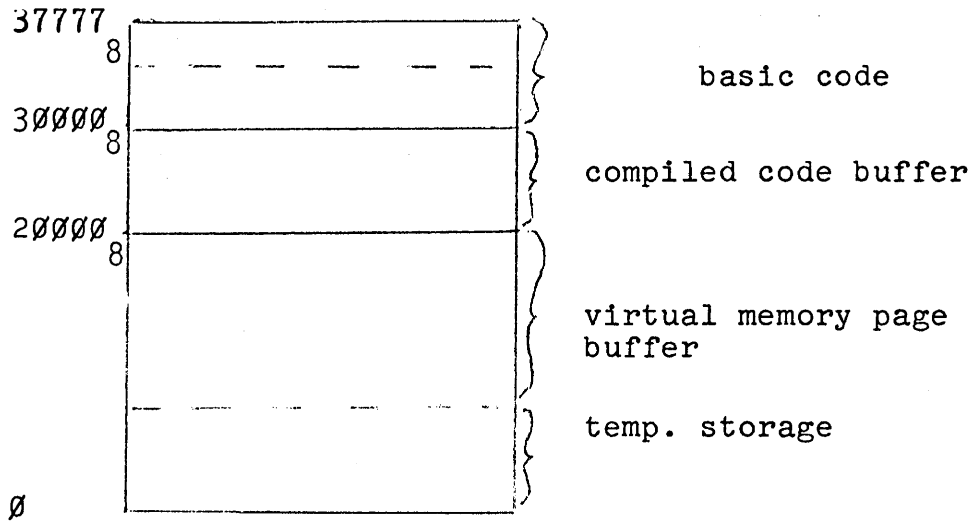
On the 940, a fork may start one or more forks subordinate to itself. In fact, all programs are subordinate forks (at some level) of the EXECUTIVE program. The EXECUTIVE itself is a fork distinguished only by the fact that it has no higher level controlling fork. When one fork starts another, they are assumed to run concurrently, although in fact the machine can only be executing one instruction at any instant.

A fork may have memory separate from or in common with its controlling fork, or both. It is this fact which gives forks their main usefulness to LISP. The 940 gives the user up to 32K of private (accessable to no other user) memory divided into 16 pages. Because the address part of a 940 instruction is 14 bits, a program can directly address only 16K of this memory. A program may, however, change its map or create a fork by which the same addresses can be made to refer to different sections of the 32K private memory. (See BRS manual "Memory Allocation and Sharing" for a detailed discussion of this).

LISP Memory Allocation

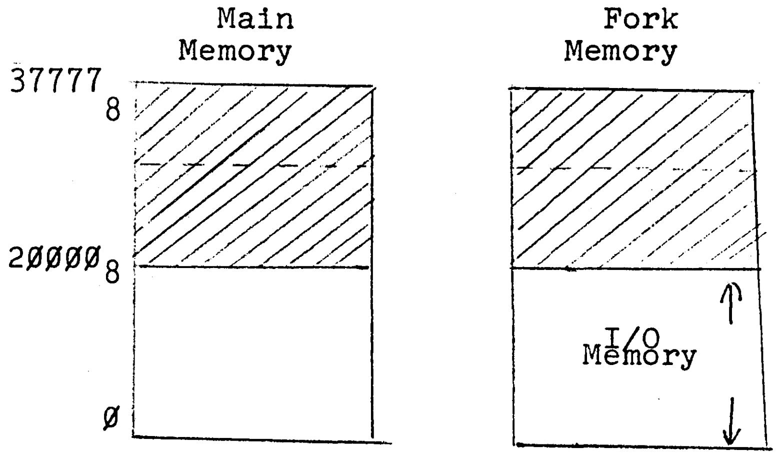
There are several different levels of storage used by LISP, and it is important to understand the distinctions. First, there is the large LISP virtual memory, so called because there is, in fact, no contiguous block of storage corresponding to it. The allocation of this memory is described in Section III.

Next, there is the core memory in which reside the basic instructions comprising the LISP interpreter, I/O routines, SUBRS, garbage collector, etc. This is the 16K of memory directly addressable by instructions. The contiguity of this memory block is also an illusion, but a very convincing one since it is implemented by the paging box. Core memory is also used for running compiled code, holding page buffers and temporary storage as shown in the figure.



LISP Core Memory Allocation

Because the LISP virtual memory is an illusion created by the LISP program, it is not possible to use it directly for I/O. That is, the hybrid processor, for example, cannot be instructed to read or write a large block of words using a LISP array because the array will in general be on the drum. Even if in core, it may be spread out over several non-contiguous page buffers. Therefore, it is necessary to allocate a contiguous block of core sufficient to hold the information to be transferred. But as can be seen from the diagram above, core is already completely allocated. The alternative is to create a fork with at least some independent memory and use it to do the I/O. The programs now written which use the fork logic allocate fork memory as shown in the figure.



The shaded area, with addresses from $2ØØØØ_8$ to 37777_8, is common to both the main and the fork memory. The area from Ø to 17777_8 in the fork is the independent memory used for I/O.

Because the page buffers are necessary to effect references to the LISP virtual memory, and the page buffers and the I/O memory do not exist in the same fork, a two-step process is necessary to move data between the virtual memory and I/O memory. This consists in first copying words from the source memory to a buffer area common to both forks then copying the buffer area to the destination memory. A program running in main memory is used to move data between the virtual memory and the buffer, and a program running in fork memory is used to move data between the I/O memory and the buffer. The LISP functions which perform these data transfers are compiled code and reside in the compiled code area when running. This area is common to both forks, and so a single function can contain instructions for execution in the fork as well as those for execution in the main program. The function must also contain space allocated for the buffer. The functions to transfer data to and from forks are called <u>storefork</u>, <u>stfk</u>, <u>readfork</u>, and <u>rdfk</u>, and are described in detail at the end of this section. The implementation details below should allow a user to define his own specialized functions for fork data transfer and running.

Implementation Details

The information necessary to start a fork has been described generally above. The specific format prescribed by the time-sharing system for this information is shown below. This information is contained in a block of seven words called the fork table (sometimes called the panic table).

Ø	Program Counter
1	A Register
2	B Register
3	X Register
4	First Relabeling Register
5	Second Relabeling Register
6	Status

To start a fork, the controlling program must place the address of the fork table in the A register and execute a BRS 9 instruction. The high order five bits of the A register also contain some control information which is used by BRS 9. More details are available in the BRS manual.

The BRS 9 causes the fork to commence operation as specified by the fork table, and both the main program and the fork are then running. When the fork is dismissed for any reason, the fork table is updated to show the latest contents of the central registers and relabeling. The status word indicates what caused the dismissal.

SPCELL+Ø	PC	
1	A	
2	B	
3	X	FIRST FORK TABLE
4	RL1	
5	RL2	
6	STAT	
7		
8		
9		
SPCELL+10	CONSTANT -	ADDRESS OF FIRST FORK TABLE
11	CONSTANT -	A REGISTER FOR BRS 9 FOR FIRST FORK
12	FDDT RL1	RELABELING FOR FDDT
13	FDDT RL2	
14	FDDT ADR	START ADDRESS FOR FDDT
15		
16		
17		
18		
19		
SPCELL+20	PC	
21	A	
22	B	SECOND FORK TABLE
23	X	
24	RL1	
25	RL2	
26	STAT	

There is a block of words in the temporary storage area of LISP available to the user for any purpose. The first location of this block is bound to the atom SPCELL. A portion of this block has been allocated for fork data as shown.

Note that there are two fork tables. This allows one fork to be transferring data to or from the fork memory while the other runs the hybrid processor. The relabeling is the same for both.

In some instances it is useful to have a fork containing DDT which can examine the running LISP. For example, this allows examination of compiled code in the binary program buffer. The following function provides that facility:

ddt[]	causes LISP to start a DDT which is looking at the running LISP. To continue LISP under this DDT, type `30002;G.` To return to LISP from DDT, hit 2 rubouts or %F. Calling ddt[] while in a LISP running under DDT will cause a return to the higher DDT, not start a lower one.

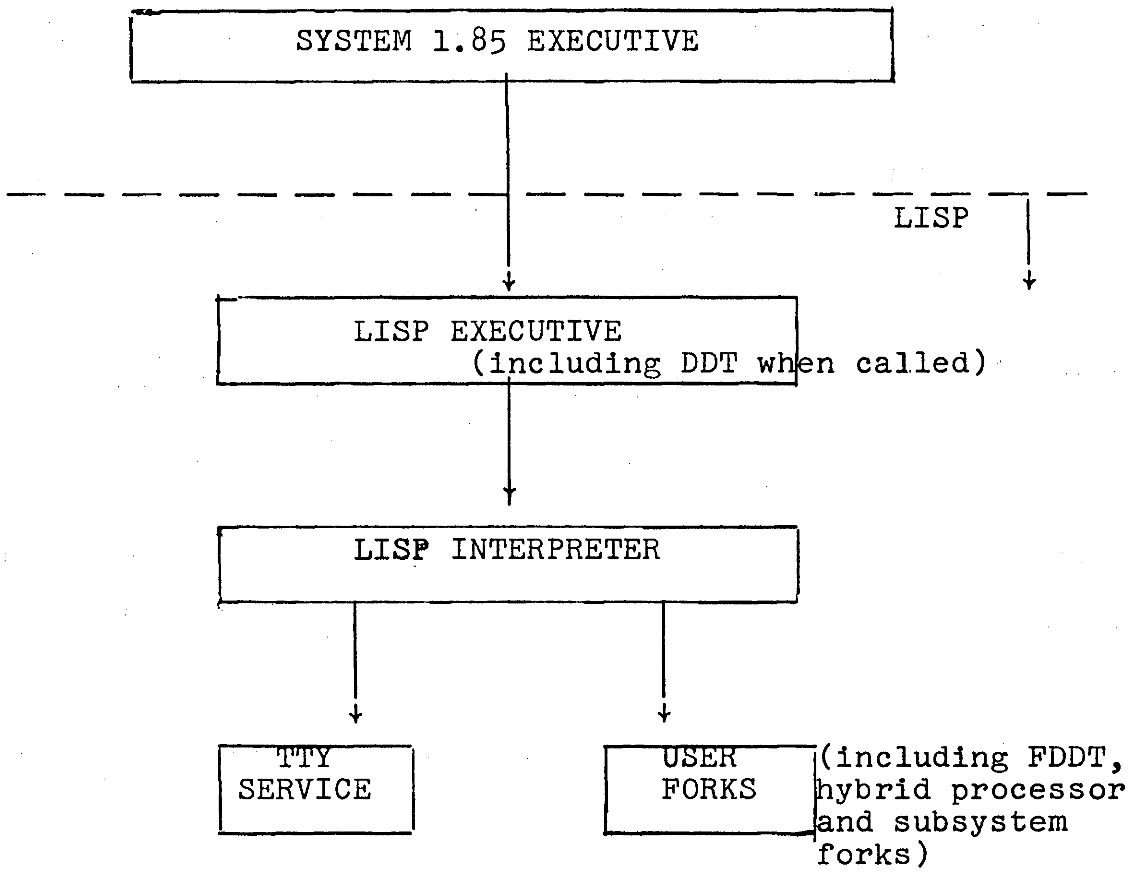
Another function called <u>fddt</u> (for fork ddt) is available in the standard system to facilitate debugging of fork programs.

fddt[]	starts a DDT (the regular subsystem program) <u>under</u> LISP and sets it to look at the <u>fork memory</u> as determined by the contents of RL1 and RL2 in the first fork table. Two rubouts cause DDT to return to LISP. This DDT is distinct from the one described earlier, called by ddt[], which looks at the running LISP memory from the position of the LISP executive.

Programmers writing fork programs should be aware of the complete fork structure of LISP as shown below in order to avoid complications. The user forks mentioned can be any that the user starts up. In addition, the user can call under LISP other 94Ø subsystems using the function <u>subsys</u>.

subsys[name;file1;file2]	starts up a 94Ø subsystem as a fork under LISP. Only those subsystems listed on the variable <u>systems</u> can be started. If <u>file1</u> is given, the subsystem accepts input from <u>file1</u>, otherwise teletype. If <u>file2</u> is given, output goes to <u>file2</u>, otherwise teletype.
utility[file]	subsys[UTILITY;file]

The LISP executive fork performs very few functions and is run only when the interpreter dismisses itself. The TTY service fork, however, runs concurrently with the interpreter and is always waiting for TTY input. If a user fork is to do TTY input, the TTY Service fork must be terminated. A BRS 108 is satisfactory for this purpose. The TTY service fork is restarted by the interpreter fork whenever it is needed but not running.


SYSTEM 1.85 EXECUTIVE
LISP
LISP EXECUTIVE (including DDT when called)
LISP INTERPRETER
TTY SERVICE
USER FORKS
(including FDDT, hybrid processor and subsystem forks)

LISP FORK STRUCTURE

Hybrid Processor and Fork Functions

As mentioned earlier, the fork capability in LISP provides a practical way of driving the hybrid processor. This section describes a number of functions that have been written to provide, in various formats, input to and output from the fork's memory, which is the link between LISP and the hybrid processor. All of these functions are necessarily low-level, but for display applications, a very general set of higher level functions have been written, and are described in the section of the manual entitled Display Capabilities in LISP. This latter section can be referred to for a more extended writeup on the use of the hybrid processor and fork functions in a particular context.

To run a process with the hybrid processor, one must first have a process number and appropriate devices assigned. The following functions are used for this purpose:

assignp[pno]	assigns process number <u>pno</u> for hybrid processor, returns <u>pno</u> or NIL if unable to assign.
assignd[dev]	assigns device <u>dev</u>, returns device number actually assigned or NIL if none available.
unassign[]	releases all devices and process numbers.

Before any fork operation can be performed, it is necessary to define the fork memory. This initialization is performed by

forkinit[]	initializes the fork

which sets the contents of the relabeling registers of the first fork table to specify the memory configuration shown earlier in this document. It also sets up the two constants in SPCELL+10 and SPCELL+11. This initialization should be performed only once, but is not preserved through a SYSOUT-SYSIN, i.e., if you save your system and resume via SYSIN, you must perform another forkinit.

The functions storefork and readfork described below permit transferring large blocks of data to and from any desired location in the fork's memory. However, the following conventions have been found to be extremely useful in communicating with the fork and the hybrid processor:

1. The fork's memory is divided into a number of distinct, non-overlapping areas or tables.

2. Each area is identified and referred to by its first location, called its handle.

3. The contents of handle, i.e., the first cell in a table, is a pointer to the first unused word in the table. When the table is completed and given to the hybrid processor to run, this will be the first cell after the end of the table.

4. Commands for the hybrid processor begin at location handle+1, and are contiguous. Data for the hybrid processor follows the commands, and continues to the cell whose address is contained in handle.

To repeat, it is not necessary to adhere to these conventions to use the fork capabilities, although the functions described here are designed to make it easier to use the fork capabilities with these conventions.

maketable[place;size]	creates a table and initializes its pointer. If place is NIL, the table begins at location 200 (the fork memory goes from location 0 to location 8192). If place is T, the table begins immediately following the previous table (assuming it was also created by maketable). Otherwise, place must be a number and specifies the actual starting location of the table. Size is an optional argument which, if given, guarantees that the next table, if created by a call to maketable with place=T, will not overwrite this one. Essentially, it is a device for reserving a block of memory for a table without having to fill up the table. It enables the user to initially divide up the fork memory into several tables before transferring any data.

The value of <u>maketable</u> is the handle of the table that was created.

storefork[ap,incr,handl,rel,np] transfers the first <u>np</u> words from the array specified by <u>ap</u>, an array pointer, into the fork's memory, beginning at locations <u>handl</u>+<u>rel</u>, and at every <u>incr</u> thereafter. If <u>incr</u> is not given, 1 is used; if <u>np</u> is not given, arraysize[ap] is used; if <u>handl</u> is not given, free variable <u>handle</u> is used; if <u>rel</u> is not given, the contents of <u>handl</u> is treated as a pointer to the first unused word in the table and this latter location is the starting point for the transfer. In this latter case, the pointer is updated after the transfer is complete. Value is <u>handl</u>.

Note: The error message (MACHINE SIZE TOO SMALL) means the user must return to the time sharing executive and change his machine size. The error message

(ATTEMPT TO STORE/READ BEYOND END OF FORK)

means the user has tried to reference a fork location > 8192.

Thus, to create a table and fill it with the arrays A1, A2, and A3, one must perform the following steps:

```
    MAKETABLE()
    200
    E (STOREFORK A1)
    200
    E (STOREFORK A2)
    200
    E (STOREFORK A3)
    200
```

readfork[ap,incr,handl,rel,np]	transfers *np* points from the fork's memory beginning at location *handl*+*rel*, and at every *incr* thereafter, into *ap*, an array pointer. *rel* must be given; if *incr*, *handl*, or *np* are suppressed, they are treated as for *storefork*. Value is *handl*.

Note: the arrays for *storefork* and *readfork* must be arrays that contain unboxed numbers, since no boxing or unboxing takes place in either of these two functions.

stfk[address,x]	stores a single word into the fork at *address*. *x* should be a number which is then unboxed by *stfk*. Value is *x*.
rdfk[address]	reads a single word from the fork at address, and boxed this quantity to return a number.

Note: maketable, storefork, readfork, stfk, and rdfk all contain error checks for out of bounds references and machine size too small error conditions. In addition, storefork and readfork check to make sure that np is not too large for the array ap.

hpstart[st,nt,lc,ld]

starts up the hybrid processor. st is the starting location for hybrid processor information with commands coming before data. nt is the iteration count for the data table, i.e., the number of times the data table is to be run (0=infinity). lc is the length of the command table for the hybrid processor and ld the length of the data table. If ld is NIL, st is assumed to be a handle, and the first word a pointer as described earlier. In this case, hpstart computes ld. Thus, the user does not have to keep track of how long his data table is. If ld≠NIL, st points directly to the first command.

hpstart uses pno as a free variable for the process number to be used. If a process is currently running, hpstart creates a new data block (BRS 132) and terminates the current data block (BRS 142) (see TSSS Manual). This means that running displays can be replaced

	by new displays without any flicker. If no process is running, hpstart creates both a command block and a data block and starts a process. If it is necessary to start an operation with a new set of commands or iteration count, be sure to first do an hpstop.
hpstop[]	performs a BRS 138 which stops the hybrid processor. Uses pno as a free variable.
hptest[]	performs a BRS 137 which tests the status of the hybrid processor. Returns T if running, NIL if not. Uses pno as a free variable. Warning: hptest immediately following an hpstart may not return T: there is a slight time lag before the process actually gets going.
hpwait[]	performs a BRS 136 which dismisses user until the hybrid processor is finished.

hprun[ap,lc,np,iter,pno]

drives the hybrid processor *directly* from LISP, i.e., this is the way displays were done before the fork capability was implemented. *ap* is an array which must lie on a single page, and therefore must be less than 253 in size. (note: all arrays of size less than 253 do not necessarily reside on a single page - see *hptable* below.) *lc* is the number of commands, *np* the number of data words, with commands preceding data in the array. *iter* is the iteration count for the data table and *pno* the process number. *hprun* waits for the process to terminate before returning.

hprun is primarily useful for short operations, such as reading a single A-D converter, because it does not require separate *maketable*, *storefork*, *hpstart*, and *readfork* operations - the value(s) are returned directly into the array.

hptable[m,n]

creates an array analogous to (ARRAY M N) in such a way as to guarantee that the result is on one page. M must be less than 253. Returns array as value.

brs[n,a,b,x]	loads the A-register with a, the B-register with b, and the X-register with x and does a BRS n. Value is the A-register. For example (BRS 81 N) will dismiss a program for N milliseconds.

SECTION XVIII

DISPLAY CAPABILITIES IN LISP

Introduction

At the present time the only display facility available on the BBN Research Computer is hardware which provides only a point by point display (e.g., no character or vector generators in hardware). Control of this display processor is achieved through the hybrid processor attached to the SDS 940. The display itself has no storage capability, therefore the image must be constantly refreshed by the hybrid processor from data stored in the computer's core memory. To enable LISP programs to make efficient use of this display, functions have been written which provide a fairly sophisticated and general display language along with low-level routines for communicating with the CRT via the hybrid processor. We expect that later hardware additions to the display will make running the display more efficient, but may require a few changes in the user programs which describe and construct displays.

Putting a display on the face of the CRT from within LISP is done in three distinct steps. The first of these involves constructing a display <u>structure</u> which specifies the points to be displayed. The second step is then to transfer the data corresponding to these points from LISP's memory into the fork's memory, (see Section XVII), from which it will be displayed. The third step is to initiate a program within the fork which starts up the hybrid processor; the hybrid processor will then maintain the display on the face of the screen independent of what is taking place in the user's LISP program.

The major effort in providing a display capability in LISP has been concentrated on the first step of the above operation, the generation of a display structure. Most users need not even be aware of the details of steps two and three, beyond the fact that they must be done, and that some straightforward functions are provided to do them. For the interested, Section XVII discusses in greater detail the use of forks and the hybrid processor. The latter part of this section discusses the details of implementation of steps 2 and 3 above, and describes the operation of the lower level functions.

Initialization

In order for the user to actually display figures on the CRT, he must acquire for his use two D-A converters and a process number. The function start is available for this purpose. It assigns process number 1, and the two D-A converters in the list converterlst initially set to (0 1). Note: in most cases, the converters will be patched to the appropriate inputs, and the user can simply use start[]. However, in the event that the last person used the D-A converters for an application other than display, it may be necessary for the user to repatch the D-A converters or to find someone who can.

Start calls start1 which performs all initialization other than the assigning of process number and devices. If the user wishes to use the functions in the display package, but does not intend to actually display, he can perform start1[], and then proceed exactly as if he had the scope assigned to him. Function calls that would normally start a display will simply print DISPLAYING on the teletype and continue. In this way several users can debug programs that use the display package, even though only one can be displaying at a time.

The Display Language

The display structure of step 1 is defined via the display language. This language provides an aesthetic way of describing displays. It revolves around the concept of a figure, which in our terminology is a display gestalt. A figure can be composed of other figures, and, in turn, be a part of a larger figure, much the same as a list can be composed of sublists, and be a part of a larger list. The interpretation of a figure, and consequently the display it produces, depends on its type. Three of the more common figure-types are:

1. Translate: if F is a figure, then (F X Y) is a figure consisting of F translated X units in the horizontal direction and Y units in the vertical direction.

2. Scale: if F is a figure, (F S) is a figure consisting of F scaled by a factor of S, i.e., (F 1)≡F. S can be a positive or negative, fixed or floating point number.

3. Combine: if F_1 ... F_n are figures then (F_1 ... F_n) is a figure consisting of the union of (the points of) the individual figures F_1 through F_n.

A primitive figure is an array consisting of X and Y coordinates in alternation. For example, the array containing the values (0 0 5 0 10 0 ... 45 0) specifies a horizontal line, 10 points long, starting at (0 0) and ending at (45 0). If H is such a line, and V a vertical line from (0 0) to (0 45), then the figure (H V (H 0 50) (V 50 0)) represents a square. Using SQ for square, we can define ROW as (SQ (SQ 50 0) (SQ 100 0) (SQ 150 0) ... (SQ 350 0)), and then CHECKERBOARD as (ROW (ROW 0 50) (ROW 0 100) ... (ROW 0 350)).

The principal advantage of such a language is that it lends itself nicely to the recursive nature of LISP and list processing. Large, complicated displays can be conveniently broken down into small, subroutine-size chunks. A secondary advantage is the reduction in storage required for displays. For example the primitive figures H and D each require 20 array cells, SQ requires 10 LISP words, and ROW, and CHECKERBOARD an additional 29 LISP words. Therefore, the entire checkerboard requires 68 LISP words and 40 binary program words, and specifies a display of 2560 points.

In addition to the three figure-types described above, the following figure-types are implemented in the display language:(*)

4. Scale: If F is a figure, (SCALE: F X Y) is a figure consisting of F scaled by X in the horizontal direction and Y in the vertical direction. (SCALE: F S S) is identical to (F S), a figure of type 2. However, figure type 4 permits individual X and Y scaling. Note: reflection can be achieved by using a positive scale factor for one coordinate and a negative scale for the other.

5. Plot: If A is a primitive figure, i.e., an array, then (PLOT: A) is a figure consisting of the values of A plotted as Y coordinates starting at X=0 with X advanced by 1 for each value of Y, i.e., the normal horizontal

(*) An atom can be used to represent a complex figure; it will be evaluated and its value treated as one of the 11 figure types discussed here.

graph. Figure type 5 can be combined with type 1 and type 4 to produce a graph at any position, any scale. Note: values of A are treated as unboxed numbers.

Other variations: (PLOT: A T) is a figure consisting of the values of A plotted as X coordinates starting at Y=0 with Y advanced by 1 for each value of X, i.e., a vertical graph.

6. Erase: (ERASE: M N) is a pseudo-figure, i.e., it does not itself transfer any points, but modifies previously transferred points. Its effect is to erase N words * starting from the M^{th} word in the top level superfigure, M and N > 0. For example, since ROW consists of 320 points or 640 words, the figure (CHECKERBOARD (ERASE: 1280 640)) would consist of a checkerboard with the third row erased.

7. Restore: (RESTORE: M N) is a pseudo-figure which reverses the action of an ERASE pseudo-figure, i.e., (F (ERASE: M N) (RESTORE: M N)) is equivalent to F.

8. Move: If F is a figure, (MOVE: F A_{11} A_{12} A_{21} A_{22}) is a figure which is the linear transformation of F specified by the matrix $\begin{pmatrix} A11 & A12 \\ A21 & A22 \end{pmatrix}$, Aij positive or negative, fixed or floating point number.

9. Rotate: If F is a figure, (MOVE: F θ) is a figure consisting of F rotated θ degrees. (MOVE: F θ) is identical to (MOVE: F COSθ SINθ -SINθ COSθ).

* 1 point = 2 words.

10. If A is an array, (A . N) is a primitive figure consisting of the first N points of A. This type of figure can be used anywhere a primitive figure can appear. Note that a figure of the form (F . N) where F is not a primitive figure, i.e., ((A 100 0) . N), is not permitted.

11. Label: If F is a figure, then (LABEL: F label) is a figure that generates the same display as F. The purpose of this figure-type is to facilitate modification of selected subfigures in a large and complicated superfigure. After the Label: figure type has been interpreted and transferred into the fork its absolute X and Y coordinates, absolute X and Y scaling, relative position in the display table, and number of words it occupies in the display table are attached (by rplacd), in that order, following the label.

New figure types may be defined by adding a definition of the form (name form1 ... formn) to the list displaymacros. Before assuming that a figure is of type 1, 2, or 3, this list will be searched for a definition using assoc and car of the figure. If such a definition is found, form1 through formn are evaluated. For example, one could define a figure type SHRINK: by adding to displaymacros

```
(SHRINK: (DISPLIST1 (CADR FIG) X Y
       (FQUOTIENT SCALEX (CADDR FIG))
       (FQUOTIENT SCALEY (CADDR FIG)))) *
```

* displist is discussed on page 18.15.

Generating Functions

A generating function is one that constructs a display figure. Several are included in the display package.

dline[x;y;dx;dy;n]	value is a primitive figure representing a line n points long starting at (*x*,*y*) with increments (*dx*,*dy*).

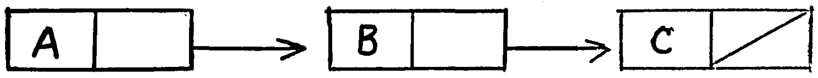
Thus from our checkerboard example, H=(DLINE 0 0 5 0 10), and V=(DLINE 0 0 0 5 10).

dvector[x0,y0,x1,y1,n]	value is a primitive figure representing a line N+1 points long starting at (x0,y0) and ending at (x1,y1). Note: *dline* is more efficient than *dvector*.
dcircle[radius, dtheta]	generates a circle of radius *radius* at [0,0]. *dtheta* represents the arc between points on the circle, i.e. 360/dtheta = number of points. Value is primitive figure.

Note: both *dline*, *dvector* and *dcircle* take two extra optional arguments *a*, and *m*. If given, the array *a* is used for the primitive figure, starting with postion *m* of the the array. In this way more than one primitive figure can be generated into the same array.

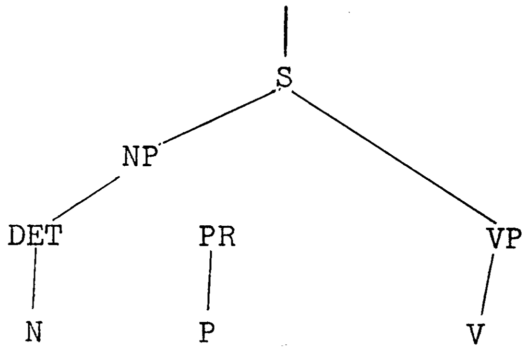
ds[s] — generates (and displays) a figure for the list structure S in conventional box-notation. Its value is a generated symbol whose value is the figure itself.

For example, DS((A B C)) will produce the display



ds is equipped to handle circularities in both car and cdr direction. Warning: since it marks each substructure to detect circularities, and subsequently restores the original structure, interrupting ds by rubout will cause the original list structure to be permanently lost.

dtree[tree] — generates (and displays) a figure for the tree structure representation of tree, i.e., car of each sublist labels the father node, cdr is treated as a list of the daughter nodes. For example, dtree[(S (NP (DET N) (PR P)) (VP V))] will produce the display



The value of dtree is a generated symbol whose value is the display figure. dtree is equipped to handle circularities in the car direction, i.e., common subtrees. Since it marks each subtree as it encounters them, and subsequently restores them, interrupting it by rubout will cause the original structure to be permanently lost.

The variable trnpts initially set at 20, specifies the distance between adjacent levels of the tree. The variable trspacing, initially set at 3, specifies the minimum distance between any two adjacent nodes on the same level. If the width of the generated tree is greater than the width of the scope, the tree is automatically scaled down in the horizontal direction. No scaling is performed in the vertical direction. However, deeper trees can be accommodated by resetting trnpts to a smaller value.

movefig[fig;opl;parl;...opn;parn]

is an NLAMBDA nonspread function. It evaluates _fig_, but treats the rest of its arguments literally as operations and parameters, e.g.,

```
(MOVEFIG FIG UP 100 LEFT 100 ROTATE 30).
```

Its value is a new figure corresponding to _fig_ with the indicated operations having been performed. For the above example, this would be

```
(MOVE: ((F 0 100) -100 0) 30)
```

movefig is designed to free the user from remembering the various conventions of figure types 1 through 9. It recognizes UP, DOWN, LEFT, RIGHT, SCALE, SCALEX, SCALEY, ROTATE, MOVE and LABEL. For MOVE, the parameter should be the matrix of transformation.

Displaying Text

A number of generating functions are available for character strings: dprint, dprin1, dprin2, dspaces, dterpri, datom, and dischar. The first five perform functions analagous to the printing functions of the same name without the d. datom and dischar are lower level routines used by dprin1, but when called directly provide certain options not available through the higher level functions. These functions are designed to allow the user to treat the scope as a teletype, if he so desires.

The size of the characters these functions generate is determined by the free variable charsize, initially set at 2. charsize is the spacing between points in a character. Since there are approximately 100 points per inch, and characters are 5x7 points, a character generated with charsize=2 would be .1"x.14", which is approximately teletype sized.

The free variables xorg and yorg determine the next character to be displayed, and correspond to the teletype position. They are numbers between -512 and 512, with (0,0) the center of the scope. xorg is adjusted by the above functions in accordance with the horizontal motion of the teletype carriage, yorg in accordance with the vertical motion of the paper. Their initial value determines the position of the first character. Their final value corresponds to the position on the scope where "printing" stopped.

The free variables lorg and rorg determine the left and right "margins" of the display. Whenever a word (an atom) would be positioned at a point to the right of rorg, i.e., when xorg is greater than rorg, dterpri is called first. This function performs a scope "carriage return" by resetting xorg to lorg and moving <u>yorg</u> down by 10*charsize. Similarly, the function <u>dspaces</u> "spaces" the display by changing <u>xorg</u>, but does not itself participate in the construction of any figures.

To complete the analogy with printing, any carriage return or blanks that appear inside of an atom will have the obvious interpretation. Furthermore, <u>dprint</u> and <u>dprin2</u> would cause this atom to be displayed in double quotes, while <u>dprinl</u> would not.

<u>dprinl</u> has one additional feature not available on the teletype: subscripting and superscripting. The characters control-U and control-D have the effect of displacing <u>yorg</u> up or down by 5*charsize. Thus the atom

"X<u>U</u>2<u>D</u> + Y<u>U</u>2<u>D</u> = Z<u>U</u>2<u>D</u>"

where <u>U</u> and <u>D</u> denote control-U and control-D respectively would be displayed as

$$X^2 + Y^2 = Z^2$$

It is important to emphasize that all of these functions do not display, but <u>merely</u> <u>generate</u> <u>figures</u>. For example, for <u>charsize</u>=2, <u>xorg</u>= -400, and <u>yorg</u>=400, the value of datom[ABC] is the list (($(a_1$ 2) -400 400) ((a_2 2) -386 400) ((a_3 2) -372 400)), where a_1, a_2, and a_3 are primitive figures for the characters a, b, and c respectively. This figure can be repositioned, rescaled, combined, and transformed the same as any other figure. It is not displayed until it is transferred into the fork, step 2, and the hybrid processor is started, step 3.

dprint[x]	performs dprin2[x] followed by dterpri[] and returns the value of dprin2
dprin2[x]	dprinl[x t]
dprinl[x;prin2]	If *xorg* is greater than *rorg*, performs dterpri[]. Successively calls *datom* and *dspaces* on the components of x and on *lpar*, *rpar*, and *period* as required, and combines the values into a figure, which is returned as its value. The second argument of *dprinl* is used as the second argument to *datom*.
dterpri[]	sets *xorg* to *lorg*, *yorg* to *yorg*-10*charsize.
dspaces[n]	sets *xorg* to *xorg*+n**charsize**7
datom[atom;prin2;x;y]	generates a figure for atom at coordinates (x,y) if given, otherwise at (xorg,yorg). It uses chcon[atom;prin2] for the list of characters to be displayed. Thus if prin2=T, and *atom* is unusually spelled, double quotes will be supplied. *datom* recognizes and treats specially the character codes for blank, carriage return,

line feed, control U and control D. Otherwise, it calls dischar.

dischar[c]

c is a number used to reference the array masktable which contains 35 bit masks for all of the teletype characters. These masks are formed by mapping the 5x7 image of the character into a string of 35 bits by starting at the lower left hand corner of the character and proceeding bottom to top, left to right, with a 1 indicating a point to be displayed. The first 21 bits, i.e., the leftmost three columns of the 5x7 image, are left justified to form the first word of the mask. The last 14 bits, also left justified form the second word. These are stored in location 2c-1 and 2c in masktable. For example, location 103 in masktable is 00403770Q, location 104 is 0040200Q, corresponding to the mask for the character T, which has ASCII code 52.

The first time a particular character is encountered, a primitive figure is generated for it, and stored in the array chartable. Subsequent use of this character will not require regeneration. Thus, the generation of figures for text is essentially a table of lookup process.

Note: arraysize[chartable]=128, and not all of these are taken by existing symbols, so the user can define new symbols by placing an appropriate mask in masktable via seta.

Decoding Figures

Step 2 of producing a display consists of transferring the data corresponding to the points in a display figure into the fork's memory. This is done by the function displist.

displist[fig;place;handle]	if handle=NIL, maketable[place] is called followed by storefork[ca] which transfers into the fork the commands necessary to display points. Then displistl is called to decode fig. The value of displist is handle.
displistl[fig;x;y;scalex;scaley]	performs the decoding of fig.

The normal way of using displist is to call it with place=NIL or T. In the first case, the display table will begin at the lowest location in the fork's memory. In the second case, it will begin at the first location after the previous table. In both cases, the value of displist is the information that will be required by drun, the function that starts the display. For more details, see the description of maketable and the discussion of handles in Section XVII.

If displist is called with handle not equal to NIL, maketable and storefork are not called. This is one way to add a figure to a previously existing table. For example, displist[fig1] followed by displist[fig2;NIL;x] where x is the value of the first call to displist, i.e., the handle, is equivalent to displist[list[fig1;fig2]], since list[fig1;fig2] is a figure of type 3, combine.

Driving the Display

Once the points have been transferred into the fork, the user can start the display by calling the hybrid processor and specifying appropriately the number of commands, number of points, number of iterations, etc. This is performed by the function drun.

drun[handl;nt]	If handl is NIL, handle is used. handle has a top level binding of 200, the first available location in the fork's memory and the value of displist when place=NIL. If nt is NIL, 0 is used which is interpreted as infinity by the hybrid processor. If start [] had been called, drun starts the display, otherwise it prints DISPLAYING.

display[fig]	Executes displist[fig], followed by <u>drun</u>, and returns a <u>gensym</u> whose value is fig. For example, E (DISPLAY (DATOM (QUOTE TESTING))) will generate the figure for this atom, transfer it into the fork, start the display, and return a <u>gensym</u> whose value is the figure. This can be used for subsequent calls to <u>displist.</u>

It is important to emphasize that <u>displist</u> only transfers points into the fork, and <u>drun</u> only calls the hybrid processor. One can execute several calls to <u>displist</u> before displaying anything and then switch rapidly back and forth from one display to another by calling <u>drun</u> with different values for its first argument. Similarly, one can execute a <u>displist</u> and a <u>drun</u>, and then be performing another <u>displist</u> while the display is running. If this latter <u>displist</u> should happen to overwrite the display table for the first one, the new points will be seen as soon as they are transferred, and the entire display will seem to melt into the new one.

Low-level Functions

The preceding discussion has presented all of the user-level display functions. These will be sufficient for most users and most applications. The following sections will describe the lower level functions, and the details of the present implementation. To make effective use of these functions, the programmer should have a greater knowledge of the computer hardware, time-sharing system, and LISP implementation. Furthermore, changes in the display hardware will result in changes, additions, and deletions to these functions. That is, programs which use these functions directly will probably be affected by changes in hardware or low-level software. Programs which use the higher-level functions will not.

Hybrid Processor and Fork Functions

If the reader has not already done so, at this point the portion of Section XVII entitled 'Hybrid Processor and Fork Functions' should be read. Briefly, this section discusses the use of the fork capability in LISP for driving the hybrid processor and the conventions for communicating with the fork. In particular, the reader should review the organization of the fork's memory into tables, and the use of the function maketable. The functions assignp, assignd, unassign, forknit, hpstart, hpstop, hptest, hpwait, and storefork are also of interest for display applications.

Storepoints

Storepoints, like storefork, is a function for transferring data from LISP's memory to the fork's memory. It takes among its arguments ap, handl, rel, and np and treats them similarly to storefork: transferring the first np words of ap into the fork's memory beginning at location handl+rel+3. The 3 reflects the presence of the pointer in the handle and the two command words. Therefore, rel is relative to the first point in the table, not the first word. If np is not given, arraysize[ap] is used; if handl is not given, the free variable handle is used; if rel is not given, the contents of handl is treated as a pointer to the first unused word in the table and this latter location is the starting point for the transfer. In this latter case, the pointer is updated after the transfer is complete. The value of storepoints, like storefork, is handl.

However, storepoints, unlike storefork, expects that its array contains the x and y coordinates, in alternation, of points to be displayed, and operates in a pair-wise fashion. It also includes among its arguments several options specifically tailored for display applications:

storepoints[ap;dx;dy;sclx;scly;handl;rel;np]	transfers np or arraysize[ap] words from ap into fork. dx and dy are the translations and sclx and scly the scale factors for the x and y coordinates, respectively. storepoints moves each pair of words into the fork's memory, at the same time multiplying by sclx, or scly, and

adding dx or dy, and converting to the format required by the hardware. If dx or dy are not given, 0 is used. If sclx or scly are not given, the free variable scalex or scaley are used. These are initially set to 1. sclx or scly may be positive or negative, fixed or floating point numbers. The elements in ap should be LISP numbers between -512 and 512, and are unboxed and shifted appropriately before being transferred.

Another option available in storepoints allows blanking out and subsequently restoring selected points. This is achieved by giving storepoints as its first argument, NIL, for erasing, or T for restoring (instead of an array). Starting at the appropriate location, the appropriate number of points will then have their low order bits set to 1, if erasing, or set to 0, if restoring. When the "blink" switch for the scope is on, points with low order bits set to 1 are not displayed. If the display were being run at the time of the operation of storepoints, these points would instantaneously disappear or reappear. Similarly, the effect of a moving display can be created by overwriting portions of a table while it is being displayed. Note that appending to a table, i.e., calling storepoints with rel=NIL, can never cause an immediate change in a display. This is because the call to the hybrid processor which starts a display must, in advance, specify how many points are to be displayed. Any additions to the table will be seen only when another drun executed. The only way to achieve the effect of points spontaneously appearing in a display while it is in progress is to initially store "invisible" points in the display, i.e., points with low order bit set to 1, and then overwrite these or turn them back on while the display is running.

Plotting Graphs

Many graphs can be specified more efficiently than by a collection of x-y coordinates as required by storepoints. It is often possible to simply give a sequence of values for one axis, and the first value and increment between successive values for the second axis. This reduces by half the amount of storage required to represent the graph. The function plotarray is provided for this purpose.

plotarray[ap;dx;dy;sclx;scly;vert;handl;rel;np]

transfers np or arraysize[ap] words from ap into twice that many words in the fork's memory. The interpretation of handl and rel is the same as for storepoints. The numbers in ap are interpreted as y-coordinates, unless vert=T. The value of the other coordinate starts at 0 and is advanced by 1 for each point. dx and dy are the translations and sclx and scly the scale factors for the x and y coordinates so that any positioning or scaling can be achieved. If dx or dy are not specified, 0 is used. If sclx or scly are not given, scalex and scaley are used. If the ap contains all boxed numbers, plotarray will unbox them.

plotarray is used by displist for figures of the form (PLOT: --).

Moving Points

The function movepoints described below provides an alternate, more efficient way of effecting linear transformations of two dimensions such as rotation and shearing. This entails performing modifications on the corresponding points *after* they have been stored into the fork's memory, rather than modifying a LISP array or several arrays and then storing the modified arrays into the fork using storepoints.

movepoints[matrix;dx;dy;handl;rel;np;from]

If handl is not given, handle is used. If rel, dx, or dy is not given, 0 is used. Modifications begin at handl+rel+3 and proceed through np words, if np is given, otherwise through entire table, i.e., up to the location specified by the pointer in handl.

matrix is a list of the form $((a_{11}\ a_{12})\ (a_{21}\ a_{22}))$ where a_{ij} are the elements of a two by two matrix, and may be positive or negative, fixed or floating point numbers. For each pair of x-y coordinates, the new value of x is given by

$$x' = a_{11}*x + a_{12}* y + dx$$

and the new value of y by

$$y' = a_{21}*x + a_{22}* y + dy$$

The state of the low order bit of x and y is not disturbed so that invisible points are transformed along with visible ones but remain invisible.

If matrix=NIL, ((1 0) (0 1)), the identity matrix, is used.
If matrix=N, a number,

((COS N -SIN N) (SIN N COS N))

is used, i.e., the effect of the transformation would be to rotate the points N degrees.

If from is given, the old values of x and y are taken from the table whose handle is from. The new ones are stored in the table specified by handl. Essentially this allows you to move from one table to another and to perform a transformation if desired.

Movepoints is used by displist for figures of the form (MOVE: --).

SECTION XIX

ADVISING

The operation of advising gives the user a way of modifying a function without necessarily knowing how the function works or even what it does. Advising consists of modifying the interface between functions as opposed to modifying the function definition itself, as in editing. break, trace, breakdown, and follow are examples of the use of this technique: they each modify user functions by placing relevant computations between the function and the rest of the programming environment.

The principal advantage of advising, aside from its convenience, is that it allows the user to treat functions, his or someone else's, as "black boxes," and to modify them without concern for their contents or details of operations. For example, the user could modify sysout so that it did not write any new files, i.e., files that did not already appear in this file directory. This could be done by:

```
ADVISE(SYSOUT (COND
          ((INPUT (INFILE U)) (CLOSEF U))
          (T (PRINT (CONS U (QUOTE (NOT FOUND))) T)
             (RETURN  NIL)))
```

As with break, advising works equally well on compiled and interpreted functions. Similarly, it is possible to effect a modification which only operates when a function is called from some other specified function, i.e., to modify the interface between two particular functions, instead of the interface between one function and the rest of the world. This latter feature is especially useful for changing the internal workings of a system function.

Consider the following obscure bug in prettyprint (which has since been fixed): if a prog had two labels for the same statement, and the first label had more than four characters in it, no spaces would be printed between the two labels. Consequently, when the function was loaded back in, only one label would be read: the concatenation of the two labels that were printed. This condition could have been remedied by:

```
ADVISE ((SPACES IN PRINTPROG)
     (COND ((ZEROP U) (SETQ U 1))))
```

Advice can also be specified to operate after a function has been called, in which case the value of the function is bound to the variable VALUE, as with BREAK. For example, execution of all commands to the LISP editor is performed under an errorset in the executive editor function edite. Frequently the user may type in a long command with a small error in it, and would prefer to be able to correct the command rather than having to retype it. The user could modify the editor to automatically save a command whenever an error occurred during its execution by performing

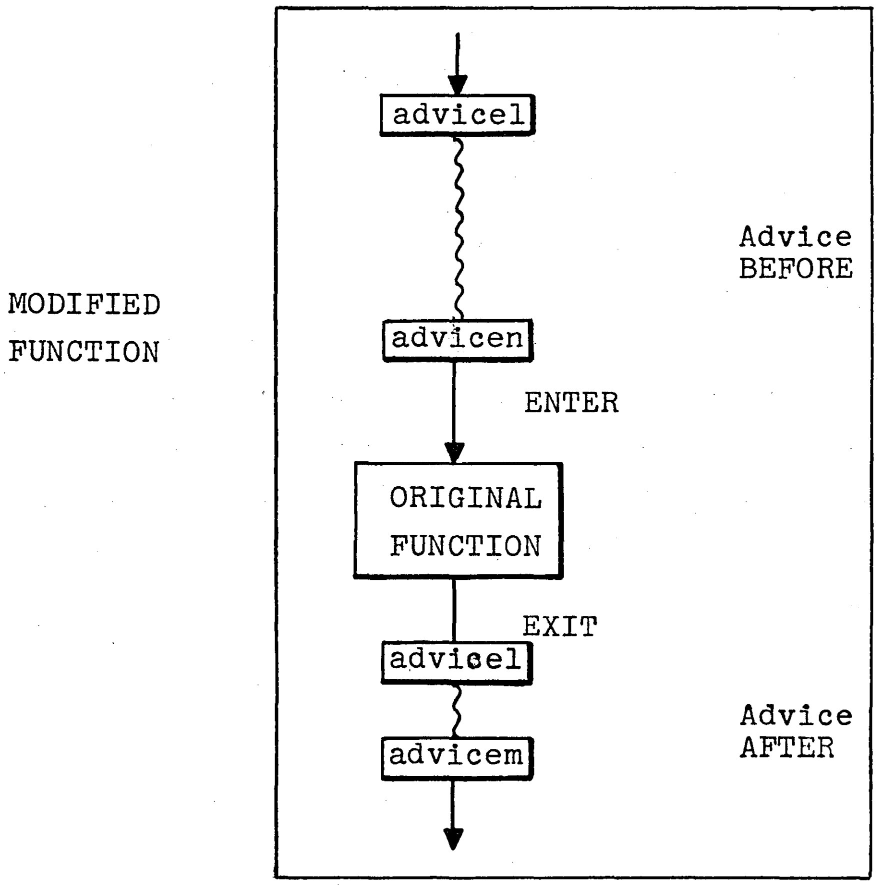
```
ADVISE ((ERRORSET IN EDITE) AFTER
             (COND ((NULL VALUE) (SETQ LASTCOM C))))
```

since the value of errorset is NIL if and only if an error occurs in the evaluation of its argument.

Note that advising spaces or errorset would have affected all calls to these very frequently used functions, whereas advising (SPACES IN PRINTPROG) and (ERRORSET IN EDITE) only affects calls to spaces from printprog or errorset from edite.

Advise

Advise is a function of four arguments: name, when, where and what. name is the function to be modified by advising, what is the modification, or piece of advise. when is either BEFORE or AFTER with the obvious interpretation. (If not given, BEFORE is assumed.) WHERE is optional and can be used to specify exactly where in the list of advice statements the advise is to be placed, e.g., FIRST or (BEFORE (ADVICE CONTAINING PRINT)), or (AFTER 3), meaning after the third piece of advice, or even (FOR ALL (ADVICE CONTAINING RETURN)). If where is specified, advise calls the editor to find and insert the advice at the appropriate location. Otherwise, the advice is inserted after any previous modifications. The structure of a function after it has been modified several times by advise is given in the following diagram:



The corresponding LISP definition is:

```
          (LAMBDA arguments (PROG (VALUE)
               (SETQ VALUE  (PROG NIL
                     advicel
                        .
                        .
                        .
                     advicen
                      (RETURN fn  arguments))))
               advicel
                  .
                  .
                  .
               advicem
                (RETURN VALUE)))
```

where _fn_ is the name of the function (generated by _advise_) which now contains the original, unadvised definition.[*]

Note that the structure of a function modified by _advise_ allows a piece of advice to bypass the original function by using the LISP function RETURN. For example, if the LISP form (COND ((ATOM X) (RETURN Y))) appeared among the advice BEFORE a function, and this function was entered with X atomic, Y would be returned as its value, i.e., VALUE would be set to Y, and control passed to the advice, if any, to be executed AFTER the function. If this same piece of advice appeared AFTER the function, Y would be returned as the value of the entire advised function.

[*] _fn_ is stored on the property list of the function _name_ under the property ADVISED.

The advice (COND ((ATOM X) (SETQ VALUE Y))) AFTER the function would have a similar effect but the rest of the advice AFTER the function would still be executed.

advise[name,when,where,what]

name is the function to be advised, when=BEFORE or AFTER, where specifies where in the advice list the advice is to be inserted, and what is the piece of advice. Both when and where are optional arguments, in the sense that they can be omitted in the call to advise. In other words, advise can be thought of as a function of two arguments: [name,what], or a function of three arguments: [name,when,what], or a function of four arguments: [name, when,where, or what]. Note that the advice is always the last argument.

If name is of the form (fn1 IN fn2), chngnm[fn2,fn1] is first performed as with break, and then fn1-IN-fn2 is used in place of name.

If name is non-atomic, every function in name is advised with the same values (but copied) for when, where, and what.

If name is broken, it is unbroken before advising.

If name is not defined, an error is generated.

If name is being advised for the first time, an appropriate S-expression definition is created, and the original definition stored on a gensym, and the gensym stored on the property list of name under the property ADVISED.

name is added to the list advisedfns.

The modification is inserted in the appropriate position indicated by where in the list of advice either BEFORE or AFTER the function depending on when. If where=NIL, the advice is added to the end of the advice. If where=FIRST or TOP, it is inserted in front of the advice. Otherwise, where is treated as a location command for the expanded editor, e.g., (BEFORE 3), (AFTER (ELEMENT CONTAINING PRINT)).

(when where what) is added to the front of a list of all calls to advise for name which is kept on the property ADVICE.

Value of advise is name.

unadvise[x]	is a non-spread NLAMBDA a la *unbreak*. It takes an indefinite number of functions to be restored to their unadvised state. unadvise[] or unadvise[ALL] will cause all functions on *advisedfns* to be unadvised. unadvising consists of restoring the original definition, removing the properties ADVISE and ADVISED from the property list, unbreaking if the function is also broken, and if the function is an alias, i.e., created by an advise (fnl IN fn2) call, the higher level function in which it appeared is also restored.
advisedump[x]	advisedump is the function that is called when an expression of the form (ADVISE fnl ... fnm) appears in the third argument to *prettydef*. If car[x] is not atomic, eval[car[x]] is used instead of x. Then for each function on x that has a property ADVICE, the reverse of the property value is put on the property READVICE. Two deflists are written: one for every function that has a property ALIAS, and one for every function with property READVICE.

readadvise[name] — is designed to be used in conjunction with _advisedump_ and _prettydef_ for dumping advised functions and then loading and restoring them to their advised state. If _name_ is of the form (fn1 IN fn2) or there is a property ALIAS on the property list of _name_, the appropriate _chngnm_ if first performed. _readvise_ then calls _advise_ for each modification on the list stored under the property READVICE.

The value of _readvise_ is _name_.

Note: if a function has both the property READVICE and the property ADVICE, _unadvise_ will first move the reversal of the property value of ADVICE onto READVICE before it removes the former. Thus if the user readvises, then executes additional calls to advise, and then unadvises, all the advice would still be dumped when the function was prettydef-ed.

In summary, _advise_ puts advice on the property ADVICE. _advisedump_ takes it from ADVICE, or else from READVICE. _readvise_ uses the advice on READVICE. _unadvise_ removes the property ADVICE but first moves ADVICE to READVICE provided both properties are present. No function removes READVICE.

cadvice[fns]

fns is a list of advised functions to be compiled with their advice. cadvice performs the appropriate modifications to their property lists before and after calling the compiler. After compilation, the function can still be unadvised, in which case the compiled code will be lost. If the function has a property EXPR this property value will be preserved through the compilation.

SECTION XX

AUTOMATIC ERROR CORRECTION IN LISP

Introduction

A surprisingly large percentage of the errors made by LISP users are of the type that could be corrected by another LISP programmer without any information about the purpose or application of the LISP program or expression in question, e.g. misspellings, certain kinds of parentheses errors, etc. We have implemented into the BBN LISP system a DWIM package, short for Do-What-I-Mean, which is designed to facilitate the correction of these type of errors. DWIM is called automatically whenever an error occurs in the execution of a LISP program (provided the user has first enabled this feature), and then proceeds to try to correct the mistake. The following output is representative of the kinds of corrections the program will handle.

```
←DEFINEQ((FACT (LAMBDA (N)
(COND ((ZEROP N9 1) ((T (TIMS N (FACTT 8SUB1 N]
(FACT)
←PRETTYPRNT((FACCT]
=PRETTYPRINT
=FACT

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N9 1)
        ((T (TIMS N (FACTT 8SUB1 N))))))))
NIL
←FACT(3)
EDITING FACT ...
N9 >>---> N)
EDITING FACT ...
(COND -- ((T --))) >>---> (COND -- (T --))
TIMS=TIMES
FACTT=FACT
EDITING FACT ...
8SUB1 >>---> (SUB1
6

←PRETTYPRINT((FACT))

(FACT
  (LAMBDA (N)
    (COND
      ((ZEROP N)
        1)
      (T (TIMES N (FACT (SUB1 N)))))))
NIL
←
```

In this example, the user first defines a function FACT, of one argument, N, whose value is to be N factorial. The function contains several errors: TIMES and FACT have been misspelled. The 9 in N9 was intended to be a right parenthesis but the teletype shift key was not depressed. Similarly, the 8 in 8SUB1 was intended to be a left parenthesis Finally, there are two left parentheses in front of the T that begins the second clause in the conditional, instead of the required one.

After defining the function FACT, the user wishes to look at its definition using PRETTYPRINT, which he unfortunately misspells. Since there is no function PRETTYPRNT in the system, an UNDEFINED FUNCTION error occurs, and the DWIM program is called. DWIM invokes its spelling corrector, which searches a list of functions frequently used (by _this_ user) for the best possible match. Finding one that is extremely close, DWIM proceeds on the assumption that PRETTYPRNT meant PRETTYPRINT, informs the user of this, and calls PRETTYPRINT.

At this point, PRETTYPRINT would normally print (FACCT NOT PRINTABLE) and exit, since FACCT has no definition. This is _not_ an error condition, so DWIM would not be called by LISP. However, it is not what the user _meant_.

In order to handle these type of situations, when the user first enables the DWIM facility, ADVISE is called to modify selected system functions. For example, PRETTYPRINT is advised that when given a function with no definition, it should call the spelling corrector. Similarly, DEFINE is advised to add the names of any new functions defined by the user to the spelling list of user functions. Thus, with the aid of DWIM, PRETTYPRINT is able to determine that the user wants to see the definition of the function FACT, and proceeds accordingly.

The user now calls his function FACT. During its execution, five errors are generated, and DWIM is called five times. At each point, the error is corrected, a comment made of the action taken, and the computation allowed to continue as if now error had occurred. Following the last correction, 6, the value of FACT(3), is printed. Finally, the user prints the new, now correct, definition of FACT.

In this particular example, the user was shown operating in a mode which gave the DWIM system the green light on all corrections. Had the user wished to interact more and approve or disapprove of the intended corrections at each stage, he could operate in a different mode. Or, operating as shown above, he could have at any point aborted the correction; or signalled his desire to see the results of a correction after it was made by typing a ? on the teletype.

Each different user may want to operate with a different "confidence factor," a parameter which indicates how sure DWIM must be before making a correction without approval. Above a certain user-established level, DWIM makes the correction and goes on. Below another level, DWIM types what it thinks is the problem, e.g., PRTYPNT = PRETTYPRINT ?, and waits for the user to respond. In the in-between area, DWIM types what it is about to do, pauses for about a second, and if the user does not respond, goes ahead and does it. The important thing to note is that since an error has occurred, the user would have to intervene in any event, so any attempt at correction is appreciated, even if wrong, as long as the correction does not cause more trouble than the original to correct. Since DWIM can recognize the difference between trivial corrections, such as misspellings, and serious corrections, such as those involving extensive editing, bad mistakes are usually avoided. When DWIM does make a mistake, the user merely aborts his computation and makes the correction <u>he would have had to anyway</u>.

Enabling DWIM

To enable the DWIM package, perform DWIM[T]. DWIM will type "SET MODE B,N, OR E", for BEGINNER, NOVICE, or EXPERT. (These will be explained later.) Respond by typing either B, N, or E; DWIM will complete the rest of the mode name.* To disable DWIM, perform DWIM[]. This is guaranteed to return the system to a pristine state.

Error Correction

Once DWIM has been enabled, errors that normally generate either UNBOUND ATOM or UNDEFINED CAR OF FORM messages (undefined function call from interpreted code), instead call the functions fixatom and fixfn, respectively. If an attempted correction is aborted by typing control-R, or if the user indicates to DWIM not to proceed with the correction, or if DWIM cannot fix the error, the action taken is exactly as though DWIM had not been enabled, i.e. the system goes into a break, as described earlier in the manual.

The general strategy followed by both fixatom and fixfn, is to return the correct S-expression for the desired evaluation and continuing the computation, and also to repair the cause of the error. DWIM in general is more cautious about making corrections to user functions than making corrections to S-expressions typed into evalquote or to break.

Unbound Atoms

Fixatom is currently programmed to handle six different types of unbound atom errors.

1. If the first character of the atom is an 8, DWIM assumes that the 8 was intended to be a left parentheses, and calls the editor on the expression in which the atom appeared.

* Alternatively, call DWIM with mode as argument, e.g. DWIM[E].

It is assumed that the user did not notice the 8, or attempt to correct for it. In other words, the user typed in the same number of right parentheses and in the same places, as he would have had the 8 actually been a left parentheses. For example, (LAMBDA (N) 8COND ((ZEROP N) 1) (T (TIMES N (FACT (SUB1 N]. If the unbound atom did not appear in another expression, e.g. if the user typed 8CAR X) into break for evaluation the appropriate expression is obtained and evaluated.

2. If the first character of the unbound atom is ' DWIM assumes that the user (intentionally) typed 'ATOM for (QUOTE ATOM) and makes the appropriate change. If the unbound atom is just ', DWIM assumes that the user typed '(LIST) for (QUOTE (LIST)) and proceeds accordingly.

3. If the last character in the atom is 9, DWIM assumes the 9 was intended to be a right parentheses and operates in a manner analagous to case 1.

4. If 8 appears as a character inside the atom, DWIM assumes the 8 was intended to be a left parentheses, splits the atom into two parts, and takes the appropriate corrective action, e.g. (CONS X8CDR Y].

5. If the unbound atom is the name of a low level function, e.g. APPEND, CONS, SUBST, etc., (anything defined before PRETTYDEF) DWIM assumes a left parentheses was omitted before the atom, but that the corresponding right parentheses was typed. In general, for parentheses errors, DWIM always assumes the error was undetected by the user and that no attempts were made to correct for it during input. If the user notices a parentheses error of the type handled by DWIM while typing in the expression, he should ignore it.

6. For all other unbound atoms, DWIM assumes that there has been a spelling error and calls the spelling corrector (described below). The spelling corrector is given a list of possible choices consisting of all variables set by rpaqq, plus, in the case the error occurred in a user function, a list of lambda variables and prog variables for that function.

Undefined Functions

Fixfn is currently programmed to handle six different types of errors.

1. If the undefined function is T, and it appears in an expression of the form (COND -- (-- (T))), or (COND -- ((T --))), or immediately following a COND as in (COND --) (T --), DWIM assumes the T was meant to start the last clause in the conditional and makes the appropriate change.

2. If the undefined function is not atomic, DWIM assumes an extra parenthesis was put in e.g. (CONS ((CAR X) Y] or (COND (((ZEROP N) 1) (T (TIMES N (FACT (SUB1 N].

3. If the undefined function has a binding, and is three or fewer characters in length, DWIM assumes an extra parenthesis was put in e.g. (CONS (X Y)).

4. If the undefined function is F/L, DWIM assumes the user (intentionally) typed (F/L expression) meaning (FUNCTION (LAMBDA (X) expression)), or if (F/L arg-list expression1 ... expressionm)), meaning (FUNCTION (LAMBDA arg-list expression1 ... expressionm)).

5. If the undefined function contains an 8, DWIM assumes a left parentheses was intended. If the error occurred on input to evalquote, e.g. ← EDITF8FOO), the appropriate form is evaluated. If the error occurred inside of another expression, e.g. (ADD18CAR X), the undefined function is split into two parts and the 8 treated as a left parentheses, as in 4. of fixatom.

6. For all other cases, DWIM assumes a misspelling has occurred and the spelling corrector called. If the depth is greater than 3, the spelling corrector is given a list of low-level functions such as add1, atom, cons, etc., i.e. those typically used inside a function. If the depth is less than 3, the list contains top-level functions such as defineq, prettydef, makefile, load, etc. In both cases, the lists include all functions defined by the user with defineq, including those loaded from files.

User Modes - Interaction with DWIM

There are currently three modes of operation in DWIM: BEGINNER, NOVICE, and EXPERT. These modes control the setting of certain parameters that in turn determine the amount of interaction for the various errorsdiscussed earlier. Essentially, DWIM always asks the user for approval when in BEGINNER mode, and its messages are more explicit and verbose. In NOVICE mode, DWIM asks the user for approval for action to be taken when the error occurred in a user function, as opposed to an expression typed by the user to evalquote or break. DWIM rarely asks the user for approval in EXPERT mode, with the exceptions noted below.

Interaction on Parentheses Errors

Errors involving parentheses errors, i.e. errors of type 1,3,4 and 5 of _fixatom_ and errors of type, 1,2,3, and 5 of _fixfn_ require editing. DWIM types an error message: ...UNBOUND ATOM or ... UNDEFINED FUNCTION in BEGINNER mode, and U.B.A. or U.D.F. in NOVICE mode. This is followed by the atom or function name, and its location (i.e. IN _function_) if the error occurred in a user function. DWIM then requests permission to make the correction. For example, ...UNDEFINED FUNCTION 8SUB1 IN FACT FIX ?

At this point, the correction that DWIM intends to makes has already been determined by the type of error; DWIM is simply asking for user approval. If the user types Y, for YES, DWIM will proceed with the correction. If the user types N, for NO, or hits control-R the correction is aborted. If the user types ↑, not only DWIM but also the subsequent break is aborted, i.e. it is equivalent to typing control-R followed by ↑.* If the user types anything else, the editor is called, for the user to edit the expression himself.

When the editing has been completed, DWIM will type the transformation that was performed, e.g. 8SUB1 >>--> (SUB1, and then CONTINUE ?

* retfrom[n] where n is position of first errorset _previous_ to call to _faulteval_ is performed. If none, retfrom[Ø] is performed.

If the user types Y for YES, the computation continues, and the appropriate expression is evaluated. The only possible cause of trouble in continuing involves an error of the form (COND -- (-- (T --))). Here the value of the conditional should have been the value of the expression immediately preceding the (T --). This expression must be reevaluated since its value is no longer around. If it cannot be reevaluated without producing a harmful effect, the user should type N for NO. The cause of the error will have been fixed, but the computation will not continue. Instead an error will be generated, and the system will go into a break. If the user types anything other than Y or N, DWIM calls the editor again to allow the user to look at the changes it has made. After exiting from the editor in the normal way by typing OK, the user is again asked: CONTINUE ?, etc.

If the user is operating in NOVICE mode and the error did not occur in a user function, or if the user is in EXPERT mode, DWIM normally does not ask for approval before attempting a correction. In this case DWIM simply types EDITING function... or EDITING... and proceeds with the correction. There are two exceptions, however. For errors of type 2 and 3 in <u>fixfn</u> e.g. (CONS ((CAR X) Y)) and (CONS (X Y)), DWIM will <u>always</u> ask for approval, because these may be the expressions the user had intended: the undefined function error may have been generated because of a <u>logical</u> problem, e.g. X was bound to a function that was not defined.

If the user is operating in a mode that does not require DWIM to interact with him, he can still signal DWIM that he wishes to examine the changes that were made before continuing the computation by typing any character while DWIM is editing. When editing has been completed, DWIM will then operate exactly as though the user were in BEGINNER mode.

If DWIM was unable to make the correction, or if the user aborted the editing by typing control-R (<u>WARNING</u> in this case some changes may have been made before the user aborted), DWIM will type COULDN'T.

Interaction on Spelling Errors

Whenever an unbound atom or undefined function error occurs that is interpreted as a misspelling, DWIM types an error message followed by the atom or function identification as described earlier. The spelling corrector then attempts to select the best match with the list of correct spellings that it has been given. If no reasonable match can be found, the spelling corrector types the offending word, followed by = ?, e.g.

```
U.B.A. FOO
FOO = ?
```

The user can then type the correct word. If he mistypes again, the spelling corrector iterates. The only way to leave the spelling corrector is to give it a suitable word, or to abort by typing control-R or ↑. If the user does type in a suitable word, i.e. a BOUND ATOM or a DEFINED FUNCTION, the S-expression in which the misspelling occurred is corrected, and this word is added to the spelling list so that it will be considered as a possible correct spelling for future mistakes. In the case of an UNBOUND ATOM, the user can also type in a non-atomic form which is then used to correct the S-expression, e.g.

```
U.B.A. FOO
FOO = (CAR FIE)
```

If the spelling corrector finds a word or words which are reasonable matches, it types them followed by ?, e.g.

```
FOO = FOO1 or FOO2 ?
```

The acceptable answers to this question are ↑, Y, for YES (only if just one word matches); N, for NO, in which case WHAT THEN ? is typed and the user is in the same situation as when no suitable match was found; D, for DELETE, in which case these words are eliminated from the spelling list and will not be considered in the future; or a number, to indicate which of the various words typed is correct. The user is not allowed to type in the correct word in response to a question of the form FOO= FOO1 or FOO2 ? He can only type ↑, Y, N, D, or a number.

If the user is operating in NOVICE mode and the error does not occur in a user function, or if the user is operating in EXPERT mode, the spelling corrector does not require interaction if the match is "good enough." There are two parameters that affect this determination. These are CFACT1, initially set at .5, and CFACT2 initially set at .8. If the match is better than CFACT2 in reliability, i.e. at least 80% sure, the spelling corrector makes the correction types it, and goes on, e.g. FOOO = FOO. If the match is better than CFACT1 but less than CFACT2, the spelling corrector types what it thinks is the correct spelling, and then waits a specified number of milliseconds, as determined by the parameter DELAYTIME (initially set at 4000). If the user types any character during this time, the spelling corrector goes into interactive mode as described earlier. Otherwise it makes the correction and goes on.

Summary of Interaction by Modes

Parenthesis error in user function FACT, e.g.
(FACT 8SUB1 N)

BEGINNER	...UNDEFINED FUNCTION 8SUB1 IN FACT FIX ?
NOVICE	...U.D.F. 8SUB1 IN FACT FIX ?
EXPERT	EDITING FACT...
	unless type 2 or 3 of fixfn, in which case same as NOVICE mode.

Parenthesis error in user typed expression; e.g.
←E (FACT 8SUB1 FOO)

BEGINNER	...UNDEFINED FUNCTION 8SUB1 FIX ?
NOVICE	EDITING...
EXOERT	EDITING...
	unless type 2 or 3 of fixfn

Note: for input like ← EDITF8FOO) to evalquote, a subcase of error 5 of fixfn, DWIM requires no interaction in any mode.

Spelling error in user function, e.g. (FACCT (SUB1 N))

```
BEGINNER          ...UNDEFINED FUNCTION FACCT IN FACT
                  FACCT= *** ?

NOVICE            ...U.D.F. FACCT IN FACT
                  FACCT= *** ?

EXPERT            FACCT=FACT(IN FACT) if certainty > CFACT2

                  FACCT=FACT(IN FACT) and pause for specified time
                                      if certainty > CFACT1.  If
                                      no input, make correction
                                      and continue.

                  ...U.D.F. FACCT IN FACT
                  FACCT= *** ?        if certainty < CFACT1 or if
                                      two choices equally good
```

Spelling error in user expression, e.g. ← FACCT(4)

```
BEGINNER          ...UNDEFINED FUNCTION FACCT
                  FACCT= *** ?

NOVICE            FACCT=FACT if certainty > CFACT2

                  FACCT=FACT          and pause for specified time
                                      if certainty > CFACT1


                  ...U.D.F. FACCT      if certainty < CFACT1 or if
                  FACCT = *** ?        two choices equally good

EXPERT            same as NOVICE
```

(*** represents list of possible candidates, if any)

Private Modes

If the user wants to define his own mode, presumably to combine certain features of BEGINNER, NOVICE, or EXPERT, or to change the confidence thresholds, he must add the definition of his mode to the end of modelst. Each mode is a list consisting of a single identifying character (hence do not use B, N, or E), followed by the rest of the mode name, followed by a list of dotted pairs of variables and values. For example, the first entry on modelst is (E XPERT (GREENLIGHT . T) (REDLIGHT) (CFACT1 .5) (CFACT2 .8) (UDF . "U.D.F.") (UBA . "U.B.A.")).

If the variable greenlight is T, as in EXPERT mode DWIM never asks for interaction, (with the few exceptions mentioned earlier). If both greenlight and redlight are NIL, as in NOVICE mode, DWIM asks for interaction only for errors involving user functions. If redlight is T, as for BEGINNER mode, DWIM always asks for interaction. The variables udf and uba are the error messages printed out for undefined function and unbound atom errors respectively.

Other uses of the spelling corrector

As mentioned in the introduction, the spelling corrector is used in the DWIM package to correct certain misspellings that would not cause LISP errors. This has been accomplished by advising certain system functions, such as prettyprint to consult the spelling corrector when given an argument that does not "make sense" in the context of the operation they perform. Userwords is a list of words that may contain the spelling the user intended. This list is built up by certain other system functions, e.g. defineq, rpaqq which have been advised to add to this list. Thus if DWIM is enabled and the user loads a file, all functions defined and variables initialized in the file will be added to userwords.

System Functions Advised by DWIM

EDITF -	if argument is (NOT EDITABLE), but has a non-atomic value, editv is called. If it has a non-null property list, editp is called. Otherwise it is treated as the name of a function on userwords.
EDITV -	if argument is (NOT EDITABLE), it is treated as the name of a variable on userwords.
EDITP -	if argument is (NOT EDITABLE), it is treated as an atom on userwords with a non-null property list.
BREAKØ	if function is not defined, it is treated as the name of a function on userwords.
UNSAVEDEF -	if function is not defined, it is treated as the name of a function on userwords.
PRETTYPRINT -	if function is not defined, it is treated as the name of a function with an S-expression definition on userwords.

In addition, after they have finished operating, EDITF, EDITV, EDITP, DEFINEQ, BREAKØ, UNSAVEDEF, and RPAQQ all add their arguments to userwords as well as to the appropriate spelling lists for error correction

Load is advised to maintain two lists: a spelling list of user files, (filesplst) and a list of user files with the functions and variables they contain (filelst), derived from the first and third argument to prettydef.* makefile consults filesplst to correct the spelling of its argument, and editf, editv, and editp add their argument to filelst under the name of the appropriate file. The function newfiles can then be used to produce an updated version of any files that have been changed.

newfiles[flg] does makefile for every file on filelst that has been modified. If flg=T, the files are also listed.

(*) This assumes that the file was created by makefile, p. 14.21 i.e. that its name is of the form /Nname/.

If the user wishes to modify his own or other system functions, the following two functions will be useful:

addspell[x;flg] Adds x to <u>userwords</u>, and, if flg = T, to the spelling list for variables, otherwise to both spelling lists for functions. Sets <u>lastword</u> to <u>x</u>. If <u>x</u> is already on <u>userfns</u>, no action is taken.

misspelled? [x;fn;splst] If <u>fn</u> is not NIL and fn[x] is not NIL, value is <u>x</u>, i.e. <u>x</u> was not misspelled. If <u>x</u> is NIL, value is <u>lastword</u> e.g. after defining a function, you can edit it by simply performing editf[]. Otherwise spelling of atom is corrected using <u>splst</u>, if given, otherwise <u>userwords</u>. If spelling correction is aborted by control-r, value is x. If ↑ is typed to spelling corrector, control is returned to errorset prior to <u>misspelled</u>?

As an example of the use of these functions, the following advising operations are performed by DWIM.

```
ADVISE(DEFINEQ AFTER (MAPC VALUE (FUNCTION ADDSPELL)))
ADVISE(RPAQQ AFTER (ADDSPELL X T))
ADVISE(UNSAVEDEF (SETQ X (MISSPELLED? X (FUNCTION FNTYP)))
```

NOFIX

The user may not want to have DWIM bother to operate on certain types of errors. To facilitate this, DWIM calls the function nofix, of no arguments, before attempting to make any corrections. If nofix returns T, the correction is aborted. nofix is currently defined as (LESSP FAULTD FAULTDEPTH), with FAULTDEPTH set to -1. If the user wanted to disable DWIM for errors occurring near the top level, or just inside of a break, he simply needs to set FAULTDEPTH to an appropriate value, e.g. 5. The user can prescribe more complicated conditions for aborting corrections by advising nofix. For example, the user might prefer to retype any misspelling shorter than four characters in length. He would then advise nofix

```
ADVISE(NOFIX (AND (ATOM FAULTX) (LESSP (NCHARS FAULTX) 4)))
```

Of course, such operatives require more intimate knowledge of the operation of DWIM. For this, contact W. Teitelman.

SECTION XXI

PRINTSTRUCTURE

In trying to work with large programs, a user can lose track of the hierarchy which defines his program structure; it is often convenient to have a map to show which functions are called by each of the functions in a system. If fn is the name of the top level function called in your system, then typing in printstructure[fn] will cause a tree printout of the function-call structure of fn. To describe this in more detail we use the printstructure program itself as an example.

```
←PRINTSTRUCTURE(PRINTSTRUCTURE)
PRINTSTRUCTURE   PROGSTRUC PRGSTRC    PRGSTRC1   PRGSTRC1
                                                 PRGSTRC
                                      PRGSTRC
                                      NOTFN
                                      PROGSTRUC
                           CALLS1     NOTFN
                                      CALLS2     CALLS1
                                      PRGSTRC
                 PRINTSTRUCTURE
                 MAKECIRC  MAKECIRC
                 TREEPRINT TREEPRINT1
                           TREEPRINT
                 VARPRINT  TREEPRINT1

PRINTSTRUCTURE   [FN,FILE; X,DONELST,NODES; ]

PROGSTRUC [FN,D; Y,Z,FLG,VARS1,VARS2; DONELST]

PRGSTRC    [X,HEAD; Y,Z; VARS1,FN,DONELST,D,FLG]

PRGSTRC1   [L; A,B,Y; Y,VARS1,VARS2]

NOTFN      [FN; DEF; ]

CALLS1     [ADR,D; M,N,X,Y,FLG,V1,V2,FLG; VARS1,INTZRO,VARS2]

CALLS2     [X; ; D]

MAKECIRC   [X; X; NODES]

TREEPRINT [X,N; Z; ]

TREEPRINT1  [X,N; ; ]

VARPRINT   [X; X,Y,N; ]
PRINTSTRUCTURE
←
```

The upper portion of this printout is the usual horizontal version of a tree. This tree is straightforwardly derived from the definitions of the functions: *printstructure* calls *progrstruc*, itself, *makecirc*, *treeprint*, and *varprint*. *progstruc* in turn calls *prgstrc* and *calls*. *prgstrc* calls *prgstrcl* itself, and *notfn*. *prgstrcl* just calls itself and *prgstrc*. Note that a function whose substructure has already been shown is not expanded in its second occurrence in the tree.

The lower portion of the printout contains information about the variables that are used in each of the functions. *printstructure* is a function of two arguments, *fn* and *file*. It binds three variables internally: *x*, *donelst*, and *nodes*. (Variables are bound internally by either progs or open lambda-expressions.) *makecirc* has only one argument, *x*, and it also binds *x* internally. It uses the variable *nodes* as a free variable.

In addition to the five functions appearing in the above output. *printstructure* calls many other low-level functions such as *getd*, *car*, *list*, *nconc*, etc. The reason these do not appear in the output is that they were defined "uninteresting" by the user for the purposes of this analysis. Two functions, *firstfn* and *lastfn*, and two variables *yesfns* and *nofns* are used for this purpose. Any function that appears on the list *yesfns* is of interest, any function appearing on *nofns* is not. Otherwise, all non-compiled functions are deemed interesting, and only those compiled functions between the two limits established by *firstfn* and *lastfn*. For example, firstfn[editf] and lastfn[bo] (the last function in the edit package) followed by printstructure[editf] will cause the structure of the editor to be printed out with only those functions that actually are part of the edit package appearing in the printout.

Three other variables, notrace, quotefns, and prdepth also affect the action of printstructure. Functions that appear on the list notrace will appear in the tree, assuming they are "interesting" functions as defined above, but will not themselves be traced, i.e., analyzed. Functions that appear on quotefns are traced, assuming they are "interesting," but when they appear as car of a form, the rest of the form is not analyzed. For example, if the function prinq were defined as (NLAMBDA X (MAPC X (FUNCTION PRIN1))) and the form (PRINQ NOW IS THE TIME) appeared in a function being analyzed, prinq would appear in the tree, but NOW, IS, THE, and TIME would not be noted as free variables if prinq were included in the list of quotefns. prdepth is a cutoff depth for analysis. It is inially set to 100.

printstructure has incorporated in it the necessary information for analyzing non-standard forms such as cond, prog and selectq. It is also capable of analyzing compiled or interpreted functions equally well. In the case of compiled functions, printstructure will automatically analyze any functions generated by the compiler, such as those caused by compiling forms beginning with ersetq, nlsetq, or function.

If printstructure encounters a form beginning with two left parentheses in the course of analyzing an interpreted function (other than a COND clause) it notes the presence of a possible parentheses error by the abbreviation P.P.E., followed by the function in which the form appears and the form itself, as in the example below. Note also that printstructure detects functions, i.e., atoms appearing as CAR of a form, that are not defined. printstructure is thus a useful tool for debugging.

```
←PRETTYPRINT((FOO))

(FOO
  (LAMBDA (X)
    (COND
      (X (FOO1 X))
      (T ((CONS X (CAR X)))))))
NIL
←PRINTSTRUCTURE(FOO)
FOO          FOO1
             CONS
             CAR

FOO          [X; ; ]

FOO1         IS NOT DEFINED.

P.P.E. IN FOO - ((CONS X (CAR X)))

FOO
←
```

Printstructure Functions

printstructure[fn,file]

analyses structure of fn and stores result on property list of fn under property PRINTSTRUCTURE. The form of this result is a list of two elements, the second of which is the tree representation of the structure, and the first a list consisting, in alternation, of the functions that appear on the tree, and a variable list vlst for that function. CAR of the variable list is a list of variables bound in the function; CDR is those variables used freely in the function.

Thus in the printstructure[printstructure] example given earlier the value of the property PRINTSTRUCTURE would be:

```
((PRINTSTRUCTURE ((FN FILE X DONELST NODES)) PROGSTRUC
    ((FN D Y Z FLG VARS1 VARS2) DONELST) PRGSTRC
    ((X HEAD Y Z) VARS1 FN DONELST D FLG) ...
    VARPRINT ((X X Y N))) tree)
```

Possible parentheses errors are indicated by a non-atomic form appearing where a function would normally occur, i.e., in an odd position of the list. It is followed by the name of the function in which the P.P.E. occurred.

	If file=NOTHING, no output is produced. If file=SCOPE or DISPLAY, the structure is displayed on the scope. In this case, the result of the analysis contains a third element, the display figure that was generated by dtree. For any other value of file, output is to that file.
	The value of printstructure is fn.
treeprint[x;n]	when given a tree representation of the structure, (see dtree, p.18.8) e.g., cadr[getp[fn,PRINTSTRUCTURE]]. treeprint prints the tree in the horizontal fashion shown in the examples above. Used by printstructure.
varprint[x]	when given the list of functions and their variables, i.e., car[getp[fn,PRINTSTRUCTURE]], varprint prints the lower half of the examples shown above.
firstfn[fn]	fn is the name of a compiled function. If fn=T, lower boundary is set to 0, i.e., all compiled functions will pass this test. If fn=NIL, lower boundary set at end of bpspace, i.e., no compiled functions will pass this test. Otherwise boundary set at fn.

lastfn[fn]	if fn=NIL, upper boundary set at end of bpspace, i.e., all compiled functions will pass this test. Otherwise boundary set at fn. Thus to accept all compiled functions, perform firstfn[T], lastfn[NIL]
calls[fn]	returns a list of three elements: a list of all (interesting) functions called by <u>fn</u>, a list of variables bound in <u>fn</u>, and a list of variables used freely in <u>fn</u>, e.g., calls[prgstrc]= ((PRGSTRC1 PRGSTRC NOTFN PROGSTRUC) (X HEAD Y Z) (VARS1 FN DONELST FLG))
vars[fn]	cdr[calls[fn]]
freevars[fn]	cadr[vars[fn]]=caddr[calls[fn]]
allcalls[fn,tr]	prints fn IS CALLED BY: and returns list of all functions that call <u>fn</u> in the tree <u>tr</u>. If <u>tr</u> is atomic, cadr[getp[tr;PRINTSTRUCTURE]] is used. Example:

```
←ALLCALLS(PRGSTRC PRINTSTRUCTURE)

PRGSTRC IS CALLED BY:
(PROGSTRUC PRGSTRC1 PRGSTRC CALLS1)
←
```

Follow

Follow is a function that enables the user to dynamically watch the flow of computation through a collection of functions. Follow uses printstructure to analyze the hierarchy of functions and to display a tree structure on the scope. Every function appearing in the tree is then modified to appropriately adjust the display when it is entered. Whenever one of these functions is entered, a circle is drawn around the corresponding node in the display, and a dotted path is displayed from the function that called it. For example, if the user performs: firstfn[editf], lastfn[bo], follow[editf] and then calls the editor, he can watch the editor as it executes his commands. (Follow is not in current system but can be loaded from file /(FLIP)CFOL/.)

follow[fn;flg]	performs printstructure[fn] and then modifies (using break) all functions in the resulting tree to appropriately update the display. If flg=NIL and follow[fn] has been done previously, follow does not regenerate the display, instead it obtains it from the property FOLLOWED on the property list of fn. unbreak restores the functions to their unmodified state.
	followspeed initially set to 0, controls the delay time, in clock ticks, between changes in the display. It can be set to slow down what might otherwise be a confusingly rapid succession of changes in the display.

SECTION XXII

MISCELLANEOUS

TIME

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)
```

date[]		Obtains date and time from system and returns it as single atom in format "mm/dd/gg hhmm:ss". where mm is month, dd day, gg year, hh hours, mm minutes, ss seconds, e.g., "02/21/68 1352:41"
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 (garbage collection time is subtracted off)
	for n=3	time spent in garbage collections
		Value of clock is in "ticks."
tickps[]		Number of ticks per second, usually 50.

BREAKDOWN

Breakdown is a function that produces an analysis of computation time by function, although it can be used for an analysis of CONSes, drum references, garbage collection time, or any other single numerical quantity. The user calls breakdown giving it a list of functions of interest. These functions are modified so that they keep track of the "charge" assessed to them. The function results gives the analysis of the statistic requested as well as the number of calls to each function. Sample output is shown below.

```
←BREAKDOWN(SUPERPRINT SUBPRINT)
(SUPERPRINT SUBPRINT)
←PRETTYDEF(FOOFNS /FOO/ (STOP))
FOOFNS
←RESULTS()
FUNCTIONS   TIME     # CALLS
SUPERPRINT  6.30       731
SUBPRINT    3.94       274
TOTAL       10.22      1005
NIL
←
```

To add or remove functions from those being monitored, the user must call breakdown giving it the entire new list of functions. However, breakdown[] can be used for simply zeroing the counters associated with the function already being monitored.

To use breakdown for some other statistic, the user must set three variables: breakdownform, the quantity of interest, e.g., (CLOCK 2) or (CONSCOUNT), label, e.g., TIME or CONSES, and interp, (optional) which is a function that will be applied to the statistic to produce the numbers printed by results. Thus, since the value of (CLOCK 2) in the above example is in clock ticks, to convert to seconds, interp had been set to

(LAMBDA (X) (FQUOTIENT X (TICKPS))).

Whenever breakdown is called with breakdownform having a non-null value, breakdown performs the necessary changes to its internal state to conform to the new analysis. In particular, if this is the first time an analysis is being run with this statistic, the compiler is called to compile the charging function. When breakdown is through initializing, it sets breakdownform back to NIL. Subsequent calls to breakdown will use the new analysis until breakdownform is again set. Sample output is shown below.

```
←SET(BREAKDOWNFORM (CONSCOUNT))
(CONSCOUNT)
←SET(LABEL CONSES)
CONSES
←SET(INTERP NIL)
NIL
←BREAKDOWN(MATCH CONSTRUCT)
(A0272 COMPILING)
(A0272 (BDX BDY) (BDLST BDPTR) 5)
(MATCH CONSTRUCT)
```

```
←PILOT(T)
PROCEED:
FLIP((A B C D E F G H C Z) (.. $1 .. #2 ..) (.. #3 ..))
(A B D E F G H Z)

>RESULTS()
FUNCTIONS    CONSES     # CALLS
MATCH        32           1
CONSTRUCT    47           1
TOTAL        79           2
NIL
```

Each time breakdown encounters a new value for breakdownform, it saves the corresponding value of label, interp, and the compiled charging function. Thus, if breakdownform is set to a form already encountered, it is not necessary to also set label and interp, nor will the compiler be called.

To restore functions modified by breakdown to their original state, use unbreak.

TIMEX

Timex is a timing function that reports statistics on number of function calls, drum references, wraparounds, etc., as well as on computation time and conses. It accepts input from the teletype for eval, and prints an analysis after each input. Exit is achieved by typing OK at which point timex also gives the distribution of drum references analysis. Sample output is given below.

```
←

←TIMEX()


>(PRETTYDEF (QUOTE FOOFNS) (QUOTE /FOO/) (QUOTE (STOP)))
24 CONSES
10.06 SECONDS
1  WRAPAROUNDS
15762 MAPPED STORES
40923 TOTAL MAPS          1.64 SECONDS
57 TOTAL DRUM REFS        0.97 SECONDS
11 DRUM WRITES            0.19 SECONDS
5161 FUNCTION CALLS       7.74 SECONDS
15 BP DRUM READS


>(COMPILE (QUOTE (MAKEPDQ)))
LISTING?
F
(OUTPUT FILE)
N
(MAKEPDQ COMPILING)
GARBAGE COLLECTION

1906 FREE WORDS
(MAKEPDQ (F) NIL 5)
GARBAGE COLLECTION
690 FREE WORDS
2761 CONSES
34.56 SECONDS
2  WRAPAROUNDS
57510 MAPPED STORES
215088 TOTAL MAPS         8.60 SECONDS
2405 TOTAL DRUM REFS     40.88 SECONDS
266 DRUM WRITES           4.52 SECONDS
890 FUNCTION CALLS       12.58 SECONDS
86 BP DRUM READS
```

```
>OK

 DISTRIBUTION OF DRUM REFERENCES
 ARRAY SPACE          1458
 LIST SPACE           665
 PNAME SPACE          31
 PARAM PUSHLIST       16
 CONTROL PUSHLIST     8
 VALUE CELLS          61
 PROPLIST CELLS       50
 FUNCTION CELLS       63
 PNAMES & HASH CELLS  110
 BOXED NUMBERS        36

 DISTRIBUTION OF DRUM REFERENCES   WRITE-ONLY
 ARRAY SPACE          4
 LIST SPACE           174
 PNAME SPACE          5
 PARAM PUSHLIST       15
 CONTROL PUSHLIST     4
 VALUE CELLS          20
 PROPLIST CELLS       1
 FUNCTION CELLS       1
 PNAMES & HASH CELLS  25

 BOXED NUMBERS        36
 NIL
 ←
```

If the first argument to timex is not NIL, it is treated as a list of inputs for eval exactly as though they had been typed by the user. If the second argument is T, the analysis after each evaluation is suppressed. The third argument, also optional, is a number given to time as its second argument and controls the number of times each individual input is to be evaluated.

Enlarging the System

The BBN LISP system uses techniques which allow the system code, i.e., all of the functions described in this manual, plus those described in the FLIP manual, to be shared by all users. In addition to this (approximately) 90K of system code, each user can acquire another (approximately) 40K 940 words of private memory; this allows the user about 20K of LISP words. The LISP system can be enlarged up to a maximum size of 256K by reassembling it. However, an individual can effectively acquire additional private memory in the existing system by "flushing" some of the system code. The function flushcode is provided for this purpose. Its effects are not reversible, but the flushed material can be reloaded from the appropriate disc files.

flushcode[from;to;flg] — flushes all binary programs between from and to. If to = NIL, flushes all binary programs from from to the end of binary program space. If flg=T, it prints names of functions being flushed. from and to can be numbers corresponding to function definitions (*), or they can be any atom on the list flushpoints, which contains the starting locations of various independent subsystems such as display, flip, compiler etc. If to=T, just the subsystem from is flushed, e.g. flushcode [printstructure; T] will flush just printstructure. These systems have been loaded so that all functions used by any particular system are loaded before it. The user can thus safely flush any subsystems following the ones

(*) (LOGAND (GETD FN) 377777Q)

he is planning to use. Of course, independent systems such as display, compiler, etc. can always be flushed without affecting the operation of other systems.

When flushcode is called with to=NIL, the reclaimed space will be noted by storage. However, if the user makes a "hole" in his binary program space, no change will be observed in storage. However, when a request is made for an array or binary program allocation that can fit in the hole, the space will be taken from there. Bpspace is a list of elements of the form (N M) corresponding to the user holes.

Note: the user should not flush the same area twice, as it will then appear twice on bpspace and hence be used twice.

Dumping Circular List Structure

It is often important to save list structures so that reloading will maintain common substructure. In particular, this is critical for circular lists which cannot be saved any other way. The function *savecl* described below allows a user to dump a representation of a set of structures onto a file; later, reading them back in with the function *loadcl* obtains an isomorphic copy of the original structures.

The algorithm for dumping is a variation of one first suggested by Minsky for garbage collection using secondary storage (described in Bobrow, "Storage Management in LISP"). Each structure to be saved is traced, and as each new LISP word is seen, it is marked by placing a mark in its *car* and a number in its *cdr*. The number indicates where this word will be reloaded. The mark is a *gensym* created for marking all words which are to be considered part of this set of structures. The dumped representation consists of triples for each word in the structure(s): its new (relative) address, its car and its cdr, both of which are either atoms or the new addresses of structure. *Note that this process destroys the structures so traced.*

savecl[lst;file;ident]

lst is a list similar to the third argument to *prettydef*, e.g.,

(A B (PROP Q R S) (DEF FN1) STOP)

will specify saving the values of A and B, the entire property lists of Q, R, and S, and the definition of FN1. STOP specifies closing *file*.

If _file_ is initially open, writing is continued. If it is not open, it is opened and the date printed. If file=NIL (omitted) the primary file is used.

ident, the third argument to _savecl_, is used by the marking process. If it is omitted, it is supplied by _savecl_. In this case, the structures dumped by this call to _savecl_ will not have common substructures with any previously dumped. To preserve common substructures across several different calls to _savecl_, give for _ident_ the value of the previous call to _savecl_. For example, if the user performs

```
SAVECL(WORDS1 /FL1/)
(A0005 1354)
SAVECL(WORDS2 /FL2/ (A0005 1354))
(A0005 2938)
```

and subsequently reloads both files using _loadcl_, he will get a structural copy of the items specified by WORDS1 and WORDS2. If he reloads just /FL1/, he will get a copy of only those items specified in WORDS1.

loadcl[file; ... filen] is a CFEXPR*. Each argument is a file name corresponding to a file generated by savecl. loadcl uses a contiguous block of storage beyond that currently in use to load all structures in these files, changes the internal state of the user's LISP to bring this block within list space bounds, and then initiates a garbage collection to clean things up. Note: if loadcl is interrupted before finishing, strange pointers may be left around in regular list space.

An additional advantage of the savecl - loadcl process is that all restored lists are linearized in storage, and compacted onto as few pages as possible.

GROUP

Group is a function for parsing lists as LISP forms. It takes a list such as (CONS CAR X CDR RPLACA X Y), and tries to make a single LISP form out of it, in this case returning (CONS (CAR X) (CDR (RPLACA X Y))). The function nargs is used to determine the number of arguments for each function encountered in the list. Group allows less than that number to be specified, i.e., group[(CONS CAR X)]=(CONS (CAR X)), but if an expression cannot be grouped without some function exceeding its number of arguments, group returns NIL, e.g., group[(CONS CAR X CDR Y Z)]=NIL. If the user has functions with extra arguments in their definition that are never used, he can indicate the true number of arguments, for the purposes of group, by putting this number on the function's property list under the property NARGS. Group looks here before calling nargs. This is also the way to handle functions that take an indefinite number of arguments, such as list. Since nargs[LIST]=1, by definition, group[(LIST X Y CAR Z)]=NIL. However, if you first perform put[LIST;NARGS;100], then group[(LIST X Y CAR Z)] will yield (LIST X Y (CAR Z)) as desired. Group is not in basic system but can be obtained by loading file /(FLIP)CPILOT/

ARM

Arm is a function for modifying the effect of control-H. Its name derives from the expression "to arm an interrupt," which is essentially what arm does. arm takes as its argument a function, or the name of a function, and modifies interruptl so that when a control-H is typed following other teletype input, the arming function (the argument to arm) is called. If it returns NIL as its value, the normal (INTERRUPTED BEFORE ...) consequence of typing control-H occurs. However, if it returns a non-null value, the computation continues exactly as though the user had typed control-H followed by OK. For example, if FOO is defined as

```
(FOO
  (LAMBDA NIL
    (COND
      ((READP)
        (SELECTQ (READC)
          (D (DRUN)
            T)
          (S (HPSTOP)
            T)
          NIL)))))
```

and arm[FOO] is executed, then whenever the user types a D followed by a control-H, the display will be started without apparently disturbing the computation in progress. Similarly, typing an S will stop the display. Typing anything else followed by control-H will cause a normal interrupt. Similarly, if nothing is typed previous to a control-H, the arming function returns NIL and a normal interrupt occurs.

The function disarm of no arguments reverses the effect of arm and restores the definition of interruptl.

Note that since the gctrp feature, p. 10.3, operates by simulating a control-H, arm can be used to automate the operation(s) the user wishes performed whenever a garbage collection is about to occur, i.e. less than N conses away. For example, suppose the user wants the display turned off when there are fewer than 200 conses left. He would then perform gctrp[200], and arm[FOO], where FOO was defined as:

```
(FOO
  (LAMBDA NIL
    (COND
      ((LESSP (GCTRP)
          200)
        (HPSTOP)
        T))))
NIL
←
```

APPENDICES

Appendix 1

Converting LISP 1.5 programs to BBN LISP

Although we have put considerable effort in the design and implementation of the BBN LISP system into making it an upwards compatible extension of LISP 1.5 as implemented on CTSS at MIT or the Q-32 at SDC, nevertheless many LISP programs, particular large systems, cannot be transferred intact from LISP 1.5 to BBN LISP and expect to run. However, the modifications required are usually quite trivial once the user is aware of what is necessary. This appendix is designed to document and call attention to the various differences between the two systems, and to facilitate this transition.

The easiest type of incompatibility to handle are functions that are defined in LISP 1.5 but not defined in BBN LISP. These are:

advance
attrib
clearbuff
common
cpl
cset
csetq
dash
digit
dump
efface
endread
errorl
evlis
excise
flag
function
intern
label
leftshift
liter
max
min
mknam
numob
onep
opchar
opdefine
pause
plb
printprop
punch
punchdef
punchlap
readlap
recip
remflag
remob
search
select
speak
special
startread
tempus-fugit
traceset
uncommon
uncount
unspecial
untraceset

The functions *advance*, *clearbuff*, *dump*, *endread*, *errorl*, *pause*, *plb*, *punch*, *punchdef*, *punchlap*, *readlap*, *startread*, and *tempus-fugit* relate to input and output, or to the time sharing systems themselves. These features are always highly system dependent.

The functions *cpl*, *dash*, *digit*, *intern*, *liter*, *mknam*, *numob*, *opchar*, and *remob* relate to the way LISP 1.5 represents and treats atoms. In BBN LISP, there are no character objects; BOFFO also does not exist. *remob* is not necessary because any atom not being used is automatically reclaimed by the garbage collector and removed from the oblist. If the user's programs perform printname manipulations, he will want to use the BBN LISP functions *pack* and *unpack*, and perhaps *chcon*, *character*, *nthchar* and *nchars*.

The functions *special*, *unspecial*, *common* and *uncommon* relate to the LISP 1.5 compiler. Because of the structure of the BBN push-down list, it is unnecessary to make a variable special (or common) when it is to be shared by several compiled (or compiled and interpreted) functions. This is taken care of automatically.

The functions *speak* and *uncount* relate to the LISP 1.5 *cons* counter. In BBN LISP, the function *conscount* gives the number of conses. However, *getrp*, p. 10.3 is somewhat analagous to the LISP 1.5 function *count*, which causes a trap if more than a specified number of conses occur. Note: the function *count* *is* defined in the BBN LISP system, but its value is the count of number of LISP words used by its argument, which is a LISP S-expression.

The functions _cset_ and _csetq_ do not exist in BBN LISP because there are no APVALS; instead _set_ and _setq_ are used at all levels to change the binding of a variable. If performed when a variable is not locally bound, they will change the top level binding.

Excise is not defined. Since BBN LISP provides a substantially larger number of free words, than LISP 1.5, the user should not miss it. However, the function _flushcode_ can be used to recover space occupied by certain system functions.

The function _traceset_ and _untraceset_ are not defined. We feel that the function _breakin_ provides this capability and more.

The function _leftshift_ is called _lsh_ in BBN LISP.

The effects of the function _printprop_ can be achieved by using _prettydef_ with an appropriate third argument.

The function _efface_ is called _dremove_. There is also a function _remove_ which returns the same value but does not destroy the original list.

Function relates to the funarg device in LISP 1.5, which does not exist in BBN LISP.

This leaves the function _attrib_, _evlis_, _flag_, _label_, _max_, _min_, _onep_, _opdefine_, _recip_, _remflag_, _search_ and _select_ which do not happen to be implemented.

The next class of incompatibilities concerns functions that are defined in both systems but whose effects are different. The functions such as _cond_, _map_, _mapcon_, _maplist_, _prin1_, _print_, and read all give the same results as their LISP 1.5 counterparts when given arguments consistent with their LISP 1.5 definitions. However, the mapping functions in BBN LISP all permit extra arguments for various options, and similarly the input-output functions permit an extra argument to designate the destination or source of the operation. The function cond will return the same value as its LISP 1.5 counterpart if given clauses containing only two elements, but also permits an arbitrary number of clauses. Therefore, these differences should cause no difficulties.

The predicates and, member, numberp, and or in LISP 1.5 all return T or NIL. In BBN LISP, they return a "useful" quantity or NIL. When used in a cond as a predicate, the effect is the same. The only anomaly the user might encounter would occur if he compared the value of a BBN predicate to T, e.g., (EQ (NUMBERP X) T). In BBN LISP, numberp[x] is either x or NIL. Thus, one can write (COND ((NUMBERP X)) (0)), or even (OR (NUMBERP X) 0), instead of (COND ((NUMBERP X) X) (T 0)), as would be required in LISP 1.5.

The functions that really differ between the two systems are apply, eval, load, pack, trace and untrace. Since there is no a-list in BBN LISP, apply and eval are functions of two and one arguments respectively. The evaluation of variables depends on the state of the push-down list. However, the function evala of two arguments is provided to emulate the LISP 1.5 function eval complete with a-list.

The function pack in LISP 1.5 takes a character object and add it to BOFFO. It is used in the process of creating a new atom. The function pack in BBN LISP takes a list of atoms as its single argument and makes a new atom out of them. Thus, pack[x] in BBN LISP is equivalent to performing the following operations in LISP 1.5:

```
CLEARBUFF()
NIL
MAP (list-of-atoms (FUNCTION (LAMBDA (X)
    (MAP (GET X (QUOTE PNAME)) (FUNCTION (LAMBDA (Y)
       (MAP (UNPACK Y) (FUNCTION PACK ))))))))
NIL
E (INTERN (MKNAM))
new-atom
```

The functions trace and untrace are part of the debugging package. Their effects are similar to those of the LISP 1.5 functions, but in BBN LISP they take an indefinite number of arguments, whereas in LISP 1.5, they take a single argument which is a list of functions. Also the output produced by tracing differs.

The function load in BBN LISP expects a list of forms for eval; in LISP 1.5 load expects a list of doublets for evalquote. This may cause some difficulty initially. However, since most symbolic files are created by prettydef, which knows the format for load, once the user has made the initial transition, no further problems should occur. Note: one easy way to convert a file from LISP 1.5 format to BBN format is to perform:

```
READFILE(FOO file)
FOO
SETQ(FOO (MAPLIST FOO (FUNCTION (LAMBDA (X)
  (LIST (QUOTE APPLY) (LIST (QUOTE QUOTE) (CAR X))
     (LIST (QUOTE QUOTE) (CADR X)))))
       (FUNCTION CDDR)))
big-hairy-list
WRITEFILE(FOO new-file)
FOO
```

In addition to the differences discussed above, the user will experience difficulty if his LISP 1.5 programs make assumptions about the internal representation of atoms, function definitions, etc. To review these differences briefly:

In BBN LISP, _car_ of an atom is always its top level binding, _cdr_ of an atom is always its property list. Consistent with this, _car_ of a number is always the number itself, _cdr_ of a number is always NIL. If the user's LISP 1.5 programs make assumptions about _car_ of an atom, they will not work in BBN LISP.

In BBN LISP, function definitions are not kept on the property list, but in a special cell which can only be accessed by the functions _getd_ and _putd_. If the user's LISP 1.5 program assumes that the function definitions are on the property list, i.e., uses DEFLIST instead of DEFINE or DEFINEQ, it will not work.

The FEXPR's of LISP 1.5 are subdivided into FEXPR's and FEXPR*'s in LISP 1.5. Similarly, EXPR's in LISP 1.5 are subdivided into EXPR's and EXPR*'s. The type in each case is indicated by either LAMBDA or NLAMBDA, and the use of an atomic or non-atomic argument list. Trying to define an FEXPR in BBN LISP by using DEFLIST will not work.

There are no properties APVAL or PNAME on the property list of atoms in BBN LISP.

There is no a-list in BBN LISP.

Finally, the following two items should be noted: there are separate fixed and floating point arithmetic functions in BBN LISP; and atoms may not be split at column 72 in BBN LISP. A carriage return in BBN LISP is a separator character and will split an atom. If your LISP 1.5 file contained atoms split at column 72 and carried over in column 1, it will not read in correctly, i.e., you will get two atoms instead of one whenever an atom was split.

Appendix 2

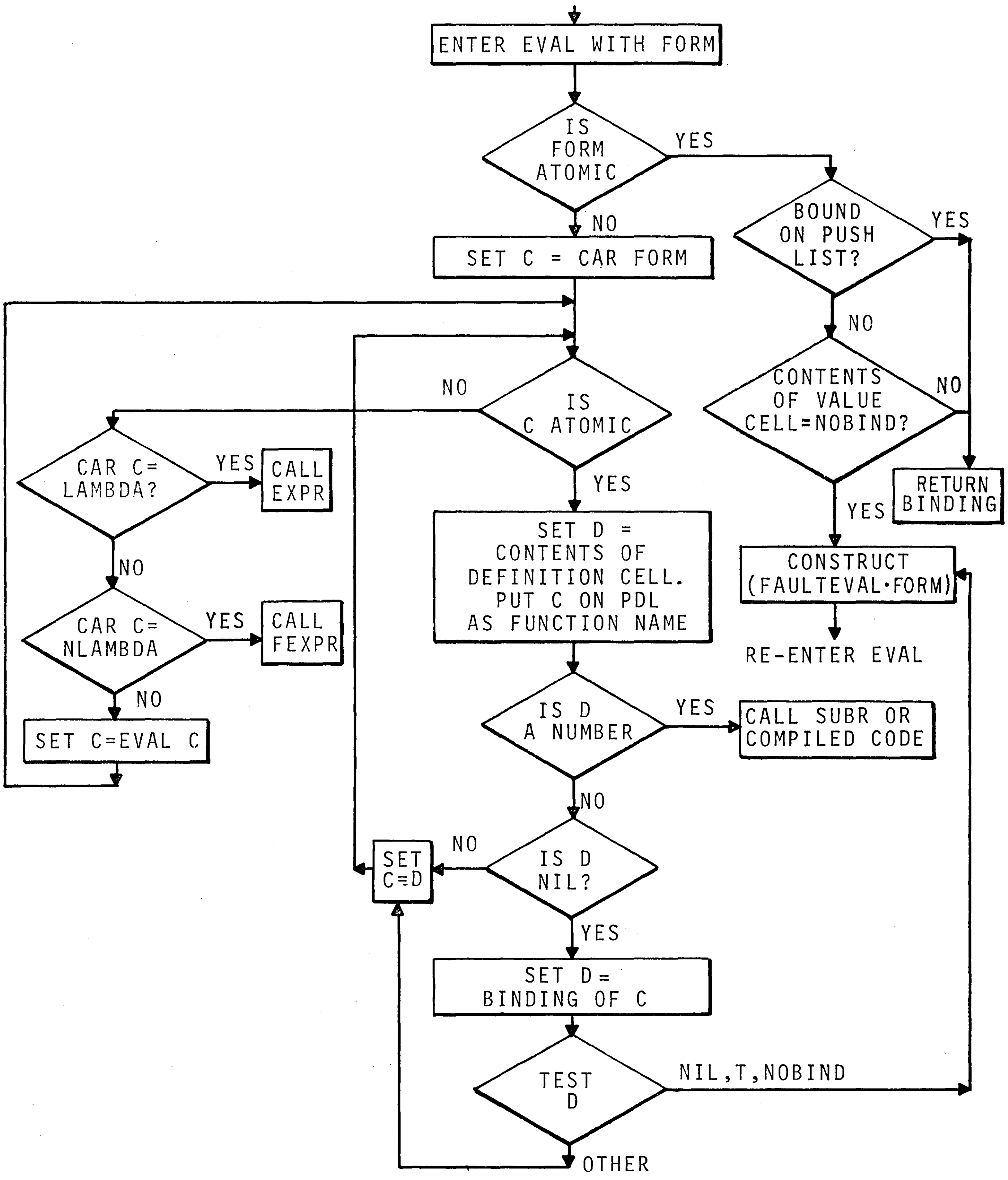
The BBN LISP Interpreter

The flow chart presented below describes the operation of the BBN LISP interpreter, and corresponds to the m-expression definition of the LISP 1.5 interpreter to be found on pp. 70-71 of the LISP 1.5 manual, McCarthy, 1966. Note that CAR of a form must identify a function, as with LISP 1.5 but the procedure the interpreter uses to obtain this function can be fairly complicated. The most common case occurs when CAR of the form is an atom. The function is then properly identified if its function cell contains:

(a) an S-expression of the form (LAMBDA ...) or (NLAMBDA ...) or
(b) a number which is the address of a SUBR or a block of compiled code.

Otherwise, if the function cell contains NIL, the atom is evaluated, and its value treated as if it were CAR of the original form. (This is the way functional arguments work.) If the evaluation of the atom produced NIL or T, or if the atom is unbound, the form is considered faulty, and faulteval is called as described in section 15.

If the function cell contains an expression other than NIL, this expression is evaluated and treated as though it were car of the original form, etc. In other words, it is possible to put the name of a function, or a form which computes the name of a function, in either the function cell or the value cell of an atom, and then to use the atom as a function.



Note: Variables c and d are for description only; they are not actually bound as regular variables.

Appendix 3

Control Characters

Several control teletype characters are available to the user for communicating directly to LISP, e.g. to abort or interrupt a computation, tell LISP to start or stop listening to the teletype, change the printlevel, etc. This section summarizes the function of these characters and references the appropriate section of the manual where a more complete description may be obtained.

1. Rubout

Clears tty input and output buffers. For example, the user would use rubout if he had "typed ahead" of the program, (typed while the program was computing but not listening), and wanted to eliminate all unprocessed input. Also this action may be useful when the program is producing a large quantity of tty output and the user wishes to skip some but not all of it. Operation of rubout when the program is doing output to tty will cause 30-40 characters to be skipped, but allow output to continue.

Rubout also terminates and restarts the tty input fork. If LISP has stopped listening because of a control-S, or its buffer is full, or some other reason, rubout will start it listening again.

Note, closely spaced repeated rubouts will terminate LISP and cause a return to the exec. This type of return may make it impossible to continue the LISP. It should only be used if control-C and/or logout absolutely fail to work.

2.	Control-R	Generates an error, type 1, p. 15.30.
3.	Control-C	Causes immediate return to evalquote top level (after garbage collection if one is in progress).
4.	Control-H	At next function call, computation is interrupted and breakl is called, p. 15.25 or other special action explicitly determined by user. (p. 22.2) Note if you are in a compiled function that does not call any other functions, the control-H will not take effect until the function is exited.
5.	Control-T	Print Time. Causes an immediate printout of the total execution time (in clock ticks) for the job, (even during garbage collections), i.e. clock[2], (p. 14.22). A series of such printouts should show increasing numbers if the program is using any CPU time. Of course, the program is not using CPU time if it is waiting for input. This information may serve to help the user determine if his LISP and/or the time sharing system has crashed.
6.	Control-F	Set Free Words. Control-F followed by a decimal number followed by a period will immediately cause minfs to be set to the indicated number, even if a garbage collection has already started. The garbage collector uses the setting of minfs about 3/4 ths of the way through, so if control-F is typed after this time, it will not take effect until the next garbage collection (p. 10.3)

7. Control-P	Set <u>printlevel</u>. Control-P followed by a decimal number followed by an exclamation point will change the printlevel to that number, i.e. it is the same as calling printlevel except that it may be performed while output is actually in progress. Control-P followed by a decimal number followed by a period will change printlevel for the current S-expression only. (14.4-14.5)
8. Control-S	Terminates LISP teletype input fork, thus preventing LISP from gobbling input and servicing other control character requests. Can be reversed by rubout - see 1 above. Normally this option would only be used if the user wished to type input to be processed by the time sharing system executive while he was still inside of LISP, e.g. he had a compilation going and wanted to leave instructions for exiting from the system. He could do this by typing LOGOUT(), control-S, and EXI.
9. Control-A,Q	Control characters for READ, see p. 2.1.

Appendix 4

Index to Variables

Following is a list of those atoms in the basic system which are initialized with top level values. Atoms that may be of interest to the user are given a brief explanation and a page reference, where applicable. Atoms which are internal and should normally not be of interest to the user, e.g. list of instruction codes, various compiler flags, etc. are listed to avoid inadvertent clobbering by the user.

In addition to the atoms with top level bindings, the variables LAMBDA, NLAMBDA, and lower case single character atoms a,b, ...z, which are printed as %A, %B ... %Z are used by the interpreter (see p. 12.4) and the user should not use them in his functions.

General	Description and/or Value
T	T
NIL	NIL
F	NIL
T	T
NOBIND	means atom has no top level binding, see p. 15.22
STATCELLS	list of dotted pairs of form (name . number), e.g. (WRAPAROUNDS . 199) where name is a statistic kept by the system, and number the location of the cell containing that statistic, e.g. openr[199] gives number of wraparounds. Used by timex, p. 22.5-7.

DFNFLG	used by <u>define</u>, <u>defineq</u>, and <u>unsavedef</u> to determine whether function definitions are saved on property lists, p. 8.3
HELPDEPTH, HELPFLAG	used by error handling routiners to determine whether or not to call <u>breakl</u>, p. 15.23, 15.25, 15.26
NHERRORS	list of non-helpable errors, i.e. those error types for which breaks do not occur, initialized to (0 1 18), see p. 15.30, 15.31, 15.35.
ESGAG	controls backtrace printout during unwinding, p. 15.34, 15.35.
FLUSHPOINTS	list of form (name . location) for use by <u>flushcode</u>, p. 22.8
BPSPACE	list of "holes" in user's binary program space, p. 22.8.
CLITRLIST	used by garbage collector
ADVISEDFNS	list of functions that have bee advised, p. 19.7
SYSTEMS	used by <u>subsys,</u> p. 17.10
ICP, CP, IPP, PP	push-down handles, p.12.11

<u>Compiler</u>

OPENFNS	list of functions to be compiled open, p. 16.12.
LAPFLG, STRF, SVFLG, NLAMA, NLAML, LCFIL, LSTFIL	answers to compset questions p.16.6-9
SYSNIL, SYST, SYSTAT, SYSNUM, SYSINT, TOPBPS, FREELW, CTEMP, INTZRO, SPCELL, PPPTR, FPPTR	p. 16.28
PREDLIST, PREDS1, PMAC, SYS, SYSFNS, SYSNOTFNS, LAMS	internal to compiler.

Break

BROKENFNS	list of functions that are broken, p. 15.13, 15.14, 15.18
BRKEVQFLG	used by breakl to determine whether user is talking to evalquote or eval when inside a break , p. 15.10.
BREAKMACROS	p. 15.8
BREAKI	number of spaces indented on each break, initially set to 3
CHNGNM	p.
NBREAKS	bound to number of recursive breaks, Ø at top level.
NOBREAKS	list of functions ignored by breakin search, p. 15.17
CHNGNM, VVNLAST, NOBREAKIN	internal

Editor

EDITMACROS	p. 9.17
MAXLEVEL	controls depth of find command, p. 9.7, initialized to 100
SUPEREDITFLG	p. 9.26, 9.27
EDITMAKROS, EDITOPS, EDRPTCNT, FINDFLAG	internal to editor

Display

DISPLAYMACROS	p. 18.6
CONVERTERLST	list of D-A converters to be used, p. 18.2
XORG, YORG, RORG, LORG, SCALEX, SCALEY, CHARSIZE	p. 18.11

HANDLE	p. 18.16
MASKTABLE	masks for characters, p. 18.14
TRNPTS, TRSPACING	used by dtree p. 18.8
ASSIGNFLG, CHARTABLE, CHARRAY, CA, PNO	internal to display

Breakdown

BREAKDOWNFORM	p. 22.4
STATFACTORS, CTTAB, MORESTATCELLS, CWTAB, BKDWNFN, BKDWNFMS, BDLST, BDPTR, BRKOVHD, BRKTERP, BRKLABEL	internal breakdown

Printstructure

YESFNS, NOFNS, QUOTEFNS, NOTRACE, PRDEPTH	p. 21.3, 21.4
FIRSTLOC, LASTLOC	internal

Edita

USERSYMS, SYMLST	p. 9.46
EDITBRK, OPCODES	internal

FLIP

FLIPBRKSAVE, FLIPBRK, A"(NIL)", D"(NIL)", "(T)", FLIPMODE, LASTREAD, UNRD, NORMBRK, SEPRS, SAVELST, $TRAN, TRAC, RTRAC, FLIPDEFAULT, INUSE, NOCONS, SAFE2USE see FLIP Manual

DWIM

CFACT1, CFACT2, DELAYTIME	Spelling correction, p. 20.11
USERWORDS	p. 20.13, 20.15
LASTWORD	p. 20.15
REDLIGHT, GREENLIGHT, UBA, UDF	modes, p. 20.13, 20.14
MODELST	list of mode definitions, p. 20.13
FAULTDEPTH	nofix, p. 20.16
FILELST, FILESPLST,	used by newfiles, p. 20.14
SPELLINGS1, SPELLINGS2, SPELLINGS3, SPELL1, SPELL2, SPELL3, FIX8, FIX9, FIXT, OPTIONSFNS	internal

PRETTYPRINT

FIRSTCOL, LASTCOL, ABBREVLST	used by comment feature, p. 14.15

MISCELLANEOUS

MAPRINTL, MAPRINTJ, MAPRINTI, APN, PATV, PFPA, PENT, PIPV, PRETN, PCLL, PMKN, PFVE, PCONS, PCDR, PCAR, SINTABLE	miscellaneous internal

Appendix 5

Properties and property lists

The following properties are used by system functions:

OPD, MACRO, COREVAL, CTYPE, CROPS	compiler
CLITRLIST	garbage collector
EXPR, CODE	savedef, unsavedef
BROKEN, BROKEN-IN, NAMESCHANGED, ALIAS	break
ADVISED, ADVICE	advise

The following atoms have their property lists initialized to other than NIL; the user should avoid clobbering them; i.e. use DEFLIST, PUT and REMPROP, not RPLACD, for modifying property lists.

PROPERTY OPD:

```
(SKD LDE STE BAC ROV OVT NOD FSB WCI WCH GCI PSTR PLAI SUC
ADC STP LDP TCI TCO CTRL BIO WIO CIO EXU NOP LCY RCY SKR
SKN SKB SKA SKM BRR BRM BRX CLAB CXA CBA CAX CLX SUB MIN
ADM EAX STX BSS MUL XAB DIV PFFV PMFN XXB CBX CAXB CLLXA
CLLX LDN PSAI NSTA BRU SKE ARGSUB CNA SARGN ARGN VSTI CSPI
CLA XMA LDA LDB STB SXMA STA SKG MPY DVD NEG EOR MRG ETR
SWAP FUNBOX FENBOX FDV FMP FAD CLLA NSTT BXC CAB CLB VAL
LDT STN STT LDV JUMP STV LDX CLL LQT RET SETIX BUI BII BUF
BIF BR2 BR1 BNLST BLST BNS BIS BNI BNA BA BNAP BAP BNN BN
BNE MSAI BRS BE LFV BI DIVIDE ADD RSH LRSH LSH BSS1 ENBOX
UNBOX CONSCLL PATV PENT PIPV PRETN PCLL PMKN PFVE PCONS
PCDR PCAR)
```

PROPERTY MACRO:

```
(// AC IFPRED MATCH4 FAILURE EVALPRED TRANSLATE3 KWOTE ASSEMBLE
ZEROP SUB1 SETA NLSETQ MINUSP MAP EVQ ERSETQ ELT DIFFERENCE
ABS RSH LRSH LSH FRPLAC NLISTP MEMB ADD1 NEQ MAPC * VAG
SETARG LOC LIST ARG)
```

PROPERTY COREVAL:

```
(ATVAL FPDLA ENTER IPV XCLL ENBOX UNBOX CONSCLL CDRCLL CARCLL
CP PP IPP ICP FPPTR PPPTR SPCELL INTZRO CTEMP FREELW TOPBPS
SYSINT SYSNUM SYSTAT SYST SYSNIL PATV PENT PIPV PRETN PCLL
PMKN PFVE PCONS PCDR PCAR RETURN)
```

PROPERTY CTYPE:

```
(LISTP LESSP DIVIDE ARRAYP CDDDDR CDDDAR CDDAAR CDADDR CDADAR
CDAADR CDAAAR CADDDR CADDAR CADADR CADAAR CAADDR CAADAR
CAAADR CAAAAR FIXP CDDADR TIMES REMAINDER QUOTIENT PLUS
OR NUMBERP NULL NOT MINUS LOGXOR LOGOR LOGAND GREATERP FTIMES
FQUOTIENT FPLUS FMINUS FLOATP EQ CDDR CDDDR CDDAR CDAR CDADR
CDAAR CADR CADDR CADAR CAAR CAADR CAAAR ATOM AND)
```

PROPERTY CROPS:

```
(CDDDDR CDDDAR CDDAAR CDADDR CDADAR CDAADR CDAAAR CADDDR
CADDAR CADADR CADAAR CAADDR CAADAR CAAADR CAAAAR CDDADR
CDDR CDDDR CDDAR CDAR CDADR CDAAR CADR CADDR CADAR CAAR
CAADR CAAAR)
```

In addition, the atom CLITRLIST has a property value LITERALS which is used by the garbage collector, and the atoms HELP-IN-UNSAVED1, PRINT-IN-PRETTYPRINT1, ERROR-IN-EDITF, ERRORSET-IN-EDITE, MAKEFILE, BREAKO, EDITP, EDITV, EDITF, RPAQQ, DEFINEQ, LOAD, EDITE, PRETTYPRINT1, and UNSAVED1 have property lists used by DWIM, see p. 20.13.

INDEX TO FUNCTIONS *

NOTE: ALL FUNCTIONS ARE LAMBDA, SPREAD UNLESS INDICATED OTHERWISE USING THE CODE NL=NLAMBDA, *=NOSPREAD

NAMES IN PARENTHESES, E.G. (DWIM), (ADVISE), REFER TO FLUSHPOINTS.

* FOR INDEX TO VARIABLES SEE APPENDIX 3, P. 23.9

NAME OF FUNCTION — DESCRIPTION PAGE

NAME OF FUNCTION	DESCRIPTION PAGE
FIXP[X]	13.4
FLOAT[U]	13.4
FLOATP[U]	13.4
FLTFMT[U]	13.4
FLUSHCODE[FROM,TO,FLG]	22.8
FMINUS[U]	13.4
FNTYP[X]	8.2
FOLLOW[FN,FLG]	21.9
FORKINIT[] (FORK)	17.13
FPLUS[--]*	13.5
FQUOTIENT[U,V]	13.5
FREEVARS[FN] (PRINTSTRUCTURE)	21.8
FTIMES[--]*	13.5
FUNCTION[--]NL*	11.1
GCGAG[U]	1Ø.4
GCTRP[U]	1Ø.3
GENSYM[]	1Ø.2
GET[X,Y]	7.2
GETD[U]	8.1
GETL[X,Y]	7.2
GETP[X,Y]	7.2
GINFN[U]	14.12

NAME OF FUNCTION	DESCRIPTION PAGE
LI[N,X] (EDIT)	6.7
LINELENGTH[U]	14.11
LIST[--]*	6.1
LISTFILE[FILES]NL* (MSC)	14.21
LISTP[X]	5.9
LO[N,X] (EDIT)	6.7
LOAD[LOADF,SVFLG,LOADFLG]	14.13
LOADCL[FILES]NL* (SAVECL)	22.12
LOC[U]	1Ø.4
LOG[X]	13.5
LOGAND[--]*	13.2
LOGOR[--]*	13.2
LOGOUT[U]	1Ø.4
LOGXOR[--]*	13.2
LRSH[X,N]	13.3
LSH[X,N]	13.3
LSUBLIS[X,Y] (MSC)	6.5
MAKEFILE[FL,FLG] (MSC)	14.21
MAKELIST[N,M] (MSC)	6.8
MAKETABLE[PLACE,SIZE] (FORK)	17.14
MAP[MAPX,MAPFN1,MAPFN2]	11.1
MAP2C[MAPX,MAPY,MAPFN1,MAPFN2]	11.3

NAME OF FUNCTION	DESCRIPTION PAGE
MAP2CAR[MAPX,MAPY,MAPFN1,MAPFN2]	11.3
MAPA[MAPARY,MAPFN1,MAPFN2,MAPN] (MSC)	11.3
MAPC[MAPX,MAPFN1,MAPFN2]	11.2
MAPCAR[MAPX,MAPFN1,MAPFN2]	11.2
MAPCON[MAPX,MAPFN1,MAPFN2]	11.3
MAPCONC[MAPX,MAPFN1,MAPFN2]	11.2
MAPDL[MPDLFN,MPDLN] (BREAK)	12.9
MAPLIST[MAPX,MAPFN1,MAPFN2]	11.2
MAPRINT[LST,FL,L,R,S,PFN,C] (MSC)	11.4
MEMB[X,Y]	5.11
MEMBER[X,Y]	5.11
MINFS[U]	10.4
MINUS[U]	13.1
MINUSP[X]	13.2
MISSPELLED?[X,FN,SPLST] (DWIM)	20.15
MOVD[FROM,TO,COPYFLG] (MSC)	8.1
MOVEFIG[MVFG]NL* (DISPLAY)	18.10
MOVEPOINTS[MATRIX,DX,DY,HANDL,REL,NP,FROM] (FORK)	18.22
NARGS[X]	8.5
NCHARS[U]	10.1
NCONC[U,V]	6.1
NCONC1[LST,X] (MSC)	6.1

NAME OF FUNCTION	DESCRIPTION PAGE
PLOTARRAY[AP,DX,DY,SCLX,SCLY,VERT,HANDL,REL,NP] (FORK)	18.21
PLUS[--]*	13.1
POSITION[U]	14.11
PRETTYDEF[PRTYX,PRTYY,PRTYL] (PRETTY) .	14.16-2Ø
PRETTYPRINT[L] (PRETTY)	14.15
PRIN1[U,V]	14.3
PRIN2[U,V]	14.3
PRIN3[U,V]	14.4
PRINT[U,V]	14.4
PRINTDATE[] (PRETTY)	14.21
PRINTDEF[E] (PRETTY)	14.16
PRINTFNS[X] (PRETTY)	14.21
PRINTLEVEL[U]	14.4
PRINTSTRUCTURE[FN,FILE] (PRINTSTRUCTURE)	21.6
PROG[--]NL*	5.7
PROG1[U,V]	5.6
PROG2[U,V]	5.6
PROGN[--]NL*	5.6
PROGN[IMPLICIT]NL*	4.3-4
PROP[X,Y,U]	7.1
PUT[X,Y,Z]	7.1
PUTD[U,V]	8.1

NAME OF FUNCTION — DESCRIPTION PAGE

NAME OF FUNCTION	DESCRIPTION PAGE
REMOVE[A,X]	6.3
REMPROP[X,Y]	7.1
RENAME[U,V,W]	12.7
RENAMEFILE[OLD,NEW] (MSC)	14.12
RESET[]	15.34
RETFROM[U,V]	12.8
RETURN[U]	5.8
REVERSE[X]	6.3
RI[M,N,X] (EDIT)	6.7
RO[N,X] (EDIT)	6.7
RPLACA[U,V]	5.3
RPLACD[U,V]	5.3
RPT[RPTN,RPTF]	5.7
RSH[X,N]	13.3
RTFRM[RTFN,RTFORM,RTN]NL (DWIM)	12.8
SASSOC[XSAS,YSAS,USAS]	7.3
SAVECL[LST,FILE,IDENT] (SAVECL)	22.10
SAVEDEF[X]	8.3
SEARCHPDL[SRCHFN,SRCHM] (DWIM)	12.10
SELECTQ[SELCQ]NL*	5.5
SET[U,V]	5.8
SETA[A,N,V]	10.7

NAME OF FUNCTION	DESCRIPTION PAGE
SETARG[U,V]	8.6
SETBRK[--]*	14.7
SETBRKC[X]NL*	14.8
SETQ[--]NL*	5.8
SETQQ[XSET,YSET]NL	5.8
SETSEPR[--]*	14.7
SETSEPRC[X]NL*	14.8
SETV[U,V,W]	12.7
SINE[THETA] (DISPLAY)	13.6
SOME[SOMEX,SOMEF] (MSC)	11.4
SPACES[U,V]	14.4
START[] (DISPLAY)	18.2
START1[] (DISPLAY)	18.2
STATISTICS[]	1Ø.5
STFK[ADDRESS,X] (FORK)	17.16
STORAGE[F]	1Ø.6
STOREFORK[AP,INCR,HANDL,REL,NP] (FORK)	17.15
STOREPOINTS[AP,DX,DY,SCLX,SCLY,HANDL,REL,NP] (FORK)	18.19
SUB1[X]	13.1
SUBLIS[A,Y]	6.4
SUBPAIR[N,V,L,FL]	6.6
SUBST[X,Y,Z]	6.3

NAME OF FUNCTION — DESCRIPTION PAGE